

LECTURE NOTES ON DATA STRUCTURES USING C

CONTENTS

BASIC CONCEPTS

Introduction to Data Structures
Data structures: organizations of data
Abstract Data Type (ADT)
Selecting a data structure to match the operation
Algorithm
Practical Algorithm design issues
Performance of a program
Classification of Algorithms
Complexity of Algorithms
Rate of Growth
Analyzing Algorithms
Exercises
Multiple Choice Questions

LINKED LISTS

Linked List Concepts
Types of Linked Lists
Single Linked List
 Source Code for the Implementation of Single Linked List
Using a header node
Array based linked lists
Double Linked List
 A Complete Source Code for the Implementation of Double Linked List
Circular Single Linked List
 Source Code for Circular Single Linked List
Circular Double Linked List
 Source Code for Circular Double Linked List
Comparison of Linked List Variations
Polynomials
 Source code for polynomial creation with help of linked list
 Addition of Polynomials
 Source code for polynomial addition with help of linked list:
Exercise
Multiple Choice Question

STACK AND QUEUE

Stack

- Representation of Stack

- Program to demonstrate a stack, using array

- Program to demonstrate a stack, using linked list

Algebraic Expressions

Converting expressions using Stack

Conversion from infix to postfix

- Program to convert an infix to postfix expression

- Conversion from infix to prefix

- Program to convert an infix to prefix expression

- Conversion from postfix to infix

- Program to convert postfix to infix expression

- Conversion from postfix to prefix

- Program to convert postfix to prefix expression

- Conversion from prefix to infix

- Program to convert prefix to infix expression

- Conversion from prefix to postfix

- Program to convert prefix to postfix expression

Evaluation of postfix expression

Applications of stacks

Queue

- Representation of Queue

- Program to demonstrate a Queue using array

- Program to demonstrate a Queue using linked list

Applications of Queue

Circular Queue

- Representation of Circular Queue

Deque

Priority Queue

Exercises

Multiple Choice Questions

RECURSION

Introduction to Recursion

Differences between recursion and iteration

Factorial of a given number

The Towers of Hanoi

Fibonacci Sequence Problem

Program using recursion to calculate the NCR of a given number

Program to calculate the least common multiple of a given number

Program to calculate the greatest common divisor

Exercises

Multiple Choice Questions

BINARY TREES

Trees

Binary Tree

Binary Tree Traversal Techniques

- Recursive Traversal Algorithms

- Building Binary Tree from Traversal Pairs

- Binary Tree Creation and Traversal Using Arrays

- Binary Tree Creation and Traversal Using Pointers

- Non Recursive Traversal Algorithms

- Expression Trees
 - Converting expressions with expression trees
- Threaded Binary Tree
- Binary Search Tree
- General Trees (m-ary tree)
 - Converting a *m-ary* tree (general tree) to a binary tree
- Search and Traversal Techniques for m-ary trees
 - Depth first search
 - Breadth first search
- Sparse Matrices
- Exercises**
- Multiple Choice Question**

SEARCHING AND SORTING

- Linear Search
 - A non-recursive program for Linear Search
 - A Recursive program for linear search
- Binary Search
 - A non-recursive program for binary search
 - A recursive program for binary search
- Bubble Sort
 - Program for Bubble Sort
- Selection Sort
 - Non-recursive Program for selection sort
 - Recursive Program for selection sort
- Quick Sort
 - Recursive program for Quick Sort
- Priority Queue and Heap and Heap Sort
- Max and Min Heap data structures
- Representation of Heap Tree
- Operations on heap tree
- Merging two heap trees
- Application of heap tree
- Heap Sort
 - Program for Heap Sort
- Priority queue implementation using heap tree
- Exercises**
- Multiple Choice Questions**

References and Selected Readings

Index

Basic Concepts

The term data structure is used to describe the way data is stored, and the term algorithm is used to describe the way data is processed. Data structures and algorithms are interrelated. Choosing a data structure affects the kind of algorithm you might use, and choosing an algorithm affects the data structures we use.

An Algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. No matter what the input values may be, an algorithm terminates after executing a finite number of instructions.

1.1. Introduction to Data Structures:

Data structure is a representation of logical relationship existing between individual elements of data. In other words, a data structure defines a way of organizing all data items that considers not only the elements stored but also their relationship to each other. The term data structure is used to describe the way data is stored.

To develop a program of an algorithm we should select an appropriate data structure for that algorithm. Therefore, data structure is represented as:

$$\text{Algorithm} + \text{Data structure} = \text{Program}$$

A data structure is said to be *linear* if its elements form a sequence or a linear list. The linear data structures like an array, stacks, queues and linked lists organize data in linear order. A data structure is said to be *non linear* if its elements form a hierarchical classification where, data items appear at various levels.

Trees and Graphs are widely used non-linear data structures. Tree and graph structures represents hierarchial relationship between individual data elements. Graphs are nothing but trees with certain restrictions removed.

Data structures are divided into two types:

- Primitive data structures.
- Non-primitive data structures.

Primitive Data Structures are the basic data structures that directly operate upon the machine instructions. They have different representations on different computers. Integers, floating point numbers, character constants, string constants and pointers come under this category.

Non-primitive data structures are more complicated data structures and are derived from primitive data structures. They emphasize on grouping same or different data items with relationship between each data item. Arrays, lists and files come under this category. Figure 1.1 shows the classification of data structures.

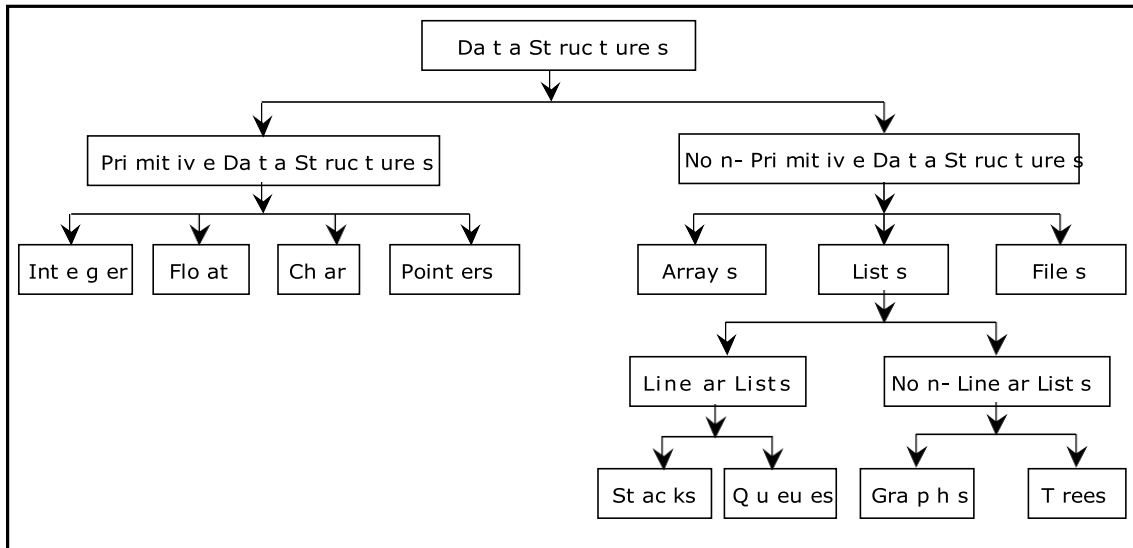


Figure 1. 1. Classification of Data Structures

1.2. Data structures: Organization of data

The collection of data you work with in a program have some kind of structure or organization. No matter how complex your data structures are they can be broken down into two fundamental types:

- Contiguous
- Non-Contiguous.

In contiguous structures, terms of data are kept together in memory (either RAM or in a file). An array is an example of a contiguous structure. Since each element in the array is located next to one or two other elements. In contrast, items in a non-contiguous structure are scattered in memory, but are linked to each other in some way. A linked list is an example of a non-contiguous data structure. Here, the nodes of the list are linked together using pointers stored in each node. Figure 1.2 below illustrates the difference between contiguous and non-contiguous structures.

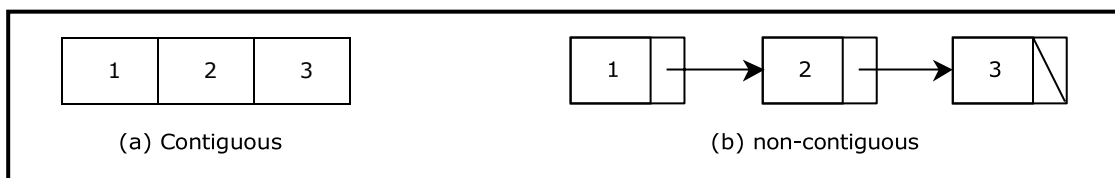


Figure 1.2 Contiguous and Non-contiguous structures compared

Contiguous structures:

Contiguous structures can be broken down further into two kinds: those that contain data items of all the same size, and those where the size may differ. Figure 1.2 shows examples of each kind. The first kind is called the array. Figure 1.3(a) shows an example of an array of numbers. In an array, each element is of the same type, and thus has the same size.

The second kind of contiguous structure is called structure, figure 1.3(b) shows a simple structure consisting of a person's name and age. In a struct, elements may be of different data types and thus may have different sizes.

For example, a person's age can be represented with a simple integer that occupies two bytes of memory. But his or her name, represented as a string of characters, may require many bytes and may even be of varying length.

Couples with the atomic types (that is, the single data-item built-in types such as integer, float and pointers), arrays and structs provide all the "mortar" you need to built more exotic form of data structure, including the non-contiguous forms.

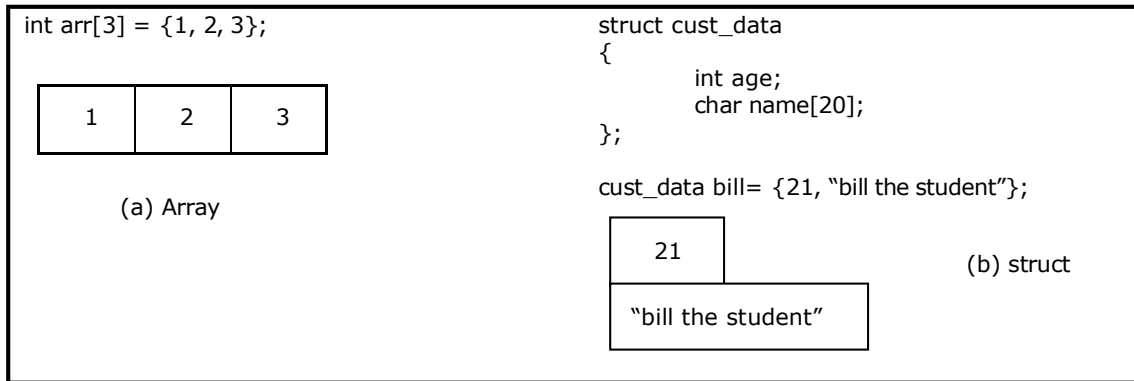


Figure 1.3 Examples of contiguous structures.

Non-contiguous structures:

Non-contiguous structures are implemented as a collection of data-items, called nodes, where each node can point to one or more other nodes in the collection. The simplest kind of non-contiguous structure is linked list.

A linked list represents a linear, one-dimension type of non-contiguous structure, where there is only the notation of backwards and forwards. A tree such as shown in figure 1.4(b) is an example of a two-dimensional non-contiguous structure. Here, there is the notion of up and down and left and right.

In a tree each node has only one link that leads into the node and links can only go down the tree. The most general type of non-contiguous structure, called a graph has no such restrictions. Figure 1.4(c) is an example of a graph.

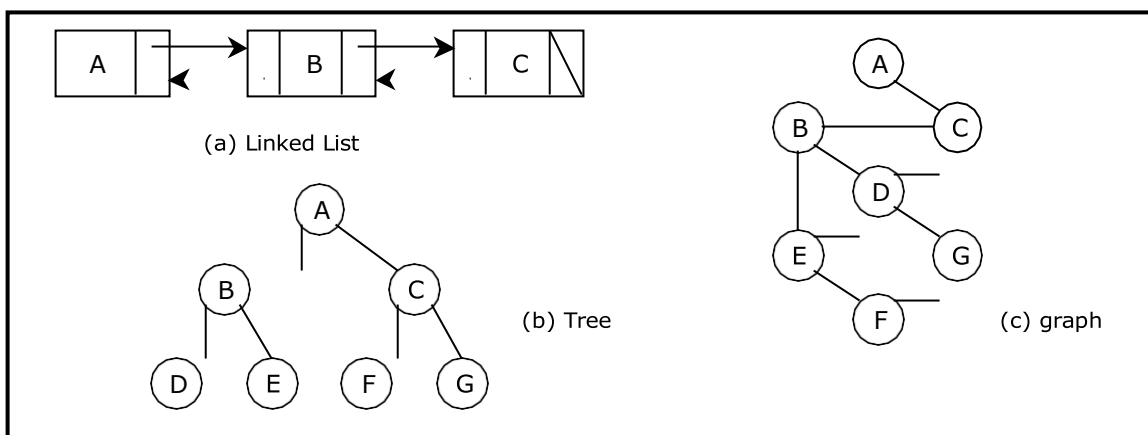


Figure 1.4. Examples of non-contiguous structures

Hybrid structures:

If two basic types of structures are mixed then it is a hybrid form. Then one part contiguous and another part non-contiguous. For example, figure 1.5 shows how to implement a double-linked list using three parallel arrays, possibly stored apart from each other in memory.

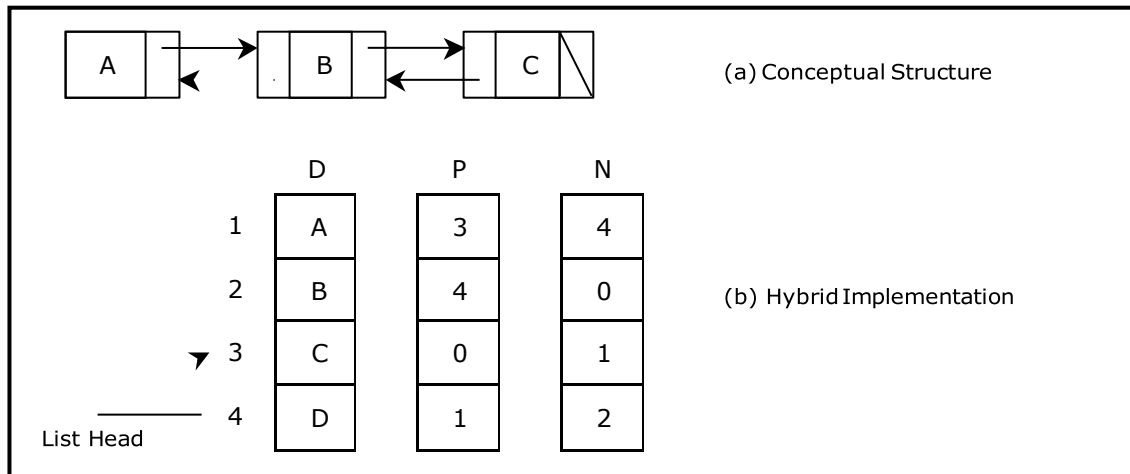


Figure 1.5. A double linked list via a hybrid data structure

The array D contains the data for the list, whereas the array P and N hold the previous and next "pointers". The pointers are actually nothing more than indexes into the D array. For instance, D[i] holds the data for node i and p[i] holds the index to the node previous to i, where may or may not reside at position i-1. Like wise, N[i] holds the index to the next node in the list.

1.3. Abstract Data Type (ADT):

The design of a data structure involves more than just its organization. You also need to plan for the way the data will be accessed and processed – that is, how the data will be interpreted actually, non-contiguous structures – including lists, tree and graphs – can be implemented either contiguously or non- contiguously like wise, the structures that are normally treated as contiguously - arrays and structures – can also be implemented non-contiguously.

The notion of a data structure in the abstract needs to be treated differently from what ever is used to implement the structure. The abstract notion of a data structure is defined in terms of the operations we plan to perform on the data.

Considering both the organization of data and the expected operations on the data, leads to the notion of an abstract data type. An *abstract data type* is a theoretical construct that consists of data as well as the operations to be performed on the data while hiding implementation.

For example, a stack is a typical abstract data type. Items stored in a stack can only be added and removed in certain order – the last item added is the first item removed. We call these operations, pushing and popping. In this definition, we haven't specified how items are stored on the stack, or how the items are pushed and popped. We have only specified the valid operations that can be performed.

For example, if we want to read a file, we wrote the code to read the physical file device. That is, we may have to write the same code over and over again. So we created what is known

today as an ADT. We wrote the code to read a file and placed it in a library for a programmer to use.

As another example, the code to read from a keyboard is an ADT. It has a data structure, character and set of operations that can be used to read that data structure.

To be made useful, an abstract data type (such as stack) has to be implemented and this is where data structure comes into play. For instance, we might choose the simple data structure of an array to represent the stack, and then define the appropriate indexing operations to perform pushing and popping.

1.4. Selecting a data structure to match the operation:

The most important process in designing a problem involves choosing which data structure to use. The choice depends greatly on the type of operations you wish to perform.

Suppose we have an application that uses a sequence of objects, where one of the main operations is delete an object from the middle of the sequence. The code for this is as follows:

```
void delete (int *seg, int &n, int posn)
// delete the item at position from an array of n elements.
{
    if (n)
    {
        int i=posn;
        n--;
        while (i < n)
        {
            seq[i] = seg[i+1];
            i++;
        }
    }
    return;
}
```

This function shifts towards the front all elements that follow the element at position *posn*. This shifting involves data movement that, for integer elements, which is too costly. However, suppose the array stores larger objects, and lots of them. In this case, the overhead for moving data becomes high. The problem is that, in a contiguous structure, such as an array the logical ordering (the ordering that we wish to interpret our elements to have) is the same as the physical ordering (the ordering that the elements actually have in memory).

If we choose non-contiguous representation, however we can separate the logical ordering from the physical ordering and thus change one without affecting the other. For example, if we store our collection of elements using a double-linked list (with previous and next pointers), we can do the deletion without moving the elements, instead, we just modify the pointers in each node. The code using double linked list is as follows:

```
void delete (node * beg, int posn)
//delete the item at posn from a list of elements.
{
    int i = posn;
    node *q = beg;
    while (i && q)
    {
```

```

        i--;
        q = q @ next;
    }

    if (q)
    {
        /* not at end of list, so detach P by making previous and
           next nodes point to each other */
        node *p = q -> prev;
        node *n = q -> next;
        if (p)
            p -> next = n;
        if (n)
            n -> prev = P;
    }
    return;
}

```

The process of detecting a node from a list is independent of the type of data stored in the node, and can be accomplished with some pointer manipulation as illustrated in figure below:

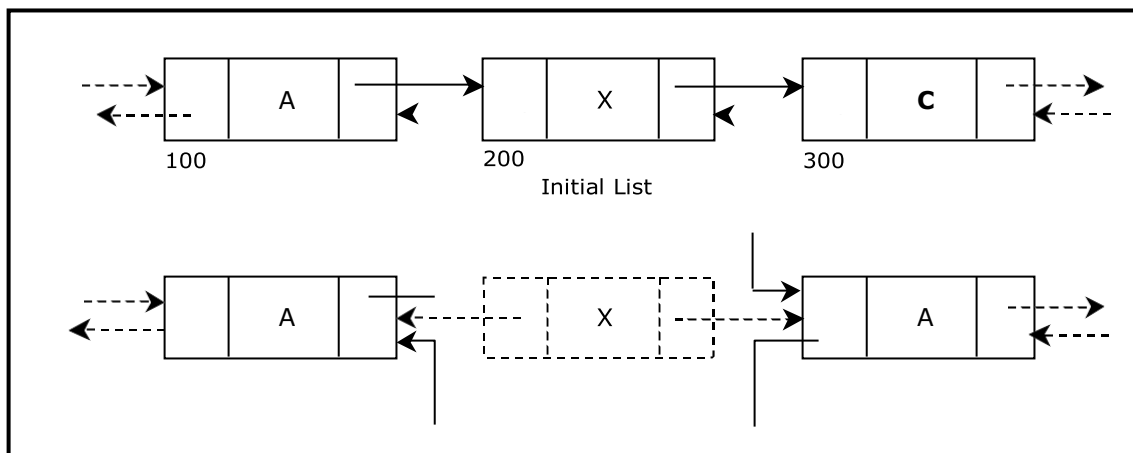


Figure 1.6 Detaching a node from a list

Since very little data is moved during this process, the deletion using linked lists will often be faster than when arrays are used.

It may seem that linked lists are superior to arrays. But is that always true? There are trade offs. Our linked lists yield faster deletions, but they take up more space because they require two extra pointers per element.

1.5. Algorithm

An **algorithm** is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. No matter what the input values may be, an algorithm terminates after executing a finite number of instructions. In addition every algorithm must satisfy the following criteria:

Input: there are zero or more quantities, which are externally supplied;

Output: at least one quantity is produced;

Definiteness: each instruction must be clear and unambiguous;

Finiteness: if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;

Effectiveness: every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite, but it must also be feasible.

In formal computer science, one distinguishes between an algorithm, and a program. A program does not necessarily satisfy the fourth condition. One important example of such a program for a computer is its operating system, which never terminates (except for system crashes) but continues in a wait loop until more jobs are entered.

We represent an algorithm using pseudo language that is a combination of the constructs of a programming language together with informal English statements.

1.6. Practical Algorithm design issues:

Choosing an efficient algorithm or data structure is just one part of the design process. Next, will look at some design issues that are broader in scope. There are three basic design goals that we should strive for in a program:

1. Try to save time (Time complexity).
2. Try to save space (Space complexity).
3. Try to have face.

A program that runs faster is a better program, so saving time is an obvious goal. Like wise, a program that saves space over a competing program is considered desirable. We want to “save face” by preventing the program from locking up or generating reams of garbled data.

1.7. Performance of a program:

The performance of a program is the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program. One is analytical, and the other experimental. In performance analysis we use analytical methods, while in performance measurement we conduct experiments.

Time Complexity:

The time needed by an algorithm expressed as a function of the size of a problem is called the **TIME COMPLEXITY** of the algorithm. The time complexity of a program is the amount of computer time it needs to run to completion.

The limiting behavior of the complexity as size increases is called the asymptotic time complexity. It is the asymptotic complexity of an algorithm, which ultimately determines the size of problems that can be solved by the algorithm.

Space Complexity:

The space complexity of a program is the amount of memory it needs to run to completion. The space need by a program has the following components:

Instruction space: Instruction space is the space needed to store the compiled version of the program instructions.

Data space: Data space is the space needed to store all constant and variable values. Data space has two components:

- Space needed by constants and simple variables in program.
- Space needed by dynamically allocated objects such as arrays and class instances.

Environment stack space: The environment stack is used to save information needed to resume execution of partially completed functions.

Instruction Space: The amount of instructions space that is needed depends on factors such as:

- The compiler used to complete the program into machine code.
- The compiler options in effect at the time of compilation
- The target computer.

1.8. Classification of Algorithms

If 'n' is the number of data items to be processed or degree of polynomial or the size of the file to be sorted or searched or the number of nodes in a graph etc.

- | | |
|-------------------------|--|
| 1 | Next instructions of most programs are executed once or at most only a few times. If all the instructions of a program have this property, we say that its running time is a constant. |
| Log n | When the running time of a program is logarithmic, the program gets slightly slower as n grows. This running time commonly occurs in programs that solve a big problem by transforming it into a smaller problem, cutting the size by some constant fraction., When n is a million, log n is a doubled whenever n doubles, log n increases by a constant, but log n does not double until n increases to n^2 . |
| n | When the running time of a program is linear, it is generally the case that a small amount of processing is done on each input element. This is the optimal situation for an algorithm that must process n inputs. |
| n. log n | This running time arises for algorithms but solve a problem by breaking it up into smaller sub-problems, solving them independently, and then combining the solutions. When n doubles, the running time more than doubles. |
| n^2 | When the running time of an algorithm is quadratic, it is practical for use only on relatively small problems. Quadratic running times typically arise in algorithms that process all pairs of data items (perhaps in a double nested loop) whenever n doubles, the running time increases four fold. |
| n^3 | Similarly, an algorithm that process triples of data items (perhaps in a triple- nested loop) has a cubic running time and is practical for use only on small problems. Whenever n doubles, the running time increases eight fold. |
| 2^n | Few algorithms with exponential running time are likely to be appropriate for practical use, such algorithms arise naturally as "brute-force" solutions to problems. Whenever n doubles, the running time squares. |

1.9. Complexity of Algorithms

The complexity of an algorithm M is the function $f(n)$ which gives the running time and/or storage space requirement of the algorithm in terms of the size 'n' of the input data. Mostly, the storage space required by an algorithm is simply a multiple of the data size 'n'. Complexity shall refer to the running time of the algorithm.

The function $f(n)$, gives the running time of an algorithm, depends not only on the size 'n' of the input data but also on the particular data. The complexity function $f(n)$ for certain cases are:

1. Best Case : The minimum possible value of $f(n)$ is called the best case.
2. Average Case : The expected value of $f(n)$.
3. Worst Case : The maximum value of $f(n)$ for any key possible input.

The field of computer science, which studies efficiency of algorithms, is known as analysis of algorithms.

Algorithms can be evaluated by a variety of criteria. Most often we shall be interested in the rate of growth of the time or space required to solve larger and larger instances of a problem. We will associate with the problem an integer, called the size of the problem, which is a measure of the quantity of input data.

1.10. Rate of Growth

Big-Oh (O), Big-Omega (Ω), Big-Theta (Θ) and Little-Oh

1. $T(n) = O(f(n))$, (pronounced order of or big oh), says that the growth rate of $T(n)$ is less than or equal (\leq) that of $f(n)$
2. $T(n) = \Omega(g(n))$ (pronounced omega), says that the growth rate of $T(n)$ is greater than or equal to (\geq) that of $g(n)$
3. $T(n) = \Theta(h(n))$ (pronounced theta), says that the growth rate of $T(n)$ equals (=) the growth rate of $h(n)$ [if $T(n) = O(h(n))$ and $T(n) = \Omega(h(n))$]
4. $T(n) = o(p(n))$ (pronounced little oh), says that the growth rate of $T(n)$ is less than the growth rate of $p(n)$ [if $T(n) = O(p(n))$ and $T(n) \neq \Theta(p(n))$].

Some Examples:

$$\begin{aligned}2n^2 + 5n - 6 &= O(2^n) \\2n^2 + 5n - 6 &= O(n^3) \\2n^2 + 5n - 6 &= O(n^2) \\2n^2 + 5n - 6 &\neq O(n)\end{aligned}$$

$$\begin{aligned}2n^2 + 5n - 6 &\neq \Omega(2^n) \\2n^2 + 5n - 6 &\neq \Omega(n^3) \\2n^2 + 5n - 6 &= \Omega(n^2) \\2n^2 + 5n - 6 &= \Omega(n)\end{aligned}$$

$$\begin{aligned}2n^2 + 5n - 6 &\neq \Theta(2^n) \\2n^2 + 5n - 6 &\neq \Theta(n^3) \\2n^2 + 5n - 6 &= \Theta(n^2) \\2n^2 + 5n - 6 &\neq \Theta(n)\end{aligned}$$

$$\begin{aligned}2n^2 + 5n - 6 &= o(2^n) \\2n^2 + 5n - 6 &= o(n^3) \\2n^2 + 5n - 6 &\neq o(n^2) \\2n^2 + 5n - 6 &\neq o(n)\end{aligned}$$

1.11. Analyzing Algorithms

Suppose 'M' is an algorithm, and suppose 'n' is the size of the input data. Clearly the complexity $f(n)$ of M increases as n increases. It is usually the rate of increase of $f(n)$ we want to examine. This is usually done by comparing $f(n)$ with some standard functions. The most common computing times are:

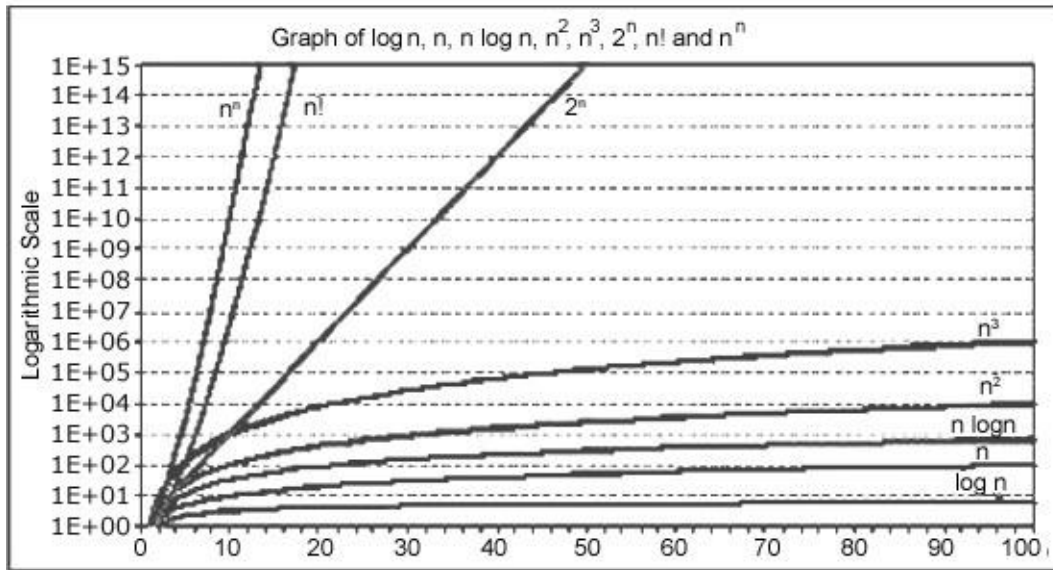
$O(1)$, $O(\log_2 n)$, $O(n)$, $O(n \cdot \log_2 n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$, $n!$ and n^n

Numerical Comparison of Different Algorithms

The execution time for six of the typical functions is given below:

S.No	$\log n$	n	$n \cdot \log n$	n^2	n^3	2^n
1	0	1	1	1	1	2
2	1	2	2	4	8	4
3	2	4	8	16	64	16
4	3	8	24	64	512	256
5	4	16	64	256	4096	65536

Graph of $\log n$, n , $n \log n$, n^2 , n^3 , 2^n , $n!$ and n^n



$O(\log n)$ does not depend on the base of the logarithm. To simplify the analysis, the convention will not have any particular units of time. Thus we throw away leading constants. We will also throw away low-order terms while computing a Big-Oh running time. Since Big-Oh is an upper bound, the answer provided is a guarantee that the program will terminate within a certain time period. The program may stop earlier than this, but never later.

One way to compare the function $f(n)$ with these standard function is to use the functional 'O' notation, suppose $f(n)$ and $g(n)$ are functions defined on the positive integers with the property that $f(n)$ is bounded by some multiple $g(n)$ for almost all 'n'. Then,

$$f(n) = O(g(n))$$

Which is read as "f(n) is of order g(n)". For example, the order of complexity for:

- Linear search is $O(n)$
- Binary search is $O(\log n)$
- Bubble sort is $O(n^2)$
- Quick sort is $O(n \log n)$

For example, if the first program takes $100n^2$ milliseconds. While the second taken $5n^3$ milliseconds. Then might not $5n^3$ program better than $100n^2$ program?

As the programs can be evaluated by comparing their running time functions, with constants by proportionality neglected. So, $5n^3$ program be better than the $100n^2$ program.

$$5n^3/100n^2 = n/20$$

for inputs $n < 20$, the program with running time $5n^3$ will be faster those the one with running time $100n^2$.

Therefore, if the program is to be run mainly on inputs of small size, we would indeed prefer the program whose running time was $O(n^3)$

However, as 'n' gets large, the ratio of the running times, which is $n/20$, gets arbitrarily larger. Thus, as the size of the input increases, the $O(n^3)$ program will take significantly more time than the $O(n^2)$ program. So it is always better to prefer a program whose running time with the lower growth rate. The low growth rate function's such as $O(n)$ or $O(n \log n)$ are always better.

Exercises

1. Define algorithm.
2. State the various steps in developing algorithms?
3. State the properties of algorithms.
4. Define efficiency of an algorithm?
5. State the various methods to estimate the efficiency of an algorithm.
6. Define time complexity of an algorithm?
7. Define worst case of an algorithm.
8. Define average case of an algorithm.
9. Define best case of an algorithm.
10. Mention the various spaces utilized by a program.

11. Define space complexity of an algorithm.
12. State the different memory spaces occupied by an algorithm.

Multiple Choice Questions

- _____ is a step-by-step recipe for solving an instance of problem. [A]
A. Algorithm
B. Complexity
C. Pseudocode
D. Analysis
- _____ is used to describe the algorithm, in less formal language. [C]
A. Cannot be defined
B. Natural Language
C. Pseudocode
D. None
- _____ of an algorithm is the amount of time (or the number of steps) needed by a program to complete its task. [D]
A. Space Complexity
B. Dynamic Programming
C. Divide and Conquer
D. Time Complexity
- _____ of a program is the amount of memory used at once by the algorithm until it completes its execution. [C]
A. Divide and Conquer
B. Time Complexity
C. Space Complexity
D. Dynamic Programming
- _____ is used to define the worst-case running time of an algorithm. [A]
A. Big-Oh notation
B. Cannot be defined
C. Complexity
D. Analysis

C. Mathematical induction.

D. Matrix Multiplication.

Chapter

3

LINKED LISTS

In this chapter, the list data structure is presented. This structure can be used as the basis for the implementation of other data structures (stacks, queues etc.). The basic linked list can be used without modification in many programs. However, some applications require enhancements to the linked list design. These enhancements fall into three broad categories and yield variations on linked lists that can be used in any combination: circular linked lists, double linked lists and lists with header nodes.

Linked lists and arrays are similar since they both store collections of data. Array is the most common data structure used to store collections of elements. Arrays are convenient to declare and provide the easy syntax to access any element by its index number. Once the array is set up, access to any element is convenient and fast. The disadvantages of arrays are:

- The size of the array is fixed. Most often this size is specified at compile time. This makes the programmers to allocate arrays, which seems "large enough" than required.
- Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.
- Deleting an element from an array is not possible.

Linked lists have their own strengths and weaknesses, but they happen to be strong where arrays are weak. Generally array's allocates the memory for all its elements in one block whereas linked lists use an entirely different strategy. Linked lists allocate memory for each element separately and only when necessary.

Here is a quick review of the terminology and rules of pointers. The linked list code will depend on the following functions:

malloc() is a system function which allocates a block of memory in the "heap" and returns a pointer to the new block. The prototype of malloc() and other heap functions are in stdlib.h. malloc() returns NULL if it cannot fulfill the request. It is defined by:

```
void *malloc (number_of_bytes)
```

Since a void * is returned the C standard states that this pointer can be converted to any type. For example,

```
char *cp;  
cp = (char *) malloc (100);
```

Attempts to get 100 bytes and assigns the starting address to cp. We can also use the sizeof() function to specify the number of bytes. For example,

```
int *ip;  
ip = (int *) malloc (100*sizeof(int));
```

free() is the opposite of **malloc()**, which de-allocates memory. The argument to **free()** is a pointer to a block of memory in the heap — a pointer which was obtained by a **malloc()** function. The syntax is:

```
free (ptr);
```

The advantage of **free()** is simply memory management when we no longer need a block.

3.1. Linked List Concepts:

A linked list is a non-sequential collection of data items. It is a dynamic data structure. For every data item in a linked list, there is an associated pointer that would give the memory location of the next data item in the linked list.

The data items in the linked list are not in consecutive memory locations. They may be anywhere, but the accessing of these data items is easier as each data item contains the address of the next data item.

Advantages of linked lists:

Linked lists have many advantages. Some of the very important advantages are:

1. Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of a program.
2. Linked lists have efficient memory utilization. Here, memory is not pre-allocated. Memory is allocated whenever it is required and it is de-allocated (removed) when it is no longer needed.
3. Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of the data item from the given position.
4. Many complex applications can be easily carried out with linked lists.

Disadvantages of linked lists:

1. It consumes more space because every node requires a additional pointer to store address of the next node.
2. Searching a particular element in list is difficult and also time consuming.

3.2. Types of Linked Lists:

Basically we can put linked lists into the following four items:

1. Single Linked List.
2. Double Linked List.
3. Circular Linked List.
4. Circular Double Linked List.

A single linked list is one in which all nodes are linked together in some sequential manner. Hence, it is also called as linear linked list.

A double linked list is one in which all nodes are linked together by multiple links which helps in accessing both the successor node (next node) and predecessor node (previous node) from any arbitrary node within the list. Therefore each node in a double linked list has two link fields (pointers) to point to the left node (previous) and the right node (next). This helps to traverse in forward direction and backward direction.

A circular linked list is one, which has no beginning and no end. A single linked list can be made a circular linked list by simply storing address of the very first node in the link field of the last node.

A circular double linked list is one, which has both the successor pointer and predecessor pointer in the circular manner.

Comparison between array and linked list:

ARRAY	LINKED LIST
Size of an array is fixed	Size of a list is not fixed
Memory is allocated from stack	Memory is allocated from heap
It is necessary to specify the number of elements during declaration (i.e., during compile time).	It is not necessary to specify the number of elements during declaration (i.e., memory is allocated during run time).
It occupies less memory than a linked list for the same number of elements.	It occupies more memory.
Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.	Inserting a new element at any position can be carried out easily.
Deleting an element from an array is not possible.	Deleting an element is possible.

Trade offs between linked lists and arrays:

FEATURE	ARRAYS	LINKED LISTS
Sequential access	efficient	efficient
Random access	efficient	inefficient
Resigning	inefficient	efficient
Element rearranging	inefficient	efficient
Overhead per elements	none	1 or 2 links

Applications of linked list:

1. Linked lists are used to represent and manipulate polynomial. Polynomials are expression containing terms with non zero coefficient and exponents. For example:

$$P(x) = a_0 X^n + a_1 X^{n-1} + \dots + a_{n-1} X + a_n$$

2. Represent very large numbers and operations of the large number such as addition, multiplication and division.
3. Linked lists are to implement stack, queue, trees and graphs.
4. Implement the symbol table in compiler construction

3.3. Single Linked List:

A linked list allocates space for each element separately in its own block of memory called a "node". The list gets an overall structure by using pointers to connect all its nodes together like the links in a chain. Each node contains two fields; a "data" field to store whatever element, and a "next" field which is a pointer used to link to the next node. Each node is allocated in the heap using malloc(), so the node memory continues to exist until it is explicitly de-allocated using free(). The front of the list is a pointer to the "start" node.

A single linked list is shown in figure 3.2.1.

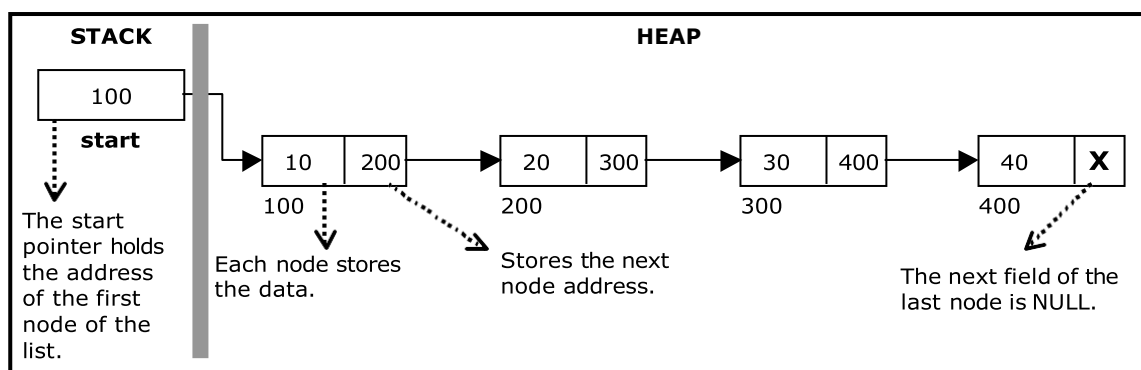


Figure 3.2.1. Single Linked List

The beginning of the linked list is stored in a "**start**" pointer which points to the first node. The first node contains a pointer to the second node. The second node contains a pointer to the third node, ... and so on. The last node in the list has its next field set to NULL to mark the end of the list. Code can access any node in the list by starting at the **start** and following the next pointers.

The **start** pointer is an ordinary local pointer variable, so it is drawn separately on the left top to show that it is in the stack. The list nodes are drawn on the right to show that they are allocated in the heap.

Implementation of Single Linked List:

Before writing the code to build the above list, we need to create a **start** node, used to create and access other nodes in the linked list. The following structure definition will do (see figure 3.2.2):

- Creating a structure with one data item and a next pointer, which will be pointing to next node of the list. This is called as self-referential structure.
- Initialise the start pointer to be NULL.

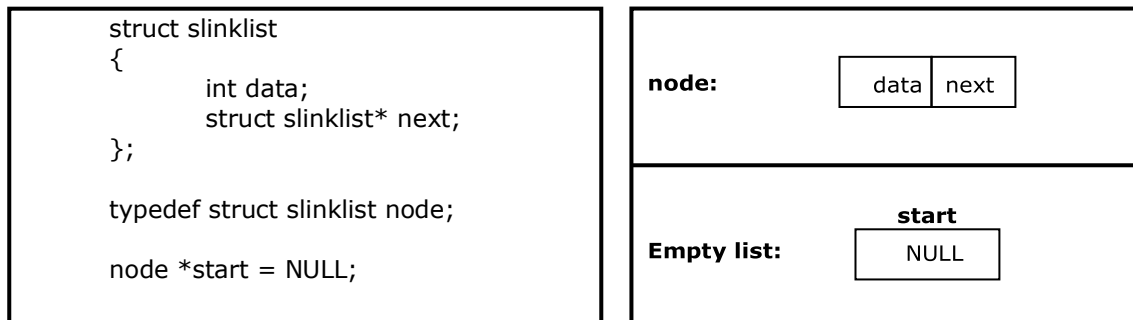


Figure 3.2.2. Structure definition, single link node and empty list

The basic operations in a single linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing.

Creating a node for Single Linked List:

Creating a singly linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the malloc() function. The function getnode(), is used for creating a node, after allocating memory for the structure of type node, the information for the item (i.e., data) has to be read from the user, set next field to NULL and finally returns the address of the node. Figure 3.2.3 illustrates the creation of a node for single linked list.

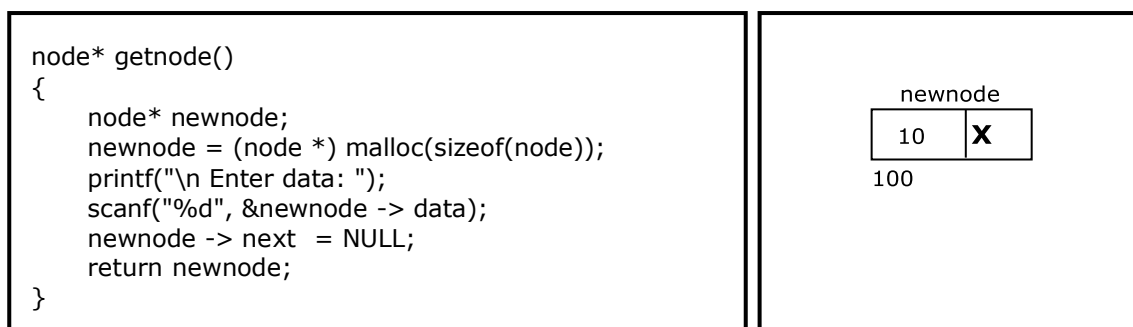


Figure 3.2.3. new node with a value of 10

Creating a Singly Linked List with 'n' number of nodes:

The following steps are to be followed to create 'n' number of nodes:

- Get the new node using `getnode()`.
`newnode = getnode();`
- If the list is empty, assign new node as start.
`start = newnode;`
- If the list is not empty, follow the steps given below:
 - The next field of the new node is made to point the first node (i.e. start node) in the list by assigning the address of the first node.
 - The start pointer is made to point the new node by assigning the address of the new node.
- Repeat the above steps 'n' times.

Figure 3.2.4 shows 4 items in a single linked list stored at different locations in memory.

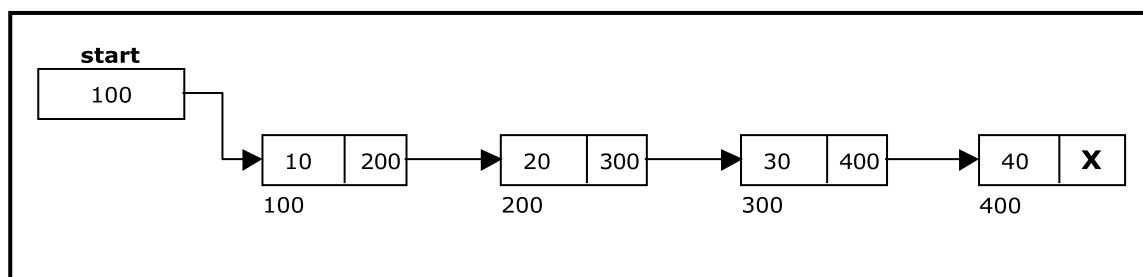


Figure 3.2.4. Single Linked List with 4 nodes

The function `createlist()`, is used to create 'n' number of nodes:

```
void createlist( int n)
{
    int i;
    node * newnode;
    node * temp;
    for( i = 0 ; i < n ; i++ )
    {
        newnode = getnode();
        if(start == NULL)
        {
            start = newnode;
        }
        else
        {
            temp = start;
            while(temp -> next != NULL)
                temp = temp -> next;
            temp -> next = newnode;
        }
    }
}
```

Insertion of a Node:

One of the most primitive operations that can be done in a singly linked list is the insertion of a node. Memory is to be allocated for the new node (in a similar way that is done while creating a list) before reading the data. The new node will contain empty data field and empty next field. The data field of the new node is then stored with the information read from the user. The next field of the new node is assigned to NULL. The new node can then be inserted at three different places namely:

- Inserting a node at the beginning.
- Inserting a node at the end.
- Inserting a node at intermediate position.

Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the list:

- Get the new node using `getnode()`.
`newnode = getnode();`
- If the list is empty then `start = newnode`.
- If the list is not empty, follow the steps given below:
`newnode -> next = start;`
`start = newnode;`

Figure 3.2.5 shows inserting a node into the single linked list at the beginning.

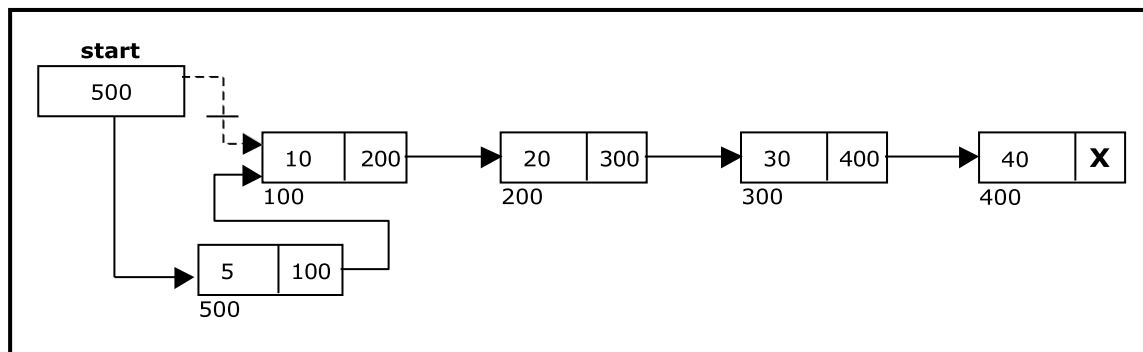


Figure 3.2.5. Inserting a node at the beginning

The function `insert_at_beg()`, is used for inserting a node at the beginning

```
void insert_at_beg()
{
    node *newnode;
    newnode = getnode();
    if(start == NULL)
    {
        start = newnode;
    }
    else
    {
        newnode -> next = start;
        start = newnode;
    }
}
```

Inserting a node at the end:

The following steps are followed to insert a new node at the end of the list:

- Get the new node using `getnode()`
`newnode = getnode();`
- If the list is empty then `start = newnode`.
- If the list is not empty follow the steps given below:
`temp = start;`
`while(temp -> next != NULL)`
`temp = temp -> next;`
`temp -> next = newnode;`

Figure 3.2.6 shows inserting a node into the single linked list at the end.

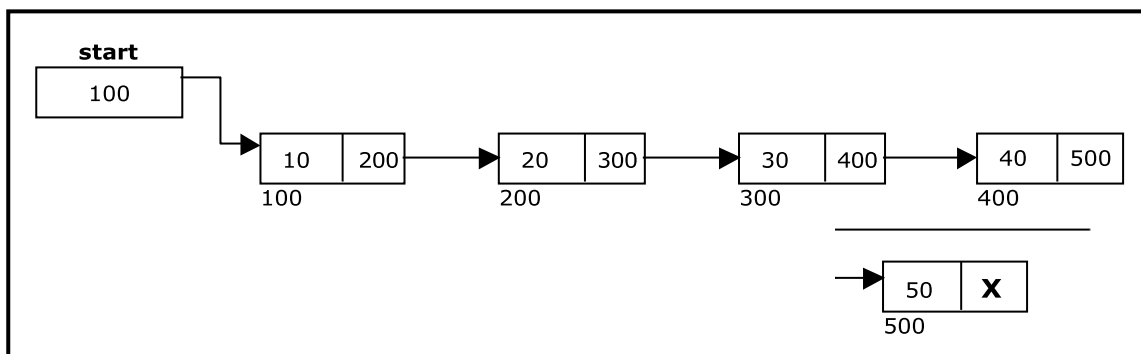


Figure 3.2.6. Inserting a node at the end.

The function `insert_at_end()`, is used for inserting a node at the end.

```
void insert_at_end()
{
    node *newnode, *temp;
    newnode = getnode();
    if(start == NULL)
    {
        start = newnode;
    }
    else
    {
        temp = start;
        while(temp -> next != NULL)
            temp = temp -> next;
        temp -> next = newnode;
    }
}
```

Inserting a node at intermediate position:

The following steps are followed, to insert a new node in an intermediate position in the list:

- Get the new node using `getnode()`.
`newnode = getnode();`

- Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.
- Store the starting address (which is in start pointer) in temp and prev pointers. Then traverse the temp pointer upto the specified position followed by prev pointer.
- After reaching the specified position, follow the steps given below:
 prev -> next = newnode;
 newnode -> next = temp;
- Let the intermediate position be 3.

Figure 3.2.7 shows inserting a node into the single linked list at a specified intermediate position other than beginning and end.

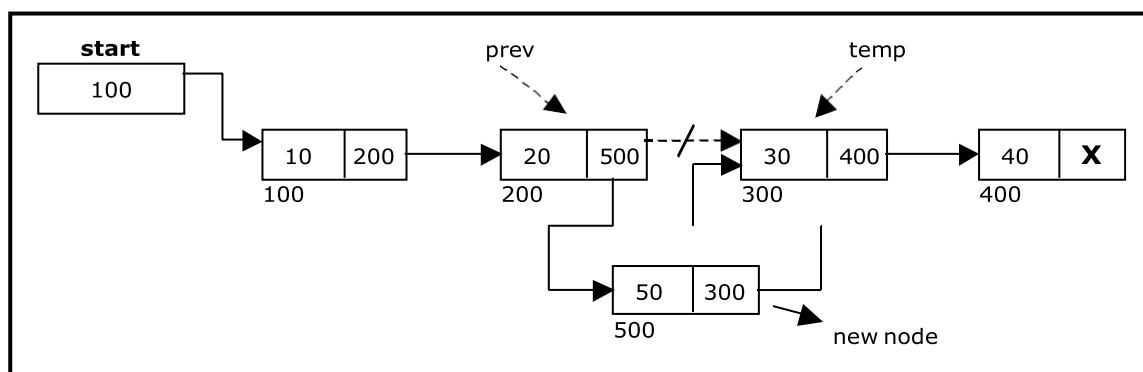


Figure 3.2.7. Inserting a node at an intermediate position.

The function insert_at_mid(), is used for inserting a node in the intermediate position.

```
void insert_at_mid()
{
    node *newnode, *temp, *prev;
    int pos, nodectr, ctr = 1;
    newnode = getnode();
    printf("\n Enter the position: ");
    scanf("%d", &pos);
    nodectr = countnode(start);
    if(pos > 1 && pos < nodectr)
    {
        temp = prev = start;
        while(ctr < pos)
        {
            prev = temp;
            temp = temp -> next;
            ctr++;
        }
        prev -> next = newnode;
        newnode -> next = temp;
    }
    else
    {
        printf("position %d is not a middle position", pos);
    }
}
```

Deletion of a node:

Another primitive operation that can be done in a singly linked list is the deletion of a node. Memory is to be released for the node to be deleted. A node can be deleted from the list from three different places namely.

- Deleting a node at the beginning.
- Deleting a node at the end.
- Deleting a node at intermediate position.

Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:
temp = start;
start = start -> next;
free(temp);

Figure 3.2.8 shows deleting a node at the beginning of a single linked list.

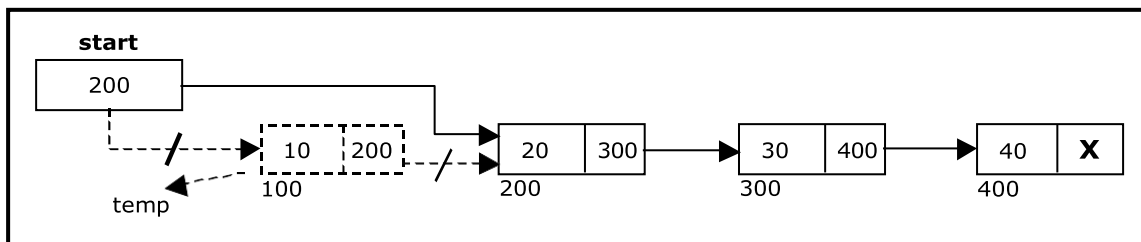


Figure 3.2.8. Deleting a node at the beginning.

The function delete_at_beg(), is used for deleting the first node in the list.

```
void delete_at_beg()
{
    node *temp;
    if(start == NULL)
    {
        printf("\n No nodes are exist..");
        return ;
    }
    else
    {
        temp = start;
        start = temp -> next;
        free(temp);
        printf("\n Node deleted ");
    }
}
```

Deleting a node at the end:

The following steps are followed to delete a node at the end of the list:

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:

```
temp = prev = start;
while(temp -> next != NULL)
{
    prev = temp;
    temp = temp -> next;
}
prev -> next = NULL;
free(temp);
```

Figure 3.2.9 shows deleting a node at the end of a single linked list.

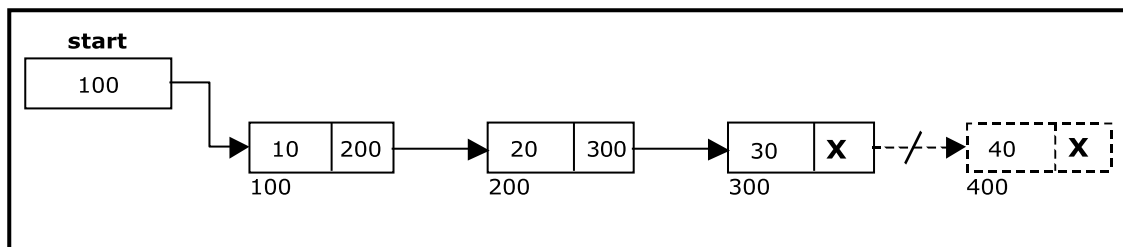


Figure 3.2.9. Deleting a node at the end.

The function `delete_at_last()`, is used for deleting the last node in the list.

```
void delete_at_last()
{
    node *temp, *prev;
    if(start == NULL)
    {
        printf("\n Empty List..");
        return ;
    }
    else
    {
        temp = start;
        prev = start;
        while(temp -> next != NULL)
        {
            prev = temp;
            temp = temp -> next;
        }
        prev -> next = NULL;
        free(temp);
        printf("\n Node deleted ");
    }
}
```


Deleting a node at Intermediate position:

The following steps are followed, to delete a node from an intermediate position in the list (List must contain more than two node).

- If list is empty then display 'Empty List' message
- If the list is not empty, follow the steps given below.

```
if(pos > 1 && pos < nodectr)
{
    temp = prev = start;
    ctr = 1;
    while(ctr < pos)
    {
        prev = temp;
        temp = temp -> next;
        ctr++;
    }
    prev -> next = temp -> next;
    free(temp);
    printf("\n node deleted..");
}
```

Figure 3.2.10 shows deleting a node at a specified intermediate position other than beginning and end from a single linked list.

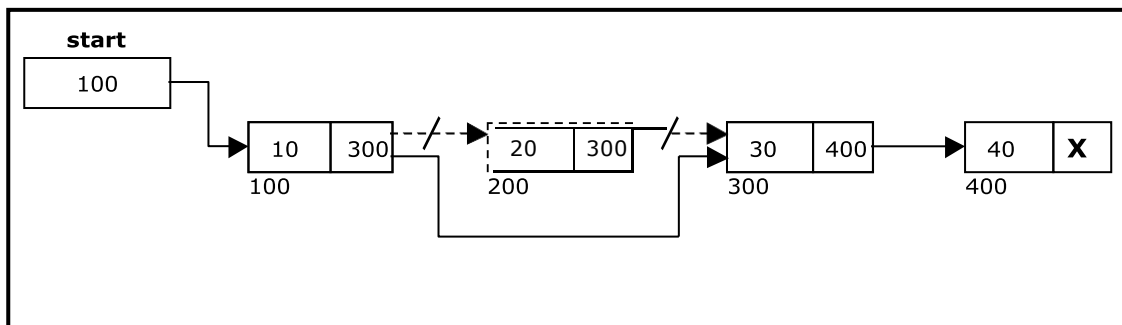


Figure 3.2.10. Deleting a node at an intermediate position.

The function `delete_at_mid()`, is used for deleting the intermediate node in the list.

```
void delete_at_mid()
{
    int ctr = 1, pos, nodectr;
    node *temp, *prev;
    if(start == NULL)
    {
        printf("\n Empty List..");
        return ;
    }
    else
    {
        printf("\n Enter position of node to delete: ");
        scanf("%d", &pos);
        nodectr = countnode(start);
        if(pos > nodectr)
        {
            printf("\n This node doesnot exist");
        }
    }
}
```

```

        if(pos > 1 && pos < nodectr)
        {
            temp = prev = start;
            while(ctr < pos)
            {
                prev = temp;
                temp = temp -> next;
                ctr ++;
            }
            prev -> next = temp -> next;
            free(temp);
            printf("\n Node deleted..");
        }
        else
        {
            printf("\n Invalid position..");
            getch();
        }
    }
}

```

Traversal and displaying a list (Left to Right):

To display the information, you have to traverse (move) a linked list, node by node from the first node, until the end of the list is reached. Traversing a list involves the following steps:

- Assign the address of start pointer to a temp pointer.
- Display the information from the data field of each node.

The function *traverse()* is used for traversing and displaying the information stored in the list from left to right.

```

void traverse()
{
    node *temp;
    temp = start;
    printf("\n The contents of List (Left to Right): \n");
    if(start == NULL )
        printf("\n Empty List");
    else
    {
        while (temp != NULL)
        {
            printf("%d ->", temp -> data);
            temp = temp -> next;
        }
        printf("X");
    }
}

```

Alternatively there is another way to traverse and display the information. That is in reverse order. The function *rev_traverse()*, is used for traversing and displaying the information stored in the list from right to left.

```

void reverse( node * st)
{
    if(st == NULL)
    {
        return;
    }
    else
    {
        reverse(st -> next);
        printf(" %d -> ", st -> data);
    }
}

```

Counting the Number of Nodes:

The following code will count the number of nodes exist in the list using *recursion*.

```

int countnode( node * st)
{
    if(st == NULL)
        return 0 ;
    else
        return(1 + countnode(st -> next));
}

```

3.3.1. Source Code for the Implementation of Single Linked List:

```

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

struct slinklist
{
    int data;
    struct slinklist *next;
};

typedef struct slinklist node;

node *start = NULL;
int menu()
{
    int ch;
    clrscr();
    printf("\n 1.Create a list ");
    printf("\n ----- ");
    printf("\n 2.Insert a node at beginning ");
    printf("\n 3.Insert a node at end");
    printf("\n 4.Insert a node at middle");
    printf("\n ----- ");
    printf("\n 5.Delete a node from beginning");
    printf("\n 6.Delete a node from Last");
    printf("\n 7.Delete a node from Middle");
    printf("\n ----- ");
    printf("\n 8.Traverse the list (Left to Right)");
    printf("\n 9.Traverse the list (Right to Left)");
}

```

```

        printf("\n----- ");
        printf("\n 10. Count nodes ");
        printf("\n 11. Exit ");
        printf("\n\n Enter your choice: ");
        scanf("%d",&ch);
        return ch;
    }

node* getnode()
{
    node * newnode;
    newnode = (node *) malloc(sizeof(node));
    printf("\n Enter data: ");
    scanf("%d", &newnode -> data);
    newnode -> next = NULL;
    return newnode;
}

int countnode(node *ptr)
{
    int count=0;
    while(ptr != NULL)
    {
        count++;
        ptr = ptr -> next;
    }
    return (count);
}

void createlist(int n)
{
    int i;
    node *newnode;
    node *temp;
    for(i = 0; i < n; i++)
    {
        newnode = getnode();
        if(start == NULL)
        {
            start = newnode;
        }
        else
        {
            temp = start;
            while(temp -> next != NULL)
                temp = temp -> next;
            temp -> next = newnode;
        }
    }
}

void traverse()
{
    node *temp;
    temp = start;
    printf("\n The contents of List (Left to Right): \n");
    if(start == NULL)
    {
        printf("\n Empty List");
        return;
    }
    else
    {

```

```

        while(temp != NULL)
        {
            printf("%d-->", temp -> data);
            temp = temp -> next;
        }
    }
    printf(" X ");
}

```

```

void rev_traverse(node *start)
{
    if(start == NULL)
    {
        return;
    }
    else
    {
        rev_traverse(start -> next);
        printf("%d -->", start -> data);
    }
}

```

```

void insert_at_beg()
{
    node *newnode;
    newnode = getnode();
    if(start == NULL)
    {
        start = newnode;
    }
    else
    {
        newnode -> next = start;
        start = newnode;
    }
}

```

```

void insert_at_end()
{
    node *newnode, *temp;
    newnode = getnode();
    if(start == NULL)
    {
        start = newnode;
    }
    else
    {
        temp = start;
        while(temp -> next != NULL)
            temp = temp -> next;
        temp -> next = newnode;
    }
}

```

```

void insert_at_mid()
{
    node *newnode, *temp, *prev;
    int pos, nodectr, ctr = 1;
    newnode = getnode();
    printf("\n Enter the position: ");
    scanf("%d", &pos);
}

```

```
nodectr = countnode(start);
```

```

        if(pos > 1 && pos < nodectr)
        {
            temp = prev = start;
            while(ctr < pos)
            {
                prev = temp;
                temp = temp -> next;
                ctr++;
            }
            prev -> next = newnode;
            newnode -> next = temp;
        }
        else
            printf("position %d is not a middle position", pos);
    }
}

```

```

void delete_at_beg()
{
    node *temp;
    if(start == NULL)
    {
        printf("\n No nodes are exist..");
        return ;
    }
    else
    {
        temp = start;
        start = temp -> next;
        free(temp);
        printf("\n Node deleted ");
    }
}

```

```

void delete_at_last()
{
    node *temp, *prev;
    if(start == NULL)
    {
        printf("\n Empty List..");
        return ;
    }
    else
    {
        temp = start;
        prev = start;
        while(temp -> next != NULL)
        {
            prev = temp;
            temp = temp -> next;
        }
        prev -> next = NULL;
        free(temp);
        printf("\n Node deleted ");
    }
}

```

```

void delete_at_mid()
{
    int ctr = 1, pos, nodectr;
    node *temp, *prev;
    if(start == NULL)
    {

```

```
printf("\n Empty List..");
```



```

        return ;
    }
    else
    {
        printf("\n Enter position of node to delete: ");
        scanf("%d", &pos);
        nodectr = countnode(start);
        if(pos > nodectr)
        {
            printf("\nThis node doesnot exist");
        }
        if(pos > 1 && pos < nodectr)
        {
            temp = prev = start;
            while(ctr < pos)
            {
                prev = temp;
                temp = temp -> next;
                ctr ++;
            }
            prev -> next = temp -> next;
            free(temp);
            printf("\n Node deleted..");
        }
        else
        {
            printf("\n Invalid position..");
            getch();
        }
    }
}

void main(void)
{
    int ch, n;
    clrscr();
    while(1)
    {
        ch = menu();
        switch(ch)
        {
            case 1:
                if(start == NULL)
                {
                    printf("\n Number of nodes you want to create: ");
                    scanf("%d", &n);
                    createlist(n);
                    printf("\n List created..");
                }
                else
                {
                    printf("\n List is already created..");
                    break;
                }
            case 2:
                insert_at_beg();
                break;
            case 3:
                insert_at_end();
                break;
            case 4:
                insert_at_mid();
                break;
        }
    }
}

```

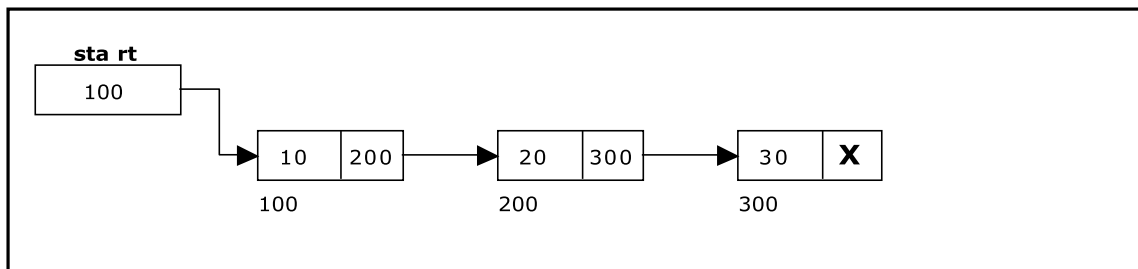
```

case 5:
    delete_at_beg();
    break;
case 6:
    delete_at_last();
    break;
case 7:
    delete_at_mid();
    break;
case 8:
    traverse();
    break;
case 9:
    printf("\n The contents of List (Right to Left): \n");
    rev_traverse(start);
    printf(" X ");
    break;
case 10:
    printf("\n No of nodes : %d ", countnode(start));
    break;
case 11 :
    exit(0);
}
getch();
}

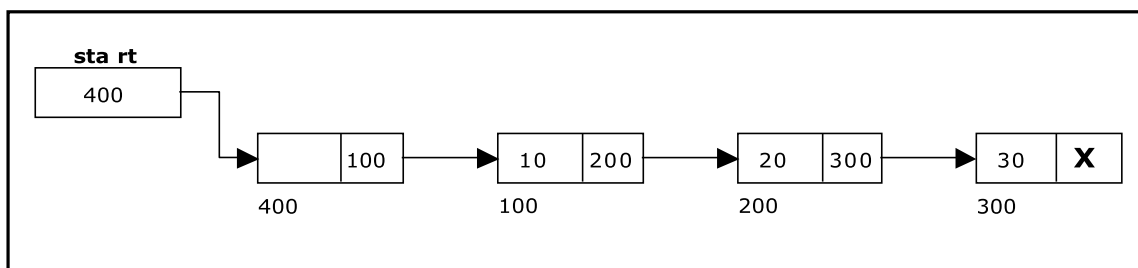
```

3.4. Using a header node:

A header node is a special dummy node found at the front of the list. The use of header node is an alternative to remove the first node in a list. For example, the picture below shows how the list with data 10, 20 and 30 would be represented using a linked list without and with a header node:



Sing le Linke d List w ith o ut a he a d e r n o d e



Sing le Linke d List w ith he a d e r n o d e

Note that if your linked lists do include a header node, there is no need for the special case code given above for the *remove* operation; node ***n*** can never be the first node in the list, so there is no need to check for that case. Similarly, having a header node can simplify the code that adds a node before a given node ***n***.

Note that if you do decide to use a header node, you must remember to initialize an empty list to contain one (dummy) node, you must remember not to include the header node in the count of "real" nodes in the list.

It is also useful when information other than that found in each node of the list is needed. For example, imagine an application in which the number of items in a list is often calculated. In a standard linked list, the list function to count the number of nodes has to traverse the entire list every time. However, if the current length is maintained in a header node, that information can be obtained very quickly.

3.5. Array based linked lists:

Another alternative is to allocate the nodes in blocks. In fact, if you know the maximum size of a list a head of time, you can pre-allocate the nodes in a single array. The result is a hybrid structure – an array based linked list. Figure 3.5.1 shows an example of null terminated single linked list where all the nodes are allocated contiguously in an array.

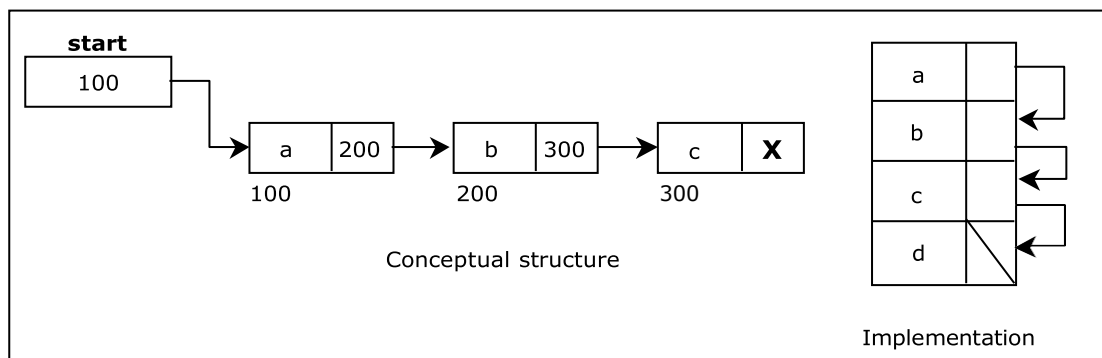


Figure 3.5.1. An array based linked list

3.6. Double Linked List:

A double linked list is a two-way list in which all nodes will have two links. This helps in accessing both successor node and predecessor node from the given node position. It provides bi-directional traversing. Each node contains three fields:

- Left link.
- Data.
- Right link.

The left link points to the predecessor node and the right link points to the successor node. The data field stores the required data.

Many applications require searching forward and backward thru nodes of a list. For example searching for a name in a telephone directory would need forward and backward scanning thru a region of the whole list.

The basic operations in a double linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing.

A double linked list is shown in figure 3.3.1.

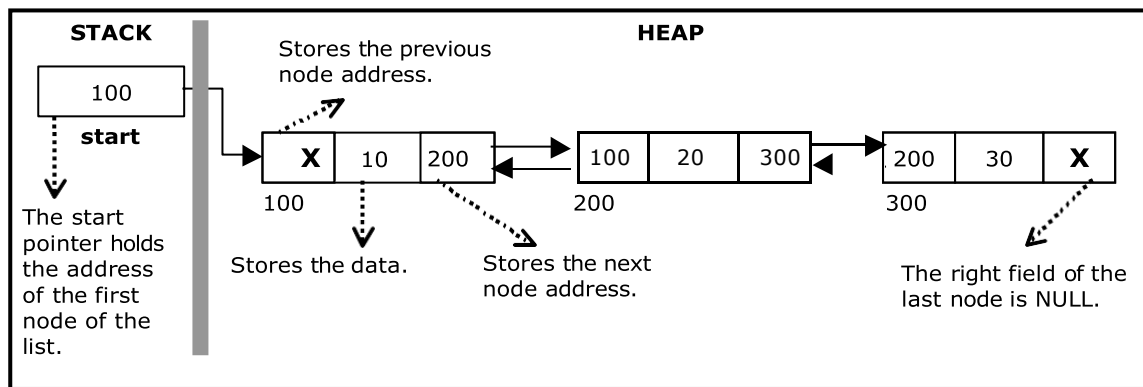


Figure 3.3.1. Double Linked List

The beginning of the double linked list is stored in a "**start**" pointer which points to the first node. The first node's left link and last node's right link is set to NULL.

The following code gives the structure definition:

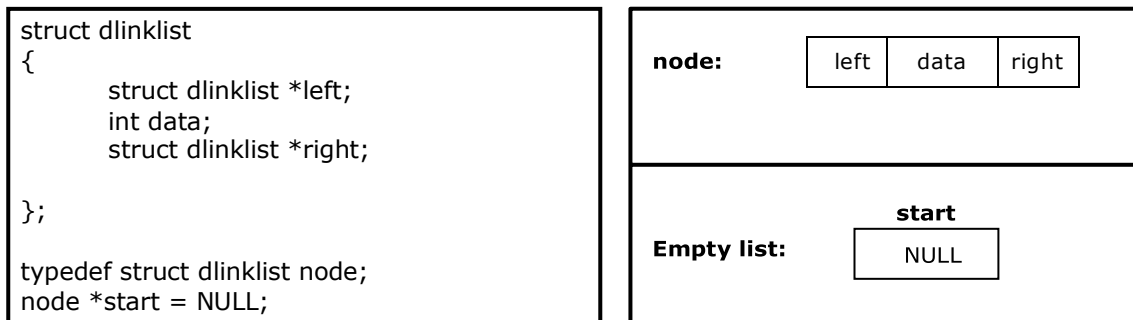


Figure 3.4.1. Structure definition, double link node and empty list

Creating a node for Double Linked List:

Creating a double linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the malloc() function. The function getnode(), is used for creating a node, after allocating memory for the structure of type node, the information for the item (i.e., data) has to be read from the user and set left field to NULL and right field also set to NULL (see figure 3.2.2).

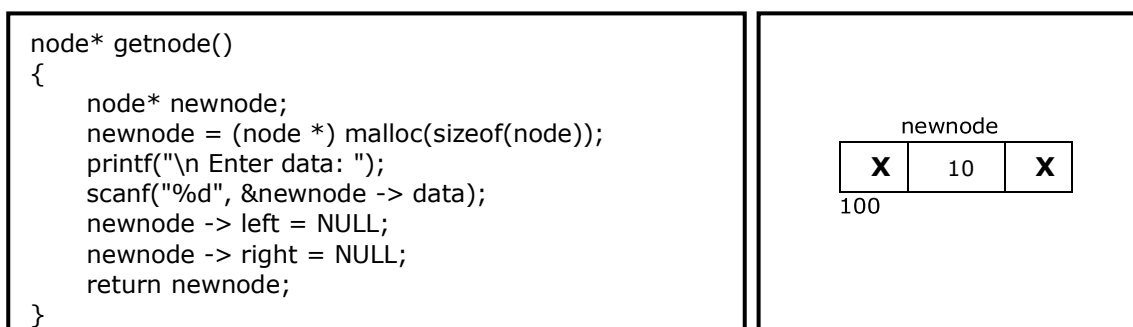


Figure 3.4.2. new node with a value of 10

Creating a Double Linked List with 'n' number of nodes:

The following steps are to be followed to create 'n' number of nodes:

- Get the new node using `getnode()`.
`newnode = getnode();`
- If the list is empty then `start = newnode`.
- If the list is not empty, follow the steps given below:
 - The left field of the new node is made to point the previous node.
 - The previous nodes right field must be assigned with address of the new node.
- Repeat the above steps 'n' times.

The function `createlist()`, is used to create 'n' number of nodes:

```
void create list( int n)
{
    int i;
    node * new node;
    node * temp;
    for( i = 0 ; i < n; i++)
    {
        new node = get node();
        if(st a rt == NU LL)
        {
            sta rt = ne w no de;
        }
        e ls e
        {
            te m p = st a rt;
            w hile(t e m p - > right)
                te m p = t e m p - > right;
            te m p - > right = ne w no de;
            ne w no de - > left = t e m p;
        }
    }
}
```

Figure 3.4.3 shows 3 items in a double linked list stored at different locations.

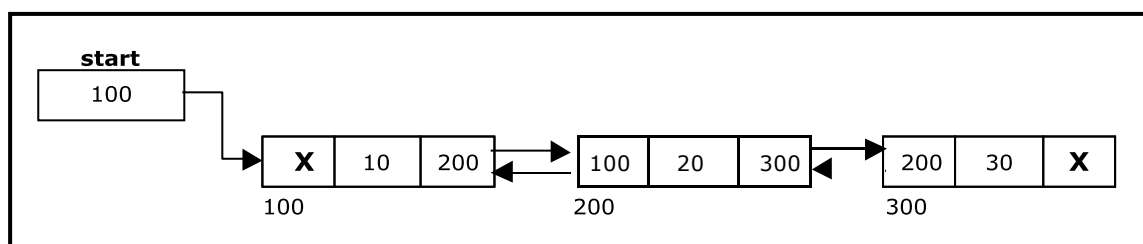


Figure 3.4.3. Double Linked List with 3 nodes

Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the list:

- Get the new node using `getnode()`.
- If the list is empty then $start = newnode$.
- If the list is not empty, follow the steps given below:

```
newnode -> right = start;  
start -> left = newnode;  
start = newnode;
```

The function `dbl_insert_beg()`, is used for inserting a node at the beginning. Figure 3.4.4 shows inserting a node into the double linked list at the beginning.

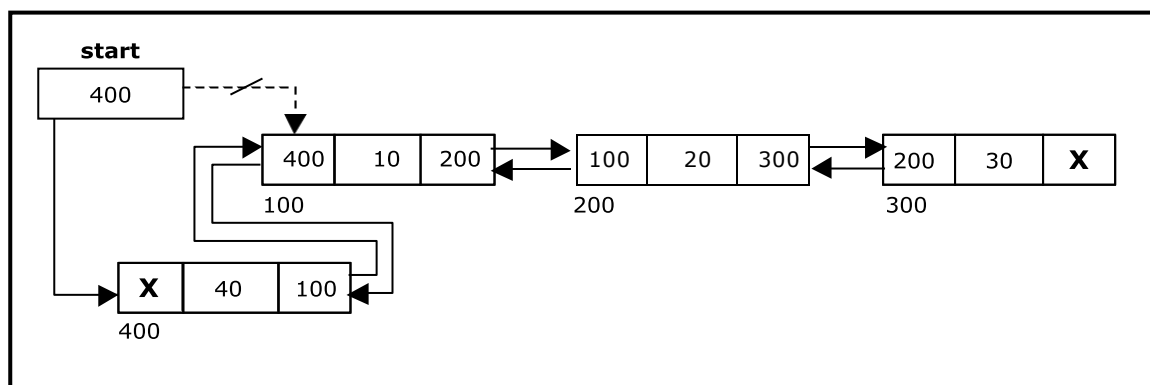


Figure 3.4.4. Inserting a node at the beginning

Inserting a node at the end:

The following steps are followed to insert a new node at the end of the list:

- Get the new node using `getnode()`
- If the list is empty then $start = newnode$.
- If the list is not empty follow the steps given below:

```
temp = start;  
while(temp -> right != NULL)  
    temp = temp -> right;  
temp -> right = newnode;  
newnode -> left = temp;
```

The function `dbl_insert_end()`, is used for inserting a node at the end. Figure 3.4.5 shows inserting a node into the double linked list at the end.

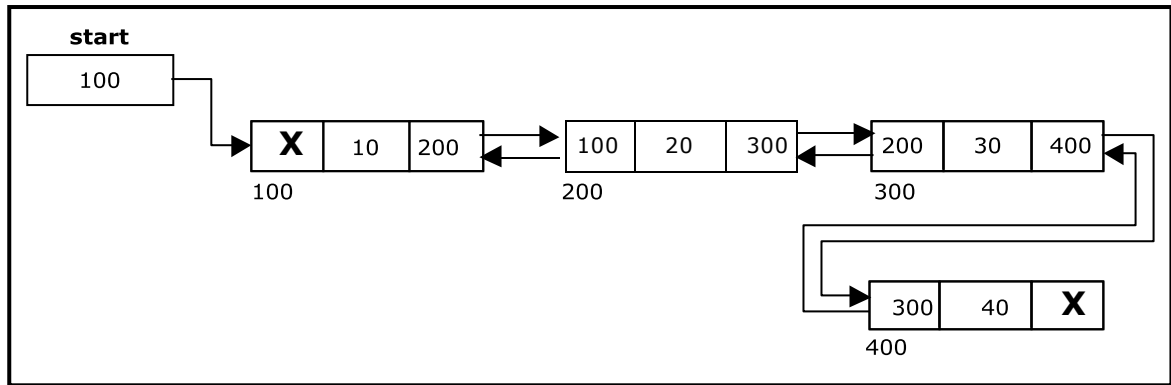


Figure 3.4.5. Inserting a node at the end

Inserting a node at an intermediate position:

The following steps are followed, to insert a new node in an intermediate position in the list:

- Get the new node using `getnode()`.
`newnode=getnode();`
- Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by `countnode()` function.
- Store the starting address (which is in start pointer) in temp and prev pointers. Then traverse the temp pointer upto the specified position followed by prev pointer.
- After reaching the specified position, follow the steps given below:

```
newnode -> left = temp;
newnode -> right = temp -> right;
temp -> right -> left = newnode;
temp -> right = newnode;
```

The function `dbl_insert_mid()`, is used for inserting a node in the intermediate position. Figure 3.4.6 shows inserting a node into the double linked list at a specified intermediate position other than beginning and end.

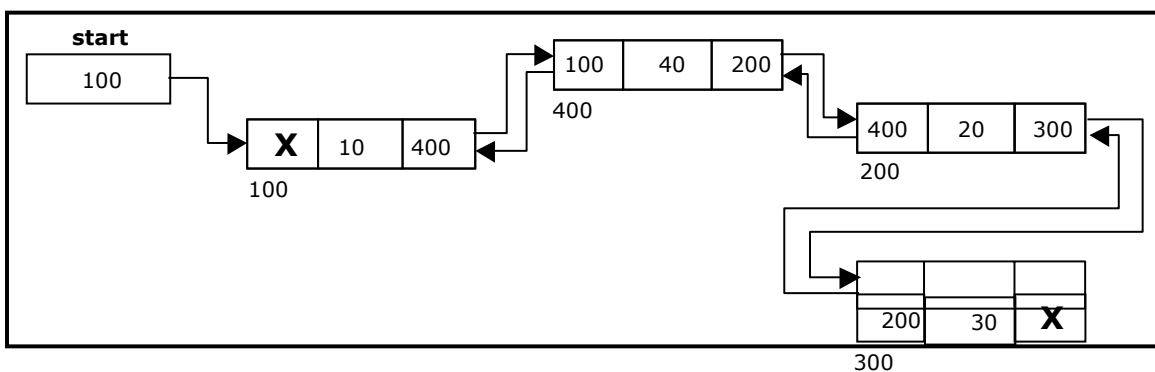


Figure 3.4.6. Inserting a node at an intermediate position

Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:

```
temp = start;  
start = start -> right;  
start -> left = NULL;  
free(temp);
```

The function `dbl_delete_beg()`, is used for deleting the first node in the list. Figure 3.4.6 shows deleting a node at the beginning of a double linked list.

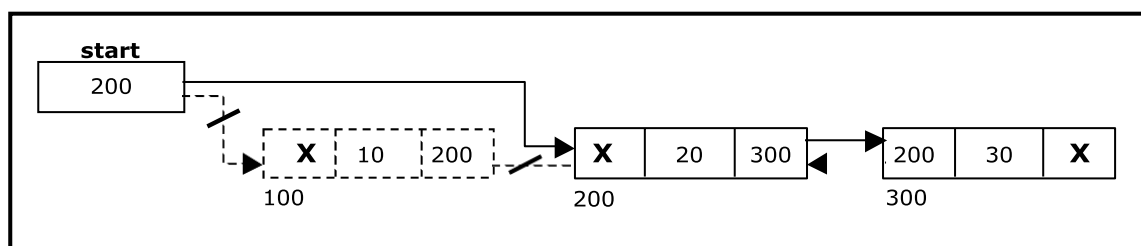


Figure 3.4.6. Deleting a node at beginning

Deleting a node at the end:

The following steps are followed to delete a node at the end of the list:

- If list is empty then display 'Empty List' message
- If the list is not empty, follow the steps given below:

```
temp = start;  
while(temp -> right != NULL)  
{  
    temp = temp -> right;  
}  
temp -> left -> right = NULL;  
free(temp);
```

The function `dbl_delete_last()`, is used for deleting the last node in the list. Figure 3.4.7 shows deleting a node at the end of a double linked list.

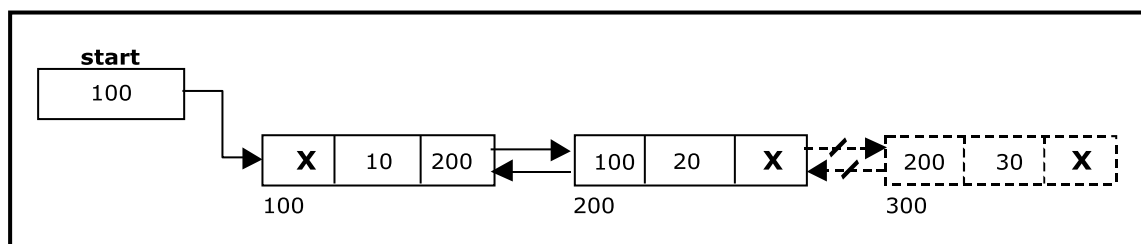


Figure 3.4.7. Deleting a node at the end

Deleting a node at Intermediate position:

The following steps are followed, to delete a node from an intermediate position in the list (List must contain more than two nodes).

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:
 - Get the position of the node to delete.
 - Ensure that the specified position is in between first node and last node. If not, specified position is invalid.
 - Then perform the following steps:

```
if(pos > 1 && pos < nodectr)
{
    temp = start;
    i = 1;
    while(i < pos)
    {
        temp = temp -> right;
        i++;
    }
    temp -> right -> left = temp -> left;
    temp -> left -> right = temp -> right;
    free(temp);
    printf("\n node deleted..");
}
```

The function `delete_at_mid()`, is used for deleting the intermediate node in the list. Figure 3.4.8 shows deleting a node at a specified intermediate position other than beginning and end from a double linked list.

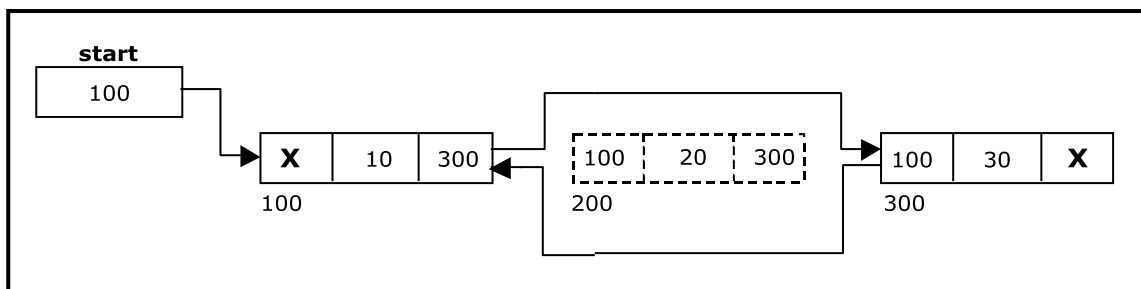


Figure 3.4.8 Deleting a node at an intermediate position

Traversal and displaying a list (Left to Right):

To display the information, you have to traverse the list, node by node from the first node, until the end of the list is reached. The function `traverse_left_right()` is used for traversing and displaying the information stored in the list from left to right.

The following steps are followed, to traverse a list from left to right:

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:

```

temp = start;
while(temp != NULL)
{
    print temp -> data;
    temp = temp -> right;
}

```

Traversal and displaying a list (Right to Left):

To display the information from right to left, you have to traverse the list, node by node from the first node, until the end of the list is reached. The function *traverse_right_left()* is used for traversing and displaying the information stored in the list from right to left. The following steps are followed, to traverse a list from right to left:

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:

```

temp = start;
while(temp -> right != NULL)
    temp = temp -> right;
while(temp != NULL)
{
    print temp -> data;
    temp = temp -> left;
}

```

Counting the Number of Nodes:

The following code will count the number of nodes exist in the list (using recursion).

```

int countno de( node * start)
{
    if(start == NULL)
        return 0;
    else
        return(1 + countno de(start -> right));
}

```

3.5. A Complete Source Code for the Implementation of Double Linked List:

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

struct dlinklist
{
    struct dlinklist *left;
    int data;
    struct dlinklist *right;
};

typedef struct dlinklist node;
node *start = NULL;

```

```

node* getnode()
{
    node * newnode;
    newnode = (node *) malloc(sizeof(node));
    printf("\n Enter data: ");
    scanf("%d", &newnode -> data);
    newnode -> left = NULL;
    newnode -> right = NULL;
    return newnode;
}

int countnode(node *start)
{
    if(start == NULL)
        return 0;
    else
        return 1 + countnode(start -> right);
}

int menu()
{
    int ch;
    clrscr();
    printf("\n 1.Create");
    printf("\n -----");
    printf("\n 2. Insert a node at beginning ");
    printf("\n 3. Insert a node at end");
    printf("\n 4. Insert a node at middle");
    printf("\n -----");
    printf("\n 5. Delete a node from beginning");
    printf("\n 6. Delete a node from Last");
    printf("\n 7. Delete a node from Middle");
    printf("\n -----");
    printf("\n 8. Traverse the list from Left to Right ");
    printf("\n 9. Traverse the list from Right to Left ");
    printf("\n -----");
    printf("\n 10.Count the Number of nodes in the list");
    printf("\n 11.Exit ");
    printf("\n\n Enter your choice: ");
    scanf("%d", &ch);
    return ch;
}

void createlist(int n)
{
    int i;
    node *newnode;
    node *temp;
    for(i = 0; i < n; i++)
    {
        newnode = getnode();
        if(start == NULL)
            start = newnode;
        else
        {
            temp = start;
            while(temp -> right)
                temp = temp -> right;
            temp -> right = newnode;
            newnode -> left = temp;
        }
    }
}

```

```

void traverse_left_to_right()
{
    node *temp;
    temp = start;
    printf("\n The contents of List: ");
    if(start == NULL )
        printf("\n Empty List");
    else
    {
        while(temp != NULL)
        {
            printf("\t %d ", temp -> data);
            temp = temp -> right;
        }
    }
}

void traverse_right_to_left()
{
    node *temp;
    temp = start;
    printf("\n The contents of List: ");
    if(start == NULL)
        printf("\n Empty List");
    else
    {
        while(temp -> right != NULL)
            temp = temp -> right;

        while(temp != NULL)
        {
            printf("\t %d", temp -> data);
            temp = temp -> left;
        }
    }
}

void dll_insert_beg()
{
    node *newnode;
    newnode = getnode();
    if(start == NULL)
        start = newnode;
    else
    {
        newnode -> right = start;
        start -> left = newnode;
        start = newnode;
    }
}

void dll_insert_end()
{
    node *newnode, *temp;
    newnode = getnode();
    if(start == NULL)
        start = newnode;
    else
    {
        temp = start;
        while(temp -> right != NULL)
            temp = temp -> right;
        temp -> right = newnode;
        newnode -> left = temp;
    }
}

```

wnode -> left = temp;

temp = start;
w

while (temp->right == NULL) {
temp = temp->right;
}

temp->right = newnode;
new

```

void dll_insert_mid()
{
    node *newnode,*temp;
    int pos, nodectr, ctr = 1;
    newnode = getnode();
    printf("\n Enter the position: ");
    scanf("%d", &pos);
    nodectr = countnode(start);
    if(pos - nodectr >= 2)
    {
        printf("\n Position is out of range..");
        return;
    }
    if(pos > 1 && pos < nodectr)
    {
        temp = start;
        while(ctr < pos - 1)
        {
            temp = temp -> right;
            ctr++;
        }
        newnode -> left = temp;
        newnode -> right = temp -> right;
        temp -> right -> left = newnode;
        temp -> right = newnode;
    }
    else
        printf("position %d of list is not a middle position ", pos);
}

```

```

void dll_delete_beg()
{
    node *temp;
    if(start == NULL)
    {
        printf("\n Empty list");
        getch();
        return ;
    }
    else
    {
        temp = start;
        start = start -> right;
        start -> left = NULL;
        free(temp);
    }
}

```

```

void dll_delete_last()
{
    node *temp;
    if(start == NULL)
    {
        printf("\n Empty list");
        getch();
        return ;
    }
    else
    {
        temp = start;

```

```
while(temp ->  
right != NULL)
```

```

        temp = temp -> right;
        temp -> left -> right = NULL;
        free(temp);
        temp = NULL;
    }
}

void dll_delete_mid()
{
    int i = 0, pos, nodectr;
    node *temp;
    if(start == NULL)
    {
        printf("\n Empty List");
        getch();
        return;
    }
    else
    {
        printf("\n Enter the position of the node to delete: ");
        scanf("%d", &pos);
        nodectr = countnode(start);
        if(pos > nodectr)
        {
            printf("\n this node does not exist");
            getch();
            return;
        }
        if(pos > 1 && pos < nodectr)
        {
            temp = start;
            i = 1;
            while(i < pos)
            {
                temp = temp -> right;
                i++;
            }
            temp -> right -> left = temp -> left;
            temp -> left -> right = temp -> right;
            free(temp);
            printf("\n node deleted..");
        }
        else
        {
            printf("\n It is not a middle position..");
            getch();
        }
    }
}

void main(void)
{
    int ch, n;
    clrscr();
    while(1)
    {
        ch = menu();
        switch( ch)
        {
            case 1 :
                printf("\n Enter Number of nodes to create: ");
                scanf("%d", &n);
                createlist(n);

```



```

        printf("\n List created..");
        break;
    case 2 :
        dll_insert_beg();
        break;
    case 3 :
        dll_insert_end();
        break;
    case 4 :
        dll_insert_mid();
        break;
    case 5 :
        dll_delete_beg();
        break;
    case 6 :
        dll_delete_last();
        break;
    case 7 :
        dll_delete_mid();
        break;
    case 8 :
        traverse_left_to_right();
        break;
    case 9 :
        traverse_right_to_left();
        break;

    case 10 :
        printf("\n Number of nodes: %d", countnode(start));
        break;
    case 11:
        exit(0);
    }
    getch();
}
}

```

3.7. Circular Single Linked List:

It is just a single linked list in which the link field of the last node points back to the address of the first node. A circular linked list has no beginning and no end. It is necessary to establish a special pointer called *start* pointer always pointing to the first node of the list. Circular linked lists are frequently used instead of ordinary linked list because many operations are much easier to implement. In circular linked list no null pointers are used, hence all pointers contain valid address.

A circular single linked list is shown in figure 3.6.1.

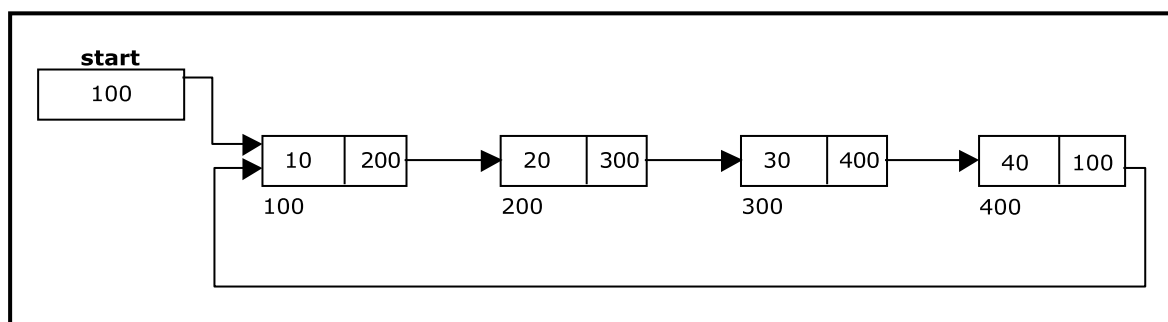


Figure 3.6.1. Circular Single Linked List

The basic operations in a circular single linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing.

Creating a circular single Linked List with 'n' number of nodes:

The following steps are to be followed to create 'n' number of nodes:

- Get the new node using `getnode()`.
`newnode = getnode();`
- If the list is empty, assign new node as start.
`start = newnode;`
- If the list is not empty, follow the steps given below:
`temp = start;`
`while(temp -> next != NULL)`
`temp = temp -> next;`
`temp -> next = newnode;`
- Repeat the above steps 'n' times.
- `newnode -> next = start;`

The function `createlist()`, is used to create 'n' number of nodes:

Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the circular list:

- Get the new node using `getnode()`.
`newnode = getnode();`
- If the list is empty, assign new node as start.
`start = newnode;`
`newnode -> next = start;`
- If the list is not empty, follow the steps given below:
`last = start;`
`while(last -> next != start)`
`last = last -> next;`
`newnode -> next = start;`
`start = newnode;`
`last -> next = start;`

The function `cil_insert_beg()`, is used for inserting a node at the beginning. Figure 3.6.2 shows inserting a node into the circular single linked list at the beginning.

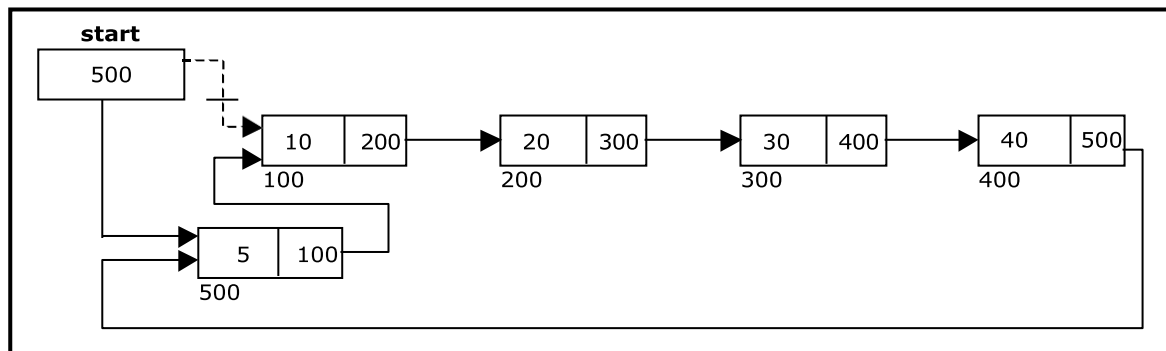


Figure 3.6.2. Inserting a node at the beginning

Inserting a node at the end:

The following steps are followed to insert a new node at the end of the list:

- Get the new node using `getnode()`.
`newnode = getnode();`
- If the list is empty, assign new node as start.
`start = newnode;`
`newnode -> next = start;`
- If the list is not empty follow the steps given below:
`temp = start;`
`while(temp -> next != start)`
 `temp = temp -> next;`
`temp -> next = newnode;`
`newnode -> next = start;`

The function `cil_insert_end()`, is used for inserting a node at the end.

Figure 3.6.3 shows inserting a node into the circular single linked list at the end.

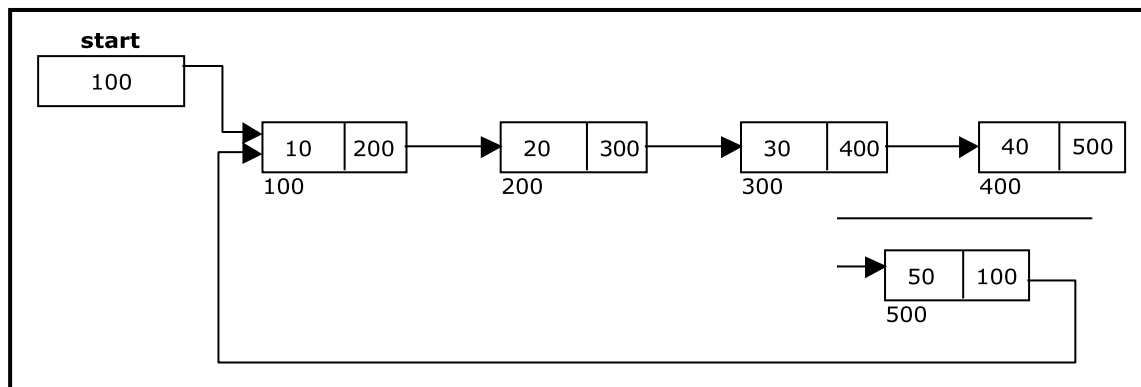


Figure 3.6.3 Inserting a node at the end.

Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

- If the list is empty, display a message 'Empty List'.
- If the list is not empty, follow the steps given below:

```
last = temp = start;  
while(last -> next != start)  
    last = last -> next;  
start = start -> next;  
last -> next = start;
```
- After deleting the node, if the list is empty then *start* = *NULL*.

The function `cil_delete_beg()`, is used for deleting the first node in the list. Figure 3.6.4 shows deleting a node at the beginning of a circular single linked list.

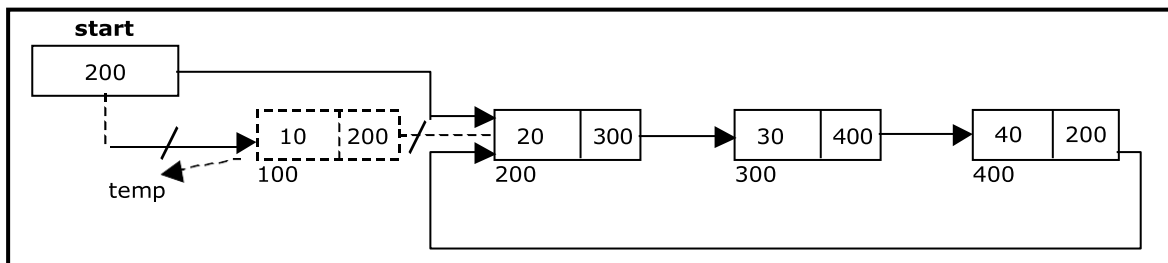


Figure 3.6.4. Deleting a node at beginning.

Deleting a node at the end:

The following steps are followed to delete a node at the end of the list:

- If the list is empty, display a message 'Empty List'.
- If the list is not empty, follow the steps given below:

```
temp = start;  
prev = start;  
while(temp -> next != start)  
{  
    prev = temp;  
    temp = temp -> next;  
}  
prev -> next = start;
```
- After deleting the node, if the list is empty then *start* = *NULL*.

The function `cil_delete_last()`, is used for deleting the last node in the list.

Figure 3.6.5 shows deleting a node at the end of a circular single linked list.

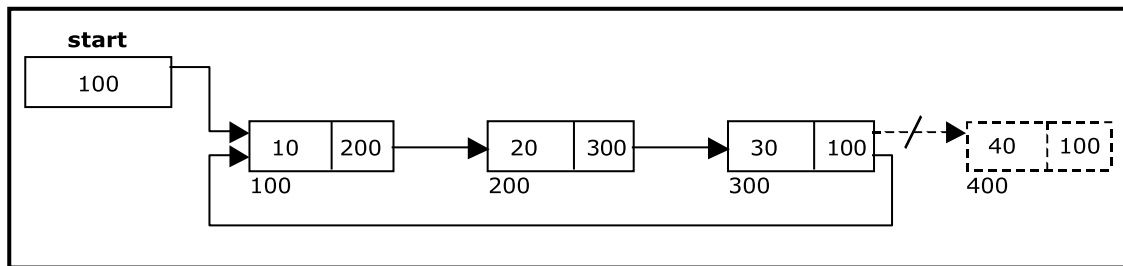


Figure 3.6.5. Deleting a node at the end.

Traversing a circular single linked list from left to right:

The following steps are followed, to traverse a list from left to right:

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:

```
temp = start;
do
{
    printf("%d ", temp -> data);
    temp = temp -> next;
} while(temp != start);
```

3.7.1. Source Code for Circular Single Linked List:

```
# include <stdio.h>
# include <conio.h>
# include <stdlib.h>

struct cslinklist
{
    int data;
    struct cslinklist *next;
};

typedef struct cslinklist node;

node *start = NULL;

int nodectr;

node* getnode()
{
    node * newnode;
    newnode = (node *) malloc(sizeof(node));
    printf("\n Enter data: ");
    scanf("%d", &newnode -> data);
    newnode -> next = NULL;
    return newnode;
}
```

```

int menu()
{
    int ch;
    clrscr();
    printf("\n 1. Create a list ");
    printf("\n\n----- ");
    printf("\n 2. Insert a node at beginning ");
    printf("\n 3. Insert a node at end");
    printf("\n 4. Insert a node at middle");
    printf("\n\n----- ");
    printf("\n 5. Delete a node from beginning");
    printf("\n 6. Delete a node from Last");
    printf("\n 7. Delete a node from Middle");
    printf("\n\n----- ");
    printf("\n 8. Display the list");
    printf("\n 9. Exit");
    printf("\n\n----- ");
    printf("\n Enter your choice: ");
    scanf("%d", &ch);
    return ch;
}

void createlist(int n)
{
    int i;
    node *newnode;
    node *temp;
    nodectr = n;
    for(i = 0; i < n ; i++)
    {
        newnode = getnode();
        if(start == NULL)
        {
            start = newnode;
        }
        else
        {
            temp = start;
            while(temp -> next != NULL)
                temp = temp -> next;
            temp -> next = newnode;
        }
    }

    newnode -> next = start;    /* last node is pointing to starting node */
}

void display()
{
    node *temp;
    temp = start;
    printf("\n The contents of List (Left to Right): ");
    if(start == NULL )
        printf("\n Empty List");
    else
    {
        do
        {
            printf("\t %d ", temp -> data);
            temp = temp -> next;
        } while(temp != start);
        printf(" X ");
    }
}

```

```

void cll_insert_beg()
{
    node *newnode, *last;
    newnode = getnode();
    if(start == NULL)
    {
        start = newnode;
        newnode -> next = start;
    }
    else
    {
        last = start;
        while(last -> next != start)
            last = last -> next;
        newnode -> next = start;
        start = newnode;
        last -> next = start;
    }

    printf("\n Node inserted at beginning..");
    nodectr++;
}

```

```

void cll_insert_end()
{
    node *newnode, *temp;
    newnode = getnode();
    if(start == NULL )
    {
        start = newnode;
        newnode -> next = start;
    }
    else
    {
        temp = start;
        while(temp -> next != start)
            temp = temp -> next;
        temp -> next = newnode;
        newnode -> next = start;
    }

    printf("\n Node inserted at end..");
    nodectr++;
}

```

```

void cll_insert_mid()
{
    node *newnode, *temp, *prev;
    int i, pos ;
    newnode = getnode();
    printf("\n Enter the position: ");
    scanf("%d", &pos);
    if(pos > 1 && pos < nodectr)
    {
        temp = start;
        prev = temp;
        i = 1;
        while(i < pos)
        {
            prev = temp;
            temp = temp -> next;
            i++;
        }
        prev -> next = newnode;
    }
}

```

```
newnode -> next = temp;
```



```

        nodectr++;
        printf("\n Node inserted at middle..");
    }
    else
    {
        printf("position %d of list is not a middle position ", pos);
    }
}

```

```

void cll_delete_beg()
{
    node *temp, *last;
    if(start == NULL)
    {
        printf("\n No nodes exist..");
        getch();
        return ;
    }
    else
    {
        last = temp = start;
        while(last -> next != start)
            last = last -> next;
        start = start -> next;
        last -> next = start;
        free(temp);
        nodectr--;
        printf("\n Node deleted..");
        if(nodectr == 0)
            start = NULL;
    }
}

```

```

void cll_delete_last()
{
    node *temp,*prev;
    if(start == NULL)
    {
        printf("\n No nodes exist..");
        getch();
        return ;
    }
    else
    {
        temp = start;
        prev = start;
        while(temp -> next != start)
        {
            prev = temp;
            temp = temp -> next;
        }
        prev -> next = start;
        free(temp);
        nodectr--;
        if(nodectr == 0)
            start = NULL;
        printf("\n Node deleted..");
    }
}

```

```

void cll_delete_mid()
{
    int i = 0, pos;
    node *temp, *prev;

    if(start == NULL)
    {
        printf("\n No nodes exist..");
        getch();
        return ;
    }
    else
    {
        printf("\n Which node to delete: ");
        scanf("%d", &pos);
        if(pos > nodectr)
        {
            printf("\n This node does not exist");
            getch();
            return;
        }
        if(pos > 1 && pos < nodectr)
        {
            temp=start;
            prev = start;
            i = 0;
            while(i < pos - 1)
            {
                prev = temp;
                temp = temp -> next ;
                i++;
            }
            prev -> next = temp -> next;
            free(temp);
            nodectr--;
            printf("\n Node Deleted..");
        }
        else
        {
            printf("\n It is not a middle position..");
            getch();
        }
    }
}

```

```

void main(void)
{
    int result;
    int ch, n;
    clrscr();
    while(1)
    {
        ch = menu();
        switch(ch)
        {
            case 1 :
                if(start == NULL)
                {
                    printf("\n Enter Number of nodes to create: ");
                    scanf("%d", &n);
                    createlist(n);
                    printf("\n List created..");
                }

```

```

else
    printf("\n List is already Exist..");
break;
case 2 :

    cll_insert_beg();
break;
case 3 :
    cll_insert_end();
break;
case 4 :
    cll_insert_mid();
break;
case 5 :
    cll_delete_beg();
break;
case 6 :
    cll_delete_last();
break;
case 7 :
    cll_delete_mid();
break;
case 8 :
    display();
break;
case 9 :
    exit(0);
}
getch();
}
}

```

3.8. Circular Double Linked List:

A circular double linked list has both successor pointer and predecessor pointer in circular manner. The objective behind considering circular double linked list is to simplify the insertion and deletion operations performed on double linked list. In circular double linked list the *right* link of the right most node points back to the *start* node and *left* link of the first node points to the last node. A circular double linked list is shown in figure 3.8.1.

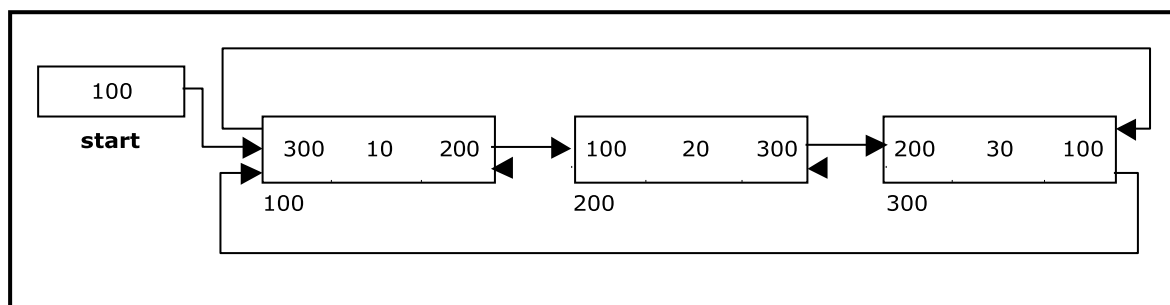


Figure 3.8.1. Circular Double Linked List

The basic operations in a circular double linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing.

Creating a Circular Double Linked List with 'n' number of nodes:

The following steps are to be followed to create 'n' number of nodes:

- Get the new node using `getnode()`.
`newnode = getnode();`
- If the list is empty, then do the following
`start = newnode;`
`newnode -> left = start;`
`newnode -> right = start;`
- If the list is not empty, follow the steps given below:
`newnode -> left = start -> left;`
`newnode -> right = start;`
`start -> left -> right = newnode;`
`start -> left = newnode;`
- Repeat the above steps 'n' times.

The function `cdll_createlist()`, is used to create 'n' number of nodes:

Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the list:

- Get the new node using `getnode()`.
`newnode = getnode();`
- If the list is empty, then
`start = newnode;`
`newnode -> left = start;`
`newnode -> right = start;`
- If the list is not empty, follow the steps given below:
`newnode -> left = start -> left;`
`newnode -> right = start;`
`start -> left -> right = newnode;`
`start -> left = newnode;`
`start = newnode;`

The function `cdll_insert_beg()`, is used for inserting a node at the beginning. Figure 3.8.2 shows inserting a node into the circular double linked list at the beginning.

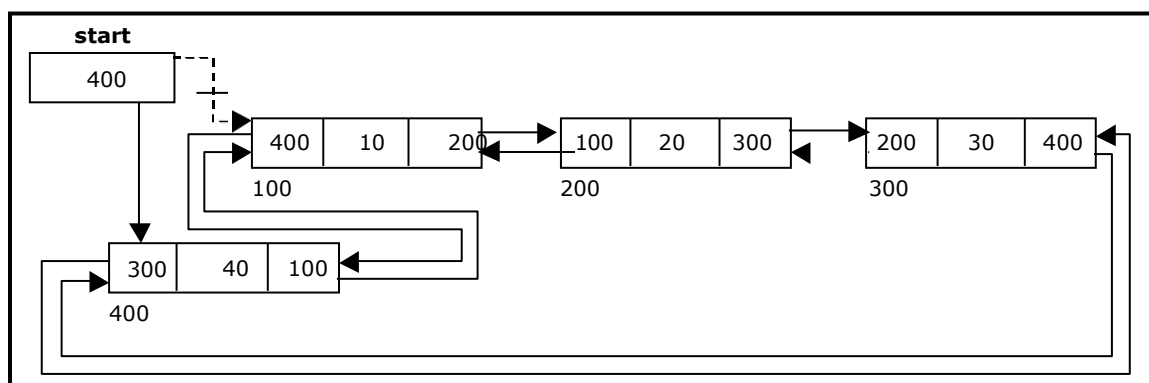


Figure 3.8.2. Inserting a node at the beginning

Inserting a node at the end:

The following steps are followed to insert a new node at the end of the list:

- Get the new node using `getnode()`
`newnode=getnode();`
- If the list is empty, then
`start = newnode;`
`newnode -> left = start;`
`newnode -> right = start;`
- If the list is not empty follow the steps given below:
`newnode -> left = start -> left;`
`newnode -> right = start;`
`start -> left -> right = newnode;`
`start -> left = newnode;`

The function `cdll_insert_end()`, is used for inserting a node at the end. Figure 3.8.3 shows inserting a node into the circular linked list at the end.

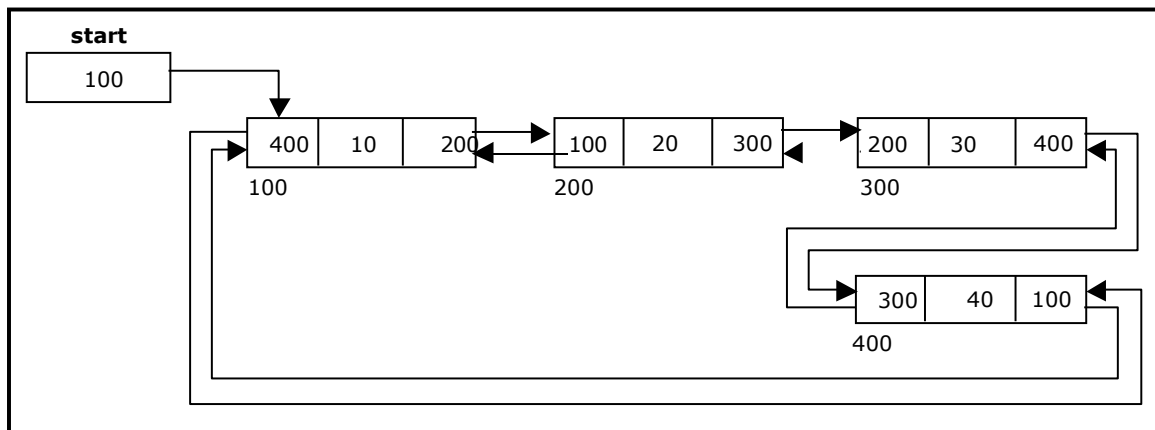


Figure 3.8.3. Inserting a node at the end

Inserting a node at an intermediate position:

The following steps are followed, to insert a new node in an intermediate position in the list:

- Get the new node using `getnode()`.
`newnode=getnode();`
- Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by `countnode()` function.
- Store the starting address (which is in start pointer) in temp. Then traverse the temp pointer upto the specified position.
- After reaching the specified position, follow the steps given below:
`newnode -> left = temp;`
`newnode -> right = temp -> right;`
`temp -> right -> left = newnode;`
`temp -> right = newnode;`
`nodectr++;`

The function `cdll_insert_mid()`, is used for inserting a node in the intermediate position. Figure 3.8.4 shows inserting a node into the circular double linked list at a specified intermediate position other than beginning and end.

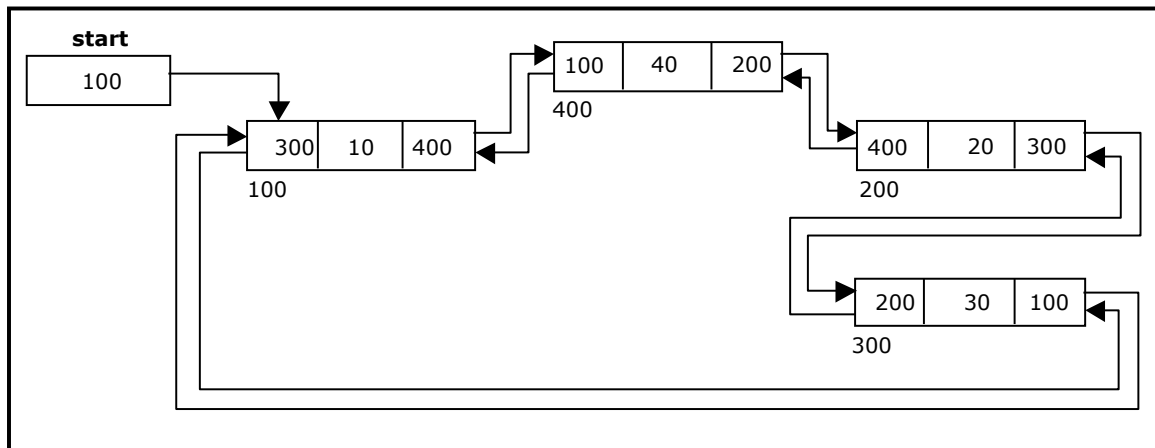


Figure 3.8.4. Inserting a node at an intermediate position

Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:

```
temp = start;
start = start -> right;
temp -> left -> right = start;
start -> left = temp -> left;
```

The function `cdll_delete_beg()`, is used for deleting the first node in the list. Figure 3.8.5 shows deleting a node at the beginning of a circular double linked list.

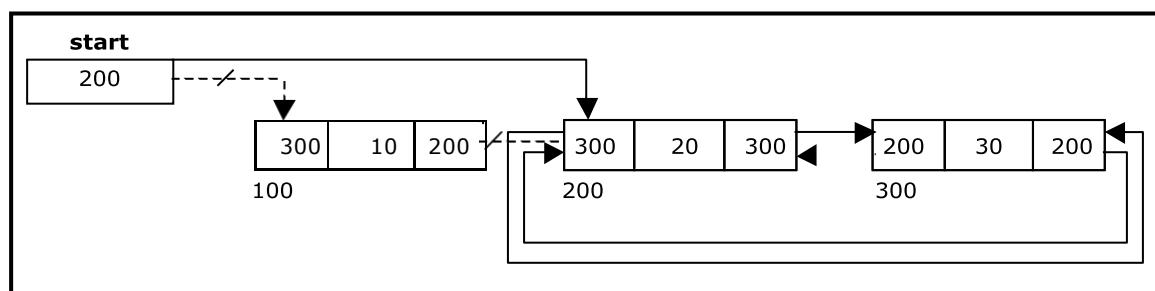


Figure 3.8.5. Deleting a node at beginning

Deleting a node at the end:

The following steps are followed to delete a node at the end of the list:

- If list is empty then display 'Empty List' message
- If the list is not empty, follow the steps given below:

```

temp = start;
while(temp -> right != start)
{
    temp = temp -> right;
}
temp -> left -> right = temp -> right;
temp -> right -> left = temp -> left;

```

The function `cdll_delete_last()`, is used for deleting the last node in the list. Figure 3.8.6 shows deleting a node at the end of a circular double linked list.

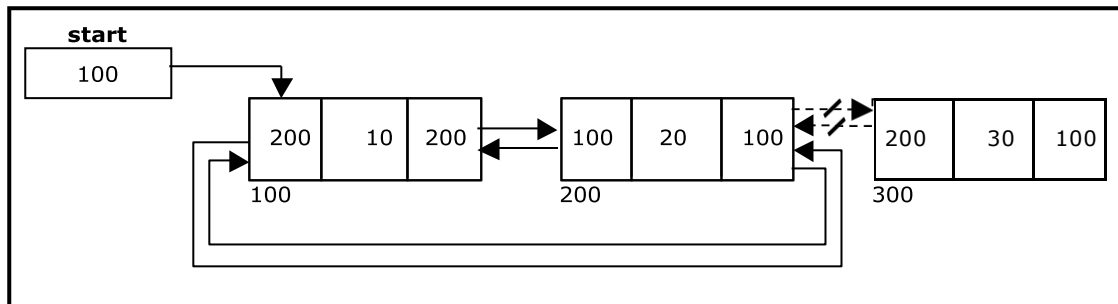


Figure 3.8.6. Deleting a node at the end

Deleting a node at Intermediate position:

The following steps are followed, to delete a node from an intermediate position in the list (List must contain more than two node).

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:
 - Get the position of the node to delete.
 - Ensure that the specified position is in between first node and last node. If not, specified position is invalid.
 - Then perform the following steps:

```

if(pos > 1 && pos < nodectr)
{
    temp = start;
    i = 1;
    while(i < pos)
    {
        temp = temp -> right ;
        i++;
    }
    temp -> right -> left = temp -> left;
    temp -> left -> right = temp -> right;
    free(temp);
    printf("\n node deleted..");
    nodectr--;
}

```

The function `cdll_delete_mid()`, is used for deleting the intermediate node in the list.

Figure 3.8.7 shows deleting a node at a specified intermediate position other than beginning and end from a circular double linked list.

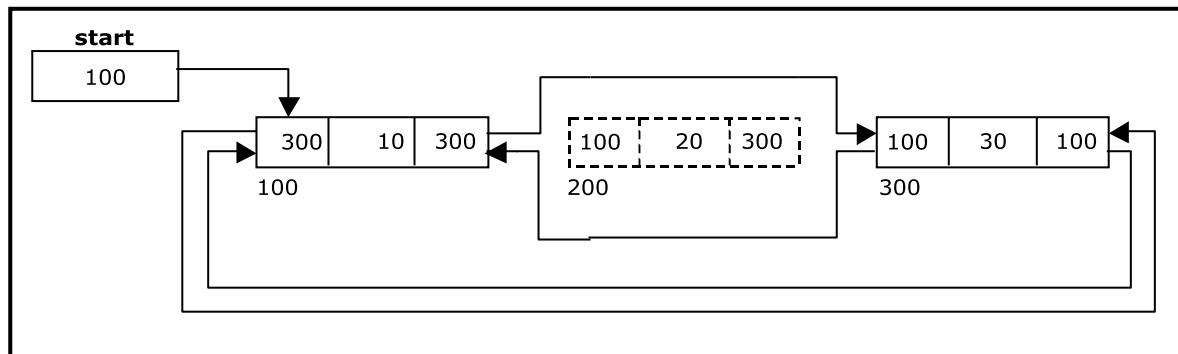


Figure 3.8.7. Deleting a node at an intermediate position

Traversing a circular double linked list from left to right:

The following steps are followed, to traverse a list from left to right:

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:
temp = start;
Print temp -> data;
temp = temp -> right;
while(temp != start)
{
print temp -> data;
temp = temp -> right;
}

The function `cdll_display_left_right()`, is used for traversing from left to right.

Traversing a circular double linked list from right to left:

The following steps are followed, to traverse a list from right to left:

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:
temp = start;
do
{
temp = temp -> left;
print temp -> data;
} while(temp != start);

The function `cdll_display_right_left()`, is used for traversing from right to left.

3.8.1. Source Code for Circular Double Linked List:

```
# include <stdio.h>
# include <stdlib.h>
# include <conio.h>
```



```

struct cdlinklist
{
    struct cdlinklist *left;
    int data;
    struct cdlinklist *right;
};

typedef struct cdlinklist node;
node *start = NULL;
int nodectr;

node* getnode()
{
    node * newnode;
    newnode = (node *) malloc(sizeof(node));
    printf("\n Enter data: ");
    scanf("%d", &newnode->data);
    newnode->left = NULL;
    newnode->right = NULL;
    return newnode;
}

int menu()
{
    int ch;
    clrscr();
    printf("\n 1. Create ");
    printf("\n\n----- ");
    printf("\n 2. Insert a node at Beginning");
    printf("\n 3. Insert a node at End");
    printf("\n 4. Insert a node at Middle");
    printf("\n\n----- ");
    printf("\n 5. Delete a node from Beginning");
    printf("\n 6. Delete a node from End");
    printf("\n 7. Delete a node from Middle");
    printf("\n\n----- ");
    printf("\n 8. Display the list from Left to Right");
    printf("\n 9. Display the list from Right to Left");
    printf("\n 10.Exit");
    printf("\n\n Enter your choice: ");
    scanf("%d", &ch);
    return ch;
}

void cdll_createlist(int n)
{
    int i;
    node *newnode, *temp;
    if(start == NULL)
    {
        nodectr = n;
        for(i = 0; i < n; i++)
        {
            newnode = getnode();
            if(start == NULL)
            {
                start = newnode;
                newnode->left = start;
                newnode->right = start;
            }
            else
            {
                newnode->left = start->left;

```

```

                                newnode -> right = start;
                                start -> left->right = newnode;
                                start -> left = newnode;
                                }
                                }
                                }
                                else
                                {
                                printf("\n List already exists..");
                                }

void cdll_display_left_right()
{
    node *temp;
    temp = start;
    if(start == NULL)
        printf("\n Empty List");
    else
    {
        printf("\n The contents of List: ");
        printf(" %d ", temp -> data);
        temp = temp -> right;
        while(temp != start)
        {
            printf(" %d ", temp -> data);
            temp = temp -> right;
        }
    }
}

void cdll_display_right_left()
{
    node *temp;
    temp = start;
    if(start == NULL)
        printf("\n Empty List");
    else
    {
        printf("\n The contents of List: ");
        do
        {
            temp = temp -> left;
            printf("\t%d", temp -> data);
        } while(temp != start);
    }
}

void cdll_insert_beg()
{
    node *newnode;
    newnode = getnode();
    nodectr++;
    if(start == NULL)
    {
        start = newnode;
        newnode -> left = start;
        newnode -> right = start;
    }
    else
    {
        newnode -> left = start -> left;
        newnode -> right = start;
        start -> left -> right = newnode;
        start -> left = newnode;
    }
}

```

```

        start = newnode;
    }
}

void cdll_insert_end()
{
    node *newnode,*temp;
    newnode = getnode();
    nodectr++;
    if(start == NULL)
    {
        start = newnode;
        newnode -> left = start;
        newnode -> right = start;
    }
    else
    {
        newnode -> left = start -> left;
        newnode -> right = start;
        start -> left -> right = newnode;
        start -> left = newnode;
    }

    printf("\n Node Inserted at End");
}

void cdll_insert_mid()
{
    node *newnode, *temp, *prev;
    int pos, ctr = 1;
    newnode = getnode();
    printf("\n Enter the position: ");
    scanf("%d", &pos);
    if(pos - nodectr >= 2)
    {
        printf("\n Position is out of range..");
        return;
    }
    if(pos > 1 && pos <= nodectr)
    {
        temp = start;
        while(ctr < pos - 1)
        {
            temp = temp -> right;
            ctr++;
        }
        newnode -> left = temp;
        newnode -> right = temp -> right;
        temp -> right -> left = newnode;
        temp -> right = newnode;
        nodectr++;
        printf("\n Node Inserted at Middle.. ");
    }
    else
        printf("position %d of list is not a middle position", pos);
}

}

void cdll_delete_beg()
{
    node *temp;
    if(start == NULL)
    {

```

```
printf("\n No nodes exist..");
```

```

        getch();
        return ;
    }
    else
    {
        nodectr--;
        if(nodectr == 0)
        {
            free(start);
            start = NULL;
        }
        else
        {
            temp = start;
            start = start -> right;
            temp -> left -> right = start;
            start -> left = temp -> left;
            free(temp);
        }

        printf("\n Node deleted at Beginning..");
    }
}

void cdll_delete_last()
{
    node *temp;
    if(start == NULL)
    {
        printf("\n No nodes exist..");
        getch();
        return;
    }
    else
    {
        nodectr--;
        if(nodectr == 0)
        {
            free(start);
            start = NULL;
        }
        else
        {
            temp = start;
            while(temp -> right != start)
                temp = temp -> right;
            temp -> left -> right = temp -> right;
            temp -> right -> left = temp -> left;
            free(temp);
        }

        printf("\n Node deleted from end ");
    }
}

void cdll_delete_mid()
{
    int ctr = 1, pos;
    node *temp;
    if( start == NULL)
    {
        printf("\n No nodes exist..");
        getch();
        return;
    }

```

}

```

else
{
    printf("\n Which node to delete: ");
    scanf("%d", &pos);
    if(pos > nodectr)
    {
        printf("\nThis node does not exist");
        getch();
        return;
    }
    if(pos > 1 && pos < nodectr)
    {
        temp = start;
        while(ctr < pos)
        {
            temp = temp -> right ;
            ctr++;
        }
        temp -> right -> left = temp -> left;
        temp -> left -> right = temp -> right;
        free(temp);
        printf("\n node deleted..");
        nodectr--;
    }
    else
    {
        printf("\n It is not a middle position..");
        getch();
    }
}
}

```

```

void main(void)
{
    int ch,n;
    clrscr();
    while(1)
    {
        ch = menu();
        switch( ch)
        {
            case 1 :
                printf("\n Enter Number of nodes to create: ");
                scanf("%d", &n);
                cdll_createlist(n);
                printf("\n List created..");
                break;
            case 2 :
                cdll_insert_beg();
                break;
            case 3 :
                cdll_insert_end();
                break;
            case 4 :
                cdll_insert_mid();
                break;
            case 5 :
                cdll_delete_beg();
                break;
            case 6 :
                cdll_delete_last();
                break;
        }
    }
}

```

```

        case 7 :
            cdll_delete_mid();
            break;
        case 8 :
            cdll_display_left_right();
            break;
        case 9 :
            cdll_display_right_left();
            break;
        case 10:
            exit(0);
    }
    getch();
}
}

```

3.9. Comparison of Linked List Variations:

The major disadvantage of doubly linked lists (over singly linked lists) is that they require more space (every node has two pointer fields instead of one). Also, the code to manipulate doubly linked lists needs to maintain the *prev* fields as well as the *next* fields; the more fields that have to be maintained, the more chance there is for errors.

The major advantage of doubly linked lists is that they make some operations (like the removal of a given node, or a right-to-left traversal of the list) more efficient.

The major advantage of circular lists (over non-circular lists) is that they eliminate some extra-case code for some operations (like deleting last node). Also, some applications lead naturally to circular list representations. For example, a computer network might best be modeled using a circular list.

3.10. Polynomials:

A polynomial is of the form: $\sum_{i=0}^n c_i x^i$

Where, c_i is the coefficient of the i^{th} term and
 n is the degree of the polynomial

Some examples are:

$$\begin{aligned}
 &5x^2 + 3x + 1 \\
 &12x^3 - 4x \\
 &5x^4 - 8x^3 + 2x^2 + 4x^1 + 9x^0
 \end{aligned}$$

It is not necessary to write terms of the polynomials in decreasing order of degree. In other words the two polynomials $1 + x$ and $x + 1$ are equivalent.

The computer implementation requires implementing polynomials as a list of pairs of coefficient and exponent. Each of these pairs will constitute a structure, so a polynomial will be represented as a list of structures. A linked list structure that represents polynomials $5x^4 - 8x^3 + 2x^2 + 4x^1 + 9x^0$ illustrates in figure 3.10.1.

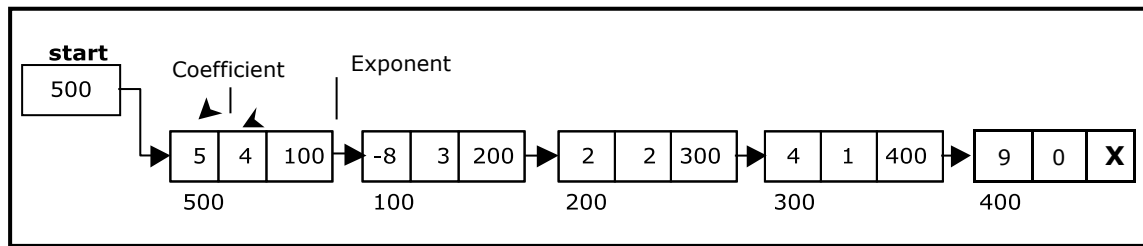


Figure 3.10.1. Single Linked List for the polynomial $F(x) = 5x^4 - 8x^3 + 2x^2 + 4x^1 + 9x^0$

3.10.1. Source code for polynomial creation with help of linked list:

```
#include <conio.h>
#include <stdio.h>
#include <malloc.h>

struct link
{
    float coef;
    int expo;
    struct link *next;
};

typedef struct link node;
node * getnode()
{
    node *tmp;
    tmp =(node *) malloc( sizeof(node) );
    printf("\n Enter Coefficient : ");
    fflush(stdin);
    scanf("%f",&tmp->coef);
    printf("\n Enter Exponent : ");
    fflush(stdin);
    scanf("%d",&tmp->expo);
    tmp->next = NULL;
    return tmp;
}

node * create_poly (node *p )
{
    char ch;
    node *temp,*newnode;
    while( 1 )
    {
        printf ("\n Do U Want polynomial node (y/n): ");
        ch = getch();
        if(ch == 'n')
            break;
        newnode = getnode();
        if( p == NULL )
            p = newnode;
        else
        {
            temp = p;
            while(temp->next != NULL )
                temp = temp->next;
            temp->next = newnode;
        }
    }

    return p;
}
```

```

void display (node *p)
{
    node *t = p;
    while (t != NULL)
    {
        printf("+ %.2f", t -> coef);
        printf("X^ %d", t -> expo);
        t = t -> next;
    }
}

void main()
{
    node *poly1 = NULL , *poly2 = NULL, *poly3=NULL;
    clrscr();
    printf("\nEnter First Polynomial..(in ascending-order of exponent)");
    poly1 = create_poly (poly1);
    printf("\nEnter Second Polynomial..(in ascending-order of exponent)");
    poly2 = create_poly (poly2);
    clrscr();
    printf("\n Enter Polynomial 1: ");
    display (poly1);
    printf("\n Enter Polynomial 2: ");
    display (poly2);
    getch();
}

```

3.10.2. Addition of Polynomials:

To add two polynomials we need to scan them once. If we find terms with the same exponent in the two polynomials, then we add the coefficients; otherwise, we copy the term of larger exponent into the sum and go on. When we reach at the end of one of the polynomial, then remaining part of the other is copied into the sum.

To add two polynomials follow the following steps:

- Read two polynomials.
- Add them.
- Display the resultant polynomial.

3.10.3. Source code for polynomial addition with help of linked list:

```

#include <conio.h>
#include <stdio.h>
#include <malloc.h>

struct link
{
    float coef;
    int expo;
    struct link *next;
};

typedef struct link node;

node * getnode()
{
    node *tmp;

```

```

        tmp =(node *) malloc( sizeof(node) );
        printf("\n Enter Coefficient : ");
        fflush(stdin);
        scanf("%f",&tmp->coef);
        printf("\n Enter Exponent : ");
        fflush(stdin);
        scanf("%d",&tmp->expo);
        tmp->next = NULL;
        return tmp;
    }

node * create_poly (node *p )
{
    char ch;
    node *temp,*newnode;
    while( 1 )
    {
        printf ("\n Do U Want polynomial node (y/n): ");
        ch = getche();
        if(ch == 'n')
            break;
        newnode = getnode();
        if( p == NULL )
            p = newnode;
        else
        {
            temp = p;
            while(temp->next != NULL )
                temp = temp->next;
            temp->next = newnode;
        }
    }
    return p;
}

void display (node *p)
{
    node *t = p;
    while (t != NULL)
    {
        printf("+ %.2f", t -> coef);
        printf("X^ %d", t -> expo);
        t = t -> next;
    }
}

void add_poly(node *p1,node *p2)
{
    node *newnode;
    while(1)
    {
        if( p1 == NULL || p2 == NULL )
            break;
        if(p1->expo == p2->expo )
        {
            printf("+ %.2f X ^%d",p1->coef+p2->coef,p1->expo);
            p1 = p1->next; p2 = p2->next;
        }
        else
        {
            if(p1->expo < p2->expo)

```

```

        {
            printf("+ %.2f X ^%d",p1->coef,p1->expo);
            p1 = p1->next;
        }
        else
        {
            printf(" + %.2f X ^%d",p2->coef,p2->expo);
            p2 = p2->next;
        }
    }
}

while(p1 != NULL )
{
    printf("+ %.2f X ^%d",p1->coef,p1->expo);
    p1 = p1->next;
}
while(p2 != NULL )
{
    printf("+ %.2f X ^%d",p2->coef,p2->expo);
    p2 = p2->next;
}

}

void main()
{
    node *poly1 = NULL ,*poly2 = NULL,*poly3=NULL;
    clrscr();
    printf("\nEnter First Polynomial..(in ascending-order of exponent)");
    poly1 = create_poly (poly1);
    printf("\nEnter Second Polynomial..(in ascending-order of exponent)");
    poly2 = create_poly (poly2);
    clrscr();
    printf("\n Enter Polynomial 1: ");
    display (poly1);
    printf("\n Enter Polynomial 2: ");
    display (poly2);
    printf( "\n Resultant Polynomial : ");
    add_poly(poly1, poly2);
    display (poly3);
    getch();
}

```

Exercise

1. Write a "C" functions to split a given list of integers represented by a single linked list into two lists in the following way. Let the list be $L = (l_0, l_1, \dots, l_n)$. The resultant lists would be $R_1 = (l_0, l_2, l_4, \dots)$ and $R_2 = (l_1, l_3, l_5, \dots)$.
2. Write a "C" function to insert a node "t" before a node pointed to by "X" in a single linked list "L".
3. Write a "C" function to delete a node pointed to by "p" from a single linked list "L".
4. Suppose that an ordered list $L = (l_0, l_1, \dots, l_n)$ is represented by a single linked list. It is required to append the list $L = (l_n, l_0, l_1, \dots, l_n)$ after another ordered list M represented by a single linked list.

5. Implement the following function as a new function for the linked list toolkit.

Precondition: head_ptr points to the start of a linked list. The list might be empty or it might be non-empty.

Postcondition: The return value is the number of occurrences of 42 in the data field of a node on the linked list. The list itself is unchanged.
6. Implement the following function as a new function for the linked list toolkit.

Precondition: head_ptr points to the start of a linked list. The list might be empty or it might be non-empty.

Postcondition: The return value is true if the list has at least one occurrence of the number 42 in the data part of a node.
7. Implement the following function as a new function for the linked list toolkit.

Precondition: head_ptr points to the start of a linked list. The list might be empty or it might be non-empty.

Postcondition: The return value is the sum of all the data components of all the nodes. NOTE: If the list is empty, the function returns 0.
8. Write a "C" function to concatenate two circular linked lists producing another circular linked list.
9. Write "C" functions to compute the following operations on polynomials represented as singly connected linked list of nonzero terms.
 1. Evaluation of a polynomial
 2. Multiplication of two polynomials.
10. Write a "C" function to represent a sparse matrix having "m" rows and "n" columns using linked list.
11. Write a "C" function to print a sparse matrix, each row in one line of output and properly formatted, with zero being printed in place of zero elements.
12. Write "C" functions to:
 1. Add two m X n sparse matrices and
 2. Multiply two m X n sparse matrices.
Where all sparse matrices are to be represented by linked lists.
13. Consider representing a linked list of integers using arrays. Write a "C" function to delete the i^{th} node from the list.

Multiple Choice Questions

1. Which among the following is a linear data structure: [D]
 A. Queue C. Linked List
 B. Stack D. all the above

2. Which among the following is a dynamic data structure: [A]
 A. Double Linked List C. Stack
 B. Queue D. all the above

3. The link field in a node contains: [A]
 A. address of the next node C. data of next node
 B. data of previous node D. data of current node

4. Memory is allocated dynamically to a data structure during execution by ----- function. [D]
 A. malloc() C. realloc()
 B. Calloc() D. all the above

5. How many null pointer/s exist in a circular double linked list? [D]
 A. 1 C. 3
 B. 2 D. 0

6. Suppose that p is a pointer variable that contains the NULL pointer. What happens if your program tries to read or write *p? []
 A. A syntax error always occurs at compilation time.
 B. A run-time error always occurs when *p is evaluated.
 C. A run-time error always occurs when the program finishes.
 D. The results are unpredictable.

7. What kind of list is best to answer questions such as: "What is the item at position n?" [A]
 A. Lists implemented with an array.
 B. Doubly-linked lists.
 C. Singly-linked lists.
 D. Doubly-linked or singly-linked lists are equally best.

8. In a single linked list which operation depends on the length of the list. [A]
 A. Delete the last element of the list
 B. Add an element before the first element of the list
 C. Delete the first element of the list
 D. Interchange the first two elements of the list

9. A double linked list is declared as follows: [A]

```

      struct dllist
      {
          struct dllist *fwd, *bwd;
          int data;
      }
    
```

Where fwd and bwd represents forward and backward links to adjacent elements of the list. Which among the following segments of code deletes the element pointed to by X from the double linked list, if it is assumed that X points to neither the first nor last element of the list?

- A. `X -> bwd -> fwd = X -> fwd;`
`X -> fwd -> bwd = X -> bwd`
 B. `X -> bwd -> fwd = X -> bwd;`
`X -> fwd -> bwd = X -> fwd`
 C. `X -> bwd -> bwd = X -> fwd;`
`X -> fwd -> fwd = X -> bwd`
 D. `X -> bwd -> bwd = X -> bwd;`
`X -> fwd -> fwd = X -> fwd`
10. Which among the following segment of code deletes the element pointed to by X from the double linked list, if it is assumed that X points to the first element of the list and **start** pointer points to beginning of the list? [B]
- A. `X -> bwd = X -> fwd;`
`X -> fwd = X -> bwd`
 B. `start = X -> fwd;`
`start -> bwd = NULL;`
 C. `start = X -> fwd;`
`X -> fwd = NULL`
 D. `X -> bwd -> bwd = X -> bwd;`
`X -> fwd -> fwd = X -> fwd`
11. Which among the following segment of code deletes the element pointed to by X from the double linked list, if it is assumed that X points to the last element of the list? [C]
- A. `X -> fwd -> bwd = NULL;`
 B. `X -> bwd -> fwd = X -> bwd;`
 C. `X -> bwd -> fwd = NULL;`
 D. `X -> fwd -> bwd = X -> bwd;`
12. Which among the following segment of code counts the number of elements in the double linked list, if it is assumed that X points to the first element of the list and *ctr* is the variable which counts the number of elements in the list? [A]
- A. `for (ctr=1; X != NULL; ctr++)`
`X = X -> fwd;`
 B. `for (ctr=1; X != NULL; ctr++)`
`X = X -> bwd;`
 C. `for (ctr=1; X -> fwd != NULL; ctr++)`
`X = X -> fwd;`
 D. `for (ctr=1; X -> bwd != NULL; ctr++)`
`X = X -> bwd;`
13. Which among the following segment of code counts the number of elements in the double linked list, if it is assumed that X points to the last element of the list and *ctr* is the variable which counts the number of elements in the list? [B]
- A. `for (ctr=1; X != NULL; ctr++)`
`X = X -> fwd;`
 B. `for (ctr=1; X != NULL; ctr++)`
`X = X -> bwd;`
 C. `for (ctr=1; X -> fwd != NULL; ctr++)`
`X = X -> fwd;`
 D. `for (ctr=1; X -> bwd != NULL; ctr++)`
`X = X -> bwd;`

14. Which among the following segment of code inserts a new node pointed by X to be inserted at the beginning of the double linked list. The **start** pointer points to beginning of the list? [B]
- A. X -> bwd = X -> fwd;
X -> fwd = X -> bwd;
 - B. X -> fwd = start;
start -> bwd = X;
start = X;
 - C. X -> bwd = X -> fwd;
X -> fwd = X -> bwd;
start = X;
 - D. X -> bwd -> bwd = X -> bwd;
X -> fwd -> fwd = X -> fwd
15. Which among the following segments of inserts a new node pointed by X to be inserted at the end of the double linked list. The **start** and **last** pointer points to beginning and end of the list respectively? [C]
- A. X -> bwd = X -> fwd;
X -> fwd = X -> bwd
 - B. X -> fwd = start;
start -> bwd = X;
 - C. last -> fwd = X;
X -> bwd = last;
 - D. X -> bwd = X -> bwd;
X -> fwd = last;
16. Which among the following segments of inserts a new node pointed by X to be inserted at any position (i.e neither first nor last) element of the double linked list? Assume **temp** pointer points to the previous position of new node. [D]
- A. X -> bwd -> fwd = X -> fwd;
X -> fwd -> bwd = X -> bwd
 - B. X -> bwd -> fwd = X -> bwd;
X -> fwd -> bwd = X -> fwd
 - C. temp -> fwd = X;
temp -> bwd = X -> fwd;
X -> fwd = x
X -> fwd -> bwd = temp
 - D. X -> bwd = temp;
X -> fwd = temp -> fwd;
temp -> fwd = X;
X -> fwd -> bwd = X;

17. A single linked list is declared as follows:

[A]

```
struct slist
{
    struct slist *next;
    int data;
}
```

Where next represents links to adjacent elements of the list.

Which among the following segments of code deletes the element pointed to by **X** from the single linked list, if it is assumed that X points to neither the first nor last element of the list? **prev** pointer points to previous element.

- A. prev -> next = X -> next;
free(X);
- B. X -> next = prev -> next;
free(X);
- C. prev -> next = X -> next;
free(prev);
- D. X -> next = prev -> next;
free(prev);

18. Which among the following segment of code deletes the element pointed to by X from the single linked list, if it is assumed that X points to the first element of the list and **start** pointer points to beginning of the list?

[B]

- A. X = start -> next;
free(X);
- B. start = X -> next;
free(X);
- C. start = start -> next;
free(start);
- D. X = X -> next;
start = X;
free(start);

19. Which among the following segment of code deletes the element pointed to by X from the single linked list, if it is assumed that X points to the last element of the list and **prev** pointer points to last but one element?

[C]

- A. prev -> next = NULL;
free(prev);
- B. X -> next = NULL;
free(X);
- C. prev -> next = NULL;
free(X);
- D. X -> next = prev;
free(prev);

20. Which among the following segment of code counts the number of elements in the single linked list, if it is assumed that X points to the first element of the list and *ctr* is the variable which counts the number of elements in the list? [A]
- A. for (ctr=1; X != NULL; ctr++)
 X = X -> next;
 - B. for (ctr=1; X != NULL; ctr--)
 X = X -> next;
 - C. for (ctr=1; X -> next != NULL; ctr++)
 X = X -> next;
 - D. for (ctr=1; X -> next != NULL; ctr--)
 X = X -> next;
21. Which among the following segment of code inserts a new node pointed by X to be inserted at the beginning of the single linked list. The **start** pointer points to beginning of the list? [B]
- A. start -> next = X;
 X = start;
 - B. X -> next = start;
 start = X
 - C. X -> next = start -> next;
 start = X
 - D. X -> next = start;
 start = X -> next
22. Which among the following segments of inserts a new node pointed by X to be inserted at the end of the single linked list. The **start** and **last** pointer points to beginning and end of the list respectively? [C]
- A. last -> next = X;
 X -> next = start;
 - B. X -> next = last;
 last -> next = NULL;
 - C. last -> next = X;
 X -> next = NULL;
 - D. last -> next = X -> next;
 X -> next = NULL;
23. Which among the following segments of inserts a new node pointed by X to be inserted at any position (i.e neither first nor last) element of the single linked list? Assume **prev** pointer points to the previous position of new node. [D]
- A. X -> next = prev -> next;
 prev -> next = X -> next;
 - B. X = prev -> next;
 prev -> next = X -> next;
 - C. X -> next = prev;
 prev -> next = X;
 - D. X -> next = prev -> next;
 prev -> next = X;

24. A circular double linked list is declared as follows:

[A]

```
struct cdllist
{
    struct cdllist *fwd, *bwd;
    int data;
}
```

Where fwd and bwd represents forward and backward links to adjacent elements of the list.

Which among the following segments of code deletes the element pointed to by X from the circular double linked list, if it is assumed that X points to neither the first nor last element of the list?

- A. X -> bwd -> fwd = X -> fwd;
X -> fwd -> bwd = X -> bwd;
- B. X -> bwd -> fwd = X -> bwd;
X -> fwd -> bwd = X -> fwd;
- C. X -> bwd -> bwd = X -> fwd;
X -> fwd -> fwd = X -> bwd;
- D. X -> bwd -> bwd = X -> bwd;
X -> fwd -> fwd = X -> fwd;

25. Which among the following segment of code deletes the element pointed to by X from the circular double linked list, if it is assumed that X points to the first element of the list and **start** pointer points to beginning of the list?

[D]

- A. start = start -> bwd;
X -> bwd -> bwd = start;
start -> bwd = X -> bwd;
- B. start = start -> fwd;
X -> fwd -> fwd = start;
start -> bwd = X -> fwd
- C. start = start -> bwd;
X -> bwd -> fwd = X;
start -> bwd = X -> bwd
- D. start = start -> fwd;
X -> bwd -> fwd = start;
start -> bwd = X -> bwd;

26. Which among the following segment of code deletes the element pointed to by X from the circular double linked list, if it is assumed that X points to the last element of the list and **start** pointer points to beginning of the list?

[B]

- A. X -> bwd -> fwd = X -> fwd;
X -> fwd -> fwd = X -> bwd;
- B. X -> bwd -> fwd = X -> fwd;
X -> fwd -> bwd = X -> bwd;
- C. X -> fwd -> fwd = X -> bwd;
X -> fwd -> bwd = X -> fwd;
- D. X -> bwd -> bwd = X -> fwd;
X -> bwd -> bwd = X -> bwd;

27. Which among the following segment of code counts the number of elements in the circular double linked list, if it is assumed that **X** and **start** points to the first element of the list and **ctr** is the variable which counts the number of elements in the list? [A]
- A. for (ctr=1; X->fwd != start; ctr++)
X = X -> fwd;
- B. for (ctr=1; X != NULL; ctr++)
X = X -> bwd;
- C. for (ctr=1; X -> fwd != NULL; ctr++)
X = X -> fwd;
- D. for (ctr=1; X -> bwd != NULL; ctr++)
X = X -> bwd;
28. Which among the following segment of code inserts a new node pointed by **X** to be inserted at the beginning of the circular double linked list. The **start** pointer points to beginning of the list? [B]
- A. X -> bwd = start;
X -> fwd = start -> fwd;
start -> bwd-> fwd = X;
start -> bwd = X;
start = X
- C. X -> fwd = start -> bwd;
X -> bwd = start;
start -> bwd-> fwd = X;
start -> bwd = X;
start = X
- B. X -> bwd = start -> bwd;
X -> fwd = start;
start -> bwd-> fwd = X;
start -> bwd = X;
start = X
- D. X -> bwd = start -> bwd;
X -> fwd = start;
start -> fwd-> fwd = X;
start -> fwd = X;
X = start;
29. Which among the following segment of code inserts a new node pointed by **X** to be inserted at the end of the circular double linked list. The **start** pointer points to beginning of the list? [C]
- A. X -> bwd = start;
X -> fwd = start -> fwd;
start -> bwd -> fwd = X;
start -> bwd = X;
start = X
- C. X -> bwd= start -> bwd;
X-> fwd = start;
start -> bwd -> fwd = X;
start -> bwd = X;
- B. X -> bwd = start -> bwd;
X -> fwd = start;
start -> bwd -> fwd = X;
start -> bwd = X;
start = X
- D. X -> bwd = start -> bwd;
X -> fwd = start;
start -> fwd-> fwd = X;
start -> fwd = X;
X = start;
30. Which among the following segments of inserts a new node pointed by **X** to be inserted at any position (i.e neither first nor last) element of the circular double linked list? Assume **temp** pointer points to the previous position of new node. [D]
- A. X -> bwd -> fwd = X -> fwd;
X -> fwd -> bwd = X -> bwd;
- C. temp -> fwd = X;
temp -> bwd = X -> fwd;
X -> fwd = X;
X -> fwd -> bwd = temp;
- B. X -> bwd -> fwd = X -> bwd;
X -> fwd -> bwd = X -> fwd;
- D. X -> bwd = temp;

```
X -> fwd =  
temp -> fwd;  
temp -> fwd  
= X;  
X -> fwd -> bwd = X;
```

Chapter 4

Stack and Queue

There are certain situations in computer science that one wants to restrict insertions and deletions so that they can take place only at the beginning or the end of the list, not in the middle. Two of such data structures that are useful are:

- *Stack.*
- *Queue.*

Linear lists and arrays allow one to insert and delete elements at any place in the list i.e., at the beginning, at the end or in the middle.

4.1. STACK:

A stack is a list of elements in which an element may be inserted or deleted only at one end, called the top of the stack. Stacks are sometimes known as LIFO (last in, first out) lists.

As the items can be added or removed only from the top i.e. the last item to be added to a stack is the first item to be removed.

The two basic operations associated with stacks are:

- *Push:* is the term used to insert an element into a stack.
- *Pop:* is the term used to delete an element from a stack.

“Push” is the term used to insert an element into a stack. “Pop” is the term used to delete an element from the stack.

All insertions and deletions take place at the same end, so the last element added to the stack will be the first element removed from the stack. When a stack is created, the stack base remains fixed while the stack top changes as elements are added and removed. The most accessible element is the top and the least accessible element is the bottom of the stack.

4.1.1. Representation of Stack:

Let us consider a stack with 6 elements capacity. This is called as the size of the stack. The number of elements to be added should not exceed the maximum size of the stack. If we attempt to add new element beyond the maximum size, we will encounter a *stack overflow* condition. Similarly, you cannot remove elements beyond the base of the stack. If such is the case, we will reach a *stack underflow* condition.

When an element is added to a stack, the operation is performed by `push()`. Figure 4.1 shows the creation of a stack and addition of elements using `push()`.

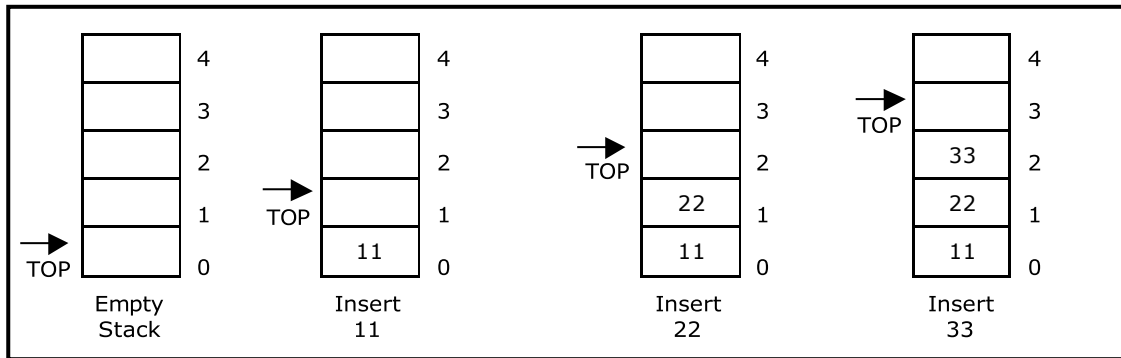


Figure 4.1. Push operations on stack

When an element is taken off from the stack, the operation is performed by pop(). Figure 4.2 shows a stack initially with three elements and shows the deletion of elements using pop().

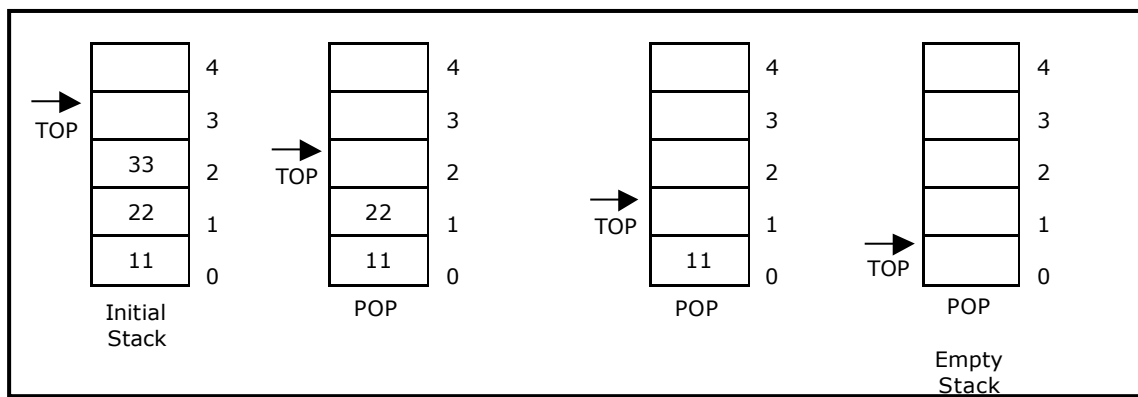


Figure 4.2. Pop operations on stack

4.1.2. Source code for stack operations, using array:

```
# include <stdio.h>
# include <conio.h>
# include <stdlib.h>
# define MAX 6
int stack[MAX];
int top = 0;
int menu()
{
    int ch;
    clrscr();
    printf("\n ... Stack operations using ARRAY... ");
    printf("\n -----***** ----- \n");
    printf("\n 1. Push ");
    printf("\n 2. Pop ");
    printf("\n 3. Display");
    printf("\n 4. Quit ");
    printf("\n Enter your choice: ");
    scanf("%d", &ch);
    return ch;
}
void display()
{
    int i;
    if(top == 0)
    {
        printf("\n\nStack empty..");
    }
}
```

```

        return;
    }
    else
    {
        printf("\n\nElements in stack:");
        for(i = 0; i < top; i++)
            printf("\t%d", stack[i]);
    }
}

void pop()
{
    if(top == 0)
    {
        printf("\n\nStack Underflow..");
        return;
    }
    else
        printf("\n\npopped element is: %d ", stack[--top]);
}

void push()
{
    int data;
    if(top == MAX)
    {
        printf("\n\nStack Overflow..");
        return;
    }
    else
    {
        printf("\n\nEnter data: ");
        scanf("%d", &data);
        stack[top] = data;
        top = top + 1;
        printf("\n\nData Pushed into the stack");
    }
}

void main()
{
    int ch;
    do
    {
        ch = menu();
        switch(ch)
        {
            case 1:
                push();
                break;
            case 2:
                pop();
                break;
            case 3:
                display();
                break;
            case 4:
                exit(0);
        }
    }
    getch();
}

```



```
} while(1);
```

4.1.3. Linked List Implementation of Stack:

We can represent a stack as a linked list. In a stack push and pop operations are performed at one end called top. We can perform similar operations at one end of list using *top* pointer. The linked stack looks as shown in figure 4.3.

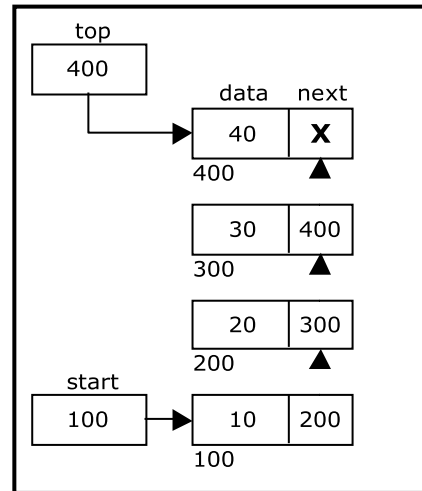


Figure 4.3. Linked stack representation

4.1.4. Source code for stack operations, using linked list:

```
# include <stdio.h>
# include <conio.h>
# include <stdlib.h>

struct stack
{
    int data;
    struct stack *next;
};

void push();
void pop();
void display();
typedef struct stack node;
node *start=NULL;
node *top = NULL;

node* getnode()
{
    node *temp;
    temp=(node *) malloc( sizeof(node)) ;
    printf("\n Enter data ");
    scanf("%d", &temp -> data);
    temp -> next = NULL;
    return temp;
}

void push(node *newnode)
{
    node *temp;
    if( newnode == NULL )
    {
        printf("\n Stack Overflow..");
        return;
    }
}
```

```

        if(start == NULL)
        {
            start = newnode;
            top = newnode;
        }
        else
        {
            temp = start;
            while( temp -> next != NULL)
                temp = temp -> next;
            temp -> next = newnode;
            top = newnode;
        }

        printf("\n\n\t Data pushed into stack");
    }
    void pop()
    {
        node *temp;
        if(top == NULL)
        {
            printf("\n\n\t Stack underflow");
            return;
        }
        temp = start;
        if( start -> next == NULL)
        {
            printf("\n\n\t Popped element is %d ", top -> data);
            start = NULL;
            free(top);
            top = NULL;
        }
        else
        {
            while(temp -> next != top)
            {
                temp = temp -> next;
            }
            temp -> next = NULL;
            printf("\n\n\t Popped element is %d ", top -> data);
            free(top);
            top = temp;
        }
    }
    void display()
    {
        node *temp;
        if(top == NULL)
        {
            printf("\n\n\t\t Stack is empty ");
        }
        else
        {
            temp = start;
            printf("\n\n\n\t\t Elements in the stack: \n");
            printf("%5d ", temp -> data);
            while(temp != top)
            {
                temp = temp -> next;
                printf("%5d ", temp -> data);
            }
        }
    }
}

```

```

char menu()
{
    char ch;
    clrscr();
    printf("\n \tStack operations using pointers.. ");
    printf("\n -----***** ----- \n");
    printf("\n 1. Push ");
    printf("\n 2. Pop ");
    printf("\n 3. Display");
    printf("\n 4. Quit ");
    printf("\n Enter your choice: ");
    ch = getche();
    return ch;
}

void main()
{
    char ch;
    node *newnode;
    do
    {
        ch = menu();
        switch(ch)
        {
            case '1' :
                newnode = getnode();
                push(newnode);
                break;
            case '2' :
                pop();
                break;
            case '3' :
                display();
                break;
            case '4':
                return;
        }
        getch();
    } while( ch != '4' );
}

```

4.2. Algebraic Expressions:

An algebraic expression is a legal combination of operators and operands. Operand is the quantity on which a mathematical operation is performed. Operand may be a variable like x, y, z or a constant like 5, 4, 6 etc. Operator is a symbol which signifies a mathematical or logical operation between the operands. Examples of familiar operators include +, -, *, /, ^ etc.

An algebraic expression can be represented using three different notations. They are infix, postfix and prefix notations:

Infix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator in between the two operands.

Example: $(A + B) * (C - D)$

Prefix: It is the form of an arithmetic notation in which we fix (place) the arithmetic operator before (pre) its two operands. The prefix notation is called as

polish notation (due to the polish mathematician Jan Lukasiewicz in the year 1920).

Example: $* + A B - C D$

Postfix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator after (post) its two operands. The postfix notation is called as *suffix notation* and is also referred to *reverse polish notation*.

Example: $A B + C D - *$

The three important features of postfix expression are:

1. The operands maintain the same order as in the equivalent infix expression.
2. The parentheses are not needed to designate the expression unambiguously.
3. While evaluating the postfix expression the priority of the operators is no longer relevant.

We consider five binary operations: $+$, $-$, $*$, $/$ and $\$$ or \uparrow (exponentiation). For these binary operations, the following in the order of precedence (highest to lowest):

OPERATOR	PRECEDENCE	VALUE
Exponentiation ($\$$ or \uparrow or \wedge)	Highest	3
$*$, $/$	Next highest	2
$+$, $-$	Lowest	1

4.3. Converting expressions using Stack:

Let us convert the expressions from one type to another. These can be done as follows:

1. Infix to postfix
2. Infix to prefix
3. Postfix to infix
4. Postfix to prefix
5. Prefix to infix
6. Prefix to postfix

4.3.1. Conversion from infix to postfix:

Procedure to convert from infix expression to postfix expression is as follows:

1. Scan the infix expression from left to right.
2. a) If the scanned symbol is left parenthesis, push it onto the stack.
b) If the scanned symbol is an operand, then place directly in the postfix expression (output).

- c) If the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.
- d) If the scanned symbol is an operator, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (*or greater than or equal*) to the precedence of the scanned operator and push the scanned operator onto the stack otherwise, push the scanned operator onto the stack.

Example 1:

Convert $((A - (B + C)) * D) \uparrow (E + F)$ infix expression to postfix form:

SYMBOL	POSTFIX STRING	STACK	REMARKS
((
(((
A	A	((
-	A	((-	
(A	((- (
B	A B	((- (
+	A B	((- (+	
C	A B C	((- (+	
)	A B C +	((-	
)	A B C + -	(
*	A B C + -	(*	
D	A B C + - D	(*	
)	A B C + - D *		
↑	A B C + - D *	↑	
(A B C + - D *	↑ (
E	A B C + - D * E	↑ (
+	A B C + - D * E	↑ (+	
F	A B C + - D * E F	↑ (+	
)	A B C + - D * E F +	↑	
End of string	A B C + - D * E F + ↑	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 2:

Convert $a + b * c + (d * e + f) * g$ the infix expression into postfix form.

SYMBOL	POSTFIX STRING	STACK	REMARKS
a	a		
+	a	+	
b	a b	+	

*	a b	+ *	
c	a b c	+ *	
+	a b c * +	+	
(a b c * +	+ (
d	a b c * + d	+ (
*	a b c * + d	+ (*	
e	a b c * + d e	+ (*	
+	a b c * + d e *	+ (+	
f	a b c * + d e * f	+ (+	
)	a b c * + d e * f +	+	
*	a b c * + d e * f +	+ *	
g	a b c * + d e * f + g	+ *	
End of string	a b c * + d e * f + g * +	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 3:

Convert the following infix expression $A + B * C - D / E * H$ into its equivalent postfix expression.

SYMBOL	POSTFIX STRING	STACK	REMARKS
A	A		
+	A	+	
B	A B	+	
*	A B	+ *	
C	A B C	+ *	
-	A B C * +	-	
D	A B C * + D	-	
/	A B C * + D	- /	
E	A B C * + D E	- /	
*	A B C * + D E /	- *	
H	A B C * + D E / H	- *	
End of string	A B C * + D E / H * -	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 4:

Convert the following infix expression $A + (B * C - (D / E \uparrow F) * G) * H$ into its equivalent postfix expression.

SYMBOL	POSTFIX STRING	STACK	REMARKS
A	A		
+	A	+	

(A	+ (
B	A B	+ (
*	A B	+ (*	
C	A B C	+ (*	
-	A B C *	+ (-	
(A B C *	+ (- (
D	A B C * D	+ (- (
/	A B C * D	+ (- (/	
E	A B C * D E	+ (- (/	
↑	A B C * D E	+ (- (/ ↑	
F	A B C * D E F	+ (- (/ ↑	
)	A B C * D E F ↑ /	+ (-	
*	A B C * D E F ↑ /	+ (- *	
G	A B C * D E F ↑ / G	+ (- *	
)	A B C * D E F ↑ / G * -	+	
*	A B C * D E F ↑ / G * -	+ *	
H	A B C * D E F ↑ / G * - H	+ *	
End of string	A B C * D E F ↑ / G * - H * +	The input is now empty. Pop the output symbols from the stack until it is empty.	

4.3.2. Program to convert an infix to postfix expression:

```
# include <string.h>
```

```
char postfix[50];
```

```
char infix[50];
```

```
char opstack[50];          /* operator stack */
```

```
int i, j, top = 0;
```

```
int lesspriority(char op, char op_at_stack)
```

```
{
```

```
    int k;
```

```
    int pv1;          /* priority value of op */
```

```
    int pv2;          /* priority value of op_at_stack */
```

```
    char operators[] = {'+', '-', '*', '/', '%', '^', '(' };
```

```
    int priority_value[] = {0,0,1,1,2,3,4};
```

```
    if( op_at_stack == '(' )
```

```
        return 0;
```

```
    for(k = 0; k < 6; k ++)
```

```
    {
```

```
        if(op == operators[k])
```

```
            pv1 = priority_value[k];
```

```
    }
```

```
    for(k = 0; k < 6; k ++)
```

```
    {
```

```
        if(op_at_stack == operators[k])
```

```
            pv2 = priority_value[k];
```

```
    }
```

```
    if(pv1 < pv2)
```

```
        return 1;
```

```
    else
```

```
        return 0;
```

```
}
```



```
void push(char op)      /* op - operator */
{
```

```
    if(top == 0)
```

```
    {
```

```
        opstack[top] = op;
```

```
        top++;
```

```
    }
```

```
    else
```

```
    {
```

```
        if(op != '(' )
```

```
        {
```

```
            while(lesspriority(op, opstack[top-1]) == 1 && top > 0)
```

```
            {
```

```
                postfix[j] = opstack[--top];
```

```
                j++;
```

```
            }
```

```
        }
```

```
        opstack[top] = op;      /* pushing onto stack */
```

```
        top++;
```

```
    }
```

```
}
```

```
pop()
```

```
{
```

```
    while(opstack[--top] != '(')      /* pop until '(' comes */
```

```
    {
```

```
        postfix[j] = opstack[top];
```

```
        j++;
```

```
    }
```

```
}
```

```
void main()
```

```
{
```

```
    char ch;
```

```
    clrscr();
```

```
    printf("\n Enter Infix Expression : ");
```

```
    gets(infix);
```

```
    while( (ch=infix[i++]) != '\0')
```

```
    {
```

```
        switch(ch)
```

```
        {
```

```
            case ' ' : break;
```

```
            case '(' :
```

```
            case '+' :
```

```
            case '-' :
```

```
            case '*' :
```

```
            case '/' :
```

```
            case '^' :
```

```
            case '%' :
```

```
                push(ch);
```

```
                break;
```

```
            case ')' :
```

```
                pop();
```

```
                break;
```

```
            default :
```

```
                postfix[j] = ch;
```

```
                j++;
```

```
        }
```

```
    }
```

```
    while(top >= 0)
```

```
    {
```

```
        postfix[j] = opstack[--top];
```

```
        j++;
```

/* before pushing the operator 'op' into the stack check priority of op with top of opstack if less then pop the operator from stack then push into postfix string else push op onto stack itself */

/* check priority and push */

```

    }
    postfix[j] = '\0';
    printf("\n Infix Expression   : %s ", infix);
    printf("\n Postfix Expression : %s ", postfix);
    getch();
}

```

4.3.3. Conversion from infix to prefix:

The precedence rules for converting an expression from infix to prefix are identical. The only change from postfix conversion is that traverse the expression from right to left and the operator is placed before the operands rather than after them. The prefix form of a complex expression is not the mirror image of the postfix form.

Example 1:

Convert the infix expression $A + B - C$ into prefix expression.

SYMBOL	PREFIX STRING	STACK	REMARKS
C	C		
-	C	-	
B	B C	-	
+	B C	- +	
A	A B C	- +	
End of string	- + A B C	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 2:

Convert the infix expression $(A + B) * (C - D)$ into prefix expression.

SYMBOL	PREFIX STRING	STACK	REMARKS
))	
D	D)	
-	D) -	
C	C D) -	
(- C D		
*	- C D	*	
)	- C D	*)	
B	B - C D	*)	
+	B - C D	*) +	
A	A B - C D	*) +	
(+ A B - C D	*	
End of string	* + A B - C D	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 3:

Convert the infix expression $A \uparrow B * C - D + E / F / (G + H)$ into prefix expression.

SYMBOL	PREFIX STRING	STACK	REMARKS
))	
H	H)	
+	H) +	
G	G H) +	
(+ G H		
/	+ G H	/	
F	F + G H	/	
/	F + G H	//	
E	E F + G H	//	
+	// E F + G H	+	
D	D // E F + G H	+	
-	D // E F + G H	+ -	
C	C D // E F + G H	+ -	
*	C D // E F + G H	+ - *	
B	B C D // E F + G H	+ - *	
\uparrow	B C D // E F + G H	+ - * \uparrow	
A	A B C D // E F + G H	+ - * \uparrow	
End of string	+ - * \uparrow A B C D // E F + G H	The input is now empty. Pop the output symbols from the stack until it is empty.	

4.3.4. Program to convert an infix to prefix expression:

```
# include <conio.h>
# include <string.h>

char prefix[50];
char infix[50];
char opstack[50];          /* operator stack */
int j, top = 0;

void insert_beg(char ch)
{
    int k;
    if(j == 0)
        prefix[0] = ch;
    else
    {
        for(k = j + 1; k > 0; k--)
            prefix[k] = prefix[k - 1];
        prefix[0] = ch;
    }
    j++;
}
```

```

int lesspriority(char op, char op_at_stack)
{
    int k;
    int pv1;          /* priority value of op */
    int pv2;          /* priority value of op_at_stack */
    char operators[] = {'+', '-', '*', '/', '%', '^', ' '};
    int priority_value[] = {0, 0, 1, 1, 2, 3, 4};
    if(op_at_stack == ' ')
        return 0;
    for(k = 0; k < 6; k++)
    {
        if(op == operators[k])
            pv1 = priority_value[k];
    }
    for(k = 0; k < 6; k++)
    {
        if( op_at_stack == operators[k] )
            pv2 = priority_value[k];
    }
    if(pv1 < pv2)
        return 1;
    else
        return 0;
}

void push(char op)          /* op - operator */
{
    if(top == 0)
    {
        opstack[top] = op;
        top++;
    }
    else
    {
        if(op != ' ')
        {
            /* before pushing the operator 'op' into the stack check priority of op with
            top of operator stack if less pop the operator from stack then push into postfix
            string else push op onto stack itself */

            while(lesspriority(op, opstack[top-1]) == 1 && top > 0)
            {
                insert_beg(opstack[--top]);
            }
            opstack[top] = op;          /* pushing onto stack */
            top++;
        }
    }
}

void pop()
{
    while(opstack[--top] != ' ')          /* pop until ')' comes; */
        insert_beg(opstack[top]);
}

void main()
{
    char ch;
    int l, i = 0;
    clrscr();
    printf("\n Enter Infix Expression : ");

```

```

gets(infix);
l = strlen(infix);
while(l > 0)
{
    ch = infix[--l];
    switch(ch)
    {
        case ' ' : break;
        case ')' :
        case '+' :
        case '-' :
        case '*' :
        case '/' :
        case '^' :
        case '%':
            push(ch);          /* check priority and push */
            break;
        case '(' :
            pop();
            break;
        default :
            insert_beg(ch);
    }
}
while( top > 0 )
{
    insert_beg( opstack[--top] );
    j++;
}
prefix[j] = '\0';
printf("\n Infix Expression   : %s ", infix);
printf("\n Prefix Expression : %s ", prefix);
getch();
}

```

4.3.5. Conversion from postfix to infix:

Procedure to convert postfix expression to infix expression is as follows:

1. Scan the postfix expression from left to right.
2. If the scanned symbol is an operand, then push it onto the stack.
3. If the scanned symbol is an operator, pop two symbols from the stack and create it as a string by placing the operator in between the operands and push it onto the stack.
4. Repeat steps 2 and 3 till the end of the expression.

Example:

Convert the following postfix expression A B C * D E F ^ / G * - H * + into its equivalent infix expression.

Symbol	Stack	Remarks
A	A	Push A
B	A B	Push B
C	A B C	Push C
*	A (B*C)	Pop two operands and place the operator in between the operands and push the string.
D	A (B*C) D	Push D
E	A (B*C) D E	Push E
F	A (B*C) D E F	Push F
^	A (B*C) D (E^F)	Pop two operands and place the operator in between the operands and push the string.
/	A (B*C) (D/(E^F))	Pop two operands and place the operator in between the operands and push the string.
G	A (B*C) (D/(E^F)) G	Push G
*	A (B*C) ((D/(E^F))*G)	Pop two operands and place the operator in between the operands and push the string.
-	A ((B*C) - ((D/(E^F))*G))	Pop two operands and place the operator in between the operands and push the string.
H	A ((B*C) - ((D/(E^F))*G)) H	Push H
*	A (((B*C) - ((D/(E^F))*G)) * H)	Pop two operands and place the operator in between the operands and push the string.
+	(A + (((B*C) - ((D/(E^F))*G)) * H))	
End of string	The input is now empty. The string formed is infix.	

4.3.6. Program to convert postfix to infix expression:

```
# include <stdio.h>
# include <conio.h>
# include <string.h>
# define MAX 100
```

```
void pop (char*);
void push(char*);
```

```
char stack[MAX] [MAX];
int top = -1;
```

```

void main()
{
    char s[MAX], str1[MAX], str2[MAX], str[MAX];
    char s1[2],temp[2];
    int i=0;
    clrscr( ) ;
    printf("\nEnter the postfix expression; ");
    gets(s);
    while (s[i]!='\0')
    {
        if(s[i] == ' ' )                /*skip whitespace, if any*/
            i++;
        if (s[i] == '^' || s[i] == '*' || s[i] == '-' || s[i] == '+' || s[i] == '/')
        {
            pop(str1);
            pop(str2);
            temp[0] = '(';
            temp[1] = '\0';
            strcpy(str, temp);
            strcat(str, str2);
            temp[0] = s[i];
            temp[1] = '\0';
            strcat(str,temp);
            strcat(str, str1);
            temp[0] = ')';
            temp[1] = '\0';
            strcat(str,temp);
            push(str);
        }
        else
        {
            temp[0]=s[i];
            temp[1]='\0';
            strcpy(s1, temp);
            push(s1);
        }
        i++;
    }

    printf("\nThe Infix expression is: %s", stack[0]);

}

void pop(char *a1)
{
    strcpy(a1,stack[top]);
    top--;
}

void push (char*str)
{
    if(top == MAX - 1)
        printf("\nstack is full");
    else
    {
        top++;
        strcpy(stack[top], str);
    }
}

```

4.3.7. Conversion from postfix to prefix:

Procedure to convert postfix expression to prefix expression is as follows:

1. Scan the postfix expression from left to right.
2. If the scanned symbol is an operand, then push it onto the stack.
3. If the scanned symbol is an operator, pop two symbols from the stack and create it as a string by placing the operator in front of the operands and push it onto the stack.
5. Repeat steps 2 and 3 till the end of the expression.

Example:

Convert the following postfix expression $A B C * D E F ^ / G * - H * +$ into its equivalent prefix expression.

Symbol	Stack	Remarks
A	A	Push A
B	A B	Push B
C	A B C	Push C
*	A *BC	Pop two operands and place the operator in front the operands and push the string.
D	A *BC D	Push D
E	A *BC D E	Push E
F	A *BC D E F	Push F
^	A *BC D ^EF	Pop two operands and place the operator in front the operands and push the string.
/	A *BC /D^EF	Pop two operands and place the operator in front the operands and push the string.
G	A *BC /D^EF G	Push G
*	A *BC */D^EFG	Pop two operands and place the operator in front the operands and push the string.
-	A - *BC*/D^EFG	Pop two operands and place the operator in front the operands and push the string.
H	A - *BC*/D^EFG H	Push H
*	A *- *BC*/D^EFGH	Pop two operands and place the operator in front the operands and push the string.
+	+A*- *BC*/D^EFGH	
End of string	The input is now empty. The string formed is prefix.	

4.3.8. Program to convert postfix to prefix expression:

```
# include <conio.h>
# include <string.h>

#define MAX 100
void pop (char *a1);
void push(char *str);
char stack[MAX][MAX];
int top = -1;

main()
{
    char s[MAX], str1[MAX], str2[MAX], str[MAX];
    char s1[2], temp[2];
    int i = 0;
    clrscr();
    printf("Enter the postfix expression; ");
    gets (s);
    while(s[i]!='\0')
    {
        /*skip whitespace, if any */
        if (s[i] == ' ')
            i++;
        if(s[i] == '^' || s[i] == '*' || s[i] == '-' || s[i] == '+' || s[i] == '/')
        {
            pop (str1);
            pop (str2);
            temp[0] = s[i];
            temp[1] = '\0';
            strcpy (str, temp);
            strcat(str, str2);
            strcat(str, str1);
            push(str);
        }
        else
        {
            temp[0] = s[i];
            temp[1] = '\0';
            strcpy (s1, temp);
            push (s1);
        }
        i++;
    }

    printf("\n The prefix expression is: %s", stack[0]);
}

void pop(char*a1)
{
    if(top == -1)
    {
        printf("\nStack is empty");
        return ;
    }
    else
    {
        strcpy (a1, stack[top]);
        top--;
    }
}
```

```

void push (char *str)
{
    if(top == MAX - 1)
        printf("\nstack is full");
    else
    {
        top++;
        strcpy(stack[top], str);
    }
}

```

4.3.9. Conversion from prefix to infix:

Procedure to convert prefix expression to infix expression is as follows:

1. Scan the prefix expression from right to left (reverse order).
2. If the scanned symbol is an operand, then push it onto the stack.
3. If the scanned symbol is an operator, pop two symbols from the stack and create it as a string by placing the operator in between the operands and push it onto the stack.
4. Repeat steps 2 and 3 till the end of the expression.

Example:

Convert the following prefix expression $+ A * - * B C * / D ^ E F G H$ into its equivalent infix expression.

Symbol	Stack	Remarks
H	H	Push H
G	H G	Push G
F	H G F	Push F
E	H G F E	Push E
^	H G (E^F)	Pop two operands and place the operator in between the operands and push the string.
D	H G (E^F) D	Push D
/	H G (D/(E^F))	Pop two operands and place the operator in between the operands and push the string.
*	H ((D/(E^F))*G)	Pop two operands and place the operator in between the operands and push the string.
C	H ((D/(E^F))*G) C	Push C
B	H ((D/(E^F))*G) C B	Push B
*	H ((D/(E^F))*G) (B*C)	Pop two operands and place the operator in front the operands and push the string.
-	H ((B*C)-((D/(E^F))*G))	Pop two operands and place the operator in front the operands and push the

				string.
*	(((B*C)-((D/(E^F))*G))*H)			Pop two operands and place the operator in front the operands and push the string.
A	(((B*C)-((D/(E^F))*G))*H)	A		Push A
+	(A+(((B*C)-((D/(E^F))*G))*H))			Pop two operands and place the operator in front the operands and push the string.
End of string	The input is now empty. The string formed is infix.			

4.3.10. Program to convert prefix to infix expression:

```
# include <string.h>
# define MAX 100

void pop (char*);
void push(char*);
char stack[MAX] [MAX];
int top = -1;

void main()
{
    char s[MAX], str1[MAX], str2[MAX], str[MAX];
    char s1[2],temp[2];
    int i=0;
    clrscr( ) ;
    printf("\nEnter the prefix expression; ");
    gets(s);
    strrev(s);
    while (s[i]!='\0')
    {
        /*skip whitespace, if any*/
        if(s[i] == ' ' )
            i++;
        if (s[i] == '^' || s[i] == '*' || s[i] == '-' || s[i] == '+' || s[i] == '/')
        {
            pop(str1);
            pop(str2);
            temp[0] = '(';
            temp[1] = '\0';
            strcpy(str, temp);
            strcat(str, str1);
            temp[0] = s[i];
            temp[1] = '\0';
            strcat(str,temp);
            strcat(str, str2);
            temp[0] = ')';
            temp[1] = '\0';
            strcat(str,temp);
            push(str);
        }
        else
        {
            temp[0]=s[i];
            temp[1]='\0';
            strcpy(s1, temp);
            push(s1);
        }
    }
}
```

```

        }
        i++;
    }
    printf("\nThe infix expression is: %s", stack[0]);
}

void pop(char *a1)
{
    strcpy(a1, stack[top]);
    top--;
}

void push (char*str)
{
    if(top == MAX - 1)
        printf("\nstack is full");
    else
    {
        top++;
        strcpy(stack[top], str);
    }
}

```

4.3.11. Conversion from prefix to postfix:

Procedure to convert prefix expression to postfix expression is as follows:

1. Scan the prefix expression from right to left (reverse order).
2. If the scanned symbol is an operand, then push it onto the stack.
3. If the scanned symbol is an operator, pop two symbols from the stack and create it as a string by placing the operator after the operands and push it onto the stack.
4. Repeat steps 2 and 3 till the end of the expression.

Example:

Convert the following prefix expression + A * - * B C * / D ^ E F G H into its equivalent postfix expression.

Symbol	Stack	Remarks					
H	<table><tr><td>H</td><td></td><td></td><td></td><td></td></tr></table>	H					Push H
H							
G	<table><tr><td>H</td><td>G</td><td></td><td></td><td></td></tr></table>	H	G				Push G
H	G						
F	<table><tr><td>H</td><td>G</td><td>F</td><td></td><td></td></tr></table>	H	G	F			Push F
H	G	F					
E	<table><tr><td>H</td><td>G</td><td>F</td><td>E</td><td></td></tr></table>	H	G	F	E		Push E
H	G	F	E				
^	<table><tr><td>H</td><td>G</td><td>EF^</td><td></td><td></td></tr></table>	H	G	EF^			Pop two operands and place the operator after the operands and push the string.
H	G	EF^					
D	<table><tr><td>H</td><td>G</td><td>EF^</td><td>D</td><td></td></tr></table>	H	G	EF^	D		Push D
H	G	EF^	D				

/	<table><tr><td>H</td><td>G</td><td>DEF^/</td><td></td></tr></table>	H	G	DEF^/		Pop two operands and place the operator after the operands and push the string.	
H	G	DEF^/					
*	<table><tr><td>H</td><td>DEF^/G*</td><td></td></tr></table>	H	DEF^/G*		Pop two operands and place the operator after the operands and push the string.		
H	DEF^/G*						
C	<table><tr><td>H</td><td>DEF^/G*</td><td>C</td><td></td></tr></table>	H	DEF^/G*	C		Push C	
H	DEF^/G*	C					
B	<table><tr><td>H</td><td>DEF^/G*</td><td>C</td><td>B</td><td></td></tr></table>	H	DEF^/G*	C	B		Push B
H	DEF^/G*	C	B				
*	<table><tr><td>H</td><td>DEF^/G*</td><td>BC*</td><td></td></tr></table>	H	DEF^/G*	BC*		Pop two operands and place the operator after the operands and push the string.	
H	DEF^/G*	BC*					
-	<table><tr><td>H</td><td>BC*DEF^/G*-</td><td></td></tr></table>	H	BC*DEF^/G*-		Pop two operands and place the operator after the operands and push the string.		
H	BC*DEF^/G*-						
*	<table><tr><td>BC*DEF^/G*-H*</td><td></td></tr></table>	BC*DEF^/G*-H*		Pop two operands and place the operator after the operands and push the string.			
BC*DEF^/G*-H*							
A	<table><tr><td>BC*DEF^/G*-H*</td><td>A</td><td></td></tr></table>	BC*DEF^/G*-H*	A		Push A		
BC*DEF^/G*-H*	A						
+	<table><tr><td>ABC*DEF^/G*-H*+</td><td></td></tr></table>	ABC*DEF^/G*-H*+		Pop two operands and place the operator after the operands and push the string.			
ABC*DEF^/G*-H*+							
End of string	The input is now empty. The string formed is postfix.						

4.3.12. Program to convert prefix to postfix expression:

```
# include <stdio.h>
# include <conio.h>
# include <string.h>

#define MAX 100

void pop (char *a1);
void push(char *str);
char stack[MAX][MAX];
int top = -1;

void main()
{
    char s[MAX], str1[MAX], str2[MAX], str[MAX];
    char s1[2], temp[2];
    int i = 0;
    clrscr();
    printf("Enter the prefix expression; ");
    gets (s);
    strrev(s);
    while(s[i]!='\0')
    {
        if (s[i] == ' ') /*skip whitespace, if any */
            i++;
        if(s[i] == '^' || s[i] == '*' || s[i] == '-' || s[i] == '+' || s[i] == '/')
        {
            pop (str1);
            pop (str2);
            temp[0] = s[i];
            temp[1] = '\0';
            strcat(str1,str2);
            strcat (str1, temp);
            strcpy(str, str1);
            push(str);
        }
    }
}
```

```

        else
        {
            temp[0] = s[i];
            temp[1] = '\0';
            strcpy (s1, temp);
            push (s1);
        }
        i++;
    }

    printf("\nThe postfix expression is: %s", stack[0]);
}
void pop(char*a1)
{
    if(top == -1)
    {
        printf("\nStack is empty");
        return ;
    }
    else
    {
        strcpy (a1, stack[top]);
        top--;
    }
}

void push (char *str)
{
    if(top == MAX - 1)
        printf("\nstack is full");
    else
    {
        top++;
        strcpy(stack[top], str);
    }
}

```

4.4. Evaluation of postfix expression:

The postfix expression is evaluated easily by the use of a stack. When a number is seen, it is pushed onto the stack; when an operator is seen, the operator is applied to the two numbers that are popped from the stack and the result is pushed onto the stack. When an expression is given in postfix notation, there is no need to know any precedence rules; this is our obvious advantage.

Example 1:

Evaluate the postfix expression: 6 5 2 3 + 8 * + 3 + *

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK	REMARKS
6				6	
5				6, 5	
2				6, 5, 2	
3				6, 5, 2, 3	The first four symbols are placed on the stack.
+	2	3	5	6, 5, 5	Next a '+' is read, so 3 and 2 are popped from the stack and their sum 5, is pushed

8	2	3	5	6, 5, 5, 8	Next 8 is pushed
*	5	8	40	6, 5, 40	Now a '*' is seen, so 8 and 5 are popped as $8 * 5 = 40$ is pushed
+	5	40	45	6, 45	Next, a '+' is seen, so 40 and 5 are popped and $40 + 5 = 45$ is pushed
3	5	40	45	6, 45, 3	Now, 3 is pushed
+	45	3	48	6, 48	Next, '+' pops 3 and 45 and pushes $45 + 3 = 48$ is pushed
*	6	48	288	288	Finally, a '*' is seen and 48 and 6 are popped, the result $6 * 48 = 288$ is pushed

Example 2:

Evaluate the following postfix expression: $6\ 2\ 3\ +\ -\ 3\ 8\ 2\ /\ +\ * \ 2\ \uparrow\ 3\ +$

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3	6	5	1	1, 3
8	6	5	1	1, 3, 8
2	6	5	1	1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2	1	7	7	7, 2
↑	7	2	49	49
3	7	2	49	49, 3
+	49	3	52	52

4.4.1. Program to evaluate a postfix expression:

```
# include <conio.h>
# include <math.h>
# define MAX 20

int isoperator(char ch)
{
    if(ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '^')
        return 1;
    else
        return 0;
}
```

```

void main(void)
{
    char postfix[MAX];
    int val;
    char ch;
    int i = 0, top = 0;
    float val_stack[MAX], val1, val2, res;
    clrscr();
    printf("\n Enter a postfix expression: ");
    scanf("%s", postfix);
    while((ch = postfix[i]) != '\0')
    {
        if(isoperator(ch) == 1)
        {
            val2 = val_stack[--top];
            val1 = val_stack[--top];
            switch(ch)
            {
                case '+':
                    res = val1 + val2;
                    break;
                case '-':
                    res = val1 - val2;
                    break;
                case '*':
                    res = val1 * val2;
                    break;
                case '/':
                    res = val1 / val2;
                    break;
                case '^':
                    res = pow(val1, val2);
                    break;
            }
            val_stack[top] = res;
        }
        else
            val_stack[top] = ch-48; /*convert character digit to integer digit */
        top++;
        i++;
    }

    printf("\n Values of %s is : %f ",postfix, val_stack[0] );
    getch();
}

```

4.5. Applications of stacks:

1. Stack is used by compilers to check for balancing of parentheses, brackets and braces.
2. Stack is used to evaluate a postfix expression.
3. Stack is used to convert an infix expression into postfix/prefix form.
4. In recursion, all intermediate arguments and return values are stored on the processor's stack.
5. During a function call the return address and arguments are pushed onto a stack and on return they are popped off.

4.6. Queue:

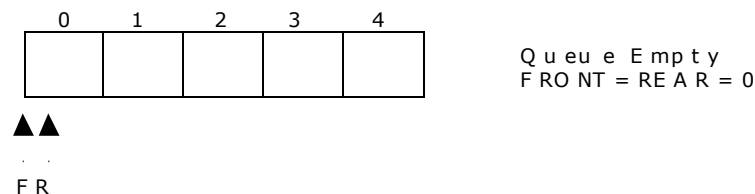
A queue is another special kind of list, where items are inserted at one end called the rear and deleted at the other end called the front. Another name for a queue is a "FIFO" or "First-in-first-out" list.

The operations for a queue are analogues to those for a stack, the difference is that the insertions go at the end of the list, rather than the beginning. We shall use the following operations on queues:

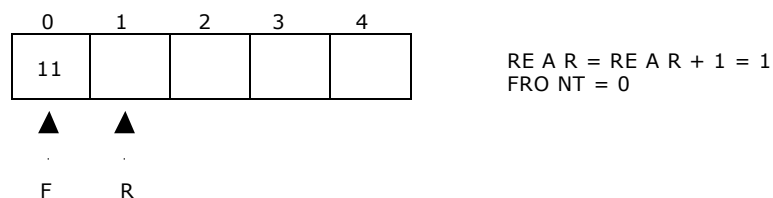
- *enqueue*: which inserts an element at the end of the queue.
- *dequeue*: which deletes an element at the start of the queue.

4.6.1. Representation of Queue:

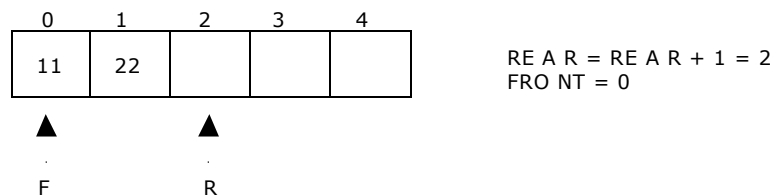
Let us consider a queue, which can hold maximum of five elements. Initially the queue is empty.



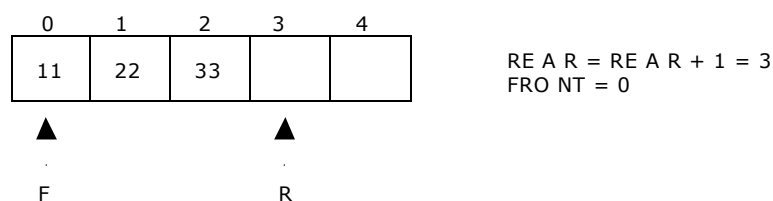
Now, insert 11 to the queue. Then queue status will be:



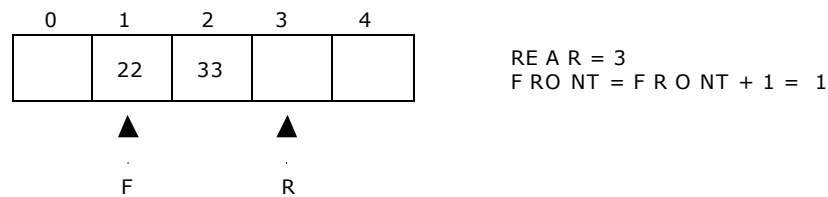
Next, insert 22 to the queue. Then the queue status is:



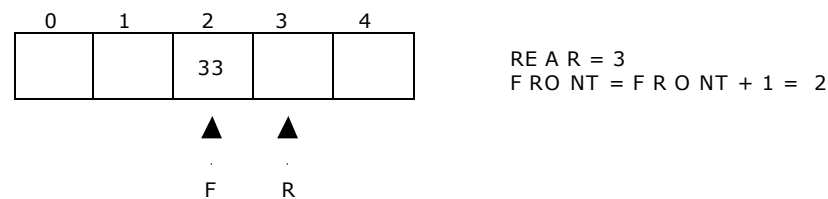
Again insert another element 33 to the queue. The status of the queue is:



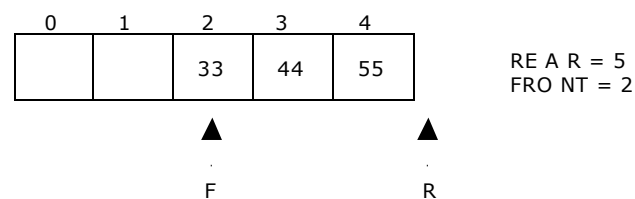
Now, delete an element. The element deleted is the element at the front of the queue. So the status of the queue is:



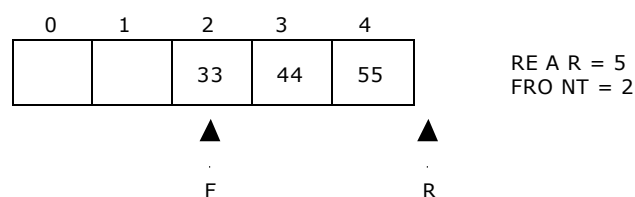
Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The queue status is as follows:



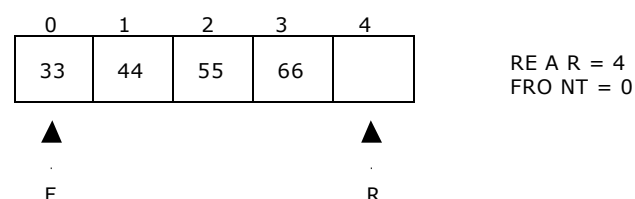
Now, insert new elements 44 and 55 into the queue. The queue status is:



Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:



Now it is not possible to insert an element 66 even though there are two vacant positions in the linear queue. To overcome this problem the elements of the queue are to be shifted towards the beginning of the queue so that it creates vacant position at the rear end. Then the FRONT and REAR are to be adjusted properly. The element 66 can be inserted at the rear end. After this operation, the queue status is as follows:



This difficulty can overcome if we treat queue position with index 0 as a position that comes after position with index 4 i.e., we treat the queue as a **circular queue**.

4.6.2. Source code for Queue operations using array:

In order to create a queue we require a one dimensional array $Q(1:n)$ and two variables *front* and *rear*. The conventions we shall adopt for these two variables are that *front* is always 1 less than the actual front of the queue and *rear* always points to the last element in the queue. Thus, $front = rear$ if and only if there are no elements in the queue. The initial condition then is $front = rear = 0$. The various queue operations to perform creation, deletion and display the elements in a queue are as follows:

1. `insertQ()`: inserts an element at the end of queue Q .
2. `deleteQ()`: deletes the first element of Q .
3. `displayQ()`: displays the elements in the queue.

```
# include <conio.h>
# define MAX 6
int Q[MAX];
int front, rear;

void insertQ()
{
    int data;
    if(rear == MAX)
    {
        printf("\n Linear Queue is full");
        return;
    }
    else
    {
        printf("\n Enter data: ");
        scanf("%d", &data);
        Q[rear] = data;
        rear++;
        printf("\n Data Inserted in the Queue ");
    }
}

void deleteQ()
{
    if(rear == front)
    {
        printf("\n\n Queue is Empty..");
        return;
    }
    else
    {
        printf("\n Deleted element from Queue is %d", Q[front]);
        front++;
    }
}

void displayQ()
{
    int i;
    if(front == rear)
    {
        printf("\n\n\t Queue is Empty");
        return;
    }
    else
    {
        printf("\n Elements in Queue are: ");
        for(i = front; i < rear; i++)
```

```

        {
            printf("%d\t", Q[i]);
        }
    }
}
int menu()
{
    int ch;
    clrscr();
    printf("\n \tQueue operations using ARRAY..");
    printf("\n -----*****----- \n");
    printf("\n 1. Insert ");
    printf("\n 2. Delete ");
    printf("\n 3. Display");
    printf("\n 4. Quit ");
    printf("\n Enter your choice: ");
    scanf("%d", &ch);
    return ch;
}
void main()
{
    int ch;
    do
    {
        ch = menu();
        switch(ch)
        {
            case 1:
                insertQ();
                break;
            case 2:
                deleteQ();
                break;
            case 3:
                displayQ();
                break;
            case 4:
                return;
        }
        getch();
    } while(1);
}

```

4.6.3. Linked List Implementation of Queue:

We can represent a queue as a linked list. In a queue data is deleted from the front end and inserted at the rear end. We can perform similar operations on the two ends of a list. We use two pointers *front* and *rear* for our linked queue implementation.

The linked queue looks as shown in figure 4.4:

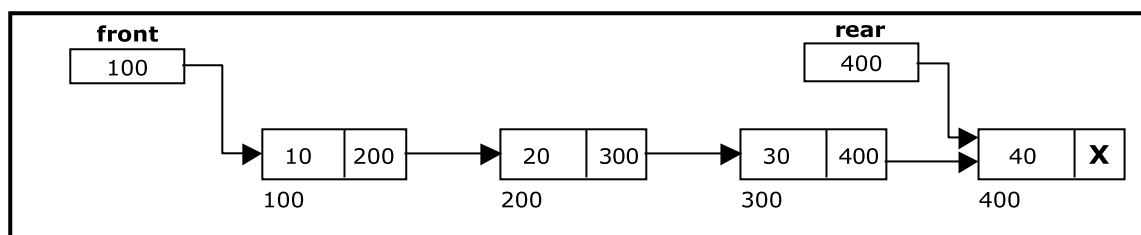


Figure 4.4. Linked Queue representation

4.6.4. Source code for queue operations using linked list:

```
# include <stdlib.h>
# include <conio.h>

struct queue
{
    int data;
    struct queue *next;
};
typedef struct queue node;
node *front = NULL;
node *rear = NULL;

node* getnode()
{
    node *temp;
    temp = (node *) malloc(sizeof(node)) ;
    printf("\n Enter data ");
    scanf("%d", &temp -> data);
    temp -> next = NULL;
    return temp;
}
void insertQ()
{
    node *newnode;
    newnode = getnode();
    if(newnode == NULL)
    {
        printf("\n Queue Full");
        return;
    }
    if(front == NULL)
    {
        front = newnode;
        rear = newnode;
    }
    else
    {
        rear -> next = newnode;
        rear = newnode;
    }

    printf("\n\n\t Data Inserted into the Queue..");
}
void deleteQ()
{
    node *temp;
    if(front == NULL)
    {
        printf("\n\n\t Empty Queue..");
        return;
    }
    temp = front;
    front = front -> next;
    printf("\n\n\t Deleted element from queue is %d ", temp -> data);
    free(temp);
}
```

```

void displayQ()
{
    node *temp;
    if(front == NULL)
    {
        printf("\n\n\t\t Empty Queue ");
    }
    else
    {
        temp = front;
        printf("\n\n\n\t\t Elements in the Queue are: ");
        while(temp != NULL )
        {
            printf("%5d ", temp -> data);
            temp = temp -> next;
        }
    }
}

char menu()
{
    char ch;
    clrscr();
    printf("\n \t..Queue operations using pointers.. ");
    printf("\n\t -----*****----- \n");
    printf("\n 1. Insert ");
    printf("\n 2. Delete ");
    printf("\n 3. Display");
    printf("\n 4. Quit ");
    printf("\n Enter your choice: ");
    ch = getche();
    return ch;
}

void main()
{
    char ch;
    do
    {
        ch = menu();
        switch(ch)
        {
            case '1' :
                insertQ();
                break;
            case '2' :
                deleteQ();
                break;
            case '3' :
                displayQ();
                break;
            case '4':
                return;
        }
        getch();
    } while(ch != '4');
}

```

4.7. Applications of Queue:

1. It is used to schedule the jobs to be processed by the CPU.
2. When multiple users send print jobs to a printer, each printing job is kept in the printing queue. Then the printer prints those jobs according to first in first out (FIFO) basis.
3. Breadth first search uses a queue data structure to find an element from a graph.

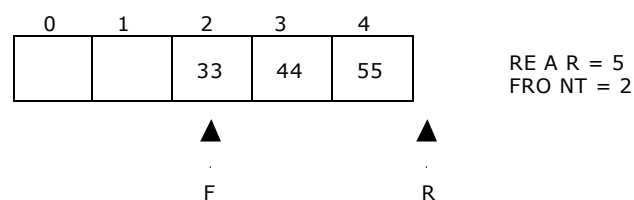
4.8. Circular Queue:

A more efficient queue representation is obtained by regarding the array $Q[\text{MAX}]$ as circular. Any number of items could be placed on the queue. This implementation of a queue is called a circular queue because it uses its storage array as if it were a circle instead of a linear list.

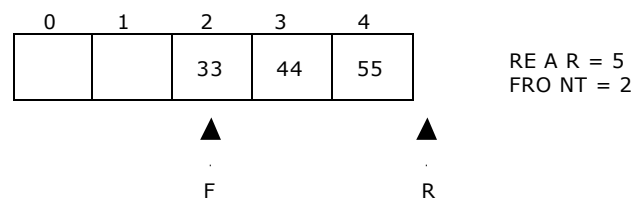
There are two problems associated with linear queue. They are:

- Time consuming: linear time to be spent in shifting the elements to the beginning of the queue.
- Signaling queue full: even if the queue is having vacant position.

For example, let us consider a linear queue status as follows:



Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:

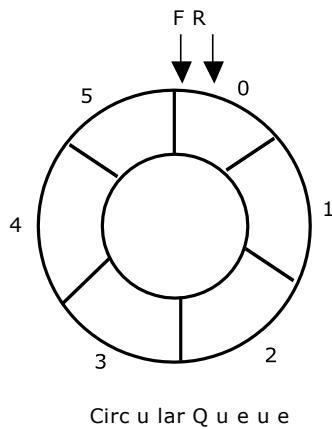


This difficulty can be overcome if we treat queue position with index zero as a position that comes after position with index four then we treat the queue as a **circular queue**.

In circular queue if we reach the end for inserting elements to it, it is possible to insert new elements if the slots at the beginning of the circular queue are empty.

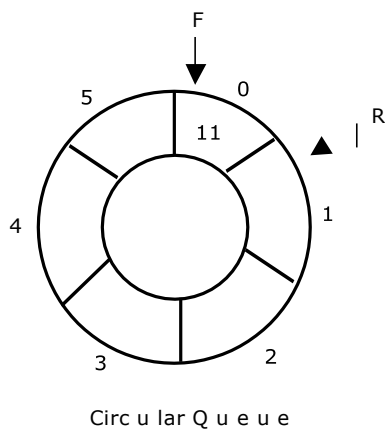
4.8.1. Representation of Circular Queue:

Let us consider a circular queue, which can hold maximum (MAX) of six elements. Initially the queue is empty.



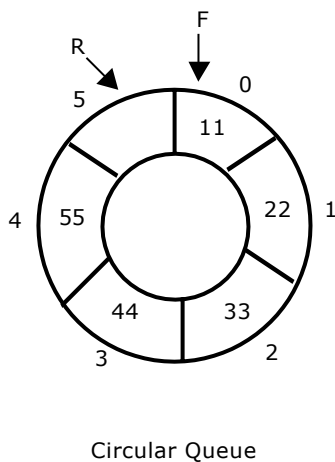
Queue Empty
MAX = 6
FRONT = REAR = 0
COUNT = 0

Now, insert 11 to the circular queue. Then circular queue status will be:



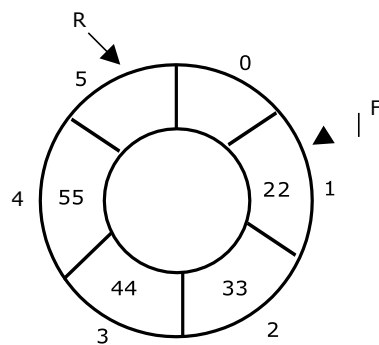
FRONT = 0
REAR = (REAR + 1) % 6 = 1
COUNT = 1

Insert new elements 22, 33, 44 and 55 into the circular queue. The circular queue status is:



FRONT = 0
REAR = (REAR + 1) % 6 = 5
COUNT = 5

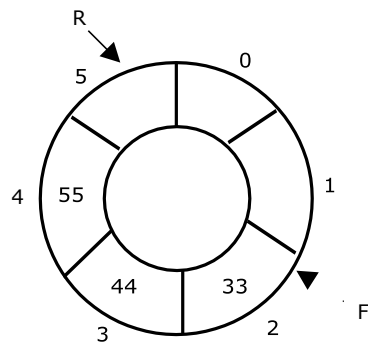
Now, delete an element. The element deleted is the element at the front of the circular queue. So, 11 is deleted. The circular queue status is as follows:



Circular Queue

$FRONT = (FRONT + 1) \% 6 = 1$
 $REAR = 5$
 $COUNT = COUNT - 1 = 4$

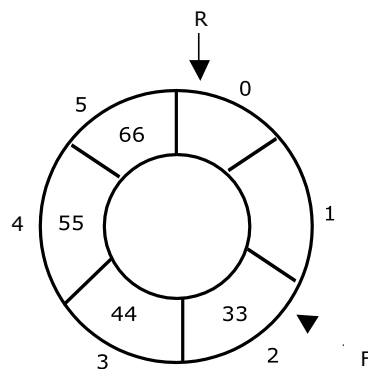
Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The circular queue status is as follows:



Circular Queue

$FRONT = (FRONT + 1) \% 6 = 2$
 $REAR = 5$
 $COUNT = COUNT - 1 = 3$

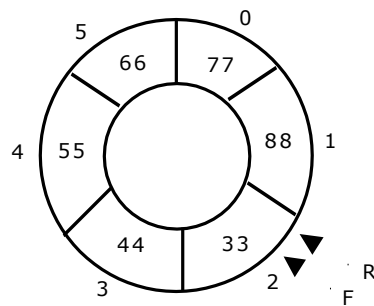
Again, insert another element 66 to the circular queue. The status of the circular queue is:



Circular Queue

$FRONT = 2$
 $REAR = (REAR + 1) \% 6 = 0$
 $COUNT = COUNT + 1 = 4$

Now, insert new elements 77 and 88 into the circular queue. The circular queue status is:



FRONT = 2, REAR = 2
 REAR = REAR % 6 = 2
 COUNT = 6

Circular Queue

Now, if we insert an element to the circular queue, as COUNT = MAX we cannot add the element to circular queue. So, the circular queue is *full*.

4.8.2. Source code for Circular Queue operations, using array:

```
# include <stdio.h>
# include <conio.h>
# define MAX 6
```

```
int CQ[MAX];
int front = 0;
int rear = 0;
int count = 0;
```

```
void insertCQ()
{
    int data;
    if(count == MAX)
    {
        printf("\n Circular Queue is Full");
    }
    else
    {
        printf("\n Enter data: ");
        scanf("%d", &data);
        CQ[rear] = data;
        rear = (rear + 1) % MAX;
        count ++;
        printf("\n Data Inserted in the Circular Queue ");
    }
}
```

```
void deleteCQ()
{
    if(count == 0)
    {
        printf("\n\nCircular Queue is Empty..");
    }
    else
    {
        printf("\n Deleted element from Circular Queue is %d ", CQ[front]);
        front = (front + 1) % MAX;
        count --;
    }
}
```

```

void displayCQ()
{
    int i, j;
    if(count == 0)
    {
        printf("\n\n\t Circular Queue is Empty ");
    }
    else
    {
        printf("\n Elements in Circular Queue are: ");
        j = count;
        for(i = front; j != 0; j--)
        {
            printf("%d\t", CQ[i]);
            i = (i + 1) % MAX;
        }
    }
}

int menu()
{
    int ch;
    clrscr();
    printf("\n \t Circular Queue Operations using ARRAY..");
    printf("\n -----***** ----- \n");
    printf("\n 1. Insert ");
    printf("\n 2. Delete ");
    printf("\n 3. Display");
    printf("\n 4. Quit ");
    printf("\n Enter Your Choice: ");
    scanf("%d", &ch);
    return ch;
}

void main()
{
    int ch;
    do
    {
        ch = menu();
        switch(ch)
        {
            case 1:
                insertCQ();
                break;
            case 2:
                deleteCQ();
                break;
            case 3:
                displayCQ();
                break;
            case 4:
                return;
            default:
                printf("\n Invalid Choice ");
        }

        getch();
    } while(1);
}

```

4.9. Deque:

In the preceding section we saw that a queue in which we insert items at one end and from which we remove items at the other end. In this section we examine an extension of the queue, which provides a means to insert and remove items at both ends of the queue. This data structure is a *deque*. The word *deque* is an acronym derived from *double-ended queue*. Figure 4.5 shows the representation of a deque.

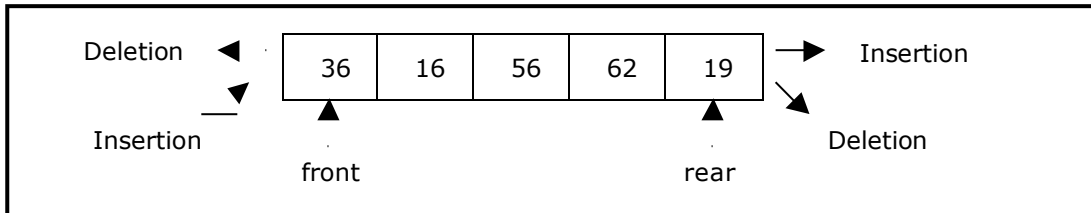


Figure 4.5. Representation of a deque.

A deque provides four operations. Figure 4.6 shows the basic operations on a deque.

- enqueue_front: insert an element at front.
- dequeue_front: delete an element at front.
- enqueue_rear: insert element at rear.
- dequeue_rear: delete element at rear.

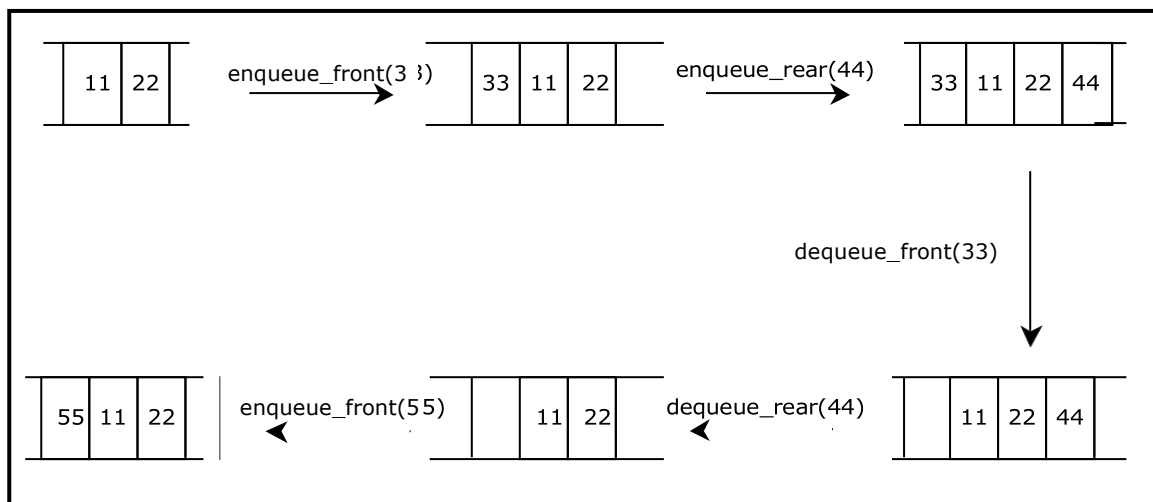


Figure 4.6. Basic operations on deque

There are two variations of deque. They are:

- Input restricted deque (IRD)
- Output restricted deque (ORD)

An Input restricted deque is a deque, which allows insertions at one end but allows deletions at both ends of the list.

An output restricted deque is a deque, which allows deletions at one end but allows insertions at both ends of the list.

4.10. Priority Queue:

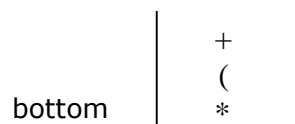
A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

1. An element of higher priority is processed before any element of lower priority.
2. two elements with same priority are processed according to the order in which they were added to the queue.

A prototype of a priority queue is time sharing system: programs of high priority are processed first, and programs with the same priority form a standard queue. An efficient implementation for the Priority Queue is to use heap, which in turn can be used for sorting purpose called heap sort.

Exercises

1. What is a linear data structure? Give two examples of linear data structures.
2. Is it possible to have two designs for the same data structure that provide the same functionality but are implemented differently?
3. What is the difference between the logical representation of a data structure and the physical representation?
4. Transform the following infix expressions to reverse polish notation:
 - a) $A \uparrow B * C - D + E / F / (G + H)$
 - b) $((A + B) * C - (D - E)) \uparrow (F + G)$
 - c) $A - B / (C * D \uparrow E)$
 - d) $(a + b \uparrow c \uparrow d) * (e + f / d)$
 - f) $3 - 6 * 7 + 2 / 4 * 5 - 8$
 - g) $(A - B) / ((D + E) * F)$
 - h) $((A + B) / D) \uparrow ((E - F) * G)$
5. Evaluate the following postfix expressions:
 - a) $P_1: 5, 3, +, 2, *, 6, 9, 7, -, /, -$
 - b) $P_2: 3, 5, +, 6, 4, -, *, 4, 1, -, 2, \uparrow, +$
 - c) $P_3: 3, 1, +, 2, \uparrow, 7, 4, -, 2, *, +, 5, -$
6. Consider the usual algorithm to convert an infix expression to a postfix expression. Suppose that you have read 10 input characters during a conversion and that the stack now contains these symbols:



Now, suppose that you read and process the 11th symbol of the input. Draw the stack for the case where the 11th symbol is:

- A. A number:
- B. A left parenthesis:
- C. A right parenthesis:
- D. A minus sign:
- E. A division sign:

7. Write a program using stack for parenthesis matching. Explain what modifications would be needed to make the parenthesis matching algorithm check expressions with different kinds of parentheses such as $()$, $[]$ and $\{\}$'s.
8. Evaluate the following prefix expressions:
 - a) $+ * 2 + / 14 2 5 1$
 - b) $- * 6 3 - 4 1$
 - c) $+ + 2 6 + - 13 2 4$
9. Convert the following infix expressions to prefix notation:
 - a) $((A + 2) * (B + 4)) - 1$
 - b) $Z - (((X + 1) * 2) - 5) / Y$
 - c) $((C * 2) + 1) / (A + B)$
 - d) $((A + B) * C - (D - E)) \uparrow (F + G)$
 - e) $A - B / (C * D \uparrow E)$
10. Write a "C" function to copy one stack to another assuming
 - a) The stack is implemented using array.
 - b) The stack is implemented using linked list.
11. Write an algorithm to construct a fully parenthesized infix expression from its postfix equivalent. Write a "C" function for your algorithm.
12. How can one convert a postfix expression to its prefix equivalent and vice-versa?
13. A double-ended queue (deque) is a linear list where additions and deletions can be performed at either end. Represent a deque using an array to store the elements of the list and write the "C" functions for additions and deletions.
14. In a circular queue represented by an array, how can one specify the number of elements in the queue in terms of "front", "rear" and MAX-QUEUE-SIZE? Write a "C" function to delete the K-th element from the "front" of a circular queue.
15. Can a queue be represented by a circular linked list with only one pointer pointing to the tail of the queue? Write "C" functions for the "add" and "delete" operations on such a queue
16. Write a "C" function to test whether a string of opening and closing parenthesis is well formed or not.
17. Represent N queues in a single one-dimensional array. Write functions for "add" and "delete" operations on the i^{th} queue
18. Represent a stack and queue in a single one-dimensional array. Write functions for "push", "pop" operations on the stack and "add", "delete" functions on the queue.

Multiple Choice Questions

1. Which among the following is a linear data structure: [D]
 A. Queue
 B. Stack
 C. Linked List
 D. all the above
2. Which among the following is a Dynamic data structure: [A]
 A. Double Linked List
 B. Queue
 C. Stack
 D. all the above
3. Stack is referred as: [A]
 A. Last in first out list
 B. First in first out list
 C. both A and B
 D. none of the above
4. A stack is a data structure in which all insertions and deletions of entries are made at: [A]
 A. One end
 B. In the middle
 C. Both the ends
 D. At any position
5. A queue is a data structure in which all insertions and deletions are made respectively at: [A]
 A. rear and front
 B. front and front
 C. front and rear
 D. rear and rear
6. Transform the following infix expression to postfix form: [D]
 $(A + B) * (C - D) / E$
 A. $A B * C + D / -$
 B. $A B C * C D / - +$
 C. $A B + C D * - / E$
 D. $A B + C D - * E /$
7. Transform the following infix expression to postfix form: [B]
 $A - B / (C * D)$
 A. $A B * C D - /$
 B. $A B C D * / -$
 C. $/ - D C * B A$
 D. $- / * A B C D$
8. Evaluate the following prefix expression: $* - + 4 3 5 / + 2 4 3$ [A]
 A. 4
 B. 8
 C. 1
 D. none of the above
9. Evaluate the following postfix expression: $1 4 18 6 / 3 + + 5 / +$ [C]
 A. 8
 B. 2
 C. 3
 D. none of the above
10. Transform the following infix expression to prefix form: [B]
 $((C * 2) + 1) / (A + B)$
 A. $A B + 1 2 C * + /$
 B. $/ + * C 2 1 + A B$
 C. $/ * + 1 2 C A B +$
 D. none of the above

11. Transform the following infix expression to prefix form: [D]
 $Z - (((X + 1) * 2) - 5) / Y$

A. / - * + X 1 2 5 Y

C. / * - + X 1 2 5 Y

B. Y 5 2 1 X + * - /

D. none of the above

12. Queue is also known as: [B]
A. Last in first out list
B. First in first out list
C. both A and B
D. none of the above

13. One difference between a queue and a stack is: [C]
 A. Queues require dynamic memory, but stacks do not.
 B. Stacks require dynamic memory, but queues do not.
 C. Queues use two ends of the structure; stacks use only one.
 D. Stacks use two ends of the structure, queues use only one.
14. If the characters 'D', 'C', 'B', 'A' are placed in a queue (in that order), and then removed one at a time, in what order will they be removed? [D]
 A. ABCD
 B. ABDC
 C. DCAB
 D. DCBA
15. Suppose we have a circular array implementation of the queue class, with ten items in the queue stored at data[2] through data[11]. The CAPACITY is 42. Where does the push member function place the new entry in the array? [D]
 A. data[1]
 B. data[2]
 C. data[11]
 D. data[12]
16. Consider the implementation of the queue using a circular array. What goes wrong if we try to keep all the items at the front of a partially-filled array (so that data[0] is always the front). [B]
 A. The constructor would require linear time.
 B. The get_front function would require linear time.
 C. The insert function would require linear time.
 D. The is_empty function would require linear time.
17. In the linked list implementation of the queue class, where does the push member function place the new entry on the linked list? [A]
 A. At the head
 B. At the tail
 C. After all other entries that are greater than the new entry.
 D. After all other entries that are smaller than the new entry.
18. In the circular array version of the queue class (with a fixed-sized array), which operations require linear time for their worst-case behavior? []
 A. front
 B. push
 C. empty
 D. None of these.
19. In the linked-list version of the queue class, which operations require linear time for their worst-case behavior? []
 A. front
 B. push
 C. empty
 D. None of these operations.
20. To implement the queue with a linked list, keeping track of a front pointer and a rear pointer. Which of these pointers will change during an insertion into a NONEMPTY queue? [B]
 A. Neither changes
 B. Only front_ptr changes.
 C. Only rear_ptr changes.
 D. Both change.
21. To implement the queue with a linked list, keeping track of a front point

er and a rear pointer. Which of these pointers will change during an insertion into an EMPTY queue?

[D]

- A. Neither changes
- B. Only front_ptr changes.
- C. Only rear_ptr changes.
- D. Both change.

22. Suppose top is called on a priority queue that has exactly two entries with equal priority. How is the return value of top selected? [B]
 A. The implementation gets to choose either one.
 B. The one which was inserted first.
 C. The one which was inserted most recently.
 D. This can never happen (violates the precondition)
23. Entries in a stack are "ordered". What is the meaning of this statement? [D]
 A. A collection of stacks can be sorted.
 B. Stack entries may be compared with the '<' operation.
 C. The entries must be stored in a linked list.
 D. There is a first entry, a second entry, and so on.
24. The operation for adding an entry to a stack is traditionally called: [D]
 A. add
 B. append
 C. insert
 D. push
25. The operation for removing an entry from a stack is traditionally called: [C]
 A. delete
 B. peek
 C. pop
 D. remove
26. Which of the following stack operations could result in stack underflow? [A]
 A. is_empty
 B. pop
 C. push
 D. Two or more of the above answers
27. Which of the following applications may use a stack? [D]
 A. A parentheses balancing program.
 B. Keeping track of local variables at run time.
 C. Syntax analyzer for a compiler.
 D. All of the above.
28. Here is an infix expression: $4 + 3 * (6 * 3 - 12)$. Suppose that we are using the usual stack algorithm to convert the expression from infix to postfix notation. What is the maximum number of symbols that will appear on the stack AT ONE TIME during the conversion of this expression? [D]
 A. 1
 B. 2
 C. 3
 D. 4
29. What is the value of the postfix expression $6\ 3\ 2\ 4\ +\ -\ *$ [A]
 A. Something between -15 and -100
 B. Something between -5 and -15
 C. Something between 5 and -5
 D. Something between 5 and 15
 E. Something between 15 and 100
30. If the expression $((2 + 3) * 4 + 5 * (6 + 7) * 8) + 9$ is evaluated with * having precedence over +, then the value obtained is same as the value of which of the following prefix expressions? [A]
 A. $++*+2\ 3\ 4\ **\ 5 + 6\ 7\ 8\ 9$
 B. $+*++2\ 3\ 4\ **\ 5 + 6\ 7\ 8\ 9$
 C. $*++++2\ 3\ 4\ **\ 5 + 6\ 7\ 8\ 9$
 D. $+*++2\ 3\ 4\ ++\ 5 * 6\ 7\ 8\ 9$

31. Evaluate the following prefix expression:
 $+ * 2 + / 14 2 5 1$

[B]

A. 50
B. 25

C. 40
D. 15

32. Parenthesis are never needed prefix or postfix expression: [A]
A. True C. Cannot be expected
B. False D. None of the above
33. A postfix expression is merely the reverse of the prefix expression: [B]
A. True C. Cannot be expected
B. False D. None of the above
34. Which among the following data structure may give overflow error, even though the current number of elements in it, is less than its size: [A]
A. Simple Queue C. Stack
B. Circular Queue D. None of the above
35. Which among the following types of expressions does not require precedence rules for evaluation: [C]
A. Fully parenthesized infix expression
B. Prefix expression
C. both A and B
D. none of the above
36. Conversion of infix arithmetic expression to postfix expression uses: [D]
A. Stack C. linked list
B. circular queue D. Queue

Recursion

Recursion is deceptively simple in statement but exceptionally complicated in implementation. Recursive procedures work fine in many problems. Many programmers prefer recursion through simpler alternatives are available. It is because recursion is elegant to use through it is costly in terms of time and space. But using it is one thing and getting involved with it is another.

In this unit we will look at "recursion" as a programmer who not only loves it but also wants to understand it! With a bit of involvement it is going to be an interesting reading for you.

2.1. Introduction to Recursion:

A function is recursive if a statement in the body of the function calls itself. Recursion is the process of defining something in terms of itself. For a computer language to be recursive, a function must be able to call itself.

For example, let us consider the function `factr()` shown below, which computes the factorial of an integer.

```
#include <stdio.h>
int factorial (int);
main()
{
    int num, fact;
    printf ("Enter a positive integer value: ");
    scanf ("%d", &num);
    fact = factorial (num);
    printf ("\n Factorial of %d =%5d\n", num, fact);
}

int factorial (int n)
{
    int result;
    if (n == 0)
        return (1);
    else
        result = n * factorial (n-1);

    return (result);
}
```

A non-recursive or iterative version for finding the factorial is as follows:

```
factorial (int n)
{
    int i, result = 1;
    if (n == 0)
```

```

        return (result);
    else
    {
        for (i=1; i<=n; i++)
            result = result * i;
    }

    return (result);
}

```

The operation of the non-recursive version is clear as it uses a loop starting at 1 and ending at the target value and progressively multiplies each number by the moving product.

When a function calls itself, new local variables and parameters are allocated storage on the stack and the function code is executed with these new variables from the start. A recursive call does not make a new copy of the function. Only the arguments and variables are new. As each recursive call returns, the old local variables and parameters are removed from the stack and execution resumes at the point of the function call inside the function.

When writing recursive functions, you must have a exit condition somewhere to force the function to return without the recursive call being executed. If you do not have an exit condition, the recursive function will recurse forever until you run out of stack space and indicate error about lack of memory, or stack overflow.

2.2. Differences between recursion and iteration:

- Both involve repetition.
- Both involve a termination test.
- Both can occur infinitely.

Iteration	Recursion
Iteration explicitly user a repetition structure.	Recursion achieves repetition through repeated function calls.
Iteration terminates when the loop continuation.	Recursion terminates when a base case is recognized.
Iteration keeps modifying the counter until the loop continuation condition fails.	Recursion keeps producing simple versions of the original problem until the base case is reached.
Iteration normally occurs within a loop so the extra memory assigned is omitted.	Recursion causes another copy of the function and hence a considerable memory space's occupied.
It reduces the processor's operating time.	It increases the processor's operating time.

2.3. Factorial of a given number:

The operation of recursive factorial function is as follows:

Start out with some natural number N (in our example, 5). The recursive definition is:

$n = 0, 0! = 1$	Base Case
$n > 0, n! = n * (n - 1)!$	Recursive Case

Recursion Factorials:

$$5! = 5 * 4! = 5 * \underline{\quad} = \underline{\quad}$$

$$4! = 4 * 3! = 4 * \underline{\quad} = \underline{\quad}$$

$$3! = 3 * 2! = 3 * \underline{\quad} = \underline{\quad}$$

$$2! = 2 * 1! = 2 * \underline{\quad} = \underline{\quad}$$

$$1! = 1 * 0! = 1 * \underline{\quad} = \underline{\quad}$$

$$0! = 1$$

$$\text{factr}(5) = 5 * \text{factr}(4) =$$

$$\text{factr}(4) = 4 * \text{factr}(3) = \underline{\quad}$$

$$\text{factr}(3) = 3 * \text{factr}(2) =$$

$$\text{factr}(2) = 2 * \text{factr}(1) = \underline{\quad}$$

$$\text{factr}(0) = \underline{\quad}$$

$$5! = 5*4! = 5*4*3! = 5*4*3*2! = 5*4*3*2*1! = 5*4*3*2*1*0! = 5*4*3*2*1*1 = 120$$

We define $0!$ to equal 1, and we define factorial N (where $N > 0$), to be $N * \text{factorial}(N-1)$. All recursive functions must have an exit condition, that is a state when it does not recurse upon itself. Our exit condition in this example is when $N = 0$.

Tracing of the flow of the factorial (factr) function:

When the factorial function is first called with, say, $N = 5$, here is what happens:

FUNCTION:

Does $N = 0$? No

Function Return Value = $5 * \text{factorial}(4)$

At this time, the function factorial is called again, with $N = 4$.

FUNCTION:

Does $N = 0$? No

Function Return Value = $4 * \text{factorial}(3)$

At this time, the function factorial is called again, with $N = 3$.

FUNCTION:

Does $N = 0$? No

Function Return Value = $3 * \text{factorial}(2)$

At this time, the function factorial is called again, with $N = 2$.

FUNCTION:

Does $N = 0$? No

Function Return Value = $2 * \text{factorial}(1)$

At this time, the function factorial is called again, with $N = 1$.

FUNCTION:

Does $N = 0$? No

Function Return Value = $1 * \text{factorial}(0)$

At this time, the function factorial is called again, with $N = 0$.

FUNCTION:

Does $N = 0$? Yes

Function Return Value = 1

Now, we have to trace our way back up! See, the factorial function was called six times. At any function level call, all function level calls above still exist! So, when we have $N = 2$, the function instances where $N = 3, 4$, and 5 are still waiting for their return values.

So, the function call where $N = 1$ gets retraced first, once the final call returns 0 . So, the function call where $N = 1$ returns $1 * 1$, or 1 . The next higher function call, where $N = 2$, returns $2 * 1$ (1 , because that's what the function call where $N = 1$ returned). You just keep working up the chain.

When $N = 2$, $2 * 1$, or 2 was returned.

When $N = 3$, $3 * 2$, or 6 was returned.

When $N = 4$, $4 * 6$, or 24 was returned.

When $N = 5$, $5 * 24$, or 120 was returned.

And since $N = 5$ was the first function call (hence the last one to be recalled), the value 120 is returned.

2.4. The Towers of Hanoi:

In the game of Towers of Hanoi, there are three towers labeled 1, 2, and 3. The game starts with n disks on tower A. For simplicity, let n is 3. The disks are numbered from 1 to 3, and without loss of generality we may assume that the diameter of each disk is the same as its number. That is, disk 1 has diameter 1 (in some unit of measure), disk 2 has diameter 2, and disk 3 has diameter 3. All three disks start on tower A in the order 1, 2, 3. The objective of the game is to move all the disks in tower 1 to entire tower 3 using tower 2. That is, at no time can a larger disk be placed on a smaller disk.

Figure 3.11.1, illustrates the initial setup of towers of Hanoi. The figure 3.11.2, illustrates the final setup of towers of Hanoi.

The rules to be followed in moving the disks from tower 1 tower 3 using tower 2 are as follows:

- Only one disk can be moved at a time.
- Only the top disc on any tower can be moved to any other tower.
- A larger disk cannot be placed on a smaller disk.

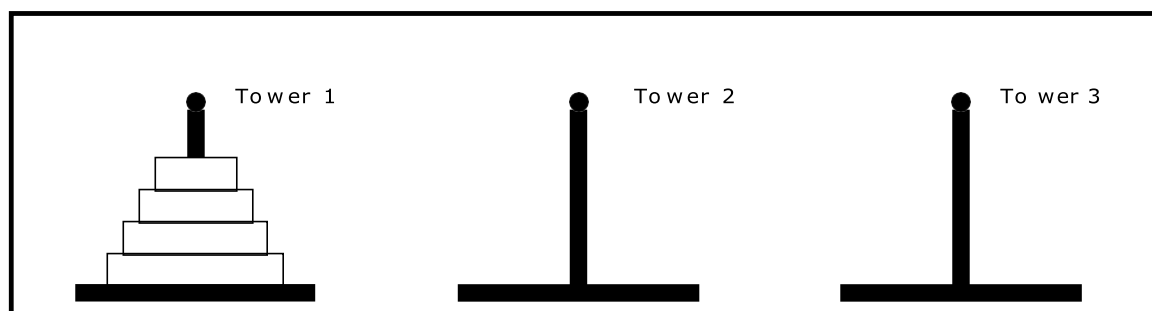


Fig. 3. 1 1. 1. In it ia l s et u p of T o w e r s of H a n o i

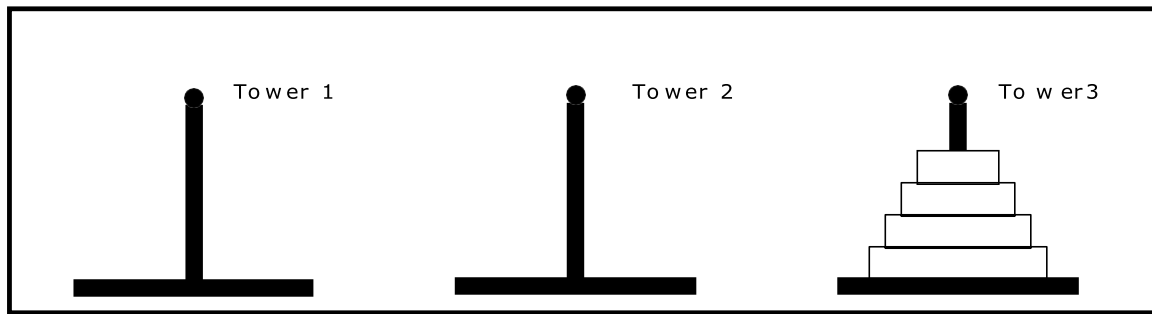


Fig 3.1 Initial setup of Towers of Hanoi

The towers of Hanoi problem can be easily implemented using recursion. To move the largest disk to the bottom of tower 3, we move the remaining $n - 1$ disks to tower 2 and then move the largest disk to tower 3. Now we have the remaining $n - 1$ disks to be moved to tower 3. This can be achieved by using the remaining two towers. We can also use tower 3 to place any disk on it, since the disk placed on tower 3 is the largest disk and continue the same operation to place the entire disks in tower 3 in order.

The program that uses recursion to produce a list of moves that shows how to accomplish the task of transferring the n disks from tower 1 to tower 3 is as follows:

```
#include <stdio.h>
#include <conio.h>

void towers_of_hanoi (int n, char *a, char *b, char *c);

int cnt=0;

int main (void)
{
    int n;
    printf("Enter number of discs: ");
    scanf("%d",&n);
    towers_of_hanoi (n, "Tower 1", "Tower 2", "Tower 3");
    getch();
}

void towers_of_hanoi (int n, char *a, char *b, char *c)
{
    if (n == 1)
    {
        ++cnt;
        printf ("\n%5d: Move disk 1 from %s to %s", cnt, a, c);
        return;
    }
    else
    {
        towers_of_hanoi (n-1, a, c, b);
        ++cnt;
        printf ("\n%5d: Move disk %d from %s to %s", cnt, n, a, c);
        towers_of_hanoi (n-1, b, a, c);
        return;
    }
}
```

Output of the program:

RUN 1:

Enter the number of discs: 3

```
1:    Move disk 1 from tower 1 to tower 3.
2:    Move disk 2 from tower 1 to tower 2.
3:    Move disk 1 from tower 3 to tower 2.
4:    Move disk 3 from tower 1 to tower 3.
5:    Move disk 1 from tower 2 to tower 1.
6:    Move disk 2 from tower 2 to tower 3.
7:    Move disk 1 from tower 1 to tower 3.
```

RUN 2:

Enter the number of discs: 4

```
1:    Move disk 1 from tower 1 to tower 2.
2:    Move disk 2 from tower 1 to tower 3.
3:    Move disk 1 from tower 2 to tower 3.
4:    Move disk 3 from tower 1 to tower 2.
5:    Move disk 1 from tower 3 to tower 1.
6:    Move disk 2 from tower 3 to tower 2.
7:    Move disk 1 from tower 1 to tower 2.
8:    Move disk 4 from tower 1 to tower 3.
9:    Move disk 1 from tower 2 to tower 3.
10:   Move disk 2 from tower 2 to tower 1.
11:   Move disk 1 from tower 3 to tower 1.
12:   Move disk 3 from tower 2 to tower 3.
13:   Move disk 1 from tower 1 to tower 2.
14:   Move disk 2 from tower 1 to tower 3.
15:   Move disk 1 from tower 2 to tower 3.
```

2.5. Fibonacci Sequence Problem:

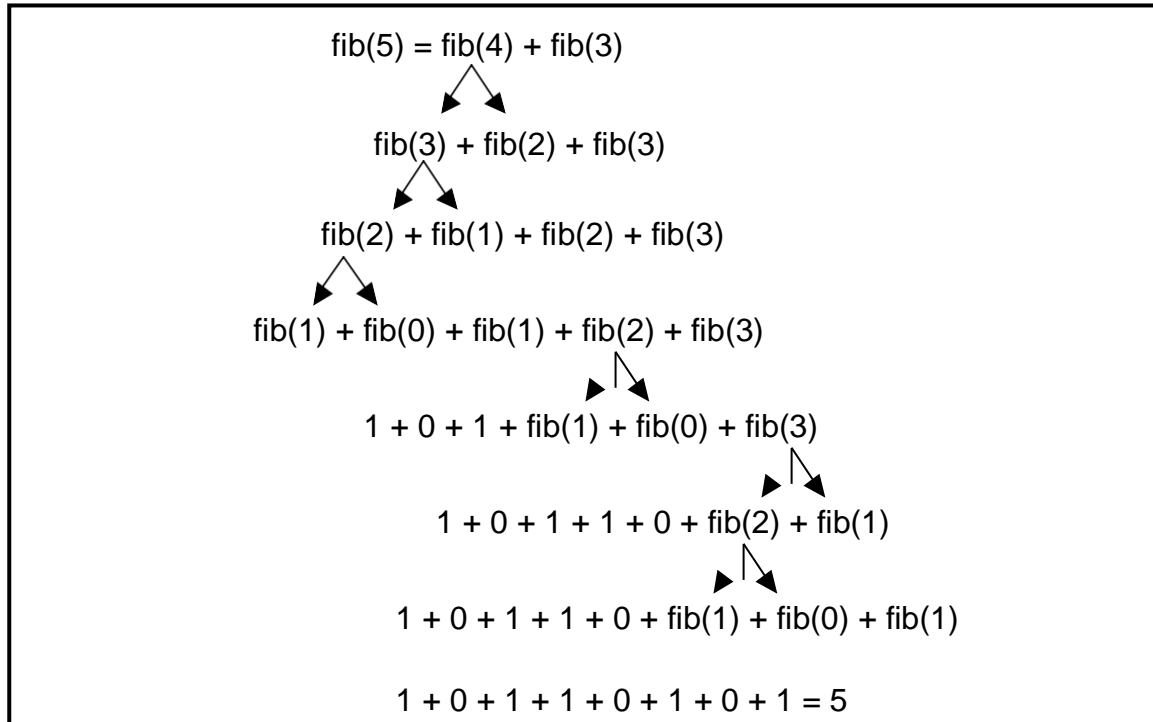
A Fibonacci sequence starts with the integers 0 and 1. Successive elements in this sequence are obtained by summing the preceding two elements in the sequence. For example, third number in the sequence is $0 + 1 = 1$, fourth number is $1 + 1 = 2$, fifth number is $1 + 2 = 3$ and so on. The sequence of Fibonacci integers is given below:

0 1 1 2 3 5 8 13 21

A recursive definition for the Fibonacci sequence of integers may be defined as follows:

$$\begin{aligned}\text{Fib}(n) &= n \text{ if } n = 0 \text{ or } n = 1 \\ \text{Fib}(n) &= \text{fib}(n-1) + \text{fib}(n-2) \text{ for } n \geq 2\end{aligned}$$

We will now use the definition to compute $\text{fib}(5)$:



We see that $\text{fib}(2)$ is computed 3 times, and $\text{fib}(3)$, 2 times in the above calculations. We save the values of $\text{fib}(2)$ or $\text{fib}(3)$ and reuse them whenever needed.

A recursive function to compute the Fibonacci number in the n^{th} position is given below:

```
main()
{
    clrscr ();
    printf ("n=5 fib(5) is %d", fib(5));
}

fib(n)
int n;
{
    int x;
    if (n==0 || n==1)
        return n;
    x=fib(n-1) + fib(n-2);
    return (x);
}
```

Output:

fib(5) is 5

2.6. Program using recursion to calculate the NCR of a given number:

```
#include<stdio.h>
float ncr (int n, int r);

void main()
{
    int n, r, result;
    printf("Enter the value of N and R :");
    scanf("%d %d", &n, &r);
    result = ncr(n, r);
    printf("The NCR value is %.3f", result);
}

float ncr (int n, int r)
{
    if(r == 0)
        return 1;
    else
        return(n * 1.0 / r * ncr (n-1, r-1));
}
```

Output:

Enter the value of N and R: 5 2
The NCR value is: 10.00

2.7. Program to calculate the least common multiple of a given number:

```
#include<stdio.h>

int alone(int a[], int n);
long int lcm(int a[], int n, int prime);

void main()
{
    int a[20], status, i, n, prime;
    printf ("Enter the limit: ");
    scanf("%d", &n);
    printf ("Enter the numbers : ");
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);
    printf ("The least common multiple is %ld", lcm(a, n, 2));
}

int alone (int a[], int n);
{
    int k;
    for (k = 0; k < n; k++)
        if (a[k] != 1)
            return 0;
    return 1;
}
```

```

long int lcm (int a[], int n, int prime)
{
    int i, status;
    status = 0;
    if (allone(a, n))
        return 1;
    for (i = 0; i < n; i++)
        if ((a[i] % prime) == 0)
        {
            status = 1;
            a[i] = a[i] / prime;
        }
    if (status == 1)
        return (prime * lcm(a, n, prime));
    else
        return (lcm (a, n, prime = (prime == 2) ? prime+1 : prime+2));
}

```

Output:

Enter the limit: 6
 Enter the numbers: 6 5 4 3 2 1
 The least common multiple is 60

2.8. Program to calculate the greatest common divisor:

```

#include<stdio.h>

int check_limit (int a[], int n, int prime);
int check_all (int a[], int n, int prime);
long int gcd (int a[], int n, int prime);

void main()
{
    int a[20], stat, i, n, prime;
    printf ("Enter the limit: ");
    scanf ("%d", &n);
    printf ("Enter the numbers: ");
    for (i = 0; i < n; i++)
        scanf ("%d", &a[i]);
    printf ("The greatest common divisor is %ld", gcd (a, n, 2));
}

int check_limit (int a[], int n, int prime)
{
    int i;
    for (i = 0; i < n; i++)
        if (prime > a[i])
            return 1;
    return 0;
}

```

```

int check_all (int a[], int n, int prime)
{
    int i;
    for (i = 0; i < n; i++)
        if ((a[i] % prime) != 0)
            return 0;
    for (i = 0; i < n; i++)
        a[i] = a[i] / prime;
    return 1;
}

long int gcd (int a[], int n, int prime)
{
    int i;
    if (check_limit(a, n, prime))
        return 1;
    if (check_all (a, n, prime))
        return (prime * gcd (a, n, prime));
    else
        return (gcd (a, n, prime = (prime == 2) ? prime+1 : prime+2));
}

```

Output:

Enter the limit: 5
 Enter the numbers: 99 55 22 77 121
 The greatest common divisor is 11

Exercises

- What is the importance of the stopping case in recursive functions?
- Write a function with one positive integer parameter called n . The function will write $2^n - 1$ integers (where $^$ is the exponentiation operation). Here are the patterns of output for various values of n :

 $n=1$: Output is: 1
 $n=2$: Output is: 1 2 1
 $n=3$: Output is: 1 2 1 3 1 2 1
 $n=4$: Output is: 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1

 And so on. Note that the output for n always consists of the output for $n-1$, followed by n itself, followed by a second copy of the output for $n-1$.
- Write a recursive function for the mathematical function:
 $f(n) = 1$ if $n = 1$
 $f(n) = 2 * f(n-1)$ if $n \geq 2$
- Which method is preferable in general?
 a) Recursive method
 b) Non-recursive method
- Write a function using Recursion to print numbers from n to 0.
- Write a function using Recursion to enter and display a string in reverse and state whether the string contains any spaces. Don't use arrays/strings.

7. Write a function using Recursion to check if a number n is prime. (You have to check whether n is divisible by any number below n)
8. Write a function using Recursion to enter characters one by one until a space is encountered. The function should return the depth at which the space was encountered.

Multiple Choice Questions

1. In a single function declaration, what is the maximum number of statements that may be recursive calls? []
 A. 1
 B. 2
 C. n (where n is the argument)
 D. There is no fixed maximum
2. What is the maximum depth of recursive calls a function may make? []
 A. 1
 B. 2
 C. n (where n is the argument)
 D. There is no fixed maximum
3. Consider the following function: []

```
void super_write_vertical (int number)
{
    if (number < 0)
    {
        printf(" - ");
        super_write_vertical(abs(number));
    }
    else if (number < 10)
        printf("%d\n", number);
    else
    {
        super_write_vertical(number/10);
        printf("%d\n", number % 10);
    }
}
```

 What values of number are directly handled by the stopping case?
 A. $\text{number} < 0$
 B. $\text{number} < 10$
 C. $\text{number} \geq 0 \ \&\& \ \text{number} < 10$
 D. $\text{number} > 10$
4. Consider the following function: []

```
void super_write_vertical(int number)
{
    if (number < 0)
    {
        printf(" - ");
        super_write_vertical (abs(number));
    }
    else if (number < 10)
        printf("%d\n", number);
    else
    {
        super_write_vertical(number/10);
        printf("%d\n", number % 10);
    }
}
```

 Which call will result in the most recursive calls?
 A. `super_write_vertical(-1023)`
 B. `super_write_vertical(0)`
 C. `super_write_vertical(100)`
 D. `super_write_vertical(1023)`

5. Consider this function declaration: []

```
void quiz (int i)
{
    if (i > 1)
    {
        quiz(i / 2);
        quiz(i / 2);
    }
    printf(" * ");
}
```

How many asterisks are printed by the function call quiz(5)?

- A. 3 B. 4
C. 7 D. 8
6. In a real computer, what will happen if you make a recursive call without making the problem smaller? []
- A. The operating system detects the infinite recursion because of the "repeated state"
B. The program keeps running until you press Ctrl-C
C. The results are non-deterministic
D. The run-time stack overflows, halting the program
7. When the compiler compiles your program, how is a recursive call treated differently than a non-recursive function call? []
- A. Parameters are all treated as reference arguments
B. Parameters are all treated as value arguments
C. There is no duplication of local variables
D. None of the above
8. When a function call is executed, which information is not saved in the activation record? []
- A. Current depth of recursion.
B. Formal parameters.
C. Location where the function should return when done.
D. Local variables
9. What technique is often used to prove the correctness of a recursive function? []
- A. Communitivity. B. Diagonalization.

Binary Trees

A data structure is said to be linear if its elements form a sequence or a linear list. Previous linear data structures that we have studied like an array, stacks, queues and linked lists organize data in linear order. A data structure is said to be non linear if its elements form a hierarchical classification where, data items appear at various levels.

Trees and Graphs are widely used non-linear data structures. Tree and graph structures represents hierarchial relationship between individual data elements. Graphs are nothing but trees with certain restrictions removed.

In this chapter in particular, we will explain special type of trees known as binary trees, which are easy to maintain in the computer.

5.1. TREES:

A tree is hierarchical collection of nodes. One of the nodes, known as the root, is at the top of the hierarchy. Each node can have at most one link coming into it. The node where the link originates is called the parent node. The root node has no parent. The links leaving a node (any number of links are allowed) point to child nodes. Trees are recursive structures. Each child node is itself the root of a subtree. At the bottom of the tree are leaf nodes, which have no children.

Trees represent a special case of more general structures known as graphs. In a graph, there is no restrictions on the number of links that can enter or leave a node, and cycles may be present in the graph. The figure 5.1.1 shows a tree and a non-tree.



Figure 5.1.1 A Tree and a not a tree

In a tree data structure, there is no distinction between the various children of a node i.e., none is the "first child" or "last child". A tree in which such distinctions are made is called an **ordered tree**, and data structures built on them are called **ordered tree data structures**. Ordered trees are by far the commonest form of tree data structure.

5.2. BINARY TREE:

In general, tree nodes can have any number of children. In a binary tree, each node can have at most two children. A binary tree is either **empty** or consists of a node called the **root** together with two binary trees called the **left subtree** and the **right subtree**.

A tree with no nodes is called as a **null** tree. A binary tree is shown in figure 5.2.1.

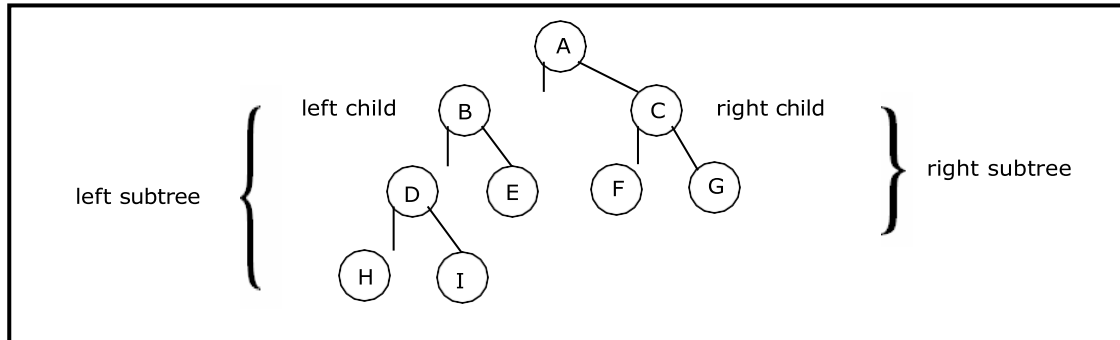


Figure 5.2.1. Binary Tree

Binary trees are easy to implement because they have a small, fixed number of child links. Because of this characteristic, binary trees are the most common types of trees and form the basis of many important data structures.

Tree Terminology:

Leaf node

A node with no children is called a *leaf* (or *external node*). A node which is not a leaf is called an *internal node*.

Path

A sequence of nodes n_1, n_2, \dots, n_k , such that n_i is the parent of n_{i+1} for $i = 1, 2, \dots, k - 1$. The length of a path is 1 less than the number of nodes on the path. Thus there is a path of length zero from a node to itself.

For the tree shown in figure 5.2.1, the path between A and I is A, B, D, I.

Siblings

The children of the same parent are called siblings.

For the tree shown in figure 5.2.1, F and G are the siblings of the parent node C and H and I are the siblings of the parent node D.

Ancestor and Descendent

If there is a path from node A to node B, then A is called an ancestor of B and B is called a descendent of A.

Subtree

Any node of a tree, with all of its descendants is a subtree.

Level

The level of the node refers to its distance from the root. The root of the tree has level 0, and the level of any other node in the tree is one more than the level of its parent. For example, in the binary tree of Figure 5.2.1 node F is at level 2 and node H is at level 3. *The maximum number of nodes at any level is 2^n .*

Height

The maximum level in a tree determines its height. The height of a node in a tree is the length of a longest path from the node to a leaf. The term depth is also used to denote height of the tree. The height of the tree of Figure 5.2.1 is 3.

Depth

The depth of a node is the number of nodes along the path from the root to that node. For instance, node 'C' in figure 5.2.1 has a depth of 1.

Assigning level numbers and Numbering of nodes for a binary tree:

The nodes of a binary tree can be numbered in a natural way, level by level, left to right. The nodes of a complete binary tree can be numbered so that the root is assigned the number 1, a left child is assigned twice the number assigned its parent, and a right child is assigned one more than twice the number assigned its parent. For example, see Figure 5.2.2.

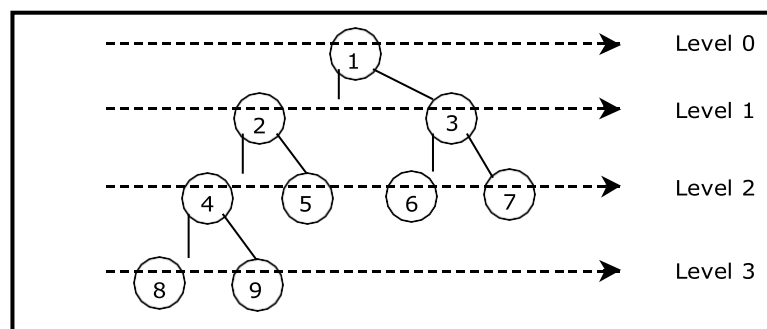


Figure 5.2.2. Level by level numbering of binary tree

Properties of binary trees:

Some of the important properties of a binary tree are as follows:

1. If h = height of a binary tree, then
 - a. Maximum number of leaves = 2^h
 - b. Maximum number of nodes = $2^{h+1} - 1$
2. If a binary tree contains m nodes at level l , it contains at most $2m$ nodes at level $l + 1$.
3. Since a binary tree can contain at most one node at level 0 (the root), it can contain at most 2^l nodes at level l .
4. The total number of edges in a full binary tree with n nodes is $n - 1$.

Strictly Binary tree:

If every non-leaf node in a binary tree has nonempty left and right subtrees, the tree is termed as strictly binary tree. Thus the tree of figure 5.2.3(a) is strictly binary. A strictly binary tree with n leaves always contains $2n - 1$ nodes.

Full Binary tree:

A full binary tree of height h has all its leaves at level h . Alternatively; All non leaf nodes of a full binary tree have two children, and the leaf nodes have no children.

A full binary tree with height h has $2^{h+1} - 1$ nodes. A full binary tree of height h is a *strictly binary tree* all of whose leaves are at level h . Figure 5.2.3(d) illustrates the full binary tree containing 15 nodes and of height 3.

A full binary tree of height h contains 2^h leaves and, $2^h - 1$ non-leaf nodes.

Thus by induction, total number of nodes (tn) = $\sum_{l=0}^h 2^l = 2^{h+1} - 1$.

For example, a full binary tree of height 3 contains $2^{3+1} - 1 = 15$ nodes.

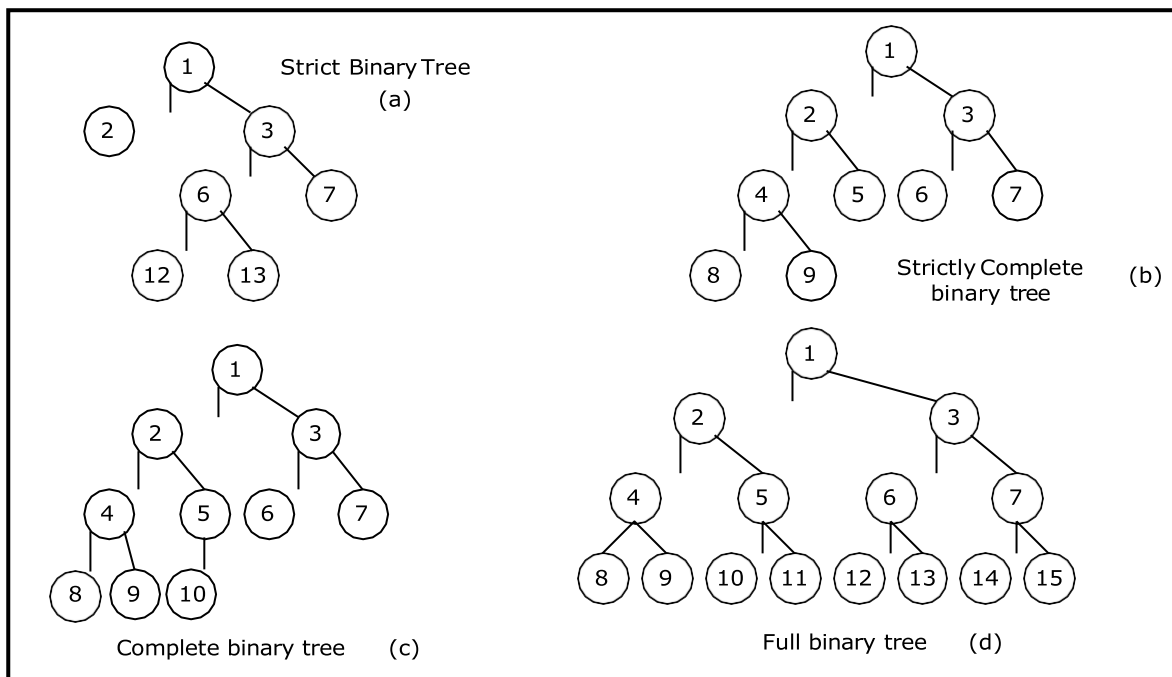


Figure 5.2.3. Examples of binary trees

Complete Binary tree:

A binary tree with n nodes is said to be **complete** if it contains all the first n nodes of the above numbering scheme. Figure 5.2.4 shows examples of complete and incomplete binary trees.

A complete binary tree of height h looks like a full binary tree down to level $h-1$, and the level h is filled from left to right.

A complete binary tree with n leaves that is *not strictly* binary has $2n$ nodes. For example, the tree of Figure 5.2.3(c) is a complete binary tree having 5 leaves and 10 nodes.

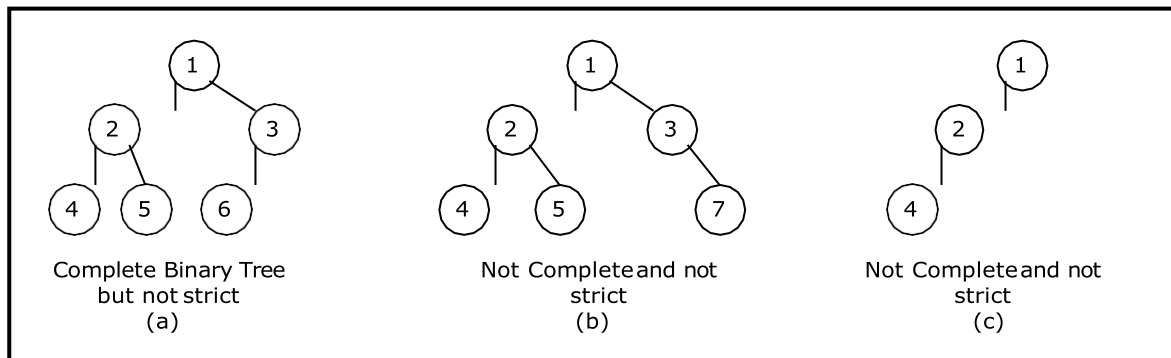


Figure 5.2.4. Examples of complete and incomplete binary trees

Internal and external nodes:

We define two terms: Internal nodes and external nodes. An internal node is a tree node having at least one key and possibly some children. It is some times convenient to have another types of nodes, called an external node, and pretend that all null child links point to such a node. An external node doesn't exist, but serves as a conceptual place holder for nodes to be inserted.

We draw internal nodes using circles, with letters as labels. External nodes are denoted by squares. The square node version is sometimes called an extended binary tree. A binary tree with n internal nodes has $n+1$ external nodes. Figure 5.2.6 shows a sample tree illustrating both internal and external nodes.

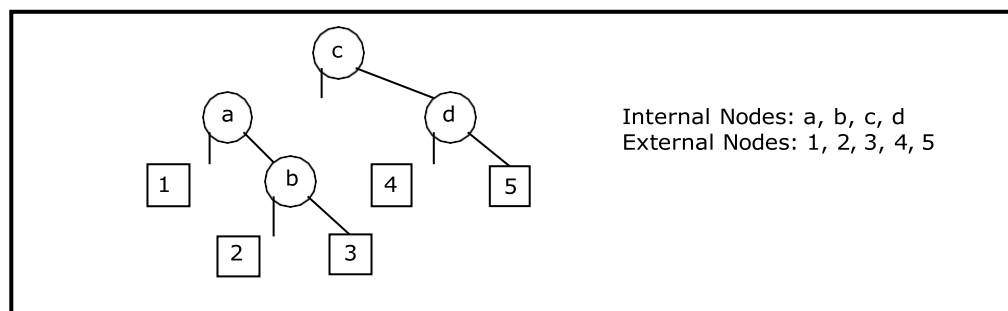


Figure 5.2.6. Internal and external nodes

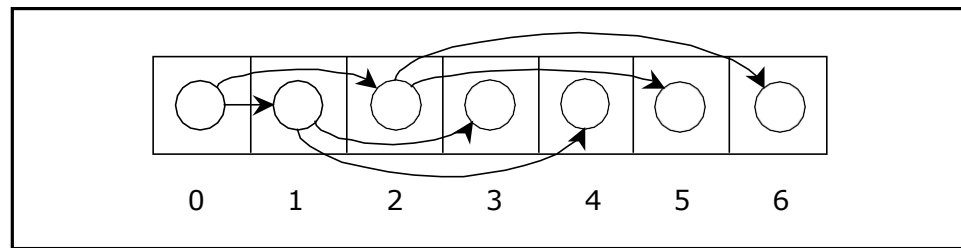
Data Structures for Binary Trees:

1. Arrays; especially suited for complete and full binary trees.
2. Pointer-based.

Array-based Implementation:

Binary trees can also be stored in arrays, and if the tree is a complete binary tree, this method wastes no space. In this compact arrangement, if a node has an index i , its children are found at indices $2i+1$ and $2i+2$, while its parent (if any) is found at index $\text{floor}((i-1)/2)$ (assuming the root of the tree stored in the array at an index zero).

This method benefits from more compact storage and better locality of reference, particularly during a preorder traversal. However, it requires contiguous memory, expensive to grow and wastes space proportional to $2^h - n$ for a tree of height h with n nodes.



Linked Representation (Pointer based):

Array representation is good for complete binary tree, but it is wasteful for many other binary trees. The representation suffers from insertion and deletion of node from the middle of the tree, as it requires the movement of potentially many nodes to reflect the change in level number of this node. To overcome this difficulty we represent the binary tree in linked representation.

In linked representation each node in a binary has three fields, the left child field denoted as *LeftChild*, data field denoted as *data* and the right child field denoted as *RightChild*. If any sub-tree is empty then the corresponding pointer's LeftChild and RightChild will store a NULL value. If the tree itself is empty the root pointer will store a NULL value.

The advantage of using linked representation of binary tree is that:

- Insertion and deletion involve no data movement and no movement of nodes except the rearrangement of pointers.

The disadvantages of linked representation of binary tree includes:

- Given a node structure, it is difficult to determine its parent node.
- Memory spaces are wasted for storing NULL pointers for the nodes, which have no subtrees.

The structure definition, node representation empty binary tree is shown in figure 5.2.6 and the linked representation of binary tree using this node structure is given in figure 5.2.7.

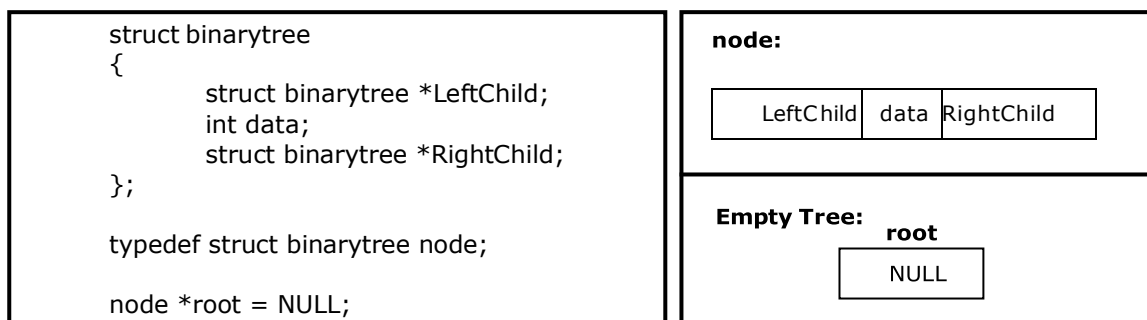


Figure 5.2.6. Structure definition, node representation and empty tree

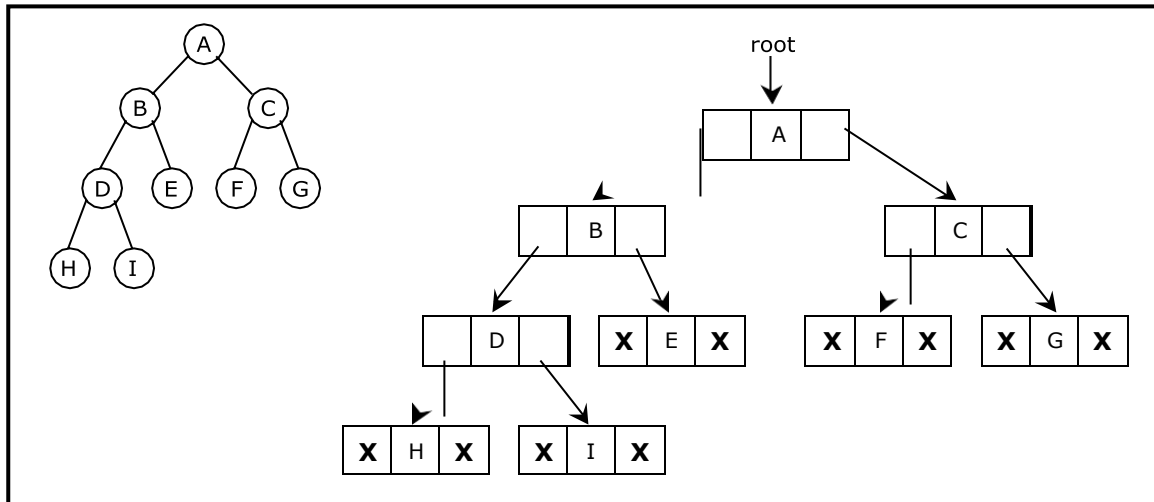


Figure 5.2.7. Linked representation for the binary tree

5.3. Binary Tree Traversal Techniques:

A tree traversal is a method of visiting every node in the tree. By visit, we mean that some type of operation is performed. For example, you may wish to print the contents of the nodes.

There are four common ways to traverse a binary tree:

1. *Preorder*
2. *Inorder*
3. *Postorder*
4. *Level order*

In the first three traversal methods, the left subtree of a node is traversed before the right subtree. The difference among them comes from the difference in the time at which a root node is visited.

5.3.1. Recursive Traversal Algorithms:

Inorder Traversal:

In the case of inorder traversal, the root of each subtree is visited after its left subtree has been traversed but before the traversal of its right subtree begins. The steps for traversing a binary tree in inorder traversal are:

1. Visit the left subtree, using inorder.
2. Visit the root.
3. Visit the right subtree, using inorder.

The algorithm for inorder traversal is as follows:

```
void inorder(node *root)
{
    if(root != NULL)
    {
        inorder(root->lchild);
```



```

        print root -> data;
        inorder(root->rchild);
    }
}

```

Preorder Traversal:

In a preorder traversal, each root node is visited before its left and right subtrees are traversed. Preorder search is also called backtracking. The steps for traversing a binary tree in preorder traversal are:

1. Visit the root.
2. Visit the left subtree, using preorder.
3. Visit the right subtree, using preorder.

The algorithm for preorder traversal is as follows:

```

void preorder(node *root)
{
    if( root != NULL )
    {
        print root -> data;
        preorder (root -> lchild);
        preorder (root -> rchild);
    }
}

```

Postorder Traversal:

In a postorder traversal, each root is visited after its left and right subtrees have been traversed. The steps for traversing a binary tree in postorder traversal are:

1. Visit the left subtree, using postorder.
2. Visit the right subtree, using postorder
3. Visit the root.

The algorithm for postorder traversal is as follows:

```

void postorder(node *root)
{
    if( root != NULL )
    {
        postorder (root -> lchild);
        postorder (root -> rchild);
        print (root -> data);
    }
}

```

Level order Traversal:

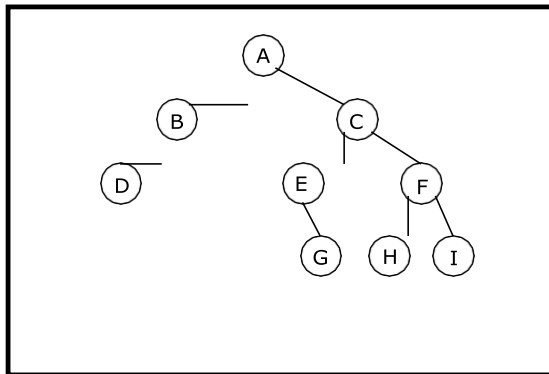
In a level order traversal, the nodes are visited level by level starting from the root, and going from left to right. The level order traversal requires a queue data structure. So, it is not possible to develop a recursive procedure to traverse the binary tree in level order. This is nothing but a breadth first search technique.

The algorithm for level order traversal is as follows:

```
void levelorder()
{
    int j;
    for(j = 0; j < ctr; j++)
    {
        if(tree[j] != NULL)
            print tree[j] -> data;
    }
}
```

Example 1:

Traverse the following binary tree in pre, post, inorder and level order.



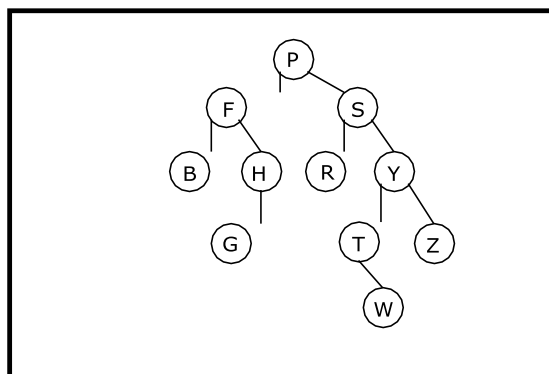
Binary Tree

- Preorder traversal yields:
A, B, D, C, E, G, F, H, I
- Postorder traversal yields:
D, B, G, E, H, I, F, C, A
- Inorder traversal yields:
D, B, A, E, G, C, H, F, I
- Level order traversal yields:
A, B, C, D, E, F, G, H, I

Pre, Post, Inorder and level order Traversing

Example 2:

Traverse the following binary tree in pre, post, inorder and level order.



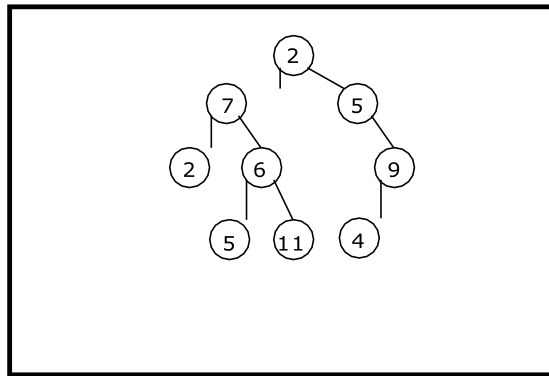
Binary Tree

- Preorder traversal yields:
P, F, B, H, G, S, R, Y, T, W, Z
- Postorder traversal yields:
B, G, H, F, R, W, T, Z, Y, S, P
- Inorder traversal yields:
B, F, G, H, P, R, S, T, W, Y, Z
- Level order traversal yields:
P, F, S, B, H, R, Y, G, T, Z, W

Pre, Post, Inorder and level order Traversing

Example 3:

Traverse the following binary tree in pre, post, inorder and level order.



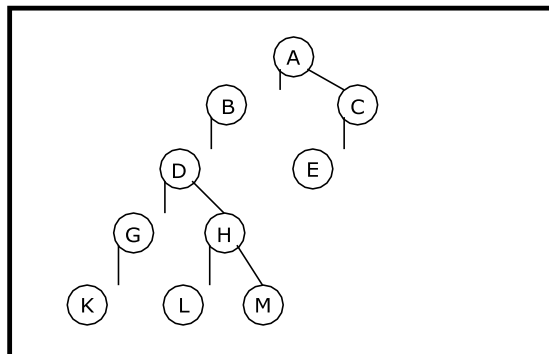
Bin ary Tree

- Preorder traversal yields:
2 , 7 , 2 , 6 , 5 , 11 , 5 , 9 , 4
- Postorder traversal yields:
2 , 5 , 11 , 6 , 7 , 4 , 9 , 5 , 2
- Inorder traversal yields:
2 , 7 , 5 , 6 , 11 , 2 , 5 , 4 , 9
- Level order traversal yields:
2 , 7 , 5 , 2 , 6 , 9 , 5 , 11 , 4

Pre, Post, Inorder and level order Traversing

Example 4:

Traverse the following binary tree in pre, post, inorder and level order.



Binary Tree

- Preorder traversal yields:
A , B , D , G , K , H , L , M , C , E
- Postorder traversal yields:
K , G , L , M , H , D , B , E , C , A
- Inorder traversal yields:
K , G , D , L , H , M , B , A , E , C
- Level order traversal yields:
A , B , C , D , E , G , H , K , L , M

Pre, Post, Inorder and level order Traversing

5.3.2. Building Binary Tree from Traversal Pairs:

Sometimes it is required to construct a binary tree if its traversals are known. From a single traversal it is not possible to construct unique binary tree. However any of the two traversals are given then the corresponding tree can be drawn uniquely:

- Inorder and preorder
- Inorder and postorder
- Inorder and level order

The basic principle for formulation is as follows:

If the preorder traversal is given, then the first node is the root node. If the postorder traversal is given then the last node is the root node. Once the root node is identified, all the nodes in the left sub-trees and right sub-trees of the root node can be identified using inorder.

Same technique can be applied repeatedly to form sub-trees.

It can be noted that, for the purpose mentioned, two traversal are essential out of which one should be inorder traversal and another preorder or postorder; alternatively, given preorder and postorder traversals, binary tree cannot be obtained uniquely.

Example 1:

Construct a binary tree from a given preorder and inorder sequence:

Preorder: A B D G C E H I F

Inorder: D G B A H E I C F

Solution:

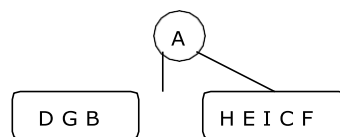
From Preorder sequence **A** B D G C E H I F, the root is: A

From Inorder sequence D G B **A** H E I C F, we get the left and right sub trees:

Left sub tree is: D G B

Right sub tree is: H E I C F

The Binary tree upto this point looks like:

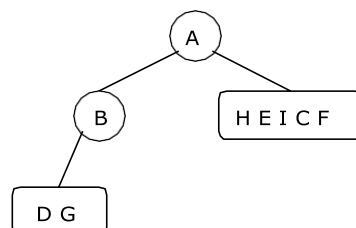


To find the root, left and right sub trees for D G B:

From the preorder sequence **B** D G, the root of tree is: B

From the inorder sequence D G **B**, we can find that D and G are to the left of B.

The Binary tree upto this point looks like:

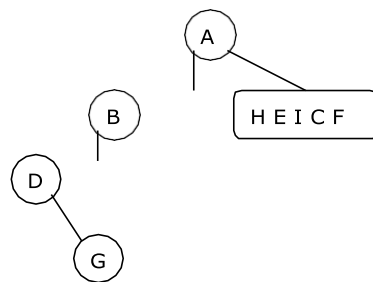


To find the root, left and right sub trees for D G:

From the preorder sequence **D** G, the root of the tree is: D

From the inorder sequence **D** G, we can find that there is no left node to D and G is at the right of D.

The Binary tree upto this point looks like:

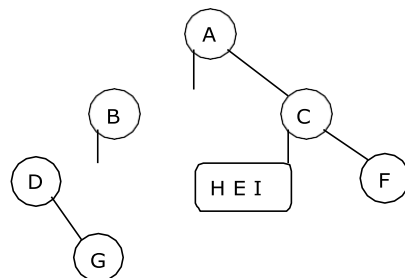


To find the root, left and right sub trees for H E I C F:

From the preorder sequence **C** E H I F, the root of the left sub tree is: C

From the inorder sequence H E I **C** F, we can find that H E I are at the left of C and F is at the right of C.

The Binary tree upto this point looks like:

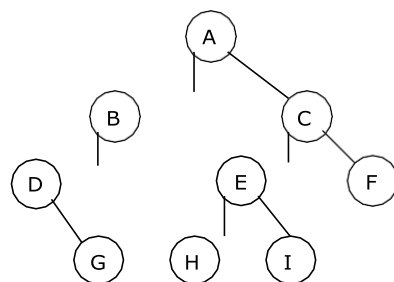


To find the root, left and right sub trees for H E I:

From the preorder sequence **E** H I, the root of the tree is: E

From the inorder sequence H **E** I, we can find that H is at the left of E and I is at the right of E.

The Binary tree upto this point looks like:



Example 2:

Construct a binary tree from a given postorder and inorder sequence:

Inorder: D G B A H E I C F

Postorder: G D B H I E F C A

Solution:

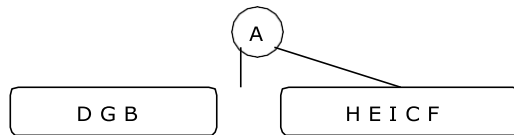
From Postorder sequence $G D B H I E F C$ **A**, the root is: A

From Inorder sequence $D G B$ **A** $H E I C F$, we get the left and right sub trees:

Left sub tree is: $D G B$

Right sub tree is: $H E I C F$

The Binary tree upto this point looks like:

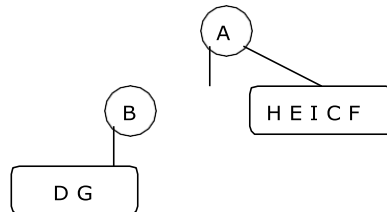


To find the root, left and right sub trees for $D G B$:

From the postorder sequence $G D B$, the root of tree is: B

From the inorder sequence $D G$ **B**, we can find that $D G$ are to the left of B and there is no right subtree for B .

The Binary tree upto this point looks like:

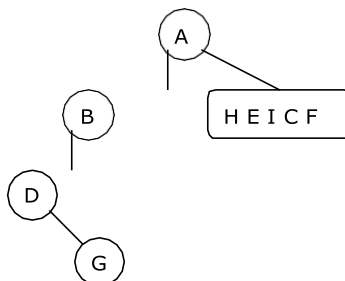


To find the root, left and right sub trees for $D G$:

From the postorder sequence G **D**, the root of the tree is: D

From the inorder sequence **D** G , we can find that there is no left subtree for D and G is to the right of D .

The Binary tree upto this point looks like:

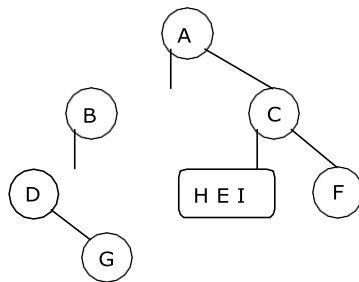


To find the root, left and right sub trees for $H E I C F$:

From the postorder sequence $H I E F$ **C**, the root of the left sub tree is: C

From the inorder sequence $H E I$ **C** F , we can find that $H E I$ are to the left of C and F is the right subtree for C .

The Binary tree upto this point looks like:

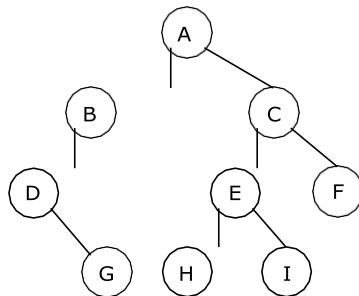


To find the root, left and right sub trees for H E I:

From the postorder sequence *H I E*, the root of the tree is: *E*

From the inorder sequence H E I, we can find that H is left subtree for E and I is to the right of E.

The Binary tree upto this point looks like:



Example 3:

Construct a binary tree from a given preorder and inorder sequence:

Inorder: n1 n2 n3 n4 n5 n6 n7 n8 n9

Preorder: n6 n2 n1 n4 n3 n5 n9 n7 n8

Solution:

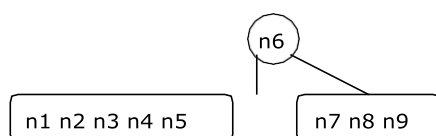
From Preorder sequence **n6** n2 n1 n4 n3 n5 n9 n7 n8, the root is: *n6*

From Inorder sequence n1 n2 n3 n4 n5 **n6** n7 n8 n9, we get the left and right sub trees:

Left sub tree is: n1 n2 n3 n4 n5

Right sub tree is: n7 n8 n9

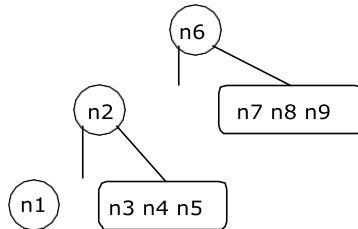
The Binary tree upto this point looks like:



To find the root, left and right sub trees for n1 n2 n3 n4 n5:

From the preorder sequence **n2** n1 n4 n3 n5, the root of tree is: n2

From the inorder sequence n1 **n2** n3 n4 n5, we can find that n1 is to the left of n2 and n3 n4 n5 are to the right of n2. The Binary tree upto this point looks like:

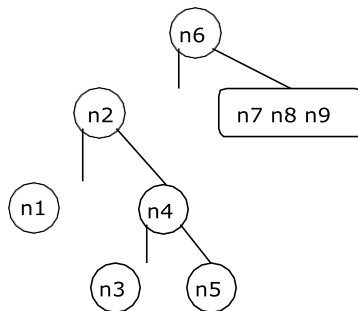


To find the root, left and right sub trees for n3 n4 n5:

From the preorder sequence **n4** n3 n5, the root of the tree is: n4

From the inorder sequence n3 **n4** n5, we can find that n3 is to the left of n4 and n5 is at the right of n4.

The Binary tree upto this point looks like:

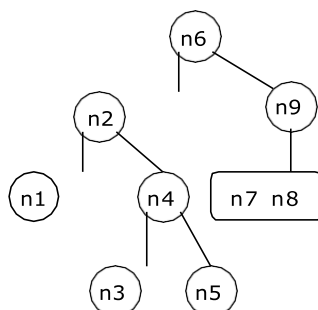


To find the root, left and right sub trees for n7 n8 n9:

From the preorder sequence **n9** n7 n8, the root of the left sub tree is: n9

From the inorder sequence n7 n8 **n9**, we can find that n7 and n8 are at the left of n9 and no right subtree of n9.

The Binary tree upto this point looks like:

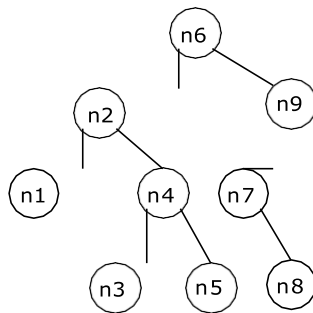


To find the root, left and right sub trees for n7 n8:

From the preorder sequence **n7** n8, the root of the tree is: n7

From the inorder sequence **n7** n8, we can find that is no left subtree for n7 and n8 is at the right of n7.

The Binary tree upto this point looks like:



Example 4:

Construct a binary tree from a given postorder and inorder sequence:

Inorder: n1 n2 n3 n4 n5 n6 n7 n8 n9

Postorder: n1 n3 n5 n4 n2 n8 n7 n9 n6

Solution:

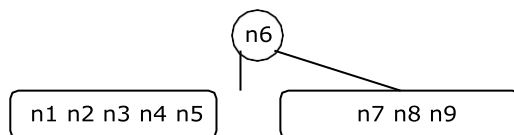
From Postorder sequence n1 n3 n5 n4 n2 n8 n7 n9 **n6**, the root is: n6

From Inorder sequence n1 n2 n3 n4 n5 **n6** n7 n8 n9, we get the left and right sub trees:

Left sub tree is: n1 n2 n3 n4 n5

Right sub tree is: n7 n8 n9

The Binary tree upto this point looks like:

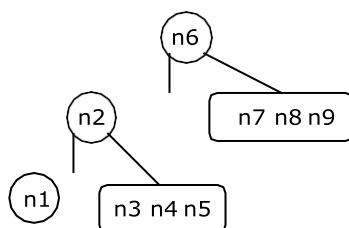


To find the root, left and right sub trees for n1 n2 n3 n4 n5:

From the postorder sequence n1 n3 n5 n4 **n2**, the root of tree is: n2

From the inorder sequence n1 **n2** n3 n4 n5, we can find that n1 is to the left of n2 and n3 n4 n5 are to the right of n2.

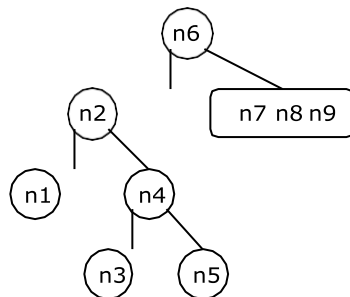
The Binary tree upto this point looks like:



To find the root, left and right sub trees for n3 n4 n5:

*From the postorder sequence n3 n5 **n4**, the root of the tree is: n4*

From the inorder sequence n3 n4 n5, we can find that n3 is to the left of n4 and n5 is to the right of n4. The Binary tree upto this point looks like:

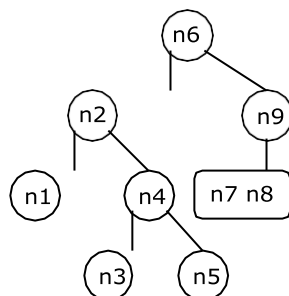


To find the root, left and right sub trees for n7 n8 and n9:

*From the postorder sequence n8 n7 **n9**, the root of the left sub tree is: n9*

From the inorder sequence n7 n8 n9, we can find that n7 and n8 are to the left of n9 and no right subtree for n9.

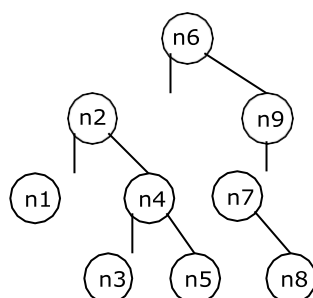
The Binary tree upto this point looks like:



To find the root, left and right sub trees for n7 and n8:

*From the postorder sequence n8 **n7**, the root of the tree is: n7*

From the inorder sequence **n7** n8, we can find that there is no left subtree for n7 and n8 is to the right of n7. The Binary tree upto this point looks like:



5.3.3. Binary Tree Creation and Traversal Using Arrays:

This program performs the following operations:

1. Creates a complete Binary Tree
2. Inorder traversal
3. Preorder traversal
4. Postorder traversal
5. Level order traversal
6. Prints leaf nodes
7. Finds height of the tree created

```
# include <stdio.h>
# include <stdlib.h>

struct tree
{
    struct tree* lchild;
    char data[10];
    struct tree* rchild;
};

typedef struct tree node;
int ctr;
node *tree[100];

node* getnode()
{
    node *temp ;
    temp = (node*) malloc(sizeof(node));
    printf("\n Enter Data: ");
    scanf("%s",temp->data);
    temp->lchild = NULL;
    temp->rchild = NULL;
    return temp;
}

void create_fbinarytree()
{
    int j, i=0;
    printf("\n How many nodes you want: ");
    scanf("%d",&ctr);
    tree[0] = getnode();
    j = ctr;
    j--;
    do
    {
        if( j > 0 )                                /* left child */
        {
            tree[ i * 2 + 1 ] = getnode();
            tree[i]->lchild = tree[i * 2 + 1];
            j--;
        }
        if( j > 0 )                                /* right child */
        {
            tree[ i * 2 + 2 ] = getnode();
            j--;
            tree[i]->rchild = tree[i * 2 + 2];
        }
        i++;
    } while( j > 0);
}
```

```

void inorder(node *root)
{
    if( root != NULL )
    {
        inorder(root->lchild);
        printf("%3s",root->data);
        inorder(root->rchild);
    }
}

```

```

void preorder(node *root)
{
    if( root != NULL )
    {
        printf("%3s",root->data);
        preorder(root->lchild);
        preorder(root->rchild);
    }
}

```

```

void postorder(node *root)
{
    if( root != NULL )
    {
        postorder(root->lchild);
        postorder(root->rchild);
        printf("%3s",root->data);
    }
}

```

```

void levelorder()
{
    int j;
    for(j = 0; j < ctr; j++)
    {
        if(tree[j] != NULL)
            printf("%3s",tree[j]->data);
    }
}

```

```

void print_leaf(node *root)
{
    if(root != NULL)
    {
        if(root->lchild == NULL && root->rchild == NULL)
            printf("%3s ",root->data);
        print_leaf(root->lchild);
        print_leaf(root->rchild);
    }
}

```

```

int height(node *root)
{
    if(root == NULL)
    {
        return 0;
    }
}

```

```

        if(root->lchild == NULL && root->rchild == NULL)
            return 0;
        else
            return (1 + max(height(root->lchild), height(root->rchild)));
    }

void main()
{
    int i;
    create_fbinarytree();
    printf("\n Inorder Traversal: ");
    inorder(tree[0]);
    printf("\n Preorder Traversal: ");
    preorder(tree[0]);
    printf("\n Postorder Traversal: ");
    postorder(tree[0]);
    printf("\n Level Order Traversal: ");
    levelorder();
    printf("\n Leaf Nodes: ");
    print_leaf(tree[0]);
    printf("\n Height of Tree: %d ", height(tree[0]));
}

```

5.3.4. Binary Tree Creation and Traversal Using Pointers:

This program performs the following operations:

1. Creates a complete Binary Tree
2. Inorder traversal
3. Preorder traversal
4. Postorder traversal
5. Level order traversal
6. Prints leaf nodes
7. Finds height of the tree created
8. Deletes last node
9. Finds height of the tree created

```

#include <stdio.h>
#include <stdlib.h>

struct tree
{
    struct tree* lchild;
    char data[10];
    struct tree* rchild;
};

typedef struct tree node;
node *Q[50];
int node_ctr;

node* getnode()
{
    node *temp ;
    temp = (node*) malloc(sizeof(node));
    printf("\n Enter Data: ");
    fflush(stdin);
    scanf("%s",temp->data);
    temp->lchild = NULL;
    temp->rchild = NULL;
    return temp;
}

```

```

void create_binarytree(node *root)
{
    char option;
    node_ctr = 1;
    if( root != NULL )
    {
        printf("\n Node %s has Left SubTree(Y/N)",root->data);
        fflush(stdin);
        scanf("%c",&option);
        if( option=='Y' || option == 'y')
        {
            root->lchild = getnode();
            node_ctr++;
            create_binarytree(root->lchild);
        }
        else
        {
            root->lchild = NULL;
            create_binarytree(root->lchild);
        }
    }

    printf("\n Node %s has Right SubTree(Y/N) ",root->data);
    fflush(stdin);
    scanf("%c",&option);
    if( option=='Y' || option == 'y')
    {
        root->rchild = getnode();
        node_ctr++;
        create_binarytree(root->rchild);
    }
    else
    {
        root->rchild = NULL;
        create_binarytree(root->rchild);
    }
}
}

```

```

void make_Queue(node *root,int parent)
{
    if(root != NULL)
    {
        node_ctr++;
        Q[parent] = root;
        make_Queue(root->lchild,parent*2+1);
        make_Queue(root->rchild,parent*2+2);
    }
}

```

```

delete_node(node *root, int parent)
{
    int index = 0;
    if(root == NULL)
        printf("\n Empty TREE ");
    else
    {
        node_ctr = 0;
        make_Queue(root,0);
        index = node_ctr-1;
        Q[index] = NULL;
        parent = (index-1) /2;
        if( 2* parent + 1 == index )
            Q[parent]->lchild = NULL;
    }
}

```

```

        else
            Q[parent]->rchild = NULL;
    }

    printf("\n Node Deleted ..");
}

void inorder(node *root)
{
    if(root != NULL)
    {
        inorder(root->lchild);
        printf("%3s",root->data);
        inorder(root->rchild);
    }
}

void preorder(node *root)
{
    if( root != NULL )
    {
        printf("%3s",root->data);
        preorder(root->lchild);
        preorder(root->rchild);
    }
}

void postorder(node *root)
{
    if( root != NULL )
    {
        postorder(root->lchild);
        postorder(root->rchild);
        printf("%3s", root->data);
    }
}

void print_leaf(node *root)
{
    if(root != NULL)
    {
        if(root->lchild == NULL && root->rchild == NULL)
            printf("%3s ",root->data);
        print_leaf(root->lchild);
        print_leaf(root->rchild);
    }
}

int height(node *root)
{
    if(root == NULL)
        return -1;
    else
        return (1 + max(height(root->lchild), height(root->rchild)));
}

void print_tree(node *root, int line)
{
    int i;
    if(root != NULL)
    {
        print_tree(root->rchild,line+1);
        printf("\n");
        for(i=0;i<line;i++)

```

```

        printf(" ");
        printf("%s", root->data);
        print_tree(root->lchild,line+1);
    }
}

void level_order(node *Q[],int ctr)
{
    int i;
    for( i = 0; i < ctr ; i++)
    {
        if( Q[i] != NULL )
            printf("%5s",Q[i]->data);
    }
}

int menu()
{
    int ch;
    clrscr();
    printf("\n 1. Create Binary Tree ");
    printf("\n 2. Inorder Traversal ");
    printf("\n 3. Preorder Traversal ");
    printf("\n 4. Postorder Traversal ");
    printf("\n 5. Level Order Traversal");
    printf("\n 6. Leaf Node ");
    printf("\n 7. Print Height of Tree ");
    printf("\n 8. Print Binary Tree ");
    printf("\n 9. Delete a node ");
    printf("\n 10. Quit ");
    printf("\n Enter Your choice: ");
    scanf("%d", &ch);
    return ch;
}

void main()
{
    int i,ch;
    node *root = NULL;
    do
    {
        ch = menu();
        switch( ch)
        {
            case 1 :
                if( root == NULL )
                {
                    root = getnode();
                    create_binarytree(root);
                }
                else
                {
                    printf("\n Tree is already Created ..");
                }
                break;
            case 2 :

                printf("\n Inorder Traversal: ");
                inorder(root);
                break;
            case 3 :
                printf("\n Preorder Traversal: ");
                preorder(root);
                break;

```



```

        case 4 :
            printf("\n Postorder Traversal: ");
            postorder(root);
            break;
        case 5:
            printf("\n Level Order Traversal ..");
            make_Queue(root,0);
            level_order(Q,node_ctr);
            break;
        case 6 :
            printf("\n Leaf Nodes: ");
            print_leaf(root);
            break;
        case 7 :
            printf("\n Height of Tree: %d ", height(root));
            break;
        case 8 :
            printf("\n Print Tree \n");
            print_tree(root, 0);
            break;
        case 9 :
            delete_node(root,0);
            break;
        case 10 :
            exit(0);
    }

    getch();
}while(1);
}

```

5.3.5. Non Recursive Traversal Algorithms:

At first glance, it appears that we would always want to use the flat traversal functions since they use less stack space. But the flat versions are not necessarily better. For instance, some overhead is associated with the use of an explicit stack, which may negate the savings we gain from storing only node pointers. Use of the implicit function call stack may actually be faster due to special machine instructions that can be used.

Inorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex, pushing each vertex onto the stack and stop when there is no left son of vertex.
2. Pop and process the nodes on stack if zero is popped then exit. If a vertex with right son exists, then set right son of vertex as current vertex and return to step one.

Algorithm inorder()

```

{
    stack[1] = 0
    vertex = root
top: while(vertex ≠ 0)
    {
        push the vertex into the stack
        vertex = leftson(vertex)
    }
}

```

```

    }

    pop the element from the stack and make it as vertex

    while(vertex ≠ 0)
    {
        print the vertex node
        if(rightson(vertex) ≠ 0)
        {
            vertex = rightson(vertex)
            goto top
        }
        pop the element from the stack and made it as vertex
    }
}

```

Preorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path by pushing the right son of vertex onto stack, if any and process each vertex. The traversing ends after a vertex with no left child exists.
2. Pop the vertex from stack, if vertex ≠ 0 then return to step one otherwise exit.

Algorithm preorder()

```

{
    stack[1] = 0
    vertex = root.
    while(vertex ≠ 0)
    {
        print vertex node
        if(rightson(vertex) ≠ 0)
            push the right son of vertex into the stack.
        if(leftson(vertex) ≠ 0)
            vertex = leftson(vertex)
        else
            pop the element from the stack and made it as vertex
    }
}

```

Postorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex. At each vertex of path push vertex on to stack and if vertex has a right son push -(right son of vertex) onto stack.
2. Pop and process the positive nodes (left nodes). If zero is popped then exit. If a negative node is popped, then ignore the sign and return to step one.

Algorithm postorder()

```

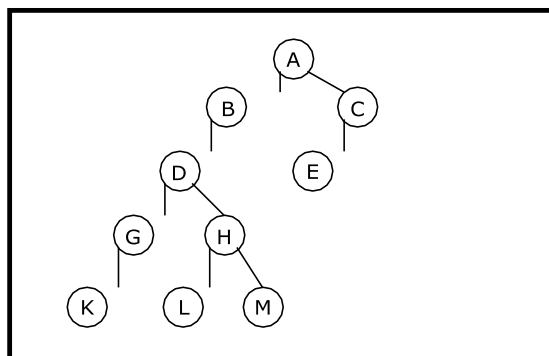
{
    stack[1] = 0
    vertex = root

top: while(vertex ≠ 0)
    {
        push vertex onto stack
        if(rightson(vertex) ≠ 0)
            push - (vertex) onto stack
        vertex = leftson(vertex)
    }
    pop from stack and make it as vertex
    while(vertex > 0)
    {
        print the vertex node
        pop from stack and make it as vertex
    }
    if(vertex < 0)
    {
        vertex = - (vertex)
        goto top
    }
}

```

Example 1:

Traverse the following binary tree in pre, post and inorder using non-recursive traversing algorithm.



Binary Tree

- Preorder traversal yields:
A, B, D, G, K, H, L, M, C, E
- Postorder traversal yields:
K, G, L, M, H, D, B, E, C, A
- Inorder traversal yields:
K, G, D, L, H, M, B, A, E, C

Pre, Post and Inorder Traversing

Inorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex, pushing each vertex onto the stack and stop when there is no left son of vertex.
2. Pop and process the nodes on stack if zero is popped then exit. If a vertex with right son exists, then set right son of vertex as current vertex and return to step one.

CURRENT VERTEX	STACK	PROCESSED NODES	REMARKS
A	0		PUSH 0
	0 A B D G K		PUSH the left most path of A
K	0 A B D G	K	POP K
G	0 A B D	K G	POP G since K has no right son
D	0 A B	K G D	POP D since G has no right son
H	0 A B	K G D	Make the right son of D as vertex
	0 A B H L	K G D	PUSH the leftmost path of H
L	0 A B H	K G D L	POP L
H	0 A B	K G D L H	POP H since L has no right son
M	0 A B	K G D L H	Make the right son of H as vertex
	0 A B M	K G D L H	PUSH the left most path of M
M	0 A B	K G D L H M	POP M
B	0 A	K G D L H M B	POP B since M has no right son
A	0	K G D L H M B A	Make the right son of A as vertex
C	0 C E	K G D L H M B A	PUSH the left most path of C
E	0 C	K G D L H M B A E	POP E
C	0	K G D L H M B A E C	Stop since stack is empty

Postorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex. At each vertex of path push vertex on to stack and if vertex has a right son push -(right son of vertex) onto stack.
2. Pop and process the positive nodes (left nodes). If zero is popped then exit. If a negative node is popped, then ignore the sign and return to step one.

CURRENT VERTEX	STACK	PROCESSED NODES	REMARKS
A	0		PUSH 0
	0 A -C B D -H G K		PUSH the left most path of A with a -ve for right sons
	0 A -C B D -H	K G	POP all +ve nodes K and G
H	0 A -C B D	K G	Pop H

	0 A -C B D H -M L	K G	PUSH the left most path of H with a -ve for right sons
L	0 A -C B D H -M	K G L	POP all +ve nodes L
M	0 A -C B D H	K G L	Pop M
	0 A -C B D H M	K G L	PUSH the left most path of M with a -ve for right sons
	0 A -C	K G L M H D B	POP all +ve nodes M, H, D and B
C	0 A	K G L M H D B	Pop C
	0 A C E	K G L M H D B	PUSH the left most path of C with a -ve for right sons
	0	K G L M H D B E C A	POP all +ve nodes E, C and A
	0	K G L M H D B E C A	Stop since stack is empty

Preorder Traversal:

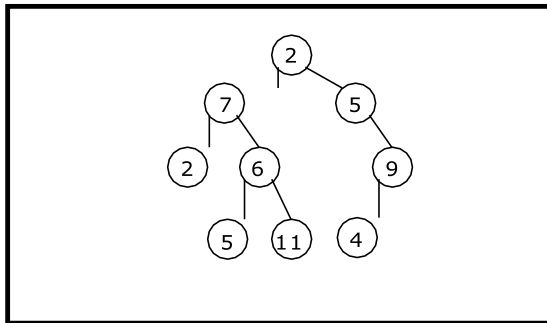
Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path by pushing the right son of vertex onto stack, if any and process each vertex. The traversing ends after a vertex with no left child exists.
2. Pop the vertex from stack, if vertex $\neq 0$ then return to step one otherwise exit.

CURRENT VERTEX	STACK	PROCESSED NODES	REMARKS
A	0		PUSH 0
	0 C H	A B D G K	PUSH the right son of each vertex onto stack and process each vertex in the left most path
H	0 C	A B D G K	POP H
	0 C M	A B D G K H L	PUSH the right son of each vertex onto stack and process each vertex in the left most path
M	0 C	A B D G K H L	POP M
	0 C	A B D G K H L M	PUSH the right son of each vertex onto stack and process each vertex in the left most path; M has no left path
C	0	A B D G K H L M	Pop C
	0	A B D G K H L M C E	PUSH the right son of each vertex onto stack and process each vertex in the left most path; C has no right son on the left most path
	0	A B D G K H L M C E	Stop since stack is empty

Example 2:

Traverse the following binary tree in pre, post and inorder using non-recursive traversing algorithm.



Binary Tree

- Preorder traversal yields:
2 , 7, 2, 6, 5, 11 , 5, 9, 4
- Postorder traversal yields:
2 , 5, 11 , 6, 7, 4, 9, 5, 2
- Inorder traversal yields:
2 , 7, 5, 6, 11 , 2, 5, 4, 9

Pre, Post and In order Traversing

Inorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex, pushing each vertex onto the stack and stop when there is no left son of vertex.
2. Pop and process the nodes on stack if zero is popped then exit. If a vertex with right son exists, then set right son of vertex as current vertex and return to step one.

CURRENT VERTEX	STACK	PROCESSED NODES	REMARKS
2	0		
	0 2 7 2		
2	0 2 7	2	
7	0 2	2 7	
6	0 2 6 5	2 7	
5	0 2 6	2 7 5	
6	0 2	2 7 5 6	
11	0 2 11	2 7 5 6	
11	0 2	2 7 5 6 11	
2	0	2 7 5 6 11 2	
5	0 5	2 7 5 6 11 2	
5	0	2 7 5 6 11 2 5	
9	0 9 4	2 7 5 6 11 2 5	
4	0 9	2 7 5 6 11 2 5 4	
9	0	2 7 5 6 11 2 5 4 9	Stop since stack is empty

Postorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex. At each vertex of path push vertex on to stack and if vertex has a right son push -(right son of vertex) onto stack.
2. Pop and process the positive nodes (left nodes). If zero is popped then exit. If a negative node is popped, then ignore the sign and return to step one.

CURRENT VERTEX	STACK	PROCESSED NODES	REMARKS
2	0		
	0 2 -5 7 -6 2		
2	0 2 -5 7 -6	2	
6	0 2 -5 7	2	
	0 2 -5 7 6 -11 5	2	
5	0 2 -5 7 6 -11	2 5	
11	0 2 -5 7 6 11	2 5	
	0 2 -5	2 5 11 6 7	
5	0 2 5 -9	2 5 11 6 7	
9	0 2 5 9 4	2 5 11 6 7	
	0	2 5 11 6 7 4 9 5 2	Stop since stack is empty

Preorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path by pushing the right son of vertex onto stack, if any and process each vertex. The traversing ends after a vertex with no left child exists.
2. Pop the vertex from stack, if vertex $\neq 0$ then return to step one otherwise exit.

CURRENT VERTEX	STACK	PROCESSED NODES	REMARKS
2	0		
	0 5 6	2 7 2	
6	0 5 11	2 7 2 6 5	
11	0 5	2 7 2 6 5 11	
	0 5	2 7 2 6 5 11	
5	0 9	2 7 2 6 5 11 5	
9	0	2 7 2 6 5 11 5 9 4	
	0	2 7 2 6 5 11 5 9 4	Stop since stack is empty

5.4. Expression Trees:

Expression tree is a binary tree, because all of the operations are binary. It is also possible for a node to have only one child, as is the case with the unary minus operator. The leaves of an expression tree are operands, such as constants or variable names, and the other (non leaf) nodes contain operators.

Once an expression tree is constructed we can traverse it in three ways:

- Inorder Traversal
- Preorder Traversal
- Postorder Traversal

Figure 5.4.1 shows some more expression trees that represent arithmetic expressions given in infix form.

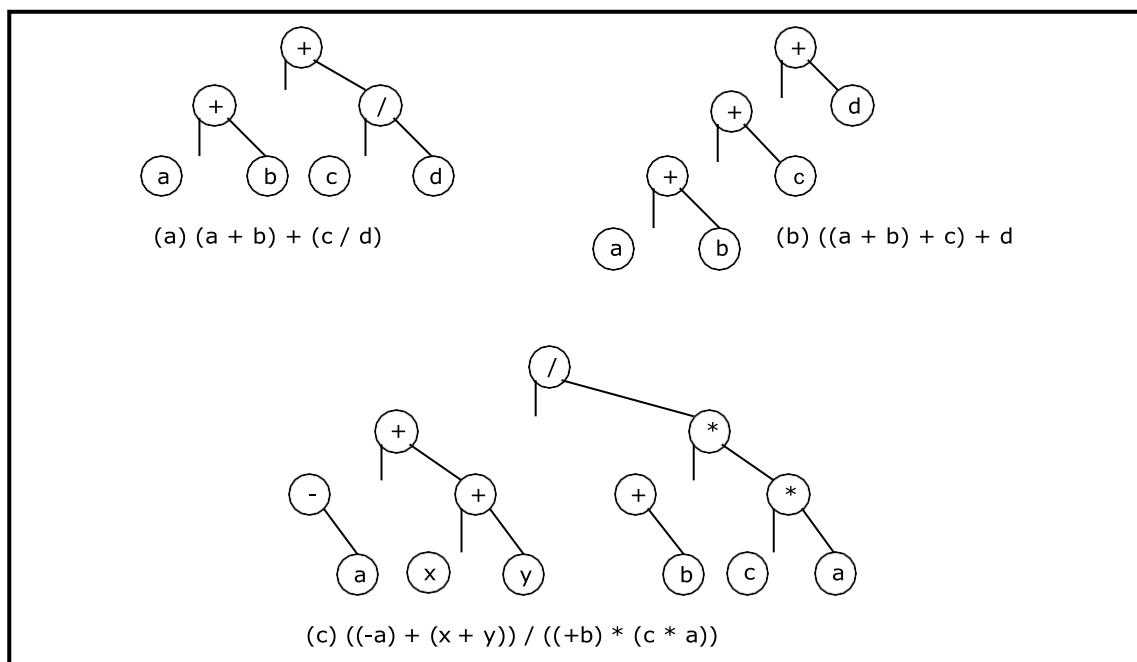


Figure 5.4.1 Expression Trees

An expression tree can be generated for the infix and postfix expressions.

An algorithm to convert a postfix expression into an expression tree is as follows:

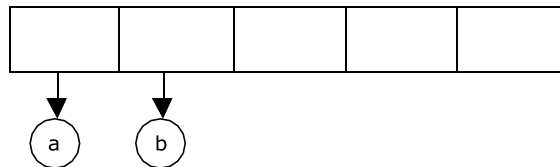
1. Read the expression one symbol at a time.
2. If the symbol is an operand, we create a one-node tree and push a pointer to it onto a stack.
3. If the symbol is an operator, we pop pointers to two trees T1 and T2 from the stack (T1 is popped first) and form a new tree whose root is the operator and whose left and right children point to T2 and T1 respectively. A pointer to this new tree is then pushed onto the stack.

Example 1:

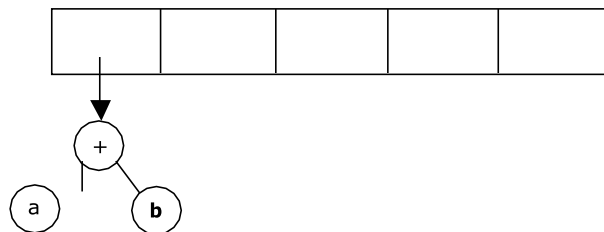
Construct an expression tree for the postfix expression: $a\ b\ +\ c\ d\ e\ +\ *\ *$

Solution:

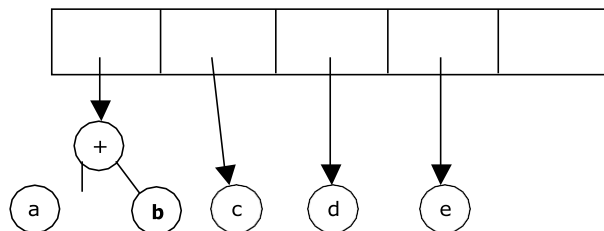
The first two symbols are operands, so we create one-node trees and push pointers to them onto a stack.



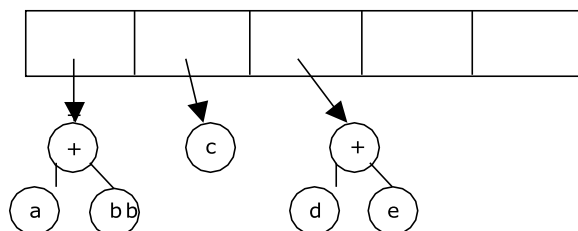
Next, a '+' is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.



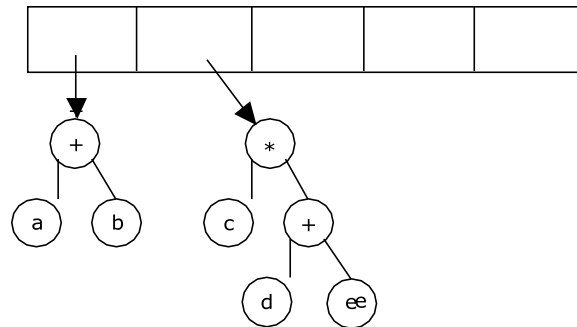
Next, c, d, and e are read, and for each one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.



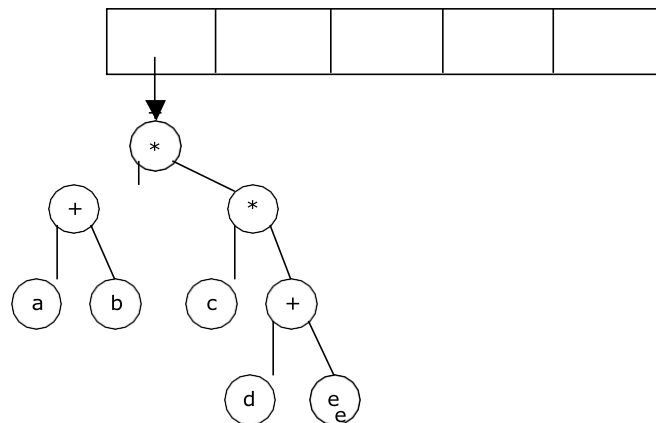
Now a '+' is read, so two trees are merged.



Continuing, a '*' is read, so we pop two tree pointers and form a new tree with a '*' as root.



Finally, the last symbol is read, two trees are merged, and a pointer to the final tree is left on the stack.



For the above tree:

Inorder form of the expression: $a + b * c * d + e$

Preorder form of the expression: $* + a b * c + d e$

Postorder form of the expression: $a b + c d e + * *$

Example 2:

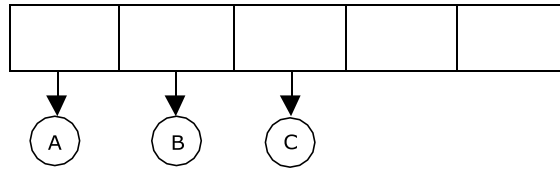
Construct an expression tree for the arithmetic expression:

$$(A + B * C) - ((D * E + F) / G)$$

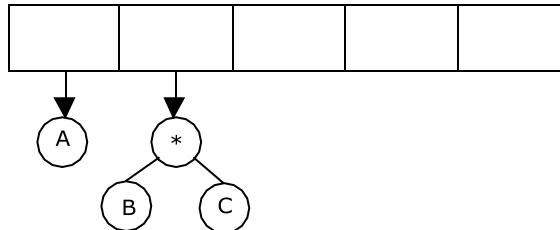
Solution:

First convert the infix expression into postfix notation. Postfix notation of the arithmetic expression is: $A B C * + D E * F + G / -$

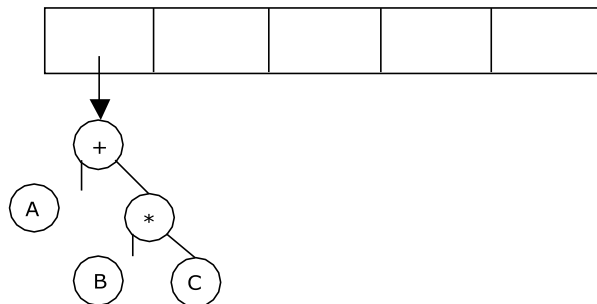
The first three symbols are operands, so we create one-node trees and pointers to three nodes pushed onto the stack.



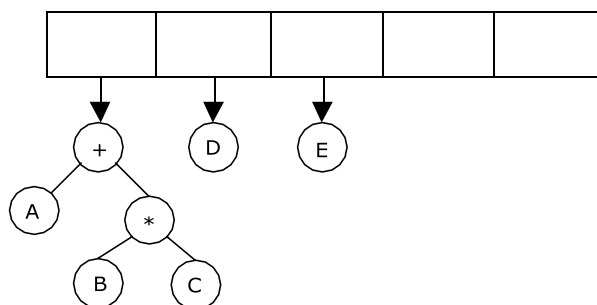
Next, a '*' is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.



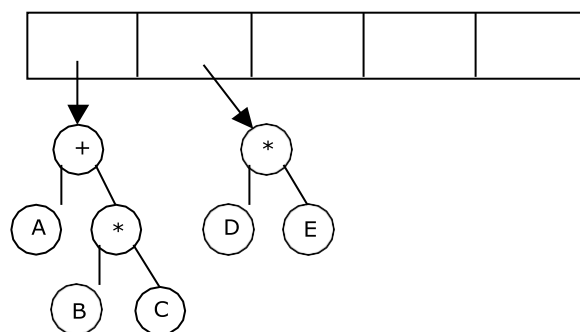
Next, a '+' is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.



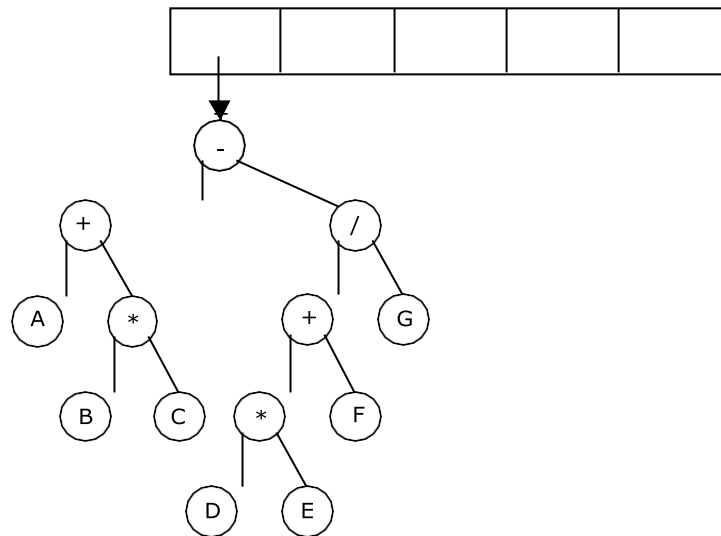
Next, D and E are read, and for each one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.



Continuing, a '*' is read, so we pop two tree pointers and form a new tree with a '*' as root.



Proceeding similar to the previous steps, finally, when the last symbol is read, the expression tree is as follows:



5.4.1. Converting expressions with expression trees:

Let us convert the following expressions from one type to another. These can be as follows:

1. Postfix to infix
2. Postfix to prefix
3. Prefix to infix
4. Prefix to postfix

1. Postfix to Infix:

The following algorithm works for the expressions whose infix form does not require parenthesis to override conventional precedence of operators.

- A. Create the expression tree from the postfix expression
- B. Run inorder traversal on the tree.

2. Postfix to Prefix:

The following algorithm works for the expressions to convert postfix to prefix:

- A. Create the expression tree from the postfix expression
- B. Run preorder traversal on the tree.

3. Prefix to Infix:

The following algorithm works for the expressions whose infix form does not require parenthesis to override conventional precedence of operators.

- A. Create the expression tree from the prefix expression
- B. Run inorder traversal on the tree.

4. Prefix to postfix:

The following algorithm works for the expressions to convert postfix to prefix:

- A. Create the expression tree from the prefix expression
- B. Run postorder traversal on the tree.

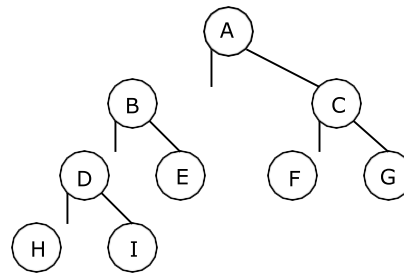
5.5. Threaded Binary Tree:

The linked representation of any binary tree has more null links than actual pointers. If there are $2n$ total links, there are $n+1$ null links. A clever way to make use of these null links has been devised by A.J. Perlis and C. Thornton.

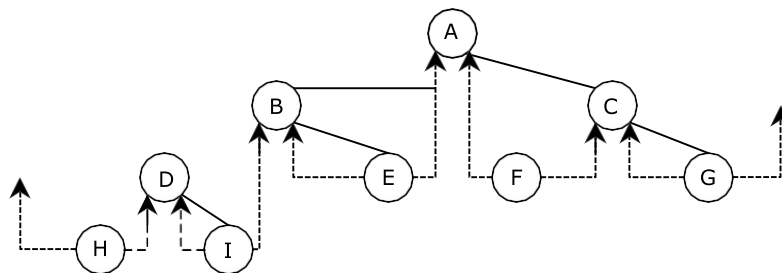
Their idea is to replace the null links by pointers called Threads to other nodes in the tree.

If the $RCHILD(p)$ is normally equal to zero, we will replace it by a pointer to the node which would be printed after P when traversing the tree in inorder.

A null $LCHILD$ link at node P is replaced by a pointer to the node which immediately precedes node P in inorder. For example, Let us consider the tree:



The Threaded Tree corresponding to the above tree is:



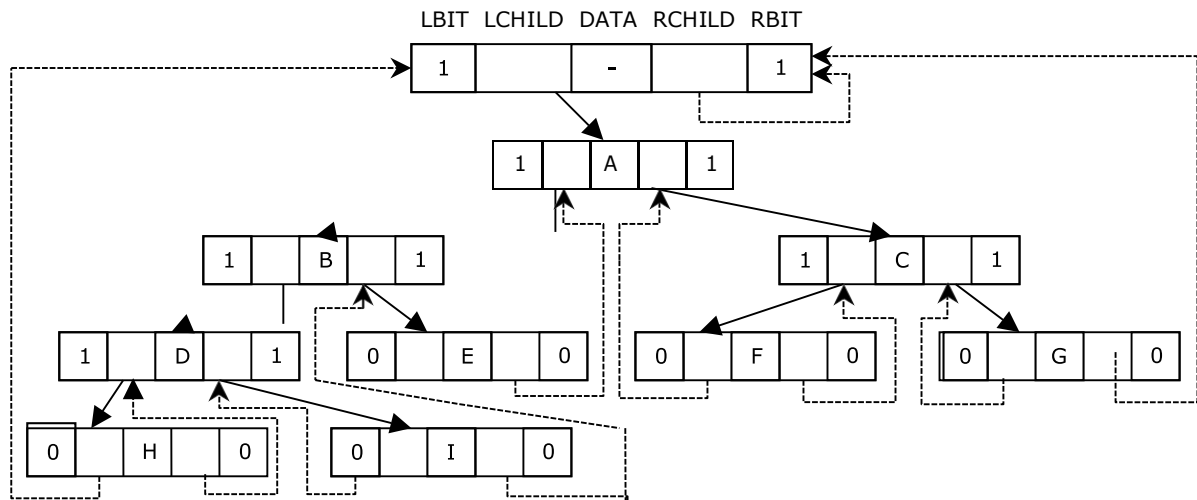
The tree has 9 nodes and 10 null links which have been replaced by Threads. If we traverse T in inorder the nodes will be visited in the order H D I B E A F C G.

For example, node 'E' has a predecessor Thread which points to 'B' and a successor Thread which points to 'A'. In memory representation Threads and normal pointers are distinguished between as by adding two extra one bit fields LBIT and RBIT.

$LBIT(P) = 1$ if $LCHILD(P)$ is a normal pointer
 $LBIT(P) = 0$ if $LCHILD(P)$ is a Thread

$RBIT(P) = 1$ if $RCHILD(P)$ is a normal pointer
 $RBIT(P) = 0$ if $RCHILD(P)$ is a Thread

In the above figure two threads have been left dangling in LCHILD(H) and RCHILD(G). In order to have no loose Threads we will assume a head node for all threaded binary trees. The Complete memory representation for the tree is as follows. The tree T is the left sub-tree of the head node.



5.6. Binary Search Tree:

A binary search tree is a binary tree. It may be empty. If it is not empty then it satisfies the following properties:

1. Every element has a key and no two elements have the same key.
2. The keys in the left subtree are smaller than the key in the root.
3. The keys in the right subtree are larger than the key in the root.
4. The left and right subtrees are also binary search trees.

Figure 5.2.5(a) is a binary search tree, whereas figure 5.2.5(b) is not a binary search tree.

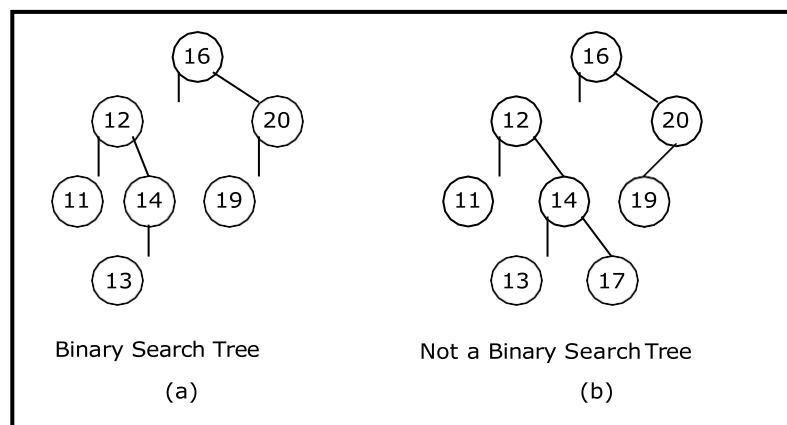


Figure 5.2.5. Examples of binary search trees

5.7. General Trees (m-ary tree):

If in a tree, the outdegree of every node is less than or equal to m , the tree is called general tree. The general tree is also called as an m -ary tree. If the outdegree of every node is exactly equal to m or zero then the tree is called a *full or complete m-ary tree*. For $m = 2$, the trees are called *binary* and *full binary trees*.

Differences between trees and binary trees:

TREE	BINARY TREE
Each element in a tree can have any number of subtrees.	Each element in a binary tree has at most two subtrees.
The subtrees in a tree are unordered.	The subtrees of each element in a binary tree are ordered (i.e. we distinguish between left and right subtrees).

5.7.1. Converting a m -ary tree (general tree) to a binary tree:

There is a one-to-one mapping between general ordered trees and binary trees. So, every tree can be uniquely represented by a binary tree. Furthermore, a forest can also be represented by a binary tree.

Conversion from general tree to binary can be done in two stages.

Stage 1:

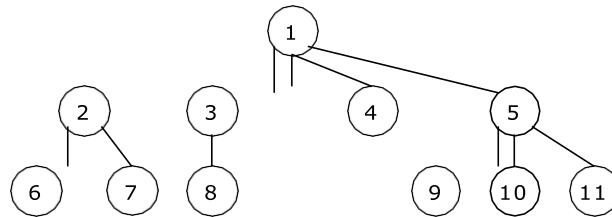
- As a first step, we delete all the branches originating in every node except the left most branch.
- We draw edges from a node to the node on the right, if any, which is situated at the same level.

Stage 2:

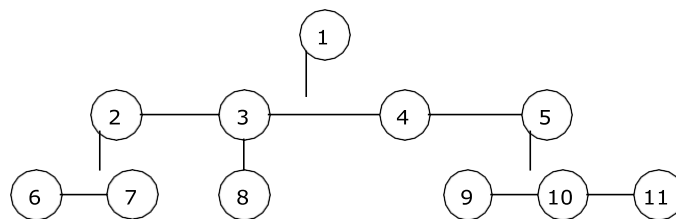
- Once this is done then for any particular node, we choose its left and right sons in the following manner:
 - The left son is the node, which is immediately below the given node, and the right son is the node to the immediate right of the given node on the same horizontal line. Such a binary tree will not have a right subtree.

Example 1:

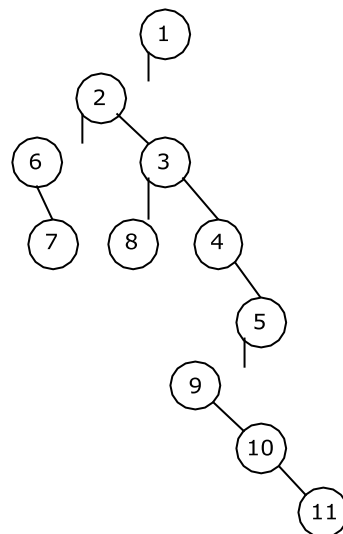
Convert the following ordered tree into a binary tree:

**Solution:**

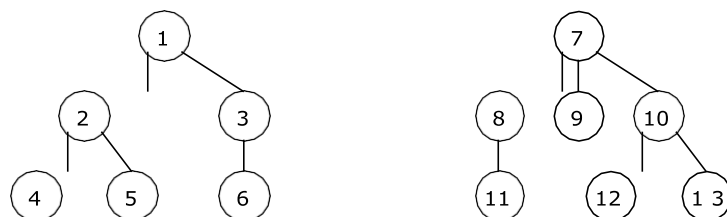
Stage 1 tree by using the above mentioned procedure is as follows:



Stage 2 tree by using the above mentioned procedure is as follows:

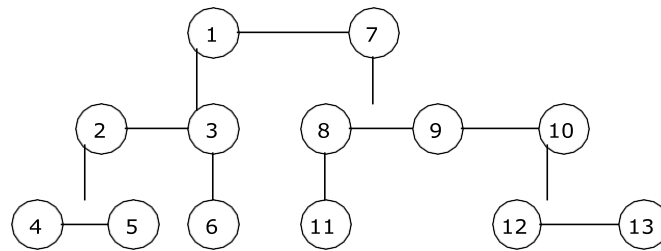
**Example 2:**

Construct a unique binary tree from the given forest.

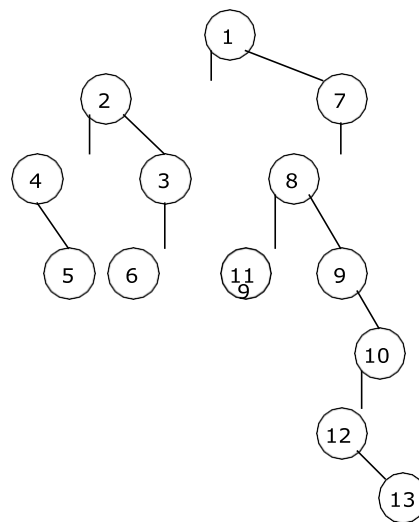


Solution:

Stage 1 tree by using the above mentioned procedure is as follows:

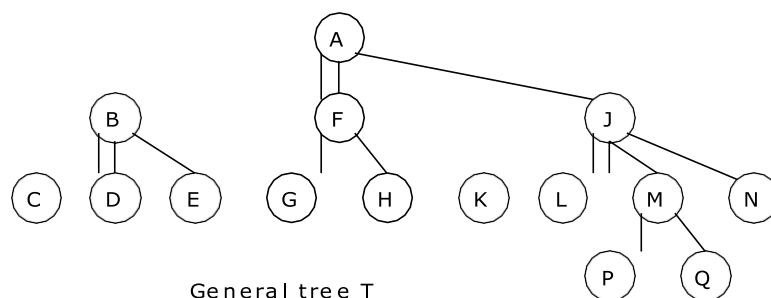


Stage 2 tree by using the above mentioned procedure is as follows (binary tree representation of forest):

**Example 3:**

For the general tree shown below:

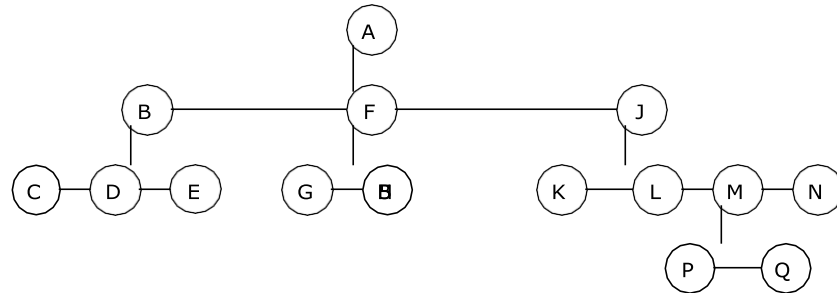
1. Find the corresponding binary tree T' .
2. Find the preorder traversal and the postorder traversal of T .
3. Find the preorder, inorder and postorder traversals of T' .
4. Compare them with the preorder and postorder traversals obtained for T' with the general tree T .



Solution:

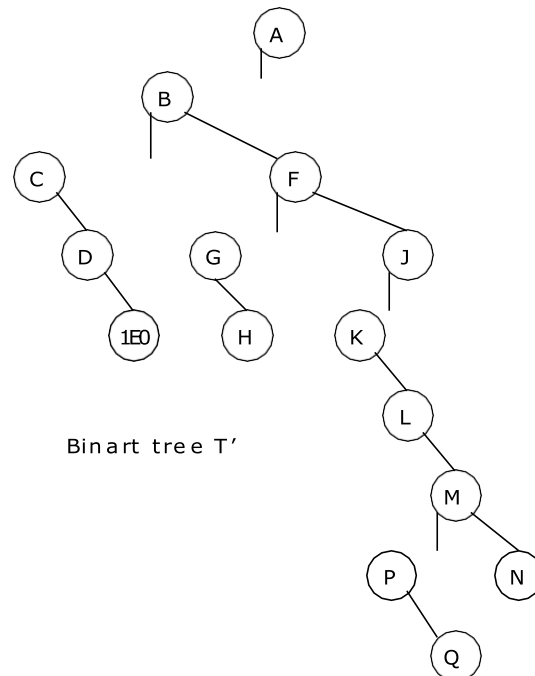
1. Stage 1:

The tree by using the above-mentioned procedure is as follows:



Stage 2:

The binary tree by using the above-mentioned procedure is as follows:



2. Suppose T is a general tree with root R and subtrees T_1, T_2, \dots, T_M . The preorder traversal and the postorder traversal of T are:

Preorder: 1) Process the root R.
2) Traverse the subtree T_1, T_2, \dots, T_M in preorder.

Postorder: 1) Traverse the subtree T_1, T_2, \dots, T_M in postorder.
2) Process the root R.

The tree T has the root A and subtrees T_1, T_2 and T_3 such that:

T_1 consists of nodes B, C, D and E.

T_2 consists of nodes F, G and H.

T_3 consists of nodes J, K, L, M, N, P and Q.

- A. The preorder traversal of T consists of the following steps:
- (i) Process root A.
 - (ii) Traverse T_1 in preorder: Process nodes B, C, D, E.
 - (iii) Traverse T_2 in preorder: Process nodes F, G, H.
 - (iv) Traverse T_3 in preorder: Process nodes J, K, L, M, P, Q, N.

The preorder traversal of T is as follows:

A, B, C, D, E, F, G, H, J, K, L, M, P, Q, N

- B. The postorder traversal of T consists of the following steps:
- (i) Traverse T_1 in postorder: Process nodes C, D, E, B.
 - (ii) Traverse T_2 in postorder: Process nodes G, H, F.
 - (iii) Traverse T_3 in postorder: Process nodes K, L, P, Q, M, N, J.
 - (iv) Process root A.

The postorder traversal of T is as follows:

C, D, E, B, G, H, F, K, L, P, Q, M, N, J, A

3. The preorder, inorder and postorder traversals of the binary tree T' are as follows:

Preorder: A, B, C, D, E, F, G, H, J, K, M, P, Q, N

Inorder: C, D, E, B, G, H, F, K, L, P, Q, M, N, J, A

Postorder: E, D, C, H, G, Q, P, N, M, L, K, J, F, B, A

4. Comparing the preorder and postorder traversals of T' with the general tree T:

We can observe that the preorder of the binary tree T' is identical to the preorder of the general T.

The inorder traversal of the binary tree T' is identical to the postorder traversal of the general tree T.

There is no natural traversal of the general tree T which corresponds to the postorder traversal of its corresponding binary tree T' .

5.8. Search and Traversal Techniques for m-ary trees:

Search involves visiting nodes in a tree in a systematic manner, and may or may not result into a visit to all nodes. When the search necessarily involved the examination of every vertex in the tree, it is called the traversal. Traversing of a tree can be done in two ways.

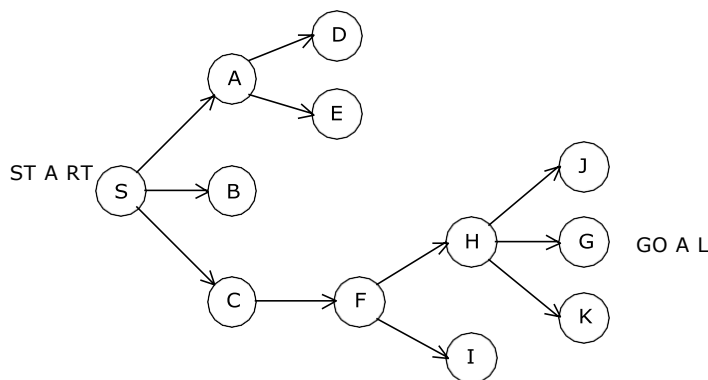
1. Depth first search or traversal.
2. Breadth first search or traversal.

5.8.1. Depth first search:

In Depth first search, we begin with root as a start state, then some successor of the start state, then some successor of that state, then some successor of that and so on, trying to reach a goal state. One simple way to implement depth first search is to use a stack data structure consisting of root node as a start state.

If depth first search reaches a state S without successors, or if all the successors of a state S have been chosen (visited) and a goal state has not get been found, then it "backs up" that means it goes to the immediately previous state or predecessor formally, the state whose successor was 'S' originally.

To illustrate this let us consider the tree shown below.



Suppose S is the start and G is the only goal state. Depth first search will first visit S, then A, then D. But D has no successors, so we must back up to A and try its second successor, E. But this doesn't have any successors either, so we back up to A again. But now we have tried all the successors of A and haven't found the goal state G so we must back to 'S'. Now 'S' has a second successor, B. But B has no successors, so we back up to S again and choose its third successor, C. C has one successor, F. The first successor of F is H, and the first of H is J. J doesn't have any successors, so we back up to H and try its second successor. And that's G, the only goal state.

So the solution path to the goal is S, C, F, H and G and the states considered were in order S, A, D, E, B, C, F, H, J, G.

Disadvantages:

1. It works very fine when search graphs are trees or lattices, but can get stuck in an infinite loop on graphs. This is because depth first search can travel around a cycle in the graph forever.

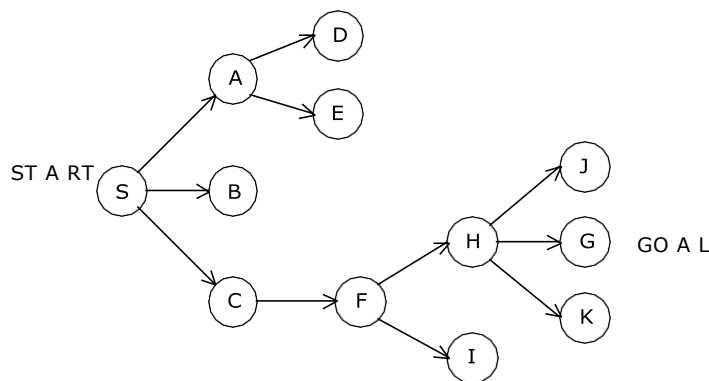
To eliminate this keep a list of states previously visited, and never permit search to return to any of them.

2. We cannot come up with shortest solution to the problem.

5.8.2. Breadth first search:

Breadth-first search starts at root node S and "discovers" which vertices are reachable from S. Breadth-first search discovers vertices in increasing order of distance. Breadth-first search is named because it visits vertices across the entire breadth.

To illustrate this let us consider the following tree:



Breadth first search finds states level by level. Here we first check all the immediate successors of the start state. Then all the immediate successors of these, then all the immediate successors of these, and so on until we find a goal node. Suppose S is the start state and G is the goal state. In the figure, start state S is at level 0; A, B and C are at level 1; D, E and F at level 2; H and I at level 3; and J, G and K at level 4.

So breadth first search, will consider in order S, A, B, C, D, E, F, H, I, J and G and then stop because it has reached the goal node.

Breadth first search does not have the danger of infinite loops as we consider states in order of increasing number of branches (level) from the start state.

One simple way to implement breadth first search is to use a queue data structure consisting of just a start state.

5.9. Sparse Matrices:

A sparse matrix is a two-dimensional array having the value of majority elements as null. The density of the matrix is the number of non-zero elements divided by the total number of matrix elements. The matrices with very low density are often good for use of the sparse format. For example,

$$A = \begin{bmatrix} 0 & 0 & 0 & 5 \\ 0 & 2 & 0 & 0 \\ 1 & 3 & 0 & 0 \\ 0 & 0 & 4 & 0 \end{bmatrix}$$

As far as the storage of a sparse matrix is concerned, storing of null elements is nothing but wastage of memory. So we should devise technique such that only non-null elements will be stored. The matrix A produces:

$$S = \begin{matrix} (3, 1) & 1 \\ (2, 2) & 2 \\ (3, 2) & 3 \\ (4, 3) & 4 \\ (1, 4) & 5 \end{matrix}$$

The printed output lists the non-zero elements of S, together with their row and column indices. The elements are sorted by columns, reflecting the internal data structure.

In large number of applications, sparse matrices are involved. One approach is to use the linked list.

The program to represent sparse matrix:

```
/*      Check whether the given matrix is sparse matrix or not, if so then print in
        alternative form for storage.      */

#include <stdio.h>
#include <conio.h>

main()
{
    int matrix[20][20], m, n, total_elements, total_zeros = 0, i, j;
    clrscr();
    printf("\n Enter Number of rows and columns: ");
    scanf("%d %d",&m, &n);
    total_elements = m * n;
    printf("\n Enter data for sparse matrix: ");
    for(i = 0; i < m ; i++)
    {
        for( j = 0; j < n ; j++)
        {
            scanf("%d", &matrix[i][j]);
            if( matrix[i][j] == 0)
            {
                total_zeros++;
            }
        }
    }
    if(total_zeros > total_elements/2 )
    {
        printf("\n Given Matrix is Sparse Matrix..");
        printf("\n The Representaion of Sparse Matrix is: \n");
        printf("\n Row \t Col \t Value ");
        for(i = 0; i < m ; i++)
        {
            for( j = 0; j < n ; j++)
            {
                if( matrix[i][j] != 0)
                {
                    printf("\n %d \t %d \t %d",i,j,matrix[i][j]);
                }
            }
        }
    }
    else
        printf("\n Given Matrix is Not a Sparse Matrix..");
}
```

EXERCISES

1. How many different binary trees can be made from three nodes that contain the key value 1, 2, and 3?
2.
 - a. Draw all the possible binary trees that have four leaves and all the nonleaf nodes have no children.
 - b. Show what would be printed by each of the following.
An inorder traversal of the tree
A postorder traversal of the tree
A preorder traversal of the tree
3.
 - a. Draw the binary search tree whose elements are inserted in the following order:
50 72 96 94 107 26 12 11 9 2 10 25 51 16 17 95
 - b. What is the height of the tree?
 - c. What nodes are on level?
 - d. Which levels have the maximum number of nodes that they could contain?
 - e. What is the maximum height of a binary search tree containing these nodes?
Draw such a tree?
 - f. What is the minimum height of a binary search tree containing these nodes?
Draw such a tree?
 - g. Show how the tree would look after the deletion of 29, 59 and 47?
 - h. Show how the (original) tree would look after the insertion of nodes containing 63, 77, 76, 48, 9 and 10 (in that order).
4. Write a "C" function to determine the height of a binary tree.
5. Write a "C" function to count the number of leaf nodes in a binary tree.
6. Write a "C" function to swap a binary tree.
7. Write a "C" function to compute the maximum number of nodes in any level of a binary tree. The maximum number of nodes in any level of a binary tree is also called the width of the tree.
8. Construct two binary trees so that their postorder traversal sequences are the same.
9. Write a "C" function to compute the internal path length of a binary tree.
10. Write a "C" function to compute the external path length of a binary tree.
11. Prove that every node in a tree except the root node has a unique parent.
12. Write a "C" function to reconstruct a binary tree from its preorder and inorder traversal sequences.
13. Prove that the inorder and postorder traversal sequences of a binary tree uniquely characterize the binary tree. Write a "C" function to reconstruct a binary tree from its postorder and inorder traversal sequences.

14. Build the binary tree from the given traversal techniques:

A. Inorder: g d h b e i a f j c
Preorder: a b d g h e i c f j

B. Inorder: g d h b e i a f j c
Postorder: g h d i e b j f c a

C. Inorder: g d h b e i a f j c
Level order: a b c d e f g h i j

15. Build the binary tree from the given traversal techniques:

A. Inorder: n1 n2 n3 n4 n5 n6 n7 n8 n9
Preorder: n6 n2 n1 n4 n3 n5 n9 n7 n8

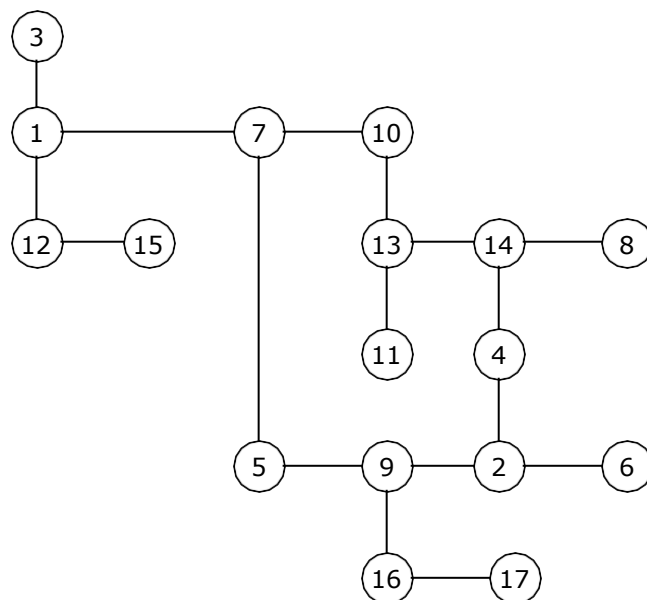
B. Inorder: n1 n2 n3 n4 n5 n6 n7 n8 n9
Postorder: n1 n3 n5 n4 n2 n8 n7 n9 n6

C. Inorder: n1 n2 n3 n4 n5 n6 n7 n8 n9
Level order: n6 n2 n9 n1 n4 n7 n3 n5 n8

16. Build the binary tree for the given inorder and preorder traversals:

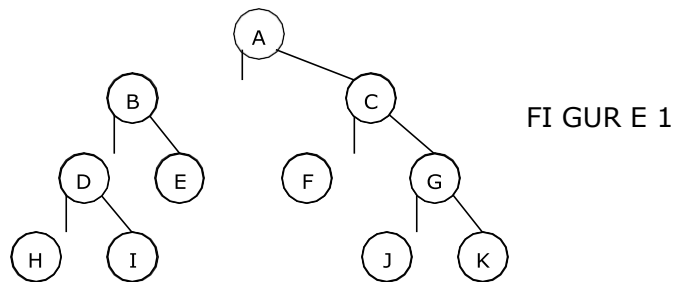
Inorder: E A C K F H D B G
Preorder: F A E K C D H G B

17. Convert the following general tree represented as a binary tree:

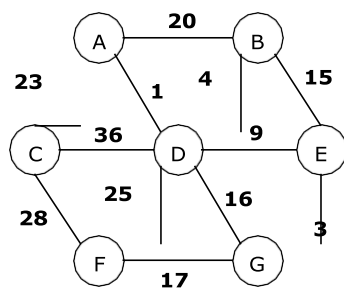


Multiple Choice Questions

1. The node that has no children is referred as: [C]
A. Parent node
B. Root node
C. Leaf node
D. Siblings
2. A binary tree in which all the leaves are on the same level is called as: [B]
A. Complete binary tree
B. Full binary tree
C. Strictly binary tree
D. Binary search tree
3. How can the graphs be represented? [C]
A. Adjacency matrix
B. Adjacency list
C. Incidence matrix
D. All of the above
4. The children of a same parent node are called as: [C]
A. adjacent node
B. non-leaf node
C. Siblings
D. leaf node
5. A tree with n vertices, consists of _____ edges. [A]
A. $n - 1$
B. $n - 2$
C. n
D. $\log n$
6. The maximum number of nodes at any level is: [B]
A. n
B. 2^n
C. $n + 1$
D. $2n$



7. For the Binary tree shown in fig. 1, the in-order traversal sequence is: [C]
A. A B C D E F G H I J K C. H D I B E A F C J G K
B. H I D E B F J K G C A D. A B D H I E C F G J K
8. For the Binary tree shown in fig. 1, the pre-order traversal sequence is: [D]
A. A B C D E F G H I J K C. H D I B E A F C J G K
B. H I D E B F J K G C A D. A B D H I E C F G J K
9. For the Binary tree shown in fig. 1, the post-order traversal sequence is: [B]
A. A B C D E F G H I J K C. H D I B E A F C J G K
B. H I D E B F J K G C A D. A B D H I E C F G J K



Node	Adjacency List
A	B C D
B	A D E
C	A D F
D	A B C E F G
E	B D G
F	C D G
G	F D E

FIGURE 2 and its adjacency list

10. Which is the correct order for Kruskal’s minimum spanning tree algorithm to add edges to the minimum spanning tree for the figure 2 shown above: [B]
- A. (A, B) then (A, C) then (A, D) then (D, E) then (C, F) then (D, G)
 B. (A, D) then (E, G) then (B, D) then (D, E) then (F, G) then (A, C)
 C. both A and B
 D. none of the above
11. For the figure 2 shown above, the cost of the minimal spanning tree is: [A]
- A. 57
 B. 68
 C. 48
 D. 32

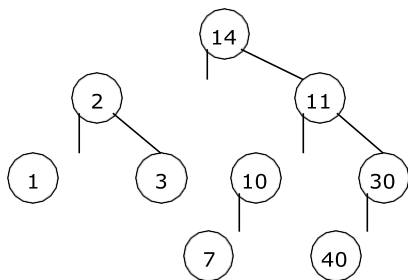


FIGURE 3

12. For the figure 3, how many leaves does it have? [B]
- A. 2
 B. 4
 C. 6
 D. 8
13. For the figure 3, how many of the nodes have at least one sibling? [A]
- A. 5
 B. 6
 C. 7
 D. 8
14. For the figure 3, How many descendants does the root have? [D]
- A. 0
 B. 2
 C. 4
 D. 8
15. For the figure 3, what is the depth of the tree? [B]
- A. 2
 B. 3
 C. 4
 D. 8
16. For the figure 3, which statement is correct? [B]

- A. The tree is neither complete nor full.
- B. The tree is complete but not full.
- C. The tree is full but not complete.
- D. The tree is both full and complete.

[A]

17. There is a tree in the box at the top of this section. What is the order of nodes visited using a pre-order traversal? []
 A. 1 2 3 7 10 11 14 30 40 C. 1 3 2 7 10 40 30 11 14
 B. 1 2 3 14 7 10 11 40 30 D. 14 2 1 3 11 10 7 30 40
18. There is a tree in the box at the top of this section. What is the order of nodes visited using an in-order traversal? []
 A. 1 2 3 7 10 11 14 30 40 C. 1 3 2 7 10 40 30 11 14
 B. 1 2 3 14 7 10 11 40 30 D. 14 2 1 3 11 10 7 30 40
19. There is a tree in the box at the top of this section. What is the order of nodes visited using a post-order traversal? []
 A. 1 2 3 7 10 11 14 30 40 C. 1 3 2 7 10 40 30 11 14
 B. 1 2 3 14 7 10 11 40 30 D. 14 2 1 3 11 10 7 30 40
20. What is the minimum number of nodes in a full binary tree with depth 3? [D]
 A. 3 C. 8
 B. 4 D. 15
21. Select the one true statement. [C]
 A. Every binary tree is either complete or full.
 B. Every complete binary tree is also a full binary tree.
 C. Every full binary tree is also a complete binary tree.
 D. No binary tree is both complete and full.
22. Suppose T is a binary tree with 14 nodes. What is the minimum possible depth of T? [B]
 A. 0 C. 4
 B. 3 D. 5
23. Select the one FALSE statement about binary trees: [A]
 A. Every binary tree has at least one node.
 B. Every non-empty tree has exactly one root node.
 C. Every node has at most two children.
 D. Every non-root node has exactly one parent.
24. Consider the node of a complete binary tree whose value is stored in data[i] for an array implementation. If this node has a right child, where will the right child's value be stored? [C]
 A. data[i+1] C. data[2*i + 1]
 B. data[i+2] D. data[2*i + 2]

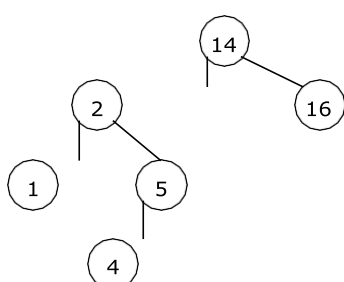
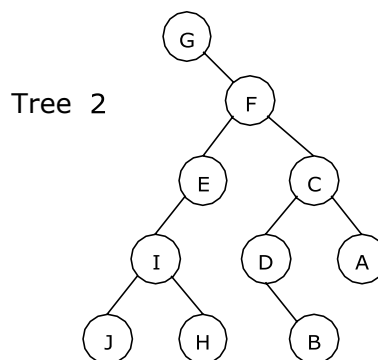
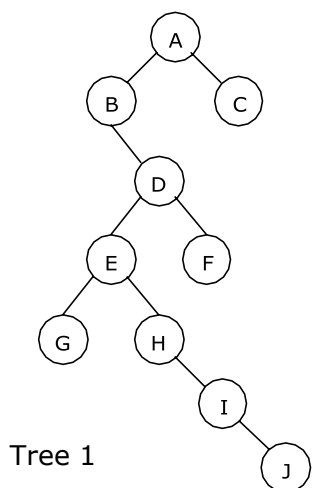


Figure 4

25. For the binary search tree shown in figure 4, Suppose we remove the root, replacing it with something from the left subtree. What will be the new root? [D]

A. 1
B. 2
C. 4

D. 5
E. 16



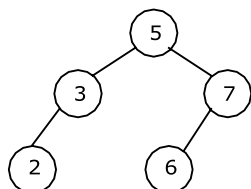
26. Which traversals of tree 1 and tree 2, will produce the same sequence of node names? [C]

A. Preorder, Postorder
B. Postorder, Postorder

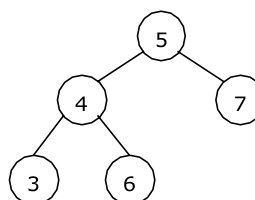
C. Postorder, Inorder
D. Inorder, Inorder

27. Which among the following is not a binary search tree? [C]

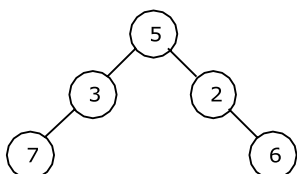
A.



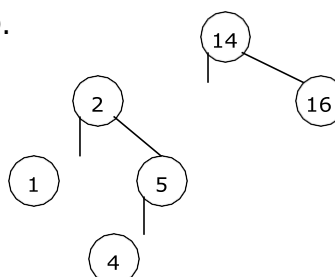
C.



B.



D.



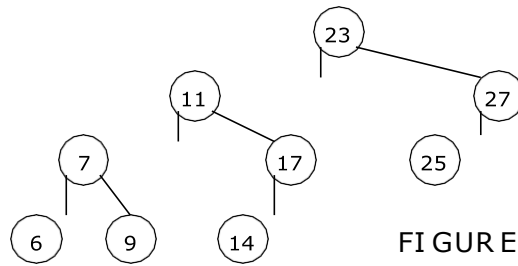


FIGURE 5

28. For the binary search tree shown in figure 5, after deleting 23 from the binary search tree what node will be at the root? []
 A. 11
 B. 25
 C. 27
 D. 14
29. For the binary search tree shown in figure 5, after deleting 23 from the binary search tree what parent → child pair does not occur in the tree? [B]
 A. 25 → 27
 B. 27 → 11
 C. 11 → 7
 D. 7 → 9
30. The number of nodes in a complete binary tree of depth d is: [B]
 A. $2d$
 B. $2^k - 1$
 C. 2^k
 D. none of the above
31. The depth of a complete binary tree with n nodes is: [C]
 A. $\log n$
 B. n^2
 C. $\leq \log_2 n + 1$
 D. $2n$
32. If the inorder and preorder traversal of a binary tree are D, B, F, E, G, H, A, C and A, B, D, E, F, G, H, C respectively then, the postorder traversal of that tree is: [A]
 A. D, F, H, G, E, B, C, A
 B. D, F, G, A, B, C, H, E
 C. F, H, D, G, E, B, C, A
 D. D, F, H, G, E, B, C, A
33. The data structure used by level order traversal of binary tree is: [A]
 A. Queue
 B. Stack
 C. linked list
 D. none of the above

Introduction to Graphs:

Graphs

Graph G is a pair (V, E) , where V is a finite set of vertices and E is a finite set of edges. We will often denote $n = |V|$, $e = |E|$.

A graph is generally displayed as figure 6.5.1, in which the vertices are represented by circles and the edges by lines.

An edge with an orientation (i.e., arrow head) is a directed edge, while an edge with no orientation is our undirected edge.

If all the edges in a graph are undirected, then the graph is an undirected graph. The graph in figure 6.5.1(a) is an undirected graph. If all the edges are directed; then the graph is a directed graph. The graph of figure 6.5.1(b) is a directed graph. A directed graph is also called as digraph. A graph G is connected if and only if there is a simple path between any two nodes in G .

A graph G is said to be complete if every node a in G is adjacent to every other node v in G . A complete graph with n nodes will have $n(n-1)/2$ edges. For example, Figure 6.5.1.(a) and figure 6.5.1.(d) are complete graphs.

A directed graph G is said to be connected, or strongly connected, if for each pair (u, v) for nodes in G there is a path from u to v and also a path from v to u . On the other hand, G is said to be unilaterally connected if for any pair (u, v) of nodes in G there is a path from u to v or a path from v to u . For example, the digraph shown in figure 6.5.1 (e) is strongly connected.

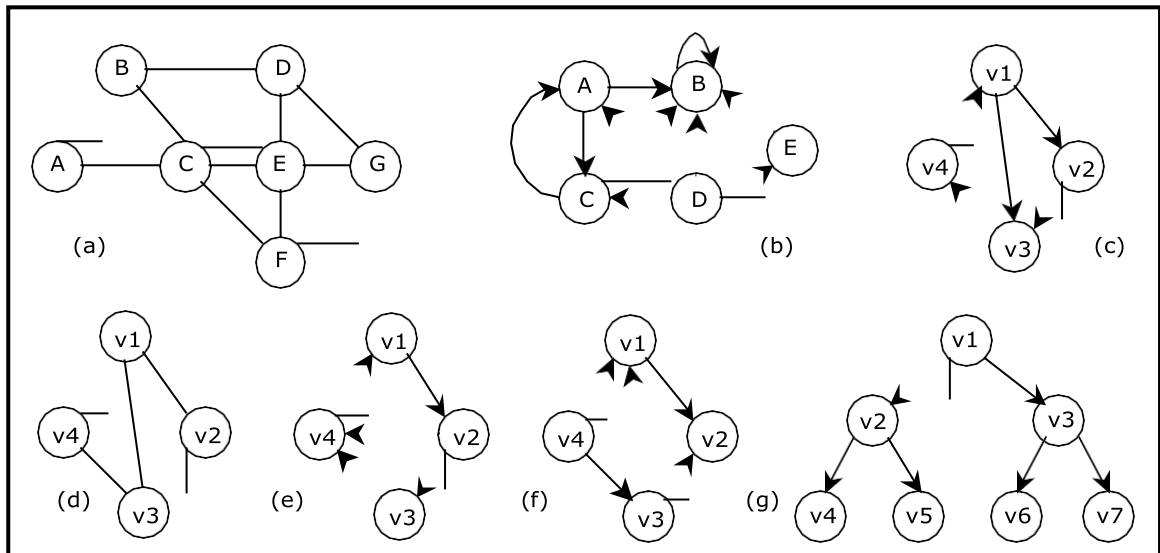


Figure 6.5.1 Various Graphs

We can assign weight function to the edges: $w_G(e)$ is a weight of edge $e \in E$. The graph which has such function assigned is called weighted graph.

The number of incoming edges to a vertex v is called in-degree of the vertex (denote $\text{indeg}(v)$). The number of outgoing edges from a vertex is called out-degree (denote $\text{outdeg}(v)$). For example, let us consider the digraph shown in figure 6.5.1(f),

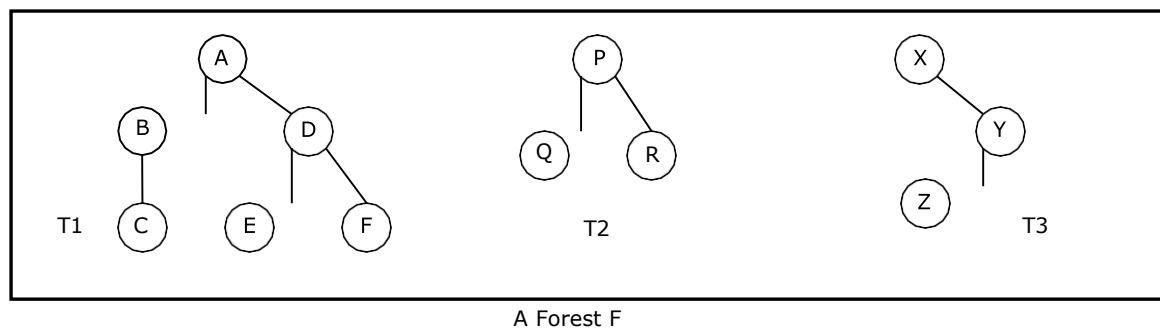
$$\begin{array}{ll} \text{indegree}(v_1) = 2 & \text{outdegree}(v_1) = 1 \\ \text{indegree}(v_2) = 2 & \text{outdegree}(v_2) = 0 \end{array}$$

A path is a sequence of vertices (v_1, v_2, \dots, v_k) , where for all i , $(v_i, v_{i+1}) \in E$. A path is simple if all vertices in the path are distinct. If there is a path containing one or more edges which starts from a vertex V_i and terminates into the same vertex then the path is known as a cycle. For example, there is a cycle in figure 6.5.1(a), figure 6.5.1(c) and figure 6.5.1(d).

If a graph (digraph) does not have any cycle then it is called **acyclic graph**. For example, the graphs of figure 6.5.1 (f) and figure 6.5.1 (g) are acyclic graphs.

A graph $G' = (V', E')$ is a sub-graph of graph $G = (V, E)$ iff $V' \subseteq V$ and $E' \subseteq E$.

A **Forest** is a set of disjoint trees. If we remove the root node of a given tree then it becomes forest. The following figure shows a forest F that consists of three trees T_1 , T_2 and T_3 .



A graph that has either self loop or parallel edges or both is called **multi-graph**.

Tree is a connected acyclic graph (there aren't any sequences of edges that go around in a loop). A spanning tree of a graph $G = (V, E)$ is a tree that contains all vertices of V and is a subgraph of G . A single graph can have multiple spanning trees.

Let T be a spanning tree of a graph G . Then

1. Any two vertices in T are connected by a unique simple path.
2. If any edge is removed from T , then T becomes disconnected.
3. If we add any edge into T , then the new graph will contain a cycle.
4. Number of edges in T is $n-1$.

6.1. Representation of Graphs:

There are two ways of representing digraphs. They are:

- Adjacency matrix.
- Adjacency List.
- Incidence matrix.

Adjacency matrix:

In this representation, the adjacency matrix of a graph G is a two dimensional $n \times n$ matrix, say $A = (a_{ij})$, where

$$a_{ij} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

The matrix is symmetric in case of undirected graph, while it may be asymmetric if the graph is directed. This matrix is also called as Boolean matrix or bit matrix.

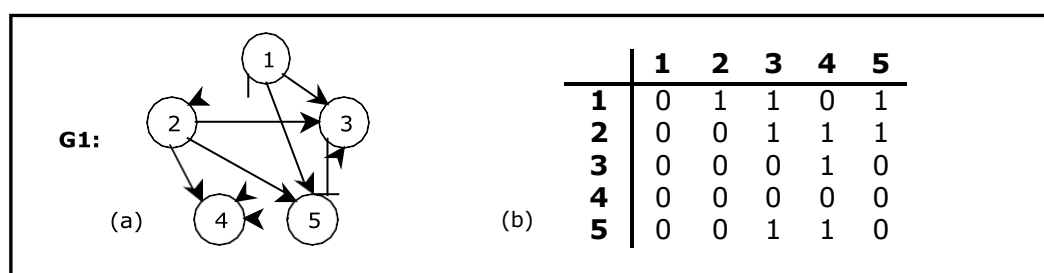


Figure 6.5.2. A graph and its Adjacency matrix

Figure 6.5.2(b) shows the adjacency matrix representation of the graph G_1 shown in figure 6.5.2(a). The adjacency matrix is also useful to store multigraph as well as weighted graph. In case of multigraph representation, instead of entry 0 or 1, the entry will be between number of edges between two vertices.

In case of weighted graph, the entries are weights of the edges between the vertices. The adjacency matrix for a weighted graph is called as cost adjacency matrix. Figure 6.5.3(b) shows the cost adjacency matrix representation of the graph G_2 shown in figure 6.5.3(a).

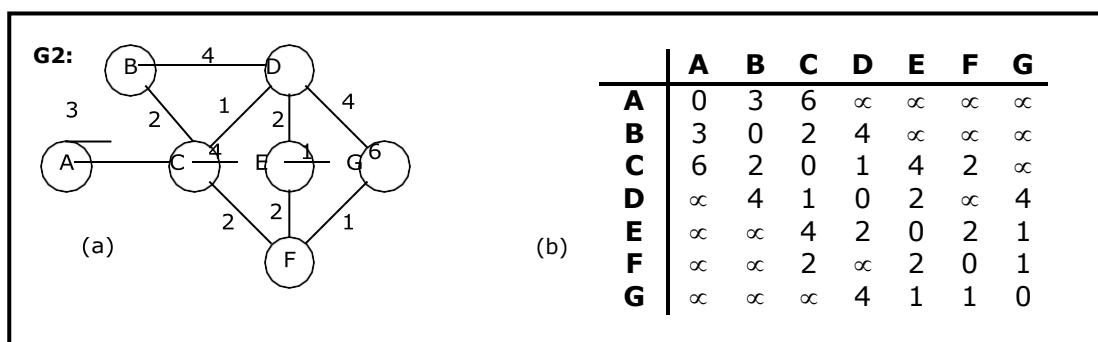


Figure 6.5.3 Weighted graph and its Cost adjacency matrix

Adjacency List:

In this representation, the n rows of the adjacency matrix are represented as n linked lists. An array $\text{Adj}[1, 2, \dots, n]$ of pointers where for $1 \leq v \leq n$, $\text{Adj}[v]$ points to a linked list containing the vertices which are adjacent to v (i.e. the vertices that can be reached from v by a single edge). If the edges have weights then these weights may also be stored in the linked list elements. For the graph G in figure 6.5.4(a), the adjacency list is shown in figure 6.5.4 (b).

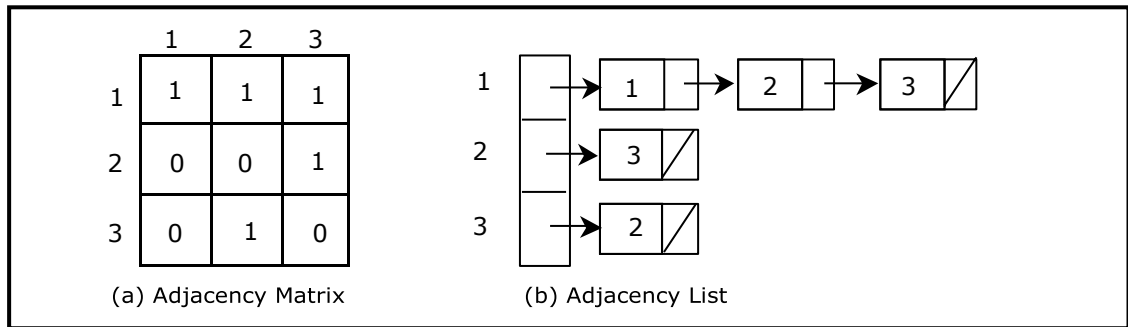


Figure 6.5.4 Adjacency matrix and adjacency list

Incidence Matrix:

In this representation, if G is a graph with n vertices, e edges and no self loops, then incidence matrix A is defined as an n by e matrix, say $A = (a_{i,j})$, where

$$a_{i,j} = \begin{cases} 1 & \text{if there is an edge } j \text{ incident to } v_i \\ 0 & \text{otherwise} \end{cases}$$

Here, n rows correspond to n vertices and e columns correspond to e edges. Such a matrix is called as vertex-edge incidence matrix or simply incidence matrix.

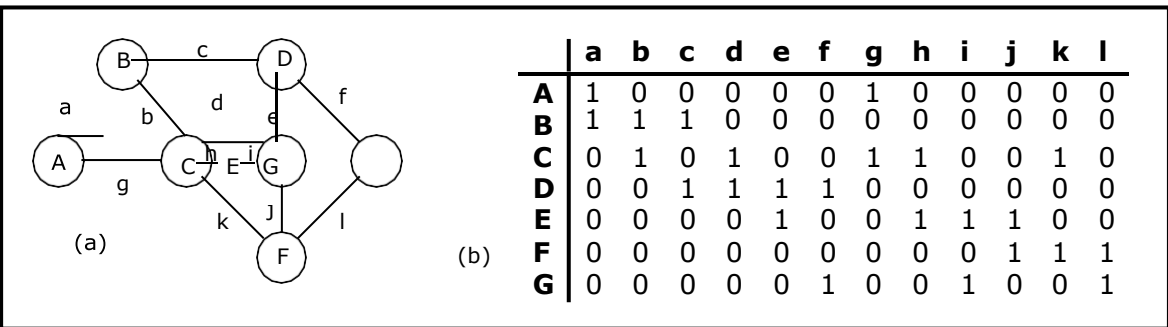


Figure 6.5.4 Graph and its incidence matrix

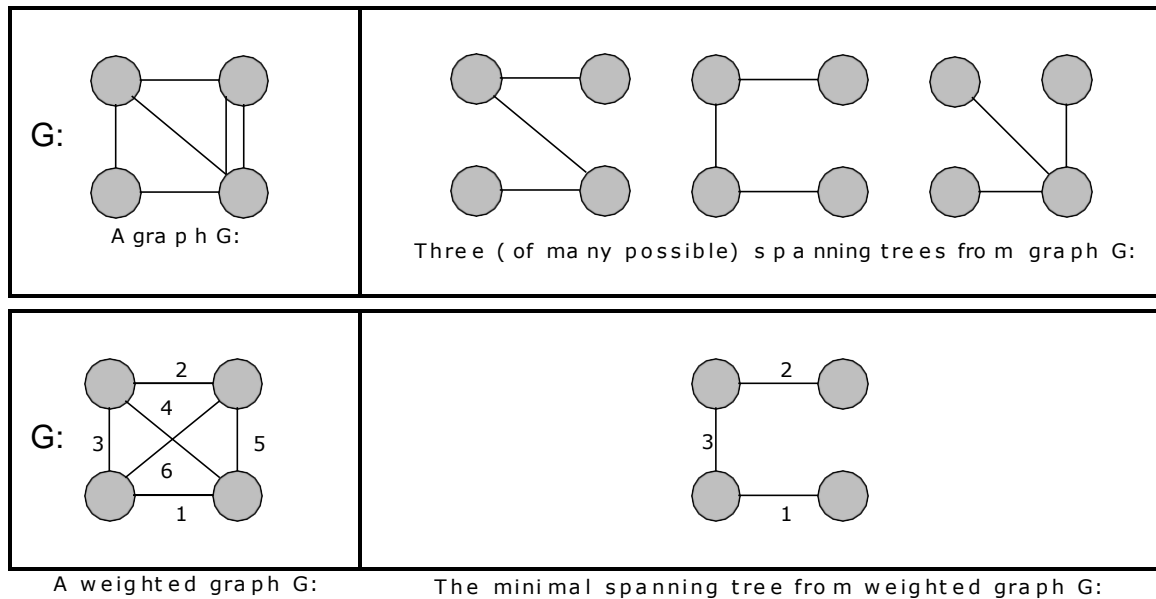
Figure 6.5.4(b) shows the incidence matrix representation of the graph G_1 shown in figure 6.5.4(a).

6.2. Minimum Spanning Tree (MST):

A spanning tree for a connected graph is a tree whose vertex set is the same as the vertex set of the given graph, and whose edge set is a subset of the edge set of the given graph. i.e., any connected graph will have a spanning tree.

Weight of a spanning tree $w(T)$ is the sum of weights of all edges in T . Minimum spanning tree (MST) is a spanning tree with the smallest possible weight.

Example:



Let's consider a couple of real-world examples on minimum spanning tree:

- One practical application of a MST would be in the design of a network. For instance, a group of individuals, who are separated by varying distances, wish to be connected together in a telephone network. Although MST cannot do anything about the distance from one connection to another, it can be used to determine the least cost paths with no cycles in this network, thereby connecting everyone at a minimum cost.
- Another useful application of MST would be finding airline routes. The vertices of the graph would represent cities, and the edges would represent routes between the cities. MST can be applied to optimize airline routes by finding the least costly paths with no cycles.

Minimum spanning tree, can be constructed using any of the following two algorithms:

1. Kruskal's algorithm and
2. Prim's algorithm.

Both algorithms differ in their methodology, but both eventually end up with the MST. *Kruskal's algorithm uses edges, and Prim's algorithm uses vertex connections* in determining the MST. In *Prim's algorithm at any instance of output it represents tree* whereas in *Kruskal's algorithm at any instance of output it may represent tree or not*.

6.3.1. Kruskal's Algorithm

This is a greedy algorithm. A greedy algorithm chooses some local optimum (i.e. picking an edge with the least weight in a MST).

Kruskal's algorithm works as follows: Take a graph with 'n' vertices, keep on adding the shortest (least cost) edge, while avoiding the creation of cycles, until $(n - 1)$ edges have been added. Sometimes two or more edges may have the same cost.

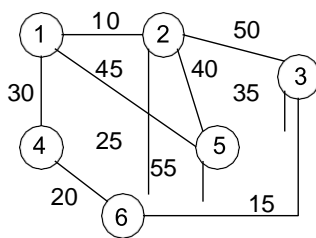
The order in which the edges are chosen, in this case, does not matter. Different MST's may result, but they will all have the same total cost, which will always be the minimum cost.

Kruskal's Algorithm for minimal spanning tree is as follows:

1. Make the tree T empty.
2. Repeat the steps 3, 4 and 5 as long as T contains less than $n - 1$ edges and E is not empty otherwise, proceed to step 6.
3. Choose an edge (v, w) from E of lowest cost.
4. Delete (v, w) from E.
5. If (v, w) does not create a cycle in T
 then Add (v, w) to T
 else discard (v, w)
6. If T contains fewer than $n - 1$ edges then print no spanning tree.

Example 1:

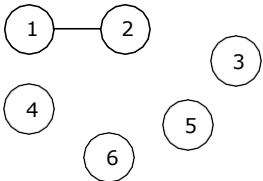
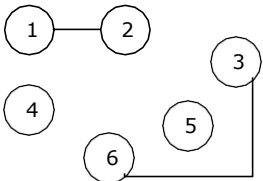
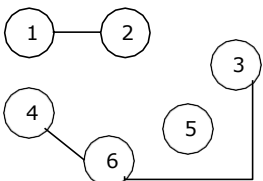
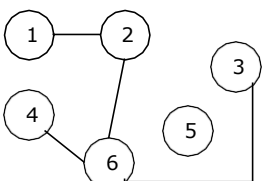
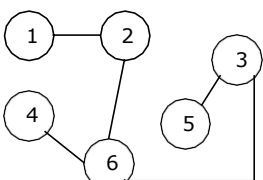
Construct the minimal spanning tree for the graph shown below:



Arrange all the edges in the increasing order of their costs:

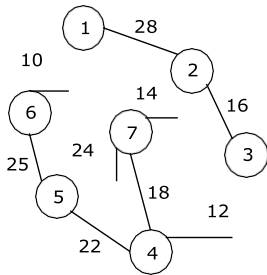
Cost	10	15	20	25	30	35	40	45	50	55
Edge	(1, 2)	(3, 6)	(4, 6)	(2, 6)	(1, 4)	(3, 5)	(2, 5)	(1, 5)	(2, 3)	(5, 6)

The stages in Kruskal's algorithm for minimal spanning tree is as follows:

EDGE	COST	STAGES IN KRUSKAL'S ALGORITHM	REMARKS
(1, 2)	10		The edge between vertices 1 and 2 is the first edge selected. It is included in the spanning tree.
(3, 6)	15		Next, the edge between vertices 3 and 6 is selected and included in the tree.
(4, 6)	20		The edge between vertices 4 and 6 is next included in the tree.
(2, 6)	25		The edge between vertices 2 and 6 is considered next and included in the tree.
(1, 4)	30	Reject	The edge between the vertices 1 and 4 is discarded as its inclusion creates a cycle.
(3, 5)	35		Finally, the edge between vertices 3 and 5 is considered and included in the tree built. This completes the tree. The cost of the minimal spanning tree is 105.

Example 2:

Construct the minimal spanning tree for the graph shown below:



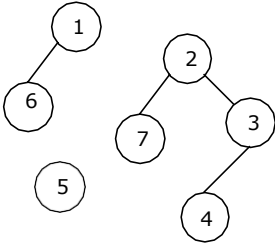
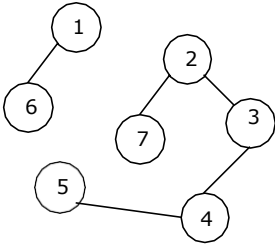
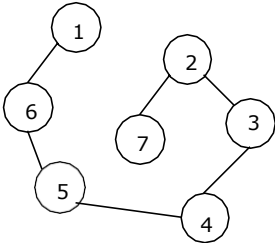
Solution:

Arrange all the edges in the increasing order of their costs:

Cost	10	12	14	16	18	22	24	25	28
Edge	(1, 6)	(3, 4)	(2, 7)	(2, 3)	(4, 7)	(4, 5)	(5, 7)	(5, 6)	(1, 2)

The stages in Kruskal’s algorithm for minimal spanning tree is as follows:

EDGE	COST	STAGES IN KRUSKAL’S ALGORITHM	REMARKS
(1, 6)	10		The edge between vertices 1 and 6 is the first edge selected. It is included in the spanning tree.
(3, 4)	12		Next, the edge between vertices 3 and 4 is selected and included in the tree.
(2, 7)	14		The edge between vertices 2 and 7 is next included in the tree.

(2, 3)	16		The edge between vertices 2 and 3 is next included in the tree.
(4, 7)	18	Reject	The edge between the vertices 4 and 7 is discarded as its inclusion creates a cycle.
(4, 5)	22		The edge between vertices 4 and 7 is considered next and included in the tree.
(5, 7)	24	Reject	The edge between the vertices 5 and 7 is discarded as its inclusion creates a cycle.
(5, 6)	25		<p>Finally, the edge between vertices 5 and 6 is considered and included in the tree built. This completes the tree.</p> <p>The cost of the minimal spanning tree is 99.</p>

6.3.2. MINIMUM-COST SPANNING TREES: PRIM'S ALGORITHM

A given graph can have many spanning trees. From these many spanning trees, we have to select a cheapest one. This tree is called as minimal cost spanning tree.

Minimal cost spanning tree is a connected undirected graph G in which each edge is labeled with a number (edge labels may signify lengths, weights other than costs). Minimal cost spanning tree is a spanning tree for which the sum of the edge labels is as small as possible

The slight modification of the spanning tree algorithm yields a very simple algorithm for finding an MST. In the spanning tree algorithm, any vertex not in the tree but connected to it by an edge can be added. To find a Minimal cost spanning tree, we must be selective - we must always add a new vertex for which the cost of the new edge is as small as possible.

This simple modified algorithm of spanning tree is called prim's algorithm for finding an Minimal cost spanning tree. Prim's algorithm is an example of a greedy algorithm.

Prim's Algorithm:

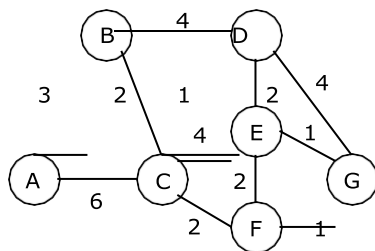
E is the set of edges in G . $\text{cost}[1:n, 1:n]$ is the cost adjacency matrix of an n vertex graph such that $\text{cost}[i, j]$ is either a positive real number or ∞ if no edge (i, j) exists. A minimum spanning tree is computed and stored as a set of edges in the array $t[1:n-1, 1:2]$. $(t[i, 1], t[i, 2])$ is an edge in the minimum-cost spanning tree. The final cost is returned.

Algorithm Prim (E, cost, n, t)

```
{
  Let  $(k, l)$  be an edge of minimum cost in  $E$ ;
  mincost := cost  $[k, l]$ ;
   $t[1, 1] := k$ ;  $t[1, 2] := l$ ;
  for  $i := 1$  to  $n$  do // Initialize near
    if  $(\text{cost}[i, l] < \text{cost}[i, k])$  then near  $[i] := l$ ;
    else near  $[i] := k$ ;
  near  $[k] := \text{near}[l] := 0$ ;
  for  $i := 2$  to  $n - 1$  do // Find  $n - 2$  additional edges for  $t$ .
  {
    Let  $j$  be an index such that near  $[j] \neq 0$  and
    cost  $[j, \text{near}[j]]$  is minimum;
     $t[i, 1] := j$ ;  $t[i, 2] := \text{near}[j]$ ;
    mincost := mincost + cost  $[j, \text{near}[j]]$ ;
    near  $[j] := 0$ ;
    for  $k := 1$  to  $n$  do // Update near[].
      if  $((\text{near}[k] \neq 0) \text{ and } (\text{cost}[k, \text{near}[k]] > \text{cost}[k, j]))$ 
      then near  $[k] := j$ ;
  }
  return mincost;
}
```

EXAMPLE:

Use Prim's Algorithm to find a minimal spanning tree for the graph shown below starting with the vertex A.



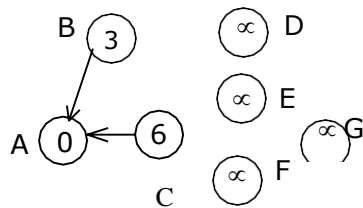
Solution:

The cost adjacency matrix is

0	3	6	∞	∞	∞	∞
3	0	2	4	∞	∞	∞
6	2	0	1	4	2	∞
∞	4	1	0	2	∞	4
∞	∞	4	2	0	2	1
∞	∞	2	∞	2	0	1
∞	∞	∞	4	1	1	0

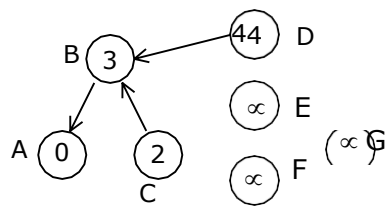
The stepwise progress of the prim's algorithm is as follows:

Step 1:



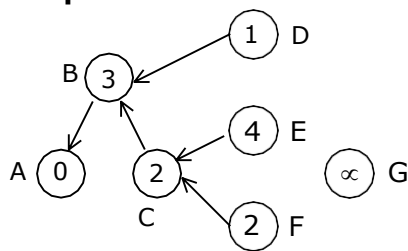
Vertex	A	B	C	D	E	F	G
Status	0	1	1	1	1	1	1
Dist.	0	3	6	∞	∞	∞	∞
Next	*	A	A	A	A	A	A

Step 2:



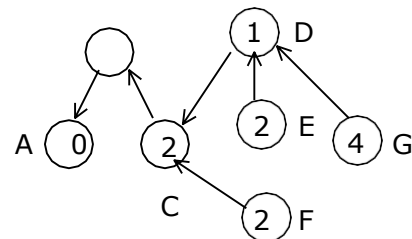
Vertex	A	B	C	D	E	F	G
Status	0	0	1	1	1	1	1
Dist.	0	3	2	4	∞	∞	∞
Next	*	A	B	B	A	A	A

Step 3:



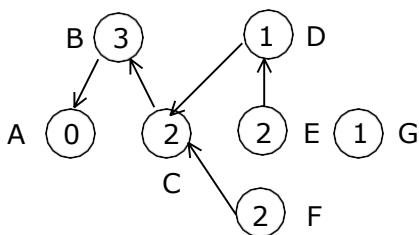
Vertex	A	B	C	D	E	F	G
Status	0	0	0	1	1	1	1
Dist.	0	3	2	1	4	2	∞
Next	*	A	B	C	C	C	A

Step 4:



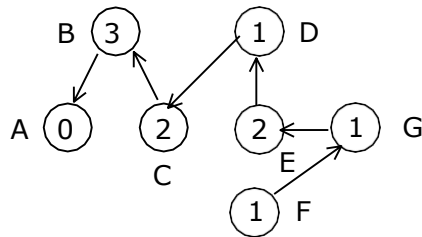
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	1	1
Dist.	0	3	2	1	2	2	4
Next	*	A	B	C	D	C	D

Step 5:



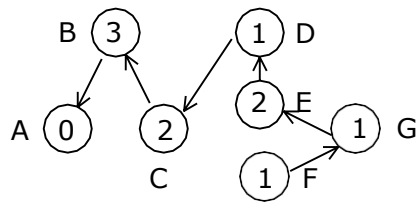
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	0	1
Dist.	0	3	2	1	2	2	1
Next	*	A	B	C	D	C	E

Step 6:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	1	0
Dist.	0	3	2	1	2	1	1
Next	*	A	B	C	D	G	E

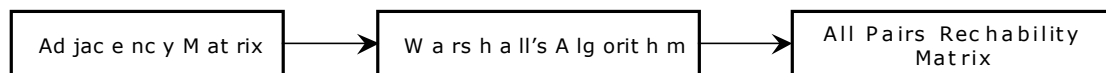
Step 7:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	0	0
Dist.	0	3	2	1	2	1	1
Next	*	A	B	C	D	G	E

6.3. Reachability Matrix (Warshall's Algorithm):

Warshall's algorithm requires knowing which edges exist and which does not. It doesn't need to know the lengths of the edges in the given directed graph. This information is conveniently displayed by adjacency matrix for the graph, in which a '1' indicates the existence of an edge and '0' indicates non-existence.



It begins with the adjacency matrix for the given graph, which is called A_0 , and then updates the matrix 'n' times, producing matrices called A_1, A_2, \dots, A_n and then stops.

In warshall's algorithm the matrix A_i contains information about the existence of i-paths. A one entry in the matrix A_i will correspond to the existence of i-paths and zero entry will correspond to non-existence. Thus when the algorithm stops, the final matrix A_n , contains the desired connectivity information.

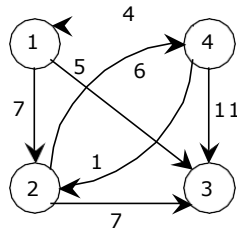
A one entry indicates a pair of vertices, which are connected and zero entry indicates a pair, which are not. This matrix is called a *reachability matrix* or *path matrix* for the graph. It is also called the *transitive closure* of the original adjacency matrix.

The update rule for computing A_i from A_{i-1} in warshall's algorithm is:

$$A_i[x, y] = A_{i-1}[x, y] \vee (A_{i-1}[x, i] \wedge A_{i-1}[i, y]) \quad \text{----} \quad (1)$$

Example 1:

Use warshall's algorithm to calculate the reachability matrix for the graph:



We begin with the adjacency matrix of the graph 'A₀'

$$A_0 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{matrix}$$

The first step is to compute 'A₁' matrix. To do so we will use the updating rule – (1).

Before doing so, we notice that only one entry in A₀ must remain one in A₁, since in Boolean algebra 1 + (anything) = 1. Since these are only nine zero entries in A₀, there are only nine entries in A₀ that need to be updated.

$$A_1[1, 1] = A_0[1, 1] \vee (A_0[1, 1] \wedge A_0[1, 1]) = 0 \vee (0 \wedge 0) = 0$$

$$A_1[1, 4] = A_0[1, 4] \vee (A_0[1, 1] \wedge A_0[1, 4]) = 0 \vee (0 \wedge 0) = 0$$

$$A_1[2, 1] = A_0[2, 1] \vee (A_0[2, 1] \wedge A_0[1, 1]) = 0 \vee (0 \wedge 0) = 0$$

$$A_1[2, 2] = A_0[2, 2] \vee (A_0[2, 1] \wedge A_0[1, 2]) = 0 \vee (0 \wedge 1) = 0$$

$$A_1[3, 1] = A_0[3, 1] \vee (A_0[3, 1] \wedge A_0[1, 1]) = 0 \vee (0 \wedge 0) = 0$$

$$A_1[3, 2] = A_0[3, 2] \vee (A_0[3, 1] \wedge A_0[1, 2]) = 0 \vee (0 \wedge 1) = 0$$

$$A_1[3, 3] = A_0[3, 3] \vee (A_0[3, 1] \wedge A_0[1, 3]) = 0 \vee (0 \wedge 1) = 0$$

$$A_1[3, 4] = A_0[3, 4] \vee (A_0[3, 1] \wedge A_0[1, 4]) = 0 \vee (0 \wedge 0) = 0$$

$$A_1[4, 4] = A_0[4, 4] \vee (A_0[4, 1] \wedge A_0[1, 4]) = 0 \vee (1 \wedge 0) = 0$$

$$A_1 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{matrix}$$

Next, A₂ must be calculated from A₁; but again we need to update the 0

$$\text{entries, } A_2[1, 1] = A_1[1, 1] \vee (A_1[1, 2] \wedge A_1[2, 1]) = 0 \vee (1 \wedge 0) = 0$$

$$A_2[1, 4] = A_1[1, 4] \vee (A_1[1, 2] \wedge A_1[2, 4]) = 0 \vee (1 \wedge 1) = 1$$

$$A_2[2, 1] = A_1[2, 1] \vee (A_1[2, 2] \wedge A_1[2, 1]) = 0 \vee (0 \wedge 0) = 0$$

$$A_2[2, 2] = A_1[2, 2] \vee (A_1[2, 2] \wedge A_1[2, 2]) = 0 \vee (0 \wedge 0) = 0$$

$$A_2[3, 1] = A_1[3, 1] \vee (A_1[3, 2] \wedge A_1[2, 1]) = 0 \vee (0 \wedge 0) = 0$$

$$A2[3, 2] = A1[3, 2] \vee (A1[3, 2] \wedge A1[2, 2]) = 0 \vee (0 \wedge 0) = 0$$

$$A2[3, 3] = A1[3, 3] \vee (A1[3, 2] \wedge A1[2, 3]) = 0 \vee (0 \wedge 1) = 0$$

$$A2[3, 4] = A1[3, 4] \vee (A1[3, 2] \wedge A1[2, 4]) = 0 \vee (0 \wedge 1) = 0$$

$$A2[4, 4] = A1[4, 4] \vee (A1[4, 2] \wedge A1[2, 4]) = 0 \vee (1 \wedge 1) = 1$$

$$A_2 = \begin{matrix} & \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{matrix}$$

This matrix has only seven 0 entries, and so to compute A_3 , we need to do only seven computations.

$$A3[1, 1] = A2[1, 1] \vee (A2[1, 3] \wedge A2[3, 1]) = 0 \vee (1 \wedge 0) = 0$$

$$A3[2, 1] = A2[2, 1] \vee (A2[2, 3] \wedge A2[3, 1]) = 0 \vee (1 \wedge 0) = 0$$

$$A3[2, 2] = A2[2, 2] \vee (A2[2, 3] \wedge A2[3, 2]) = 0 \vee (1 \wedge 0) = 0$$

$$A3[3, 1] = A2[3, 1] \vee (A2[3, 3] \wedge A2[3, 1]) = 0 \vee (0 \wedge 0) = 0$$

$$A3[3, 2] = A2[3, 2] \vee (A2[3, 3] \wedge A2[3, 2]) = 0 \vee (0 \wedge 0) = 0$$

$$A3[3, 3] = A2[3, 3] \vee (A2[3, 3] \wedge A2[3, 3]) = 0 \vee (0 \wedge 0) = 0$$

$$A3[3, 4] = A2[3, 4] \vee (A2[3, 3] \wedge A2[3, 4]) = 0 \vee (0 \wedge 0) = 0$$

$$A_3 = \begin{matrix} & \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{matrix}$$

Once A_3 is calculated, we use the update rule to calculate A_4 and stop. This matrix is the reachability matrix for the graph.

$$A4[1, 1] = A3[1, 1] \vee (A3[1, 4] \wedge A3[4, 1]) = 0 \vee (1 \wedge 1) = 0 \vee 1 = 1$$

$$A4[2, 1] = A3[2, 1] \vee (A3[2, 4] \wedge A3[4, 1]) = 0 \vee (1 \wedge 1) = 0 \vee 1 = 1$$

$$A4[2, 2] = A3[2, 2] \vee (A3[2, 4] \wedge A3[4, 2]) = 0 \vee (1 \wedge 1) = 0 \vee 1 = 1$$

$$A4[3, 1] = A3[3, 1] \vee (A3[3, 4] \wedge A3[4, 1]) = 0 \vee (0 \wedge 1) = 0 \vee 0 = 0$$

$$A4[3, 2] = A3[3, 2] \vee (A3[3, 4] \wedge A3[4, 2]) = 0 \vee (0 \wedge 1) = 0 \vee 0 = 0$$

$$A4[3, 3] = A3[3, 3] \vee (A3[3, 4] \wedge A3[4, 3]) = 0 \vee (0 \wedge 1) = 0 \vee 0 = 0$$

$$A4[3, 4] = A3[3, 4] \vee (A3[3, 4] \wedge A3[4, 4]) = 0 \vee (0 \wedge 1) = 0 \vee 0 = 0$$

$$A_4 = \begin{matrix} & \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{matrix}$$

Note that according to the algorithm vertex 3 is not reachable from itself 1. This is because as can be seen in the graph, there is no path from vertex 3 back to itself.

6.4. Traversing a Graph

Many graph algorithms require one to systematically examine the nodes and edges of a graph G . There are two standard ways to do this. They are:

- Breadth first traversal (BFT)
- Depth first traversal (DFT)

The BFT will use a queue as an auxiliary structure to hold nodes for future processing and the DFT will use a STACK.

During the execution of these algorithms, each node N of G will be in one of three states, called the *status* of N , as follows:

1. STATUS = 1 (Ready state): The initial state of the node N .
2. STATUS = 2 (Waiting state): The node N is on the QUEUE or STACK, waiting to be processed.
3. STATUS = 3 (Processed state): The node N has been processed.

Both BFS and DFS impose a tree (the BFS/DFS tree) on the structure of graph. So, we can compute a spanning tree in a graph. The computed spanning tree is not a minimum spanning tree. The spanning trees obtained using depth first search are called depth first spanning trees. The spanning trees obtained using breadth first search are called Breadth first spanning trees.

6.5.1. Breadth first search and traversal:

The general idea behind a breadth first traversal beginning at a starting node A is as follows. First we examine the starting node A . Then we examine all the neighbors of A . Then we examine all the neighbors of neighbors of A . And so on. We need to keep track of the neighbors of a node, and we need to guarantee that no node is processed more than once. This is accomplished by using a QUEUE to hold nodes that are waiting to be processed, and by using a field STATUS that tells us the current status of any node. The spanning trees obtained using BFS are called Breadth first spanning trees.

Breadth first traversal algorithm on graph G is as follows:

This algorithm executes a BFT on graph G beginning at a starting node A .

Initialize all nodes to the ready state (STATUS = 1).

1. Put the starting node A in QUEUE and change its status to the waiting state (STATUS = 2).
2. Repeat the following steps until QUEUE is empty:
 - a. Remove the front node N of QUEUE. Process N and change the status of N to the processed state (STATUS = 3).
 - b. Add to the rear of QUEUE all the neighbors of N that are in the ready state (STATUS = 1), and change their status to the waiting state (STATUS = 2).
3. Exit.

6.5.2. Depth first search and traversal:

Depth first search of undirected graph proceeds as follows: First we examine the starting node V. Next an unvisited vertex 'W' adjacent to 'V' is selected and a depth first search from 'W' is initiated. When a vertex 'U' is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited, which has an unvisited vertex 'W' adjacent to it and initiate a depth first search from W. The search terminates when no unvisited vertex can be reached from any of the visited ones.

This algorithm is similar to the inorder traversal of binary tree. DFT algorithm is similar to BFS except now use a STACK instead of the QUEUE. Again field STATUS is used to tell us the current status of a node.

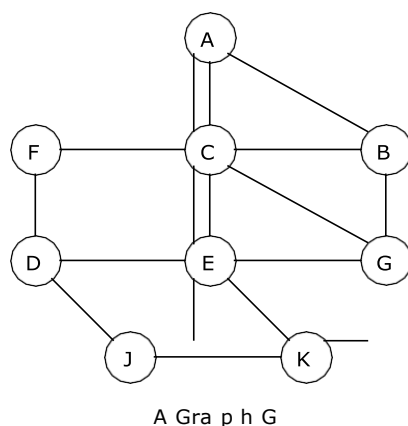
The algorithm for depth first traversal on a graph G is as follows.

This algorithm executes a DFT on graph G beginning at a starting node A.

1. Initialize all nodes to the ready state (STATUS = 1).
2. Push the starting node A into STACK and change its status to the waiting state (STATUS = 2).
3. Repeat the following steps until STACK is empty:
 - a. Pop the top node N from STACK. Process N and change the status of N to the processed state (STATUS = 3).
 - b. Push all the neighbors of N that are in the ready state (STATUS = 1), and change their status to the waiting state (STATUS = 2).
4. Exit.

Example 1:

Consider the graph shown below. Traverse the graph shown below in breadth first order and depth first order.



Node	Adjacency List
A	F, C, B
B	A, C, G
C	A, B, D, E, F, G
D	C, F, E, J
E	C, D, G, J, K
F	A, C, D
G	B, C, E, K
J	D, E, K
K	E, G, J

Adjacency list for graph G

Breadth-first search and traversal:

The steps involved in breadth first traversal are as follows:

Current Node	QUEUE	Processed Nodes	Status								
			A	B	C	D	E	F	G	J	K
			1	1	1	1	1	1	1	1	1
	A		2	1	1	1	1	1	1	1	1
A	F C B	A	3	2	2	1	1	2	1	1	1
F	C B D	A F	3	2	2	2	1	3	1	1	1
C	B D E G	A F C	3	2	3	2	2	3	2	1	1
B	D E G	A F C B	3	3	3	2	2	3	2	1	1
D	E G J	A F C B D	3	3	3	3	2	3	2	2	1
E	G J K	A F C B D E	3	3	3	3	3	3	2	2	2
G	J K	A F C B D E G	3	3	3	3	3	3	3	2	2
J	K	A F C B D E G J	3	3	3	3	3	3	3	3	2
K	EMPTY	A F C B D E G J K	3	3	3	3	3	3	3	3	3

For the above graph the breadth first traversal sequence is: **A F C B D E G J K**.

Depth-first search and traversal:

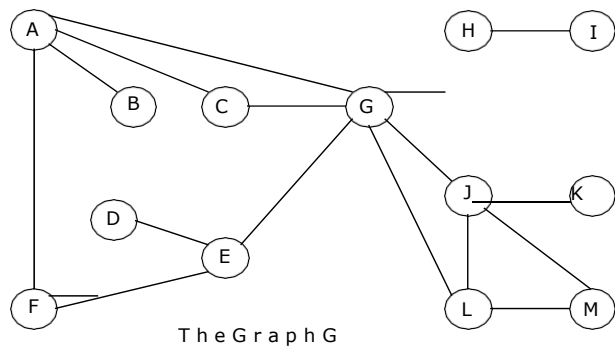
The steps involved in depth first traversal are as follows:

Current Node	Stack	Processed Nodes	Status								
			A	B	C	D	E	F	G	J	K
			1	1	1	1	1	1	1	1	1
	A		2	1	1	1	1	1	1	1	1
A	B C F	A	3	2	2	1	1	2	1	1	1
F	B C D	A F	3	2	2	2	1	3	1	1	1
D	B C E J	A F D	3	2	2	3	2	3	1	2	1
J	B C E K	A F D J	3	2	2	3	2	3	1	3	2
K	B C E G	A F D J K	3	2	2	3	2	3	2	3	3
G	B C E	A F D J K G	3	2	2	3	2	3	3	3	3
E	B C	A F D J K G E	3	2	2	3	3	3	3	3	3
C	B	A F D J K G E C	3	2	3	3	3	3	3	3	3
B	EMPTY	A F D J K G E C B	3	3	3	3	3	3	3	3	3

For the above graph the depth first traversal sequence is: **A F D J K G E C B**.

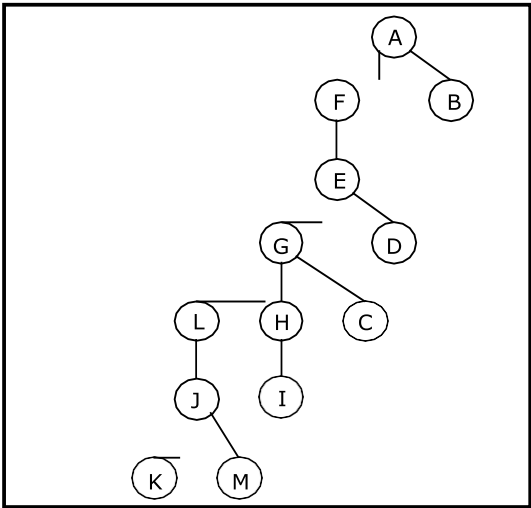
Example 2:

Traverse the graph shown below in breadth first order, depth first order and construct the breadth first and depth first spanning trees.

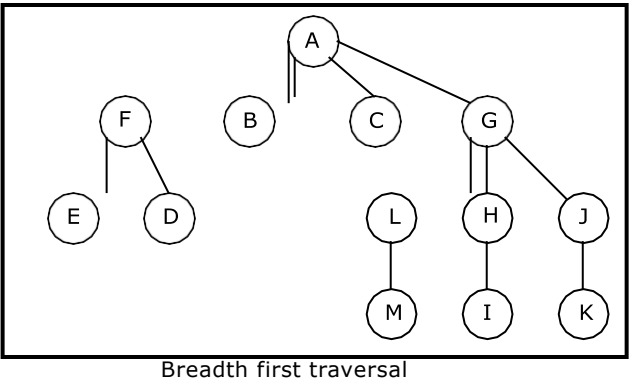


Node	Adjacency List
A	F, B, C, G
B	A
C	A, G
D	E, F
E	G, D, F
F	A, E, D
G	A, L, E, H, J, C
H	G, I
I	H
J	G, L, K, M
K	J
L	G, J, M
M	G, J, L

If the depth first traversal is initiated from vertex A, then the vertices of graph G are visited in the order: **A F E G L J K M H I C D B**. The depth first spanning tree is shown in the figure given below:

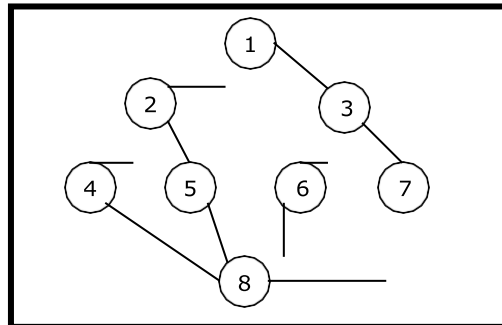


If the breadth first traversal is initiated from vertex A, then the vertices of graph G are visited in the order: **A F B C G E D L H J M I K**. The breadth first spanning tree is shown in the figure given below:



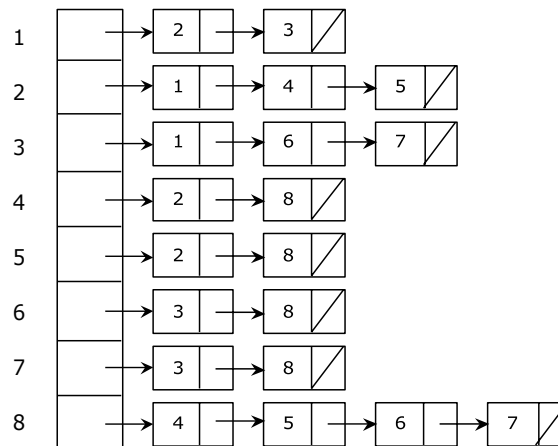
Example 3:

Traverse the graph shown below in breadth first order, depth first order and construct the breadth first and depth first spanning trees.



Graph G

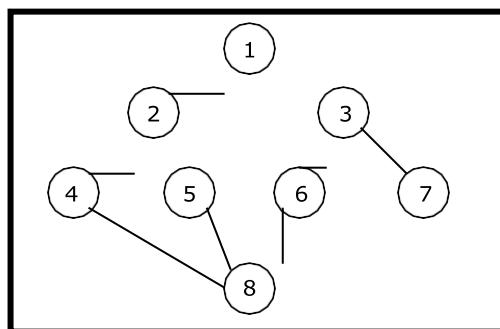
Head Nodes



Adjacency list for graph G

Depth first search and traversal:

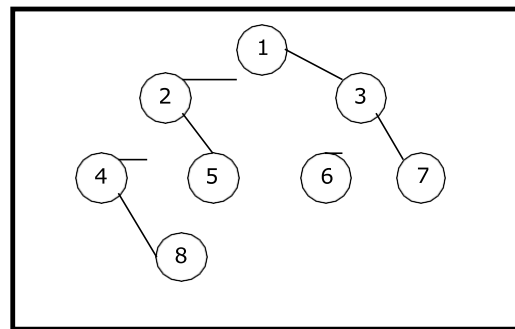
If the depth first is initiated from vertex 1, then the vertices of graph G are visited in the order: 1, 2, 4, 8, 5, 6, 3, 7. The depth first spanning tree is as follows:



Depth First Spanning Tree

Breadth first search and traversal:

If the breadth first search is initiated from vertex 1, then the vertices of G are visited in the order: 1, 2, 3, 4, 5, 6, 7, 8. The breadth first spanning tree is as follows:

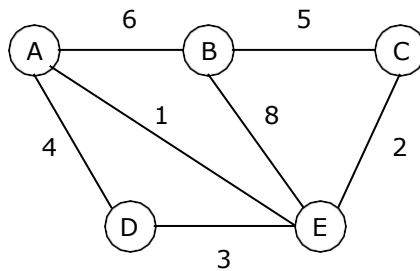


Breadth First Spanning Tree

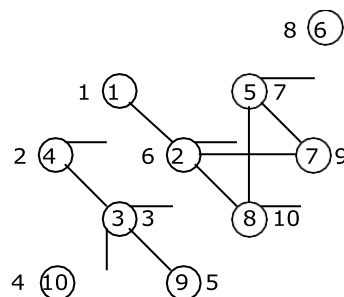
EXERCISES

1. Show that the sum of degrees of all vertices in an undirected graph is twice the number of edges.
2. Show that the number of vertices of odd degree in a finite graph is even.
3. How many edges are contained in a complete graph of "n" vertices.
4. Show that the number of spanning trees in a complete graph of "n" vertices is $2^{n-1} - 1$.
5. Prove that the edges explored by a breadth first or depth first traversal of a connected graph form a tree.
6. Explain how existence of a cycle in an undirected graph may be detected by traversing the graph in a depth first manner.
7. Write a "C" function to generate the incidence matrix of a graph from its adjacency matrix.
8. Give an example of a connected directed graph so that a depth first traversal of that graph yields a forest and not a spanning tree of the graph.
9. Rewrite the algorithms "BFSearch" and "DFSearch" so that it works on adjacency matrix representation of graphs.
10. Write a "C" function to find out whether there is a path between any two vertices in a graph (i.e. to compute the transitive closure matrix of a graph)
11. Write a "C" function to delete an existing edge from a graph represented by an adjacency list.
12. Construct a weighted graph for which the minimal spanning trees produced by Kruskal's algorithm and Prim's algorithm are different.

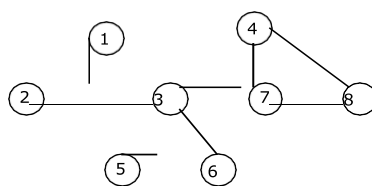
13. Describe the algorithm to find a minimum spanning tree T of a weighted graph G . Find the minimum spanning tree T of the graph shown below.



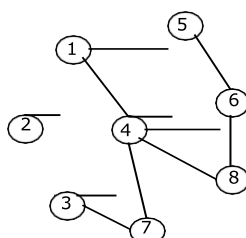
14. For the graph given below find the following:
- Linked representation of the graph.
 - Adjacency list.
 - Depth first spanning tree.
 - Breadth first spanning tree.
 - Minimal spanning tree using Kruskal's and Prim's algorithms.



15. For the graph given below find the following:
- Linked representation of the graph.
 - Adjacency list.
 - Depth first spanning tree.
 - Breadth first spanning tree.
 - Minimal spanning tree using Kruskal's and Prim's algorithms.

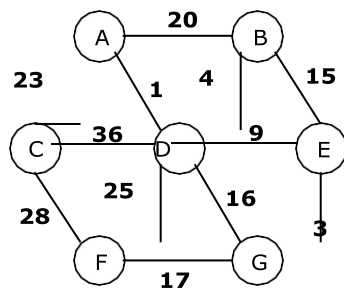


16. For the graph given below find the following:
- Linked representation of the graph.
 - Adjacency list.
 - Depth first spanning tree.
 - Breadth first spanning tree.
 - Minimal spanning tree using Kruskal's and Prim's algorithms.



Multiple Choice Questions

1. How can the graphs be represented? [D]
 A. Adjacency matrix C. Incidence matrix
 B. Adjacency list D. All of the above
2. The depth-first traversal in graph is analogous to tree traversal: [C]
 A. In-order C. Pre-order
 B. Post-order D. Level order
3. The children of a same parent node are called as: [C]
 A. adjacent node C. Siblings
 B. non-leaf node D. leaf node
4. Complete graphs with n nodes will have_____edges. [C]
 A. $n - 1$ C. $n(n-1)/2$
 B. $n/2$ D. $(n - 1)/2$
5. A graph with no cycle is called as: [C]
 A. Sub-graph C. Acyclic graph
 B. Directed graph D. none of the above
6. The maximum number of nodes at any level is: [B]
 A. n C. $n + 1$
 B. 2^n D. $2n$



Node	Adjacency List
A	B C D
B	A D E
C	A D F
D	A B C E F G
E	B D G
F	C D G
G	F D E

FIGURE 1 and its adjacency list

7. For the figure 1 shown above, the depth first spanning tree visiting sequence is: [B]
 A. A B C D E F G C. A B C D E F G
 B. A B D C F G E D. none of the above
8. For the figure 1 shown above, the breadth first spanning tree visiting sequence is: [B]
 A. A B D C F G E C. A B C D E F G
 B. A B C D E F G D. none of the above
9. Which is the correct order for Kruskal's minimum spanning tree algorithm to add edges to the minimum spanning tree for the figure 1 shown above: [B]
 A. (A, B) then (A, C) then (A, D) then (D, E) then (C, F) then (D, G)
 B. (A, D) then (E, G) then (B, D) then (D, E) then (F, G) then (A, C)

- C. both A and B
- D. none of the above

10. For the figure 1 shown above, the cost of the minimal spanning tree is: [A]

- | | |
|-------|-------|
| A. 57 | C. 48 |
| B. 68 | D. 32 |

11. A simple graph has no loops. What other property must a simple graph have? [D]
- A. It must be directed. C. It must have at least one vertex.
B. It must be undirected. D. It must have no multiple edges.
12. Suppose you have a directed graph representing all the flights that an airline flies. What algorithm might be used to find the best sequence of connections from one city to another? [D]
- A. Breadth first search. C. A cycle-finding algorithm.
B. Depth first search. D. A shortest-path algorithm.
13. If G is an directed graph with 20 vertices, how many boolean values will be needed to represent G using an adjacency matrix? [D]
- A. 20 C. 200
B. 40 D. 400
14. Which graph representation allows the most efficient determination of the existence of a particular edge in a graph? [B]
- A. An adjacency matrix. C. Incidence matrix
B. Edge lists. D. none of the above
15. What graph traversal algorithm uses a queue to keep track of vertices which need to be processed? [A]
- A. Breadth-first search. C Level order search
B. Depth-first search. D. none of the above
16. What graph traversal algorithm uses a stack to keep track of vertices which need to be processed? [B]
- A. Breadth-first search. C Level order search
B. Depth-first search. D. none of the above
17. What is the expected number of operations needed to loop through all the edges terminating at a particular vertex given an adjacency matrix representation of the graph? (Assume n vertices are in the graph and m edges terminate at the desired node.) [D]
- A. $O(m)$ C. $O(m^2)$
B. $O(n)$ D. $O(n^2)$
18. What is the expected number of operations needed to loop through all the edges terminating at a particular vertex given an adjacency list representation of the graph? (Assume n vertices are in the graph and m edges terminate at the desired node.) [A]
- 19.19. A. $O(m)$ C. $O(m^2)$
B. $O(n)$ D. $O(n^2)$

[B]

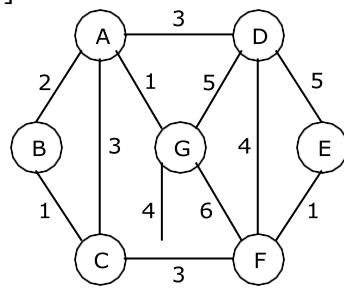


FIGURE 3

For the figure 3, starting at vertex A, which is a correct order for Prim's minimum spanning tree algorithm to add edges to the minimum spanning tree?

Searching and Sorting

There are basically two aspects of computer programming. One is data organization also commonly called as data structures. Till now we have seen about data structures and the techniques and algorithms used to access them. The other part of computer programming involves choosing the appropriate algorithm to solve the problem. Data structures and algorithms are linked each other. After developing programming techniques to represent information, it is logical to proceed to manipulate it. This chapter introduces this important aspect of problem solving.

Searching is used to find the location where an element is available. There are two types of search techniques. They are:

1. Linear or sequential search
2. Binary search

Sorting allows an efficient arrangement of elements within a given data structure. It is a way in which the elements are organized systematically for some purpose. For example, a dictionary in which words are arranged in alphabetical order and telephone directory in which the subscriber names are listed in alphabetical order. There are many sorting techniques out of which we study the following.

1. Bubble sort
2. Quick sort
3. Selection sort and
4. Heap sort

There are two types of sorting techniques:

1. Internal sorting
2. External sorting

If all the elements to be sorted are present in the main memory then such sorting is called **internal sorting** on the other hand, if some of the elements to be sorted are kept on the secondary storage, it is called **external sorting**. Here we study only internal sorting techniques.

7.1. Linear Search:

This is the simplest of all searching techniques. In this technique, an ordered or unordered list will be searched one by one from the beginning until the desired element is found. If the desired element is found in the list then the search is successful otherwise unsuccessful.

Suppose there are 'n' elements organized sequentially on a List. The number of comparisons required to retrieve an element from the list, purely depends on where the element is stored in the list. If it is the first element, one comparison will do; if it is second element two comparisons are necessary and so on. On an average you need $[(n+1)/2]$ comparison's to search an element. If search is not successful, you would need 'n' comparisons.

The time complexity of linear search is **$O(n)$** .

Algorithm:

Let array a[n] stores n elements. Determine whether element 'x' is present or not.

```
linsrch(a[n], x)
{
    index = 0;
    flag = 0;
    while (index < n) do
    {
        if (x == a[index])
        {
            flag = 1;
            break;
        }
        index ++;
    }
    if(flag == 1)
        printf("Data found at %d position", index);
    else
        printf("data not found");
}
```

Example 1:

Suppose we have the following unsorted list: 45, 39, 8, 54, 77, 38, 24, 16, 4, 7, 9, 20

If we are searching for:	45, we'll look at 1 element before success
	39, we'll look at 2 elements before success
	8, we'll look at 3 elements before success
	54, we'll look at 4 elements before success
	77, we'll look at 5 elements before success
	38 we'll look at 6 elements before success
	24, we'll look at 7 elements before success
	16, we'll look at 8 elements before success
	4, we'll look at 9 elements before success
	7, we'll look at 10 elements before success
	9, we'll look at 11 elements before success
	20, we'll look at 12 elements before success

For any element not in the list, we'll look at 12 elements before failure.

Example 2:

Let us illustrate linear search on the following 9 elements:

Index	0	1	2	3	4	5	6	7	8
Elements	-15	-6	0	7	9	23	54	82	101

Searching different elements is as follows:

1. Searching for x = 7 Search successful, data found at 3rd position.
2. Searching for x = 82 Search successful, data found at 7th position.
3. Searching for x = 42 Search un-successful, data not found.

7.1.1. A non-recursive program for Linear Search:

```
# include <stdio.h>
# include <conio.h>

main()
{
    int number[25], n, data, i, flag = 0;
    clrscr();
    printf("\n Enter the number of elements: ");
    scanf("%d", &n);
    printf("\n Enter the elements: ");
    for(i = 0; i < n; i++)
        scanf("%d", &number[i]);
    printf("\n Enter the element to be Searched: ");
    scanf("%d", &data);
    for( i = 0; i < n; i++)
    {
        if(number[i] == data)
        {
            flag = 1;
            break;
        }
    }
    if(flag == 1)
        printf("\n Data found at location: %d", i+1);
    else
        printf("\n Data not found ");
}
```

7.1.2. A Recursive program for linear search:

```
# include <stdio.h>
# include <conio.h>

void linear_search(int a[], int data, int position, int n)
{
    if(position < n)
```

```

        {
            if(a[position] == data)
                printf("\n Data Found at %d ", position);
            else
                linear_search(a, data, position + 1, n);
        }
    else

        printf("\n Data not found");
}

void main()
{

    int a[25], i, n, data;
    clrscr();
    printf("\n Enter the number of elements: ");
    scanf("%d", &n);
    printf("\n Enter the elements: ");
    for(i = 0; i < n; i++)
    {
        scanf("%d", &a[i]);
    }
    printf("\n Enter the element to be seached: ");
    scanf("%d", &data);
    linear_search(a, data, 0, n);
    getch();
}

```

7.2. BINARY SEARCH

If we have 'n' records which have been ordered by keys so that $x_1 < x_2 < \dots < x_n$. When we are given a element 'x', binary search is used to find the corresponding element from the list. In case 'x' is present, we have to determine a value 'j' such that $a[j] = x$ (successful search). If 'x' is not in the list then j is to set to zero (un successful search).

In Binary search we jump into the middle of the file, where we find key $a[mid]$, and compare 'x' with $a[mid]$. If $x = a[mid]$ then the desired record has been found. If $x < a[mid]$ then 'x' must be in that portion of the file that precedes $a[mid]$. Similarly, if $a[mid] > x$, then further search is only necessary in that part of the file which follows $a[mid]$.

If we use recursive procedure of finding the middle key $a[mid]$ of the un-searched portion of a file, then every un-successful comparison of 'x' with $a[mid]$ will eliminate roughly half the un-searched portion from consideration.

Since the array size is roughly halved after each comparison between 'x' and $a[mid]$, and since an array of length 'n' can be halved only about $\log_2 n$ times before reaching a trivial length, the worst case complexity of Binary search is about $\log_2 n$.

Algorithm:

Let array $a[n]$ of elements in increasing order, $n \geq 0$, determine whether 'x' is present, and if so, set j such that $x = a[j]$ else return 0.

```

binsrch(a[], n, x)
{
    low = 1; high = n;
    while (low ≤ high) do
    {
        mid = (low + high)/2
        if (x < a[mid])
            high = mid - 1;
        else if (x > a[mid])
            low = mid + 1;
        else return mid;
    }
    return 0;
}

```

low and *high* are integer variables such that each time through the loop either 'x' is found or *low* is increased by at least one or *high* is decreased by at least one. Thus we have two sequences of integers approaching each other and eventually *low* will become greater than *high* causing termination in a finite number of steps if 'x' is not present.

Example 1:

Let us illustrate binary search on the following 12 elements:

Index	1	2	3	4	5	6	7	8	9	10	11	12
Elements	4	7	8	9	16	20	24	38	39	45	54	77

If we are searching for $x = 4$: (This needs 3 comparisons)

low = 1, high = 12, mid = $13/2 = 6$, check 20

low = 1, high = 5, mid = $6/2 = 3$, check 8

low = 1, high = 2, mid = $3/2 = 1$, check 4, **found**

If we are searching for $x = 7$: (This needs 4 comparisons)

low = 1, high = 12, mid = $13/2 = 6$, check 20

low = 1, high = 5, mid = $6/2 = 3$, check 8

low = 1, high = 2, mid = $3/2 = 1$, check 4

low = 2, high = 2, mid = $4/2 = 2$, check 7, **found**

If we are searching for $x = 8$: (This needs 2 comparisons)

low = 1, high = 12, mid = $13/2 = 6$, check 20

low = 1, high = 5, mid = $6/2 = 3$, check 8, **found**

If we are searching for $x = 9$: (This needs 3 comparisons)

low = 1, high = 12, mid = $13/2 = 6$, check 20

low = 1, high = 5, mid = $6/2 = 3$, check 8

low = 4, high = 5, mid = $9/2 = 4$, check 9, **found**

If we are searching for $x = 16$: (This needs 4 comparisons)

low = 1, high = 12, mid = $13/2 = 6$, check 20

low = 1, high = 5, mid = $6/2 = 3$, check 8

low = 4, high = 5, mid = $9/2 = 4$, check 9

low = 5, high = 5, mid = $10/2 = 5$, check 16, **found**

If we are searching for $x = 20$: (This needs 1 comparison)

low = 1, high = 12, mid = $13/2 = 6$, check 20, **found**

If we are searching for $x = 24$: (This needs 3 comparisons)
low = 1, high = 12, mid = $13/2 = 6$, check 20
low = 7, high = 12, mid = $19/2 = 9$, check 39
low = 7, high = 8, mid = $15/2 = 7$, check 24, **found**

If we are searching for $x = 38$: (This needs 4 comparisons)
low = 1, high = 12, mid = $13/2 = 6$, check 20
low = 7, high = 12, mid = $19/2 = 9$, check 39
low = 7, high = 8, mid = $15/2 = 7$, check 24
low = 8, high = 8, mid = $16/2 = 8$, check 38, **found**

If we are searching for $x = 39$: (This needs 2 comparisons)
low = 1, high = 12, mid = $13/2 = 6$, check 20
low = 7, high = 12, mid = $19/2 = 9$, check 39, **found**

If we are searching for $x = 45$: (This needs 4 comparisons)
low = 1, high = 12, mid = $13/2 = 6$, check 20
low = 7, high = 12, mid = $19/2 = 9$, check 39
low = 10, high = 12, mid = $22/2 = 11$, check 54
low = 10, high = 10, mid = $20/2 = 10$, check 45, **found**

If we are searching for $x = 54$: (This needs 3 comparisons)
low = 1, high = 12, mid = $13/2 = 6$, check 20
low = 7, high = 12, mid = $19/2 = 9$, check 39
low = 10, high = 12, mid = $22/2 = 11$, check 54, **found**

If we are searching for $x = 77$: (This needs 4 comparisons)
low = 1, high = 12, mid = $13/2 = 6$, check 20
low = 7, high = 12, mid = $19/2 = 9$, check 39
low = 10, high = 12, mid = $22/2 = 11$, check 54
low = 12, high = 12, mid = $24/2 = 12$, check 77, **found**

The number of comparisons necessary by search element:

20 – requires 1 comparison;
8 and 39 – requires 2 comparisons;
4, 9, 24, 54 – requires 3 comparisons and
7, 16, 38, 45, 77 – requires 4 comparisons

Summing the comparisons, needed to find all twelve items and dividing by 12, yielding $37/12$ or approximately 3.08 comparisons per successful search on the average.

Example 2:

Let us illustrate binary search on the following 9 elements:

Index	0	1	2	3	4	5	6	7	8
Elements	-15	-6	0	7	9	23	54	82	101

Solution:

The number of comparisons required for searching different elements is as follows:

1. If we are searching for $x = 101$: (Number of comparisons = 4)

low	high	mid
1	9	5
6	9	7
8	9	8
9	9	9

found

2. Searching for $x = 82$: (Number of comparisons = 3)

low	high	mid
1	9	5
6	9	7
8	9	8

found

3. Searching for $x = 42$: (Number of comparisons = 4)

low	high	mid
1	9	5
6	9	7
6	6	6
7	6	not found

4. Searching for $x = -14$: (Number of comparisons = 3)

low	high	mid
1	9	5
1	4	2
1	1	1
2	1	not found

Continuing in this manner the number of element comparisons needed to find each of nine elements is:

Index	1	2	3	4	5	6	7	8	9
Elements	-15	-6	0	7	9	23	54	82	101
Comparisons	3	2	3	4	1	3	2	3	4

No element requires more than 4 comparisons to be found. Summing the comparisons needed to find all nine items and dividing by 9, yielding $25/9$ or approximately 2.77 comparisons per successful search on the average.

There are ten possible ways that an un-successful search may terminate depending upon the value of x .

If $x < a(1)$, $a(1) < x < a(2)$, $a(2) < x < a(3)$, $a(5) < x < a(6)$, $a(6) < x < a(7)$ or $a(7) < x < a(8)$ the algorithm requires 3 element comparisons to determine that ' x ' is not present. For all of the remaining possibilities BINSRCH requires 4 element comparisons.

Thus the average number of element comparisons for an unsuccessful search is:

$$(3 + 3 + 3 + 4 + 4 + 3 + 3 + 3 + 4 + 4) / 10 = 34/10 = 3.4$$

Time Complexity:

The time complexity of binary search in a successful search is $O(\log n)$ and for an unsuccessful search is $O(\log n)$.

7.2.1. A non-recursive program for binary search:

```
# include <stdio.h>
# include <conio.h>

main()
{
    int number[25], n, data, i, flag = 0, low, high, mid;
    clrscr();
    printf("\n Enter the number of elements: ");
    scanf("%d", &n);
    printf("\n Enter the elements in ascending order: ");
    for(i = 0; i < n; i++)
        scanf("%d", &number[i]);
    printf("\n Enter the element to be searched: ");
    scanf("%d", &data);
    low = 0; high = n-1;
    while(low <= high)
    {
        mid = (low + high)/2;
        if(number[mid] == data)
        {
            flag = 1;
            break;
        }
        else
        {
            if(data < number[mid])
                high = mid - 1;
            else
                low = mid + 1;
        }
    }

    if(flag == 1)
        printf("\n Data found at location: %d", mid + 1);
    else
        printf("\n Data Not Found ");
}
```

7.2.2. A recursive program for binary search:

```
# include <stdio.h>
# include <conio.h>

void bin_search(int a[], int data, int low, int high)
{
    int mid ;
    if( low <= high)
    {
        mid = (low + high)/2;
        if(a[mid] == data)
            printf("\n Element found at location: %d ", mid + 1);
        else
        {
            if(data < a[mid])
```

```
bin_searc    h(a, data, low, mid-1);  
    else
```

```

        bin_search(a, data, mid+1, high);
    }
}
else

    printf("\n Element not found");
}
void main()
{

    int a[25], i, n, data;
    clrscr();
    printf("\n Enter the number of elements: ");
    scanf("%d", &n);
    printf("\n Enter the elements in ascending order: ");
    for(i = 0; i < n; i++)
        scanf("%d", &a[i]);
    printf("\n Enter the element to be searched: ");
    scanf("%d", &data);
    bin_search(a, data, 0, n-1);
    getch();
}

```

7.3. Bubble Sort:

The bubble sort is easy to understand and program. The basic idea of bubble sort is to pass through the file sequentially several times. In each pass, we compare each element in the file with its successor i.e., $X[i]$ with $X[i+1]$ and interchange two element when they are not in proper order. We will illustrate this sorting technique by taking a specific example. Bubble sort is also called as exchange sort.

Example:

Consider the array $x[n]$ which is stored in memory as shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	X[5]
33	44	22	11	66	55

Suppose we want our array to be stored in ascending order. Then we pass through the array 5 times as described below:

Pass 1: (first element is compared with all other elements).

We compare $X[i]$ and $X[i+1]$ for $i = 0, 1, 2, 3$, and 4 , and interchange $X[i]$ and $X[i+1]$ if $X[i] > X[i+1]$. The process is shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	X[5]	Remarks
33	44	22	11	66	55	
	22	44				
		11	44			
			44	66		
				55	66	
33	22	11	44	55	66	

The biggest number 66 is moved to (bubbled up) the right most position in the array.

Pass 2: (second element is compared).

We repeat the same process, but this time we don't include X[5] into our comparisons. i.e., we compare X[i] with X[i+1] for i=0, 1, 2, and 3 and interchange X[i] and X[i+1] if $X[i] > X[i+1]$. The process is shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	Remarks
33	22	11	44	55	
22	33				
	11	33			
		33	44		
			44	55	
22	11	33	44	55	

The second biggest number 55 is moved now to X[4].

Pass 3: (third element is compared).

We repeat the same process, but this time we leave both X[4] and X[5]. By doing this, we move the third biggest number 44 to X[3].

X[0]	X[1]	X[2]	X[3]	Remarks
22	11	33	44	
11	22			
	22	33		
		33	44	
11	22	33	44	

Pass 4: (fourth element is compared).

We repeat the process leaving X[3], X[4], and X[5]. By doing this, we move the fourth biggest number 33 to X[2].

X[0]	X[1]	X[2]	Remarks
11	22	33	
11	22		
	22	33	

Pass 5: (fifth element is compared).

We repeat the process leaving X[2], X[3], X[4], and X[5]. By doing this, we move the fifth biggest number 22 to X[1]. At this time, we will have the smallest number 11 in X[0]. Thus, we see that we can sort the array of size 6 in 5 passes.

For an array of size n, we required (n-1) passes.

7.3.1. Program for Bubble Sort:

```
#include <stdio.h>
#include <conio.h>
void bubblesort(int x[], int n)
{
    int i, j, temp;
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n-i-1 ; j++)
        {
            if (x[j] > x[j+1])
            {
                temp = x[j];
                x[j] = x[j+1];
                x[j+1] = temp;
            }
        }
    }
}

main()
{
    int i, n, x[25];
    clrscr();
    printf("\n Enter the number of elements: ");
    scanf("%d", &n);
    printf("\n Enter Data:");
    for(i = 0; i < n ; i++)
        scanf("%d", &x[i]);
    bubblesort(x, n);
    printf ("\n Array Elements after sorting: ");
    for (i = 0; i < n; i++)
        printf ("%5d", x[i]);
}
```

Time Complexity:

The bubble sort method of sorting an array of size n requires $(n-1)$ passes and $(n-1)$ comparisons on each pass. Thus the total number of comparisons is $(n-1) * (n-1) = n^2 - 2n + 1$, which is $O(n^2)$. Therefore bubble sort is very inefficient when there are more elements to sorting.

7.4. Selection Sort:

Selection sort will not require no more than $n-1$ interchanges. Suppose x is an array of size n stored in memory. The selection sort algorithm first selects the smallest element in the array x and place it at array position 0; then it selects the next smallest element in the array x and place it at array position 1. It simply continues this procedure until it places the biggest element in the last position of the array.

The array is passed through $(n-1)$ times and the smallest element is placed in its respective position in the array as detailed below:

Pass 1: Find the location j of the smallest element in the array $x[0], x[1], x[n-1]$, and then interchange $x[j]$ with $x[0]$. Then $x[0]$ is sorted.

Pass 2: Leave the first element and find the location j of the smallest element in the sub-array $x[1], x[2], \dots, x[n-1]$, and then interchange $x[1]$ with $x[j]$. Then $x[0], x[1]$ are sorted.

Pass 3: Leave the first two elements and find the location j of the smallest element in the sub-array $x[2], x[3], \dots, x[n-1]$, and then interchange $x[2]$ with $x[j]$. Then $x[0], x[1], x[2]$ are sorted.

Pass (n-1): Find the location j of the smaller of the elements $x[n-2]$ and $x[n-1]$, and then interchange $x[j]$ and $x[n-2]$. Then $x[0], x[1], \dots, x[n-2]$ are sorted. Of course, during this pass $x[n-1]$ will be the biggest element and so the entire array is sorted.

Time Complexity:

In general we prefer selection sort in case where the insertion sort or the bubble sort requires excessive swapping. In spite of superiority of the selection sort over bubble sort and the insertion sort (there is significant decrease in run time), its efficiency is also $O(n^2)$ for n data items.

Example:

Let us consider the following example with 9 elements to analyze selection Sort:

1	2	3	4	5	6	7	8	9	Remarks
65	70	75	80	50	60	55	85	45	find the first smallest element
i								j	swap $a[i]$ & $a[j]$
45	70	75	80	50	60	55	85	65	find the second smallest element
	i			j					swap $a[i]$ and $a[j]$
45	50	75	80	70	60	55	85	65	Find the third smallest element
		i				j			swap $a[i]$ and $a[j]$
45	50	55	80	70	60	75	85	65	Find the fourth smallest element
			i		j				swap $a[i]$ and $a[j]$
45	50	55	60	70	80	75	85	65	Find the fifth smallest element
				i				j	swap $a[i]$ and $a[j]$
45	50	55	60	65	80	75	85	70	Find the sixth smallest element
					i			j	swap $a[i]$ and $a[j]$
45	50	55	60	65	70	75	85	80	Find the seventh smallest element
						i j			swap $a[i]$ and $a[j]$
45	50	55	60	65	70	75	85	80	Find the eighth smallest element
							i	J	swap $a[i]$ and $a[j]$
45	50	55	60	65	70	75	80	85	The outer loop ends.

7.4.1. Non-recursive Program for selection sort:

```
# include<stdio.h>
# include<conio.h>

void selectionSort( int low, int high );

int a[25];

int main()
{
    int num, i= 0;
    clrscr();
    printf( "Enter the number of elements: " );
    scanf("%d", &num);
    printf( "\nEnter the elements:\n" );
    for(i=0; i < num; i++)
        scanf( "%d", &a[i] );
    selectionSort( 0, num - 1 );
    printf( "\nThe elements after sorting are: " );
    for( i=0; i< num; i++ )
        printf( "%d  ", a[i] );
    return 0;
}

void selectionSort( int low, int high )
{
    int i=0, j=0, temp=0, minindex;
    for( i=low; i <= high; i++ )
    {
        minindex = i;
        for( j=i+1; j <= high; j++ )
        {
            if( a[j] < a[minindex] )
                minindex = j;
        }
        temp = a[i];
        a[i] = a[minindex];
        a[minindex] = temp;
    }
}
```

7.4.2. Recursive Program for selection sort:

```
#include <stdio.h>
#include<conio.h>

int x[6] = {77, 33, 44, 11, 66};
selectionSort(int);

main()
{
    int i, n = 0;
    clrscr();
    printf ( " Array Elements before sorting: ");
    for (i=0; i<5; i++)
```



```

        printf ("%d ", x[i]);
    selectionSort(n);                /* call selection sort */
    printf ("\n Array Elements after sorting: ");
    for (i=0; i<5; i++)
        printf ("%d ", x[i]);
}

selectionSort( int n)
{
    int k, p, temp, min;
    if (n== 4)
        return (-1);
    min = x[n];
    p = n;
    for (k = n+1; k<5; k++)
    {
        if (x[k] <min)
        {
            min = x[k];
            p = k;
        }
    }
    temp = x[n];          /* interchange x[n] and x[p] */
    x[n] = x[p];
    x[p] = temp;
    n++;
    selectionSort(n);
}

```

7.5. Quick Sort:

The quick sort was invented by Prof. C. A. R. Hoare in the early 1960's. It was one of the first most efficient sorting algorithms. It is an example of a class of algorithms that work by "divide and conquer" technique.

The quick sort algorithm partitions the original array by rearranging it into two groups. The first group contains those elements less than some arbitrary chosen value taken from the set, and the second group contains those elements greater than or equal to the chosen value. The chosen value is known as the *pivot* element. Once the array has been rearranged in this way with respect to the *pivot*, the same partitioning procedure is recursively applied to each of the two subsets. When all the subsets have been partitioned and rearranged, the original array is sorted.

The function partition() makes use of two pointers up and down which are moved toward each other in the following fashion:

1. Repeatedly increase the pointer 'up' until $a[up] \geq pivot$.
2. Repeatedly decrease the pointer 'down' until $a[down] \leq pivot$.
3. If $down > up$, interchange $a[down]$ with $a[up]$
4. Repeat the steps 1, 2 and 3 till the 'up' pointer crosses the 'down' pointer. If 'up' pointer crosses 'down' pointer, the position for pivot is found and place pivot element in 'down' pointer position.

The program uses a recursive function quicksort(). The algorithm of quick sort function sorts all elements in an array 'a' between positions 'low' and 'high'.

1. It terminates when the condition $low \geq high$ is satisfied. This condition will be satisfied only when the array is completely sorted.
2. Here we choose the first element as the 'pivot'. So, $pivot = x[low]$. Now it calls the partition function to find the proper position j of the element $x[low]$ i.e. pivot. Then we will have two sub-arrays $x[low], x[low+1], \dots, x[j-1]$ and $x[j+1], x[j+2], \dots, x[high]$.
3. It calls itself recursively to sort the left sub-array $x[low], x[low+1], \dots, x[j-1]$ between positions low and $j-1$ (where j is returned by the partition function).
4. It calls itself recursively to sort the right sub-array $x[j+1], x[j+2], \dots, x[high]$ between positions $j+1$ and $high$.

The time complexity of quick sort algorithm is of **$O(n \log n)$** .

Algorithm

Sorts the elements $a[p], \dots, a[q]$ which reside in the global array $a[n]$ into ascending order. The $a[n + 1]$ is considered to be defined and must be greater than all elements in $a[n]$; $a[n + 1] = +\infty$

quicksort (p, q)

```
{
    if ( p < q ) then
    {
        call j = PARTITION(a, p, q+1);    // j is the position of the partitioning element
        call quicksort(p, j - 1);
        call quicksort(j + 1 , q);
    }
}
```

partition(a, m, p)

```
{
    v = a[m]; up = m; down = p;          // a[m] is the partition element
    do
    {
        repeat
            up = up + 1;
        until (a[up] ≥ v);

        repeat
            down = down - 1;
        until (a[down] ≤ v);
        if (up < down) then call interchange(a, up, down);
    } while (up ≥ down);

    a[m] = a[down];
    a[down] = v;
    return (down);
}
```

```

interchange(a, up, down)
{
    p = a[up];
    a[up] = a[down];
    a[down] = p;
}

```

Example:

Select first element as the pivot element. Move 'up' pointer from left to right in search of an element larger than pivot. Move the 'down' pointer from right to left in search of an element smaller than pivot. If such elements are found, the elements are swapped.

This process continues till the 'up' pointer crosses the 'down' pointer. If 'up' pointer crosses 'down' pointer, the position for pivot is found and interchange pivot and element at 'down' position.

Let us consider the following example with 13 elements to analyze quick sort:

1	2	3	4	5	6	7	8	9	10	11	12	13	Remarks
38	08	16	06	79	57	24	56	02	58	04	70	45	
pivot				up						down			swap up & down
pivot				04						79			
pivot					up			down					swap up & down
pivot					02			57					
pivot						down	up						swap pivot & down
(24	08	16	06	04	02)	38	(56	57	58	79	70	45)	
pivot					down	up							swap pivot & down
(02	08	16	06	04)	24								
pivot, down	up												swap pivot & down
02	(08	16	06	04)									
	pivot	up		down									swap up & down
	pivot	04		16									
	pivot		down	Up									
	(06	04)	08	(16)									swap pivot & down
	pivot	down	up										
	(04)	06											swap pivot & down
	04												
	pivot, down, up												
				16									
				pivot, down, up									
(02	04	06	08	16	24)	38							

							(56	57	58	79	70	45)	
							pivot	up				down	swap up & down
							pivot	45				57	
							pivot	down	up				swap pivot & down
							(45)	56	(58	79	70	57)	
							45 pivot, down, up						swap pivot & down
									(58 pivot	79 up	70	57) down	swap up & down
										57		79	
										down	up		
									(57)	58	(70	79)	swap pivot & down
									57 pivot, down, up				
											(70	79)	
											pivot, down	up	swap pivot & down
											70		
												79 pivot, down, up	
							(45	56	57	58	70	79)	
02	04	06	08	16	24	38	45	56	57	58	70	79	

7.5.1. Recursive program for Quick Sort:

```
# include<stdio.h>
# include<conio.h>

void quicksort(int, int);
int partition(int, int);
void interchange(int, int);
int array[25];

int main()
{
    int num, i = 0;
    clrscr();
    printf( "Enter the number of elements: " );
    scanf( "%d", &num);
    printf( "Enter the elements: " );
    for(i=0; i < num; i++)
        scanf( "%d", &array[i] );
    quicksort(0, num -1);
    printf( "\nThe elements after sorting are: " );
}
```

```

        for(i=0; i < num; i++)
            printf("%d ", array[i]);
        return 0;
    }

void quicksort(int low, int high)
{
    int pivotpos;
    if( low < high )
    {
        pivotpos = partition(low, high + 1);
        quicksort(low, pivotpos - 1);
        quicksort(pivotpos + 1, high);
    }
}

int partition(int low, int high)
{
    int pivot = array[low];
    int up = low, down = high;

    do
    {
        do
            up = up + 1;
        while(array[up] < pivot );

        do
            down = down - 1;
        while(array[down] > pivot);

        if(up < down)
            interchange(up, down);

    } while(up < down);
    array[low] = array[down];
    array[down] = pivot;
    return down;
}

void interchange(int i, int j)
{
    int temp;
    temp = array[i];
    array[i] = array[j];
    array[j] = temp;
}

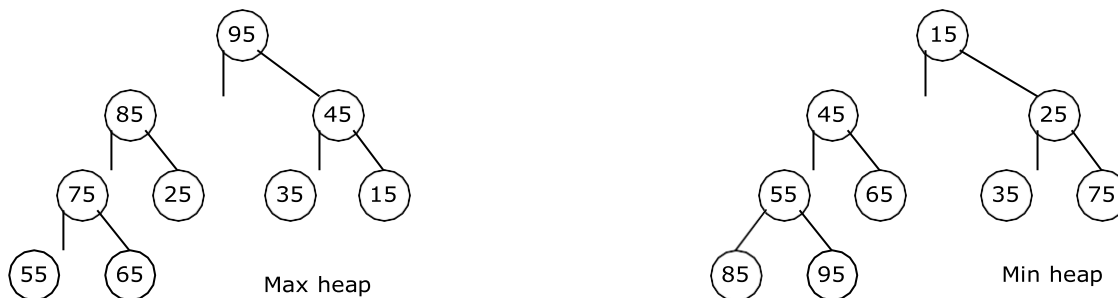
```

7.6. Priority Queue, Heap and Heap Sort:

Heap is a data structure, which permits one to insert elements into a set and also to find the largest element efficiently. A data structure, which provides these two operations, is called a priority queue.

7.6.1. Max and Min Heap data structures:

A max heap is an almost complete binary tree such that the value of each node is greater than or equal to those in its children.



A min heap is an almost complete binary tree such that the value of each node is less than or equal to those in its children.

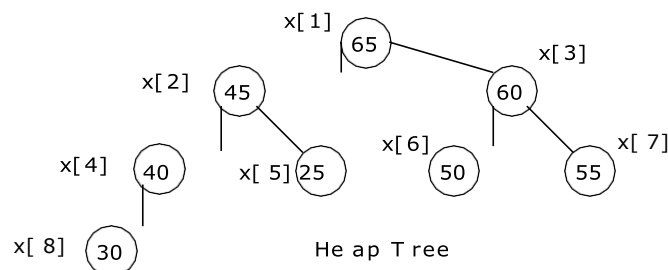
7.6.2. Representation of Heap Tree:

Since heap is a complete binary tree, a heap tree can be efficiently represented using one dimensional array. This provides a very convenient way of figuring out where children belong to.

- The root of the tree is in location 1.
- The left child of an element stored at location i can be found in location $2*i$.
- The right child of an element stored at location i can be found in location $2*i+1$.
- The parent of an element stored at location i can be found at location $\text{floor}(i/2)$.

The elements of the array can be thought of as lying in a tree structure. A heap tree represented using a single array looks as follows:

X[1]	X[2]	X[3]	X[4]	X[5]	X[6]	X[7]	X[8]
65	45	60	40	25	50	55	30



7.6.3. Operations on heap tree:

The major operations required to be performed on a heap tree:

1. Insertion,
2. Deletion and
3. Merging.

Insertion into a heap tree:

This operation is used to insert a node into an existing heap tree satisfying the properties of heap tree. Using repeated insertions of data, starting from an empty heap tree, one can build up a heap tree.

Let us consider the heap (max) tree. The principle of insertion is that, first we have to adjoin the data in the complete binary tree. Next, we have to compare it with the data in its parent; if the value is greater than that at parent then interchange the values. This will continue between two nodes on path from the newly inserted node to the root node till we get a parent whose value is greater than its child or we reached the root.

For illustration, 35 is added as the right child of 80. Its value is compared with its parent's value, and to be a max heap, parent's value greater than child's value is satisfied, hence interchange as well as further comparisons are no more required.

As another illustration, let us consider the case of insertion 90 into the resultant heap tree. First, 90 will be added as left child of 40, when 90 is compared with 40 it requires interchange. Next, 90 is compared with 80, another interchange takes place. Now, our process stops here, as 90 is now in root node. The path on which these comparisons and interchanges have taken places are shown by *dashed line*.

The algorithm Max_heap_insert to insert a data into a max heap tree is as follows:

```
Max_heap_insert (a, n)
{
    //inserts the value in a[n] into the heap which is stored at a[1] to a[n-1]
    int i, n;
    i = n;
    item = a[n];
    while ( (i > 1) and (a[ ≤ i/2 f ] < item ) do
    {
        a[i] = a[ ≤ i/2 f ] ;           // move the parent down
        i = ≤ i/2 f ;
    }
    a[i] = item ;
    return true ;
}
```

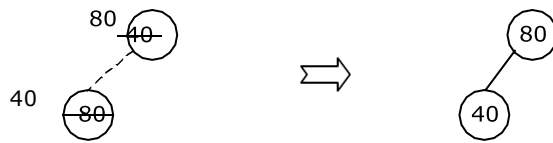
Example:

Form a heap using the above algorithm for the data: 40, 80, 35, 90, 45, 50, 70.

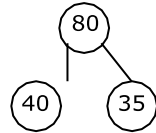
1. Insert 40:



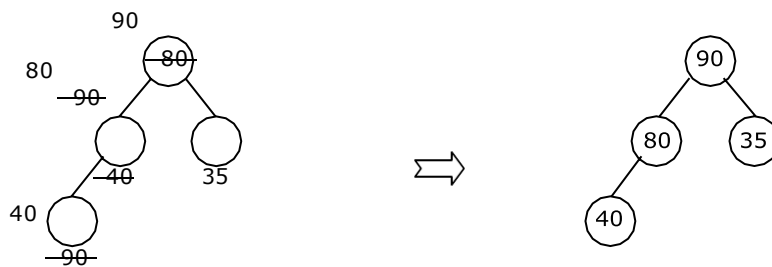
2. Insert 80:



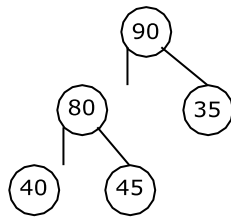
3. Insert 35:



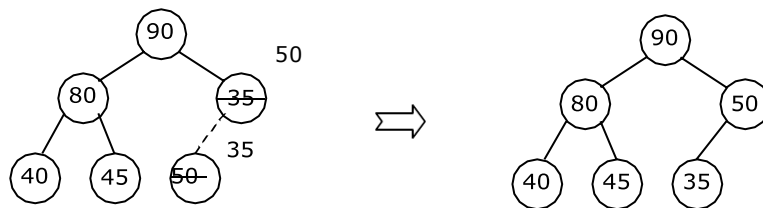
4. Insert 90:



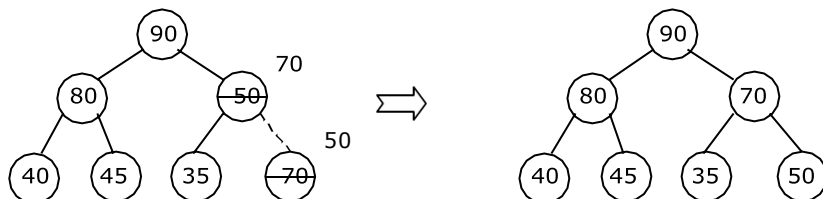
5. Insert 45:



6. Insert 50:



7. Insert 70:



Deletion of a node from heap tree:

Any node can be deleted from a heap tree. But from the application point of view, deleting the root node has some special importance. The principle of deletion is as follows:

- Read the root node into a temporary storage say, ITEM.
- Replace the root node by the last node in the heap tree. Then re-heap the tree as stated below:
 - Let newly modified root node be the current node. Compare its value with the value of its two child. Let X be the child whose value is the largest. Interchange the value of X with the value of the current node.
 - Make X as the current node.
 - Continue re-heap, if the current node is not an empty node.

The algorithm for the above is as follows:

delmax (a, n, x)

```
// delete the maximum from the heap a[n] and store it in x
{
    if (n = 0) then
    {
        write ("heap is empty");
        return false;
    }
    x = a[1]; a[1] = a[n];
    adjust (a, 1, n-1);
    return true;
}
```

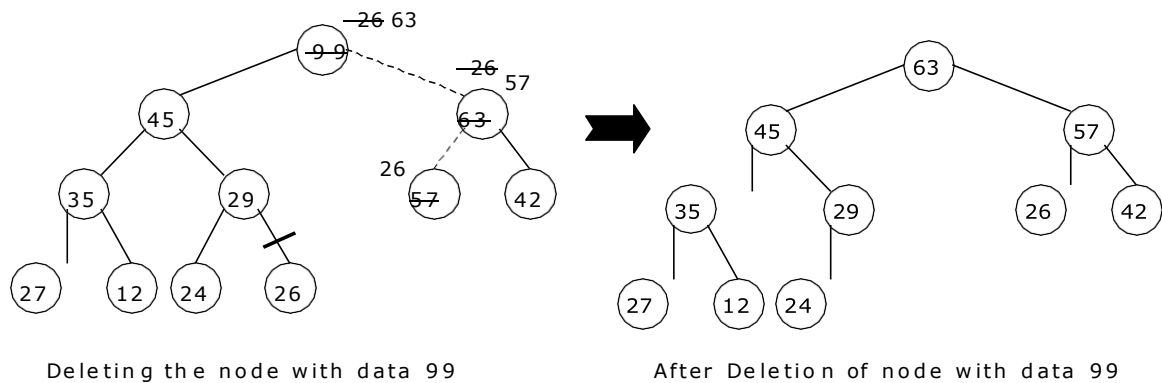
adjust (a, i, n)

// The complete binary trees with roots $a(2*i)$ and $a(2*i + 1)$ are combined with $a(i)$ to form a single heap, $1 \leq i \leq n$. No node has an address greater than n or less than 1. //

```
{
    j = 2 * i ;
    item = a[i] ;
    while (j ≤ n) do
    {
        if ((j < n) and (a (j) < a (j + 1))) then j @ j + 1;
        // compare left and right child and let j be the larger child
        if (item ≥ a (j)) then break;
        // a position for item is found
        else a [ ≤ j / 2 f ] = a[j] // move the larger child up a level
        j = 2 * j;
    }
    a [ ≤ j / 2 f ] = item;
}
```

Here the root node is 99. The last node is 26, it is in the level 3. So, 99 is replaced by 26 and this node with data 26 is removed from the tree. Next 26 at root node is compared with its two child 45 and 63. As 63 is greater, they are interchanged. Now,

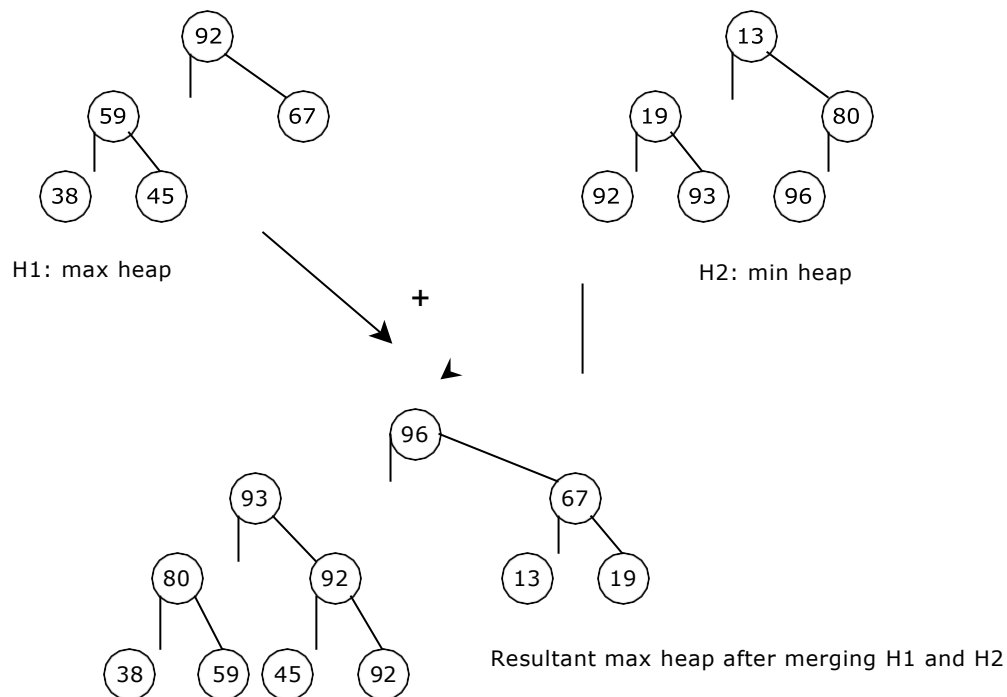
26 is compared with its children, namely, 57 and 42, as 57 is greater, so they are interchanged. Now, 26 appears as the leave node, hence re-heap is completed.



7.6.4. Merging two heap trees:

Consider, two heap trees H1 and H2. Merging the tree H2 with H1 means to include all the node from H2 to H1. H2 may be min heap or max heap and the resultant tree will be min heap if H1 is min heap else it will be max heap. Merging operation consists of two steps: Continue steps 1 and 2 while H2 is not empty:

1. Delete the root node, say x, from H2. Re-heap H2.
2. Insert the node x into H1 satisfying the property of H1.



7.6.5. Application of heap tree:

They are two main applications of heap trees known are:

1. Sorting (Heap sort) and
2. Priority queue implementation.

7.7. HEAP SORT:

A heap sort algorithm works by first organizing the data to be sorted into a special type of binary tree called a heap. Any kind of data can be sorted either in ascending order or in descending order using heap tree. It does this with the following steps:

1. Build a heap tree with the given set of data.
2. a. Remove the top most item (the largest) and replace it with the last element in the heap.
b. Re-heapify the complete binary tree.
c. Place the deleted node in the output.
3. Continue step 2 until the heap tree is empty.

Algorithm:

This algorithm sorts the elements $a[n]$. Heap sort rearranges them in-place in non-decreasing order. First transform the elements into a heap.

heapsort(a, n)

```
{
    heapify(a, n);
    for i = n to 2 by - 1 do
    {
        temp = a[i];
        a[i] = a[1];
        a[1] = temp;
        adjust (a, 1, i - 1);
    }
}
```

heapify (a, n)

//Readjust the elements in $a[n]$ to form a heap.

```
{
    for i @  $\leq n/2$  f to 1 by - 1 do adjust (a, i, n);
}
```

adjust (a, i, n)

// The complete binary trees with roots $a(2*i)$ and $a(2*i + 1)$ are combined with $a(i)$ to form a single heap, $1 \leq i \leq n$. No node has an address greater than n or less than 1. //

```
{
    j = 2 * i ;
    item = a[i] ;
    while (j  $\leq$  n) do
    {
        if ((j < n) and (a (j) < a (j + 1))) then j @ j + 1;
        // compare left and right child and let j be the larger child
        if (item  $\geq$  a (j)) then break;
        // a position for item is found
        else a [  $\leq j / 2$  f ] = a[j] // move the larger child up a level
        j = 2 * j;
    }
    a [  $\leq j / 2$  f ] = item;
}
```

Time Complexity:

Each 'n' insertion operations takes $O(\log k)$, where 'k' is the number of elements in the heap at the time. Likewise, each of the 'n' remove operations also runs in time $O(\log k)$, where 'k' is the number of elements in the heap at the time.

Since we always have $k \leq n$, each such operation runs in $O(\log n)$ time in the worst case.

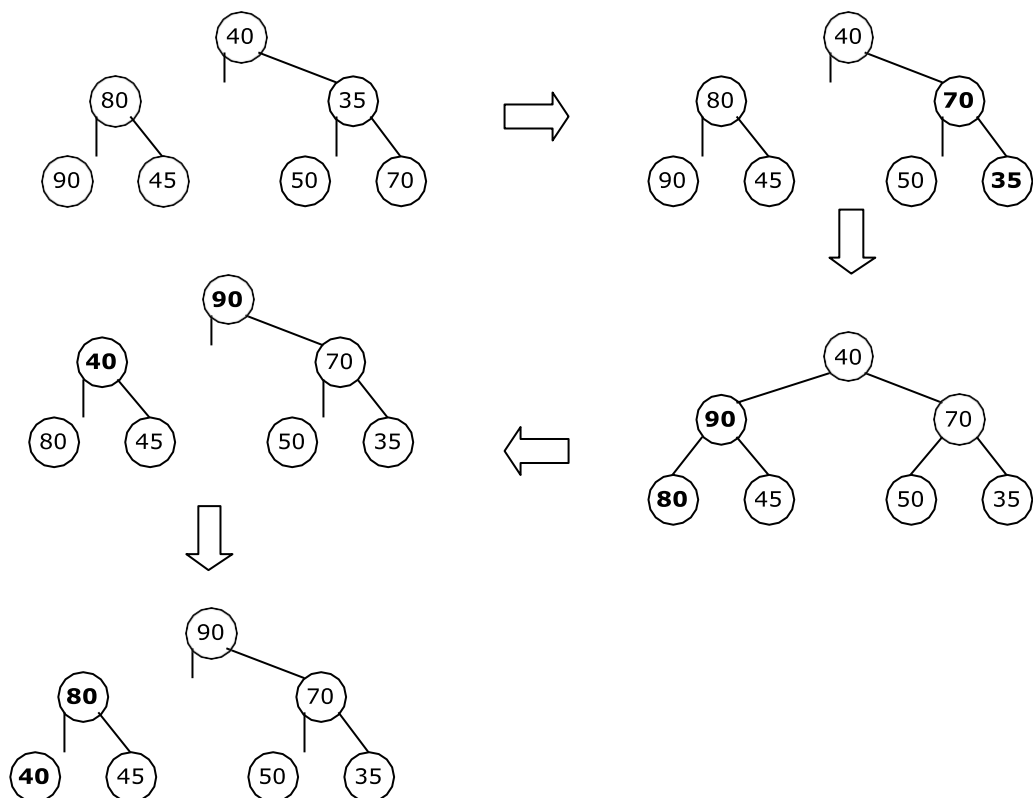
Thus, for 'n' elements it takes $O(n \log n)$ time, so the priority queue sorting algorithm runs in $O(n \log n)$ time when we use a heap to implement the priority queue.

Example 1:

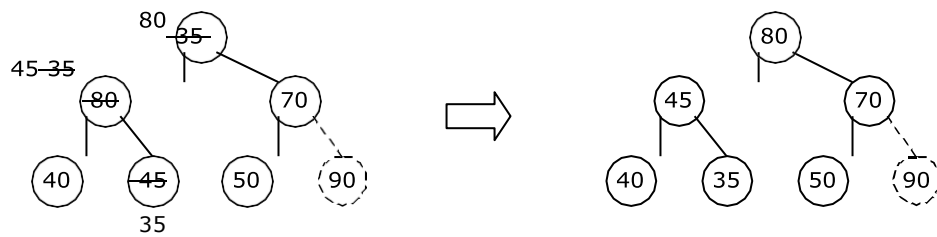
Form a heap from the set of elements (40, 80, 35, 90, 45, 50, 70) and sort the data using heap sort.

Solution:

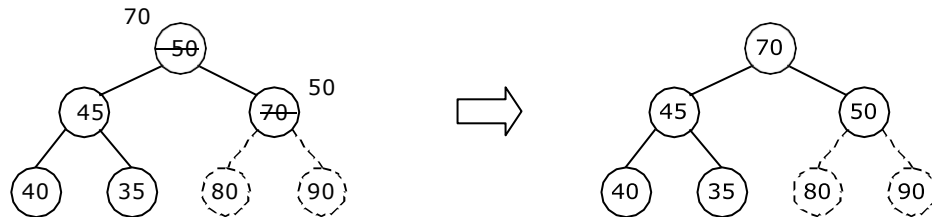
First form a heap tree from the given set of data and then sort by repeated deletion operation:



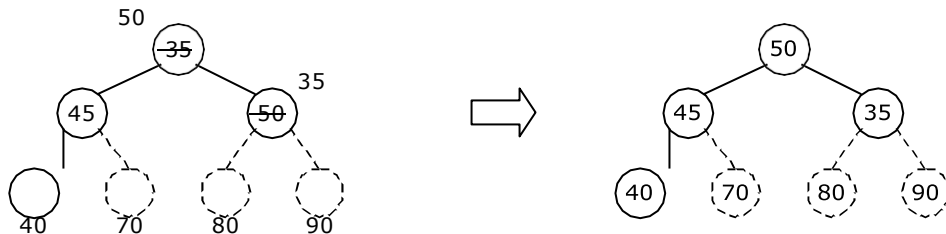
1. Exchange root 90 with the last element 35 of the array and re-heapify



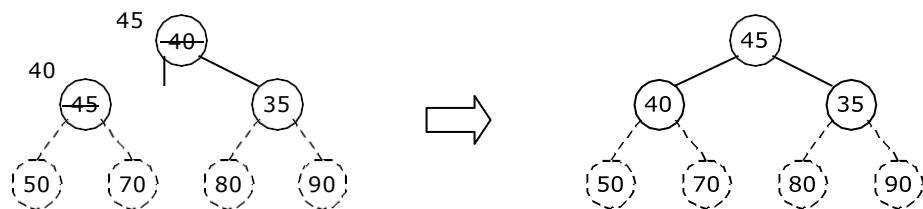
2. Exchange root 80 with the last element 50 of the array and re-heapify



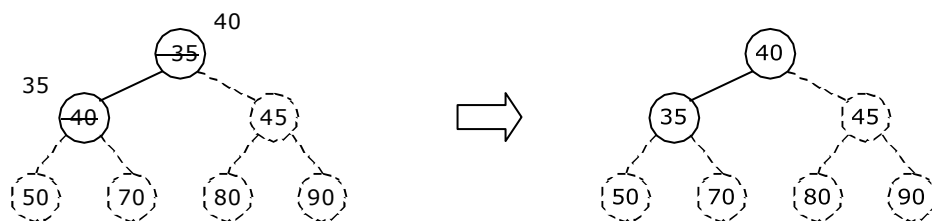
3. Exchange root 70 with the last element 35 of the array and re-heapify



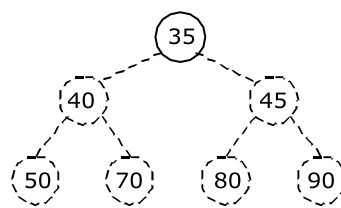
4. Exchange root 50 with the last element 40 of the array and re-heapify



5. Exchange root 45 with the last element 35 of the array and re-heapify



6. Exchange root 40 with the last element 35 of the array and re-heapify



The sorted tree

7.7.1. Program for Heap Sort:

```
void adjust(int i, int n, int a[])
{
    int j, item;
    j = 2 * i;
    item = a[i];
    while(j <= n)
    {
        if((j < n) && (a[j] < a[j+1]))
            j++;
        if(item >= a[j])
            break;
        else
        {
            a[j/2] = a[j];
            j = 2*j;
        }
    }
    a[j/2] = item;
}

void heapify(int n, int a[])
{
    int i;
    for(i = n/2; i > 0; i--)
        adjust(i, n, a);
}

void heapsort(int n, int a[])
{
    int temp, i;
    heapify(n, a);
    for(i = n; i > 0; i--)
    {
        temp = a[i];
        a[i] = a[1];
        a[1] = temp;
        adjust(1, i - 1, a);
    }
}

void main()
{
    int i, n, a[20];
    clrscr();
    printf("\n How many element you want: ");
    scanf("%d", &n);
    printf("Enter %d elements: ", n);
    for (i=1; i<=n; i++)
        scanf("%d", &a[i]);
    heapsort(n, a);
    printf("\n The sorted elements are: \n");
    for (i=1; i<=n; i++)
        printf("%5d", a[i]);
    getch();
}
```

}

7.8. Priority queue implementation using heap tree:

Priority queue can be implemented using circular array, linked list etc. Another simplified implementation is possible using heap tree; the heap, however, can be represented using an array. This implementation is therefore free from the complexities of circular array and linked list but getting the advantages of simplicities of array.

As heap trees allow the duplicity of data in it. Elements associated with their priority values are to be stored in from of heap tree, which can be formed based on their priority values. The top priority element that has to be processed first is at the root; so it can be deleted and heap can be rebuilt to get the next element to be processed, and so on. As an illustration, consider the following processes with their priorities:

Process	P ₁	P ₂	P ₃	P ₄	P ₅	P ₆	P ₇	P ₈	P ₉	P ₁₀
Priority	5	4	3	4	5	5	3	2	1	5

These processes enter the system in the order as listed above at time 0, say. Assume that a process having higher priority value will be serviced first. The heap tree can be formed considering the process priority values. The order of servicing the process is successive deletion of roots from the heap.

Exercises

1. Write a recursive "C" function to implement binary search and compute its time complexity.
2. Find the expected number of passes, comparisons and exchanges for bubble sort when the number of elements is equal to "10". Compare these results with the actual number of operations when the given sequence is as follows: 7, 1, 3, 4, 10, 9, 8, 6, 5, 2.
3. An array contains "n" elements of numbers. The several elements of this array may contain the same number "x". Write an algorithm to find the total number of elements which are equal to "x" and also indicate the position of the first such element in the array.
4. When a "C" function to sort a matrix row-wise and column-wise. Assume that the matrix is represented by a two dimensional array.
5. A very large array of elements is to be sorted. The program is to be run on a personal computer with limited memory. Which sort would be a better choice: Heap sort or Quick sort? Why?
6. Here is an array of ten integers: 5 3 8 9 1 7 0 2 6 4
Suppose we partition this array using quicksort's partition function and using 5 for the pivot. Draw the resulting array after the partition finishes.
7. Here is an array which has just been partitioned by the first step of quicksort: 3, 0, 2, 4, 5, 8, 7, 6, 9. Which of these elements could be the pivot? (There may be more than one possibility!)
8. Show the result of inserting 10, 12, 1, 14, 6, 5, 8, 15, 3, 9, 7, 4, 11, 13, and 2, one at a time, into an initially empty binary heap.
9. Sort the sequence 3, 1, 4, 5, 9, 2, 6, 5 using insertion sort.

10. Show how heap sort processes the input 142, 543, 123, 65, 453, 879, 572, 434, 111, 242, 811, 102.
11. Sort 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5 using quick sort with median-of-three partitioning and a cutoff of 3.

Multiple Choice Questions

1. What is the worst-case time for serial search finding a single item in an array? [D]
 A. Constant time
 B. Quadratic time
 C. Logarithmic time
 D. Linear time
2. What is the worst-case time for binary search finding a single item in an array? [B]
 A. Constant time
 B. Quadratic time
 C. Logarithmic time
 D. Linear time
3. What additional requirement is placed on an array, so that binary search may be used to locate an entry? [C]
 A. The array elements must form a heap.
 B. The array must have at least 2 entries
 C. The array must be sorted.
 D. The array's size must be a power of two.
4. Which searching can be performed recursively ? [B]
 A. linear search
 B. both
 C. Binary search
 D. none
5. Which searching can be performed iteratively ? [B]
 A. linear search
 B. both
 C. Binary search
 D. none
6. In a selection sort of n elements, how many times is the swap function called in the complete execution of the algorithm? [B]
 A. 1
 B. n^2
 C. $n - 1$
 D. $n \log n$
7. Selection sort and quick sort both fall into the same category of sorting algorithms. What is this category? [B]
 A. $O(n \log n)$ sorts
 B. Interchange sorts
 C. Divide-and-conquer sorts
 D. Average time is quadratic
8. Suppose that a selection sort of 100 items has completed 42 iterations of the

th
e

main loop. How many items are now guaranteed to be in their final spot (never to be moved again)? [C]

- A. 21
- B. 41
- C. 42
- D. 43

9. When is insertion sort a good choice for sorting an array? [B]
- A. Each component of the array requires a large amount of memory
 - B. The array has only a few items out of place
 - C. Each component of the array requires a small amount of memory
 - D. The processor speed is fast

10. What is the worst-case time for quick sort to sort an array of n elements? [D]
 A. $O(\log n)$ C. $O(n \log n)$
 B. $O(n)$ D. $O(n^2)$
11. Suppose we are sorting an array of eight integers using quick sort, and we have just finished the first partitioning with the array looking like this:
 2 5 1 7 9 12 11 10 Which statement is correct? [A]
 A. The pivot could be either the 7 or the 9.
 B. The pivot is not the 7, but it could be the 9.
 C. The pivot could be the 7, but it is not the 9.
 D. Neither the 7 nor the 9 is the pivot
12. What is the worst-case time for heap sort to sort an array of n elements? [C]
 A. $O(\log n)$ C. $O(n \log n)$
 B. $O(n)$ D. $O(n^2)$
13. Suppose we are sorting an array of eight integers using heap sort, and we have just finished one of the reheapifications downward. The array now looks like this: 6 4 5 1 2 7 8 [B]
 How many reheapifications downward have been performed so far?
 A. 1 C. 2
 B. 3 or 4 D. 5 or 6
14. Time complexity of inserting an element to a heap of n elements is of the order of [A]
 A. $\log_2 n$ C. $n \log_2 n$
 B. n^2 D. n
15. A min heap is the tree structure where smallest element is available at the [B]
 A. leaf C. intermediate parent
 B. root D. any where
16. In the quick sort method , a desirable choice for the portioning element will be [C]
 A. first element of list C. median of list
 B. last element of list D. any element of list
17. Quick sort is also known as [D]
 A. merge sort C. heap sort
 B. bubble sort D. none
18. Which design algorithm technique is used for quick sort . [A]
 A. Divide and conqueror C. backtrack
 B. greedy D. dynamic programming

19. Which among the following is fastest sorting technique (for unordered data) [C]
A. Heap sort
B. Selection Sort
C. Quick Sort
D. Bubble sort
20. In which searching technique elements are eliminated by half in each pass . [C]
A. Linear search
B. both
C. Binary search
D. none
21. Running time of Heap sort algorithm is ----- [B]
A. $O(\log_2 n)$
B. $O(n \log_2 n)$
C. $O(n)$
D. $O(n^2)$

22. Running time of Bubble sort algorithm is ----- [D]
 A. $O(\log_2 n)$ C. $O(n)$
 B. $O(n \log_2 n)$ D. $O(n^2)$
23. Running time of Selection sort algorithm is ----- [D]
 A. $O(\log_2 n)$ C. $O(n)$
 B. $O(n \log_2 n)$ D. $O(n^2)$
24. The Max heap constructed from the list of numbers 30,10,80,60,15,55 is [C]
 A. 60,80,55,30,10,15 C. 80,55,60,15,10,30
 B. 80,60,55,30,10,15 D. none
25. The number of swappings needed to sort the numbers 8,22,7,9,31,19,5,13 in ascending order using bubble sort is [D]
 A. 11 C. 13
 B. 12 D. 14
26. Time complexity of insertion sort algorithm in best case is [C]
 A. $O(\log_2 n)$ C. $O(n)$
 B. $O(n \log_2 n)$ D. $O(n^2)$
27. Binary search algorithm performs efficiently on a [C]
 A. linked list C. array
 B. both D. none
28. Which is a stable sort ? [D]
 A. Bubble sort C. Quick sort
 B. Selection Sort D. none
29. Heap is a good data structure to implement [A]
 A. priority Queue C. linear queue
 B. Deque D. none
30. Always Heap is a [A]
 A. complete Binary tree C. Full Binary tree
 B. Binary Search Tree D. none

Chapter-5:Polymorphism

Polymorphism in Java with example

Polymorphism is one of the [OOps](#) feature that allows us to perform a single action in different ways. For example, lets say we have a class `Animal` that has a method `sound()`. Since this is a generic class so we can't give it a implementation like: Roar, Meow, Oink etc. We had to give a generic message.

```
public class Animal{
    ...
    public void sound(){
        System.out.println("Animal is making a sound");
    }
}
```

Now lets say we two subclasses of `Animal` class: `Horse` and `Cat` that extends (see [Inheritance](#)) `Animal` class. We can provide the implementation to the same method like this:

```
public class Horse extends Animal{
    ...
    @Override
    public void sound(){
        System.out.println("Neigh");
    }
}
```

and

```
public class Cat extends Animal{
    ...
    @Override
    public void sound(){
        System.out.println("Meow");
    }
}
```

As you can see that although we had the common action for all subclasses `sound()` but there were different ways to do the same action. This is a perfect example of polymorphism (feature that allows us to perform a single action in different ways). It would not make any sense to just call the generic `sound()` method as each `Animal` has a different sound. Thus we can say that the action this method performs is based on the type of object.

What is polymorphism in programming?

Polymorphism is the capability of a method to do different things based on the object that it is acting upon. In other words, polymorphism allows you define one interface and have multiple implementations. As we have seen in the

above example that we have defined the method `sound()` and have the multiple implementations of it in the different-2 sub classes. Which `sound()` method will be called is determined at runtime so the example we gave above is a **runtime polymorphism example**.

Types of polymorphism and method overloading & overriding are covered in the separate tutorials. You can refer them here:

1. [Method Overloading in Java](#) – This is an example of compile time (or static polymorphism)
2. [Method Overriding in Java](#) – This is an example of runtime time (or dynamic polymorphism)
3. [Types of Polymorphism – Runtime and compile time](#) – This is our next tutorial where we have covered the types of polymorphism in detail. I would recommend you to go through method overloading and overriding before going through this topic. Let's write down the complete code of it:

Example 1: Polymorphism in Java

Runtime Polymorphism example:

Animal.java

```
public class Animal{
    public void sound(){
        System.out.println("Animal is making a sound");
    }
}
```

Horse.java

```
class Horse extends Animal{
    @Override
    public void sound(){
        System.out.println("Neigh");
    }
    public static void main(String args[]){
        Animal obj = new Horse();
        obj.sound();
    }
}
```

Output:

```
Neigh
```

Cat.java

```
public class Cat extends Animal{
    @Override
    public void sound(){
        System.out.println("Meow");
    }
    public static void main(String args[]){
```



```

        Animal obj = new Cat();
        obj.sound();
    }
}

```

Output:

Meow

Example 2: Compile time Polymorphism

Method Overloading on the other hand is a compile time polymorphism example.

```

class Overload
{
    void demo (int a)
    {
        System.out.println ("a: " + a);
    }
    void demo (int a, int b)
    {
        System.out.println ("a and b: " + a + "," + b);
    }
    double demo(double a) {
        System.out.println("double a: " + a);
        return a*a;
    }
}
class MethodOverloading
{
    public static void main (String args [])
    {
        Overload Obj = new Overload();
        double result;
        Obj .demo(10);
        Obj .demo(10, 20);
        result = Obj .demo(5.5);
        System.out.println("O/P : " + result);
    }
}

```

Here the method `demo()` is overloaded 3 times: first method has 1 int parameter, second method has 2 int parameters and third one is having double parameter. Which method is to be called is determined by the arguments we pass while calling methods. This happens at compile time so this type of polymorphism is known as compile time polymorphism.

Output:

```

a: 10
a and b: 10,20
double a: 5.5
O/P : 30.25

```

Types of polymorphism in java- Runtime and Compile time polymorphism

BY CHAITANYA SINGH | FILED UNDER: [OOPS CONCEPT](#)

In the last tutorial we discussed [Polymorphism in Java](#). In this guide we will see **types of polymorphism**. There are two types of polymorphism in java:

- 1) **Static Polymorphism** also known as compile time polymorphism
- 2) **Dynamic Polymorphism** also known as runtime polymorphism

Compile time Polymorphism (or Static polymorphism)

Polymorphism that is resolved during compiler time is known as static polymorphism. Method overloading is an example of compile time polymorphism.

Method Overloading: This allows us to have more than one method having the same name, if the parameters of methods are different in number, sequence and data types of parameters.

Method Overloading in Java with examples

Method Overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different. It is similar to [constructor overloading](#) in Java, that allows a class to have more than one constructor having different argument lists.

let's get back to the point, when I say argument list it means the parameters that a method has: For example the argument list of a method `add(int a, int b)` having two parameters is different from the argument list of the method `add(int a, int b, int c)` having three parameters.

Three ways to overload a method

In order to overload a method, the argument lists of the methods must differ in either of these:

1. Number of parameters.

For example: This is a valid case of overloading

```
add(int, int)
add(int, int, int)
```

2. Data type of parameters.

For example:

```
add(int, int)
add(int, float)
```

3. Sequence of Data type of parameters.

For example:

```
add(int, float)
add(float, int)
```

Invalid case of method overloading:

When I say argument list, I am not talking about return type of the method, for example if two methods have same name, same parameters and have different return type, then this is not a valid method overloading example. This will throw compilation error.

```
int add(int, int)
float add(int, int)
```

Method overloading is an example of [Static Polymorphism](#). We will discuss [polymorphism](#) and types of it in a separate tutorial.

Points to Note:

1. Static Polymorphism is also known as compile time binding or early binding.
2. [Static binding](#) happens at compile time. Method overloading is an example of static binding where binding of method call to its definition happens at Compile time.

Method Overloading examples

As discussed in the beginning of this guide, method overloading is done by declaring same method with different parameters. The parameters must be different in either of these: number, sequence or types of parameters (or arguments). Lets see examples of each of these cases.

Argument list is also known as parameter list

Example 1: Overloading - Different Number of parameters in argument list

This example shows how method overloading is done by having different number of parameters

```
class DisplayOverloading
{
    public void disp(char c)
```

```

    {
        System.out.println(c);
    }
    public void disp(char c, int num)
    {
        System.out.println(c + " "+num);
    }
}
class Sample
{
    public static void main(String args[])
    {
        DisplayOverloading obj = new DisplayOverloading();
        obj.disp('a');
        obj.disp('a',10);
    }
}

```

Output:

```

a
a 10

```

In the above example – method `disp()` is overloaded based on the number of parameters – We have two methods with the name `disp` but the parameters they have are different. Both are having different number of parameters.

Example 2: Overloading - Difference in data type of parameters

In this example, method `disp()` is overloaded based on the data type of parameters – We have two methods with the name `disp()`, one with parameter of char type and another method with the parameter of int type.

```

class DisplayOverloading2
{
    public void disp(char c)
    {
        System.out.println(c);
    }
    public void disp(int c)
    {
        System.out.println(c );
    }
}

class Sample2
{
    public static void main(String args[])
    {
        DisplayOverloading2 obj = new DisplayOverloading2();
        obj.disp('a');
        obj.disp(5);
    }
}

```

Output:

```
a
5
```

Example3: Overloading - Sequence of data type of arguments

Here method `disp()` is overloaded based on sequence of data type of parameters – Both the methods have different sequence of data type in argument list. First method is having argument list as (char, int) and second is having (int, char). Since the sequence is different, the method can be overloaded without any issues.

```
class DisplayOverloading3
{
    public void disp(char c, int num)
    {
        System.out.println("I'm the first definition of method disp");
    }
    public void disp(int num, char c)
    {
        System.out.println("I'm the second definition of method disp" );
    }
}
class Sample3
{
    public static void main(String args[])
    {
        DisplayOverloading3 obj = new DisplayOverloading3();
        obj.disp('x', 51 );
        obj.disp(52, 'y');
    }
}
```

Output:

```
I'm the first definition of method disp
I'm the second definition of method disp
```

Method Overloading and Type Promotion

When a data type of smaller size is promoted to the data type of bigger size than this is called type promotion, for example: byte data type can be promoted to short, a short data type can be promoted to int, long, double etc.

What it has to do with method overloading?

Well, it is very important to understand type promotion else you will think that the program will throw compilation error but in fact that program will run fine because of type promotion.

Lets take an example to see what I am talking here:

```
class Demo{
    void disp(int a, double b){
        System.out.println("Method A");
    }
}
```

```

    }
    void disp(int a, double b, double c){
        System.out.println("Method B");
    }
    public static void main(String args[]){
        Demo obj = new Demo();
        /* I am passing float value as a second argument but
        * it got promoted to the type double, because there
        * wasn't any method having arg list as (int, float)
        */
        obj.disp(100, 20.67f);
    }
}

```

Output:

Method A

As you can see that I have passed the float value while calling the disp() method but it got promoted to the double type as there wasn't any method with argument list as (int, float)

But this type promotion doesn't always happen, let's see another example:

```

class Demo{
    void disp(int a, double b){
        System.out.println("Method A");
    }
    void disp(int a, double b, double c){
        System.out.println("Method B");
    }
    void disp(int a, float b){
        System.out.println("Method C");
    }
    public static void main(String args[]){
        Demo obj = new Demo();
        /* This time promotion won't happen as there is
        * a method with arg list as (int, float)
        */
        obj.disp(100, 20.67f);
    }
}

```

Output:

Method C

As you see that this time type promotion didn't happen because there was a method with matching argument type.

Type Promotion table:

The data type on the left side can be promoted to any of the data types present in the right side of it.

```

byte → short → int → long
short → int → long
int → long → float → double
float → double

```

long → float → double

Lets see few Valid/invalid cases of method overloading

Case 1:

```
int mymethod(int a, int b, float c)
int mymethod(int var1, int var2, float var3)
```

Result: Compile time error. Argument lists are exactly same. Both methods are having same number, data types and same sequence of data types.

Case 2:

```
int mymethod(int a, int b)
int mymethod(float var1, float var2)
```

Result: Perfectly fine. Valid case of overloading. Here data types of arguments are different.

Case 3:

```
int mymethod(int a, int b)
int mymethod(int num)
```

Result: Perfectly fine. Valid case of overloading. Here number of arguments are different.

Case 4:

```
float mymethod(int a, float b)
float mymethod(float var1, int var2)
```

Result: Perfectly fine. Valid case of overloading. Sequence of the data types of parameters are different, first method is having (int, float) and second is having (float, int).

Case 5:

```
int mymethod(int a, int b)
float mymethod(int var1, int var2)
```

Result: Compile time error. Argument lists are exactly same. Even though return type of methods are different, it is not a valid case. Since return type of method doesn't matter while overloading a method.

Guess the answers before checking it at the end of programs:

Question 1 – return type, method name and argument list same.

```
class Demo
```

```

{
    public int myMethod(int num1, int num2)
    {
        System.out.println("First myMethod of class Demo");
        return num1+num2;
    }
    public int myMethod(int var1, int var2)
    {
        System.out.println("Second myMethod of class Demo");
        return var1-var2;
    }
}
class Sample4
{
    public static void main(String args[])
    {
        Demo obj1= new Demo();
        obj1.myMethod(10,10);
        obj1.myMethod(20,12);
    }
}

```

Answer:

It will throw a compilation error: More than one method with same name and argument list cannot be defined in a same class.

Question 2 – return type is different. Method name & argument list same.

```

class Demo2
{
    public double myMethod(int num1, int num2)
    {
        System.out.println("First myMethod of class Demo");
        return num1+num2;
    }
    public int myMethod(int var1, int var2)
    {
        System.out.println("Second myMethod of class Demo");
        return var1-var2;
    }
}
class Sample5
{
    public static void main(String args[])
    {
        Demo2 obj2= new Demo2();
        obj2.myMethod(10,10);
        obj2.myMethod(20,12);
    }
}

```

Answer:

It will throw a compilation error: More than one method with same name and argument list cannot be given in a class even though their return type is different. **Method return type doesn't matter in case of overloading.**

Example of static Polymorphism

Method overloading is one of the way java supports static polymorphism. Here we have two definitions of the same method add() which add method would be called is determined by the parameter list at the compile time. That is the reason this is also known as compile time polymorphism.

```
class SimpleCalculator
{
    int add(int a, int b)
    {
        return a+b;
    }
    int add(int a, int b, int c)
    {
        return a+b+c;
    }
}
public class Demo
{
    public static void main(String args[])
    {
        SimpleCalculator obj = new SimpleCalculator();
        System.out.println(obj.add(10, 20));
        System.out.println(obj.add(10, 20, 30));
    }
}
```

Output:

```
30
60
```

Runtime Polymorphism (or Dynamic polymorphism)

It is also known as Dynamic Method Dispatch. Dynamic polymorphism is a process in which a call to an overridden method is resolved at runtime, thats why it is called runtime polymorphism.

Method overriding in java with example

Declaring a method in **sub class** which is already present in **parent class** is known as method overriding. Overriding is done so that a child class can give its own implementation to a method which is already provided by the parent class. In this case the method in parent class is called overridden method and the method in child class is called overriding method. In this guide, we will see what is method overriding in Java and why we use it.

Method Overriding Example

Lets take a simple example to understand this. We have two classes: A child class Boy and a parent class Human. The Boy class extends Human class. Both the classes have a common method void eat(). Boy class is giving its own implementation to the eat() method or in other words it is overriding the eat() method.

The purpose of Method Overriding is clear here. Child class wants to give its own implementation so that when it calls this method, it prints Boy is eating instead of Human is eating.

```
class Human{
    //Overridden method
    public void eat()
    {
        System.out.println("Human is eating");
    }
}
class Boy extends Human{
    //Overriding method
    public void eat(){
        System.out.println("Boy is eating");
    }
    public static void main( String args[]) {
        Boy obj = new Boy();
        //This will call the child class version of eat()
        obj.eat();
    }
}
```

Output:

```
Boy is eating
```

Advantage of method overriding

The main advantage of method overriding is that the class can give its own specific implementation to a inherited method **without even modifying the parent class code**.

This is helpful when a class has several child classes, so if a child class needs to use the parent class method, it can use it and the other classes that want to have different implementation can use overriding feature to make changes without touching the parent class code.

Method Overriding and Dynamic Method Dispatch

Method Overriding is an example of [runtime polymorphism](#). When a parent class reference points to the child class object then the call to the overridden method is determined at runtime, because during method call which method(parent class or child class) is to be executed is determined by the

type of object. This process in which call to the overridden method is resolved at runtime is known as dynamic method dispatch. Lets see an example to understand this:

```
class ABC{
    //Overridden method
    public void disp()
    {
        System.out.println("disp() method of parent class");
    }
}
class Demo extends ABC{
    //Overriding method
    public void disp(){
        System.out.println("disp() method of Child class");
    }
    public void newMethod(){
        System.out.println("new method of child class");
    }
    public static void main( String args[]) {
        /* When Parent class reference refers to the parent class object
        * then in this case overridden method (the method of parent class)
        * is called.
        */
        ABC obj = new ABC();
        obj.disp();

        /* When parent class reference refers to the child class object
        * then the overriding method (method of child class) is called.
        * This is called dynamic method dispatch and runtime polymorphism
        */
        ABC obj2 = new Demo();
        obj2.disp();
    }
}
```

Output:

```
disp() method of parent class
disp() method of Child class
```

In the above example the call to the disp() method using second object (obj2) is runtime polymorphism (or dynamic method dispatch).

Note: In dynamic method dispatch the object can call the overriding methods of child class and all the non-overridden methods of base class but it cannot call the methods which are newly declared in the child class. In the above example the object obj2 is calling the disp(). However if you try to call the newMethod() method (which has been newly declared in Demo class) using obj2 then you would give compilation error with the following message:

```
Exception in thread "main" java.lang.Error: Unresolved compilation
problem: The method xyz() is undefined for the type ABC
```

Rules of method overriding in Java

1. Argument list: The argument list of overriding method (method of child class) must match the Overridden method(the method of parent class). The data types of the arguments and their sequence should exactly match.
2. Access Modifier of the overriding method (method of subclass) cannot be more restrictive than the overridden method of parent class. For e.g. if the Access Modifier of parent class method is public then the overriding method (child class method) cannot have private, protected and default Access modifier,because all of these three access modifiers are more restrictive than public.

For e.g. This is **not allowed** as child class disp method is more restrictive(protected) than base class(public)

```
3. class MyBaseClass{
4.     public void disp()
5.     {
6.         System.out.println("Parent class method");
7.     }
8. }
9. class MyChildClass extends MyBaseClass{
10.    protected void disp(){
11.        System.out.println("Child class method");
12.    }
13.    public static void main( String args[]) {
14.        MyChildClass obj = new MyChildClass();
15.        obj.disp();
16.    }
}
```

Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation
problem: Cannot reduce the visibility of the inherited method from
MyBaseClass
```

However this is perfectly valid scenario as public is less restrictive than protected. Same access modifier is also a valid one.

```
class MyBaseClass{
    protected void disp()
    {
        System.out.println("Parent class method");
    }
}
class MyChildClass extends MyBaseClass{
    public void disp(){
        System.out.println("Child class method");
    }
    public static void main( String args[]) {
        MyChildClass obj = new MyChildClass();
        obj.disp();
    }
}
```

Output:

```
Child class method
```

17. private, static and final methods cannot be overridden as they are local to the class. However static methods can be re-declared in the sub class, in this case the sub-class method would act differently and will have nothing to do with the same static method of parent class.
18. Overriding method (method of child class) can throw unchecked exceptions, regardless of whether the overridden method(method of parent class) throws any exception or not. However the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. We will discuss this in detail with example in the upcoming tutorial.
19. Binding of overridden methods happen at runtime which is known as dynamic binding.
20. If a class is extending an abstract class or implementing an interface then it has to override all the abstract methods unless the class itself is a abstract class.

Super keyword in Method Overriding

The super keyword is used for calling the parent class method/constructor. `super.myMethod()` calls the `myMethod()` method of base class while `super()` calls the constructor of base class. Let's see the use of super in method Overriding.

As we know that we we override a method in child class, then call to the method using child class object calls the overridden method. By using super we can call the overridden method as shown in the example below:

```
class ABC{
    public void myMethod()
    {
        System.out.println("Overridden method");
    }
}
class Demo extends ABC{
    public void myMethod(){
        //This will call the myMethod() of parent class
        super.myMethod();
        System.out.println("Overriding method");
    }
    public static void main( String args[]) {
        Demo obj = new Demo();
        obj.myMethod();
    }
}
```

Output:

```
Class ABC: mymethod()
Class Test: mymethod()
```

As you see using super keyword, we can access the overridden method.

Example

In this example we have two classes ABC and XYZ. ABC is a parent class and XYZ is a child class. The child class is overriding the method myMethod() of parent class. In this example we have child class object assigned to the parent class reference so in order to determine which method would be called, the type of the object would be determined at run-time. It is the type of object that determines which version of the method would be called (not the type of reference).

To understand the concept of overriding, you should have the basic knowledge of [inheritance in Java](#).

```
class ABC{
    public void myMethod(){
        System.out.println("Overridden Method");
    }
}
public class XYZ extends ABC{

    public void myMethod(){
        System.out.println("Overriding Method");
    }
    public static void main(String args[]){
        ABC obj = new XYZ();
        obj.myMethod();
    }
}
```

Output:

Overriding Method

When an overridden method is called through a reference of parent class, then type of the object determines which method is to be executed. Thus, this determination is made at run time.

Since both the classes, child class and parent class have the same method `animalSound`. Which version of the method(child class or parent class) will be called is determined at runtime by JVM.

Few more overriding examples:

```
ABC obj = new ABC();
obj.myMethod();
// This would call the myMethod() of parent class ABC

XYZ obj = new XYZ();
obj.myMethod();
// This would call the myMethod() of child class XYZ

ABC obj = new XYZ();
```

```
obj.myMethod();  
// This would call the myMethod() of child class XYZ
```

In the third case the method of child class is to be executed because which method is to be executed is determined by the type of object and since the object belongs to the child class, the child class version of myMethod() is called.


Constructor Overloading in Java with examples

Like methods, [constructors](#) can also be overloaded. In this guide we will see Constructor overloading with the help of examples. Before we proceed further let's understand what is constructor overloading and why we do it.

Constructor overloading is a concept of having more than one constructor with different parameters list, in such a way so that each constructor performs a different task. For e.g. `Vector` class has 4 types of constructors. If you do not want to specify the initial capacity and capacity increment then you can simply use default constructor of [Vector class](#) like this `Vector v = new Vector();`; however if you need to specify the capacity and increment then you call the parameterized constructor of `Vector` class with two int arguments like this: `Vector v= new Vector(10, 5);`

Beginnersbook.com

```
public class Demo {  
    Demo() {  
        ..  
    }  
    Demo(String s) {  
        ...  
    }  
    Demo(int i) {  
        ...  
    }  
    .....  
}
```



Three overloaded constructors -
They must have
different
Parameters list

You must have understood the purpose of constructor overloading. Lets see how to overload a constructor with the help of following java program.

Constructor Overloading Example

Here we are creating two objects of class `StudentData`. One is with default constructor and another one using parameterized constructor. Both the constructors have different initialization code, similarly you can create any number of constructors with different-2 initialization codes for different-2 purposes.

StudentData.java

```
class StudentData
{
    private int stuID;
    private String stuName;
    private int stuAge;
    StudentData()
    {
        //Default constructor
        stuID = 100;
        stuName = "New Student";
        stuAge = 18;
    }
    StudentData(int num1, String str, int num2)
    {
        //Parameterized constructor
        stuID = num1;
        stuName = str;
        stuAge = num2;
    }
    //Getter and setter methods
    public int getStuID() {
        return stuID;
    }
    public void setStuID(int stuID) {
        this.stuID = stuID;
    }
    public String getStuName() {
        return stuName;
    }
    public void setStuName(String stuName) {
        this.stuName = stuName;
    }
    public int getStuAge() {
        return stuAge;
    }
    public void setStuAge(int stuAge) {
        this.stuAge = stuAge;
    }
}

public static void main(String args[])
{
    //This object creation would call the default constructor
    StudentData myobj = new StudentData();
    System.out.println("Student Name is: "+myobj.getStuName());
    System.out.println("Student Age is: "+myobj.getStuAge());
    System.out.println("Student ID is: "+myobj.getStuID());

    /*This object creation would call the parameterized
    * constructor StudentData(int, String, int)*/
    StudentData myobj2 = new StudentData(555, "Chaitanya", 25);
}
```



```

        System.out.println("Student Name is: "+myobj2.getStuName());
        System.out.println("Student Age is: "+myobj2.getStuAge());
        System.out.println("Student ID is: "+myobj2.getStuID());
    }
}

```

Output:

```

Student Name is: New Student
Student Age is: 18
Student ID is: 100
Student Name is: Chaitanya
Student Age is: 25
Student ID is: 555

```

Let's understand the role of this () in constructor overloading

```

public class OverloadingExample2
{
    private int rollNum;
    OverloadingExample2()
    {
        rollNum =100;
    }
    OverloadingExample2(int rnum)
    {
        this();
        /*this() is used for calling the default
        * constructor from parameterized constructor.
        * It should always be the first statement
        * inside constructor body.
        */
        rollNum = rollNum+ rnum;
    }
    public int getRollNum() {
        return rollNum;
    }
    public void setRollNum(int rollNum) {
        this.rollNum = rollNum;
    }
    public static void main(String args[])
    {
        OverloadingExample2 obj = new OverloadingExample2(12);
        System.out.println(obj.getRollNum());
    }
}

```

Output:

112

As you can see in the above program that we called the parameterized constructor during object creation. Since we have this() placed in parameterized constructor, the default constructor got invoked from it and initialized the variable rollNum.

Test your skills – Guess the output of the following program

```
public class OverloadingExample2
{
    private int rollNum;
    OverloadingExample2()
    {
        rollNum =100;
    }
    OverloadingExample2(int rnum)
    {

        rollNum = rollNum+ rnum;
        this();
    }
    public int getRollNum() {
        return rollNum;
    }
    public void setRollNum(int rollNum) {
        this.rollNum = rollNum;
    }
    public static void main(String args[])
    {
        OverloadingExample2 obj = new OverloadingExample2(12);
        System.out.println(obj.getRollNum());
    }
}
```

Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation
problem:Constructor call must be the first statement in a constructor
```

Program gave a compilation error. **Reason:** this() should be the first statement inside a constructor.

Another Constructor overloading Example

Another important point to note while overloading a constructor is: When we don't implement any constructor, the java compiler inserts the default constructor into our code during compilation, however if we implement any constructor then compiler doesn't do it. See the example below.

```
public class Demo
{
    private int rollNum;
    //We are not defining a no-arg constructor here

    Demo(int rnum)
    {
        rollNum = rollNum+ rnum;
    }
    //Getter and Setter methods

    public static void main(String args[])
```

```
{  
    //This statement would invoke no-arg constructor  
    Demo obj = new Demo();  
}  
}
```

Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation  
problem:The constructor Demo() is undefined
```

Difference between method Overloading and Overriding in java

In this we will discuss the difference between overloading and overriding in Java.

1. Method overloading in java
2. Method overriding in java

Overloading vs Overriding in Java

1. Overloading happens at **compile-time** while Overriding happens at **runtime**: The binding of overloaded method call to its definition happens at compile-time however binding of overridden method call to its definition happens at runtime.
2. Static methods can be overloaded which means a class can have more than one static method of same name. Static methods cannot be overridden, even if you declare a same static method in child class it has nothing to do with the same method of parent class.
3. The most basic difference is that overloading is being done in the same class while for overriding base and child classes are required. Overriding is all about giving a specific implementation to the inherited method of parent class.
4. **Static binding** is being used for overloaded methods and **dynamic binding** is being used for overridden/overriding methods.
5. Performance: Overloading gives better performance compared to overriding. The reason is that the binding of overridden methods is being done at runtime.
6. private and final methods can be overloaded but they cannot be overridden. It means a class can have more than one private/final methods of same name but a child class cannot override the private/final methods of their base class.

7. Return type of method does not matter in case of method overloading, it can be same or different. However in case of method overriding the overriding method can have more specific return type ([refer this](#)).
8. Argument list should be different while doing method overloading. Argument list should be same in method Overriding.

Overloading example

```
//A class for adding upto 5 numbers
class Sum
{
    int add(int n1, int n2)
    {
        return n1+n2;
    }
    int add(int n1, int n2, int n3)
    {
        return n1+n2+n3;
    }
    int add(int n1, int n2, int n3, int n4)
    {
        return n1+n2+n3+n4;
    }
    int add(int n1, int n2, int n3, int n4, int n5)
    {
        return n1+n2+n3+n4+n5;
    }
    public static void main(String args[])
    {
        Sum obj = new Sum();
        System.out.println("Sum of two numbers: "+obj.add(20, 21));
        System.out.println("Sum of three numbers: "+obj.add(20, 21, 22));
        System.out.println("Sum of four numbers: "+obj.add(20, 21, 22, 23));
        System.out.println("Sum of five numbers: "+obj.add(20, 21, 22, 23, 24));
    }
}
```

Output:

```
Sum of two numbers: 41
Sum of three numbers: 63
Sum of four numbers: 86
Sum of five numbers: 110
```

Here we have 4 versions of same method `add`. We are overloading the method `add()` here.

Overriding example

```
package beginnersbook.com;
class CarClass
{
    public int speedLimit()
    {
        return 100;
    }
}
```

```

}
class Ford extends CarClass
{
    public int speedLimit()
    {
        return 150;
    }
    public static void main(String args[])
    {
        CarClass obj = new Ford();
        int num= obj.speedLimit();
        System.out.println("Speed Limit is: "+num);
    }
}

```

Output:

```
Speed Limit is: 150
```

Here speedLimit() method of class Ford is overriding the speedLimit() method of class CarClass.

Typecasting is converting one data type to another.

Up-casting – Converting a subclass type to a superclass type is known as up casting.

Example

```

class Super {
    void Sample() {
        System.out.println("method of super class");
    }
}

public class Sub extends Super {
    void Sample() {
        System.out.println("method of sub class");
    }

    public static void main(String args[]) {
        Super obj =(Super) new Sub(); obj.Sample();
    }
}

```

```
}
```

Down-casting – Converting a superclass type to a subclass type is known as downcasting.

Example

```
class Super {  
    void Sample() {  
        System.out.println("method of super class");  
    }  
}  
  
public class Sub extends Super {  
    void Sample() {  
        System.out.println("method of sub class");  
    }  
  
    public static void main(String args[]) {  
        Super obj = new Sub();  
        Sub sub = (Sub) obj; sub.Sample();  
    }  
}
```

What is Upcasting and Downcasting in Java

Perhaps in your daily Java coding, you see (and use) **upcasting** and **downcasting** occasionally. You may hear the terms ‘casting’, ‘upcasting’, ‘downcasting’ from someone or somewhere, and you may be confused about them.

As you read on, you will realize that upcasting and downcasting are really simple.

Before we go into the details, suppose that we have the following class hierarchy:

Mammal > Animal > Dog, Cat

Mammal is the super interface:

```
1 public interface Mammal {  
2     public void eat();  
3     public void move();  
4 }
```

```

5     public void sleep();
6 }
7

```

Animal is the abstract class:

```

1
2 public abstract class Animal implements Mammal {
3     public void eat() {
4         System.out.println("Eating...");
5     }
6     public void move() {
7         System.out.println("Moving...");
8     }
9     public void sleep() {
10        System.out.println("Sleeping...");
11    }
12
13 }
14

```

Dog and **Cat** are the two concrete sub classes:

```

1
2 public class Dog extends Animal {
3     public void bark() {
4         System.out.println("Gow gow!");
5     }
6     public void eat() {
7         System.out.println("Dog is eating...");
8     }
9 }
10 public class Cat extends Animal {
11     public void meow() {
12         System.out.println("Meow Meow!");
13     }
14 }

```

1. What is Upcasting in Java?

Upcasting is casting a subtype to a supertype, upward to the inheritance tree. Let's see an example:

```

1 Dog dog = new Dog();
2 Animal anim = (Animal) dog;
3 anim.eat();

```

Here, we cast the **Dog** type to the **Animal** type. Because **Animal** is the supertype of **Dog**, this casting is called upcasting.

Note that the actual object type does not change because of casting. The **Dog** object is still a **Dog** object. Only the reference type gets changed. Hence the above code produces the following output:

```

1 Dog is eating...

```

Upcasting is always safe, as we treat a type to a more general one. In the above example, an **Animal** has all behaviors of a **Dog**.

This is also another example of upcasting:

```
1 Mammal mam = new Cat();
2 Animal anim = new Dog();
```

2. Why is Upcasting in Java?

Generally, upcasting is not necessary. However, we need upcasting when we want to write general code that deals with only the supertype. Consider the following class:

```
1 public class AnimalTrainer {
2     public void teach(Animal anim) {
3         anim.move();
4         anim.eat();
5     }
6 }
```

Here, the `teach()` method can accept any object which is subtype of `Animal`. So objects of type `Dog` and `Cat` will be upcasted to `Animal` when they are passed into this method:

```
1 Dog dog = new Dog();
2 Cat cat = new Cat();
3
4 AnimalTrainer trainer = new AnimalTrainer();
5 trainer.teach(dog);
6 trainer.teach(cat);
```

3. What is Downcasting in Java?

Downcasting is casting to a subtype, downward to the inheritance tree. Let's see an example:

```
1 Animal anim = new Cat();
2 Cat cat = (Cat) anim;
```

Here, we cast the `Animal` type to the `Cat` type. As `Cat` is subclass of `Animal`, this casting is called downcasting.

Unlike upcasting, downcasting can fail if the actual object type is not the target object type.

For example:

```
1 Animal anim = new Cat();
2 Dog dog = (Dog) anim;
```

This will throw a `ClassCastException` because the actual object type is `Cat`. And a `Cat` is not a `Dog` so we cannot cast it to a `Dog`.

The Java language provides the `instanceof` keyword to check type of an object before casting.

For example:

```
1 if (anim instanceof Cat) {
2     Cat cat = (Cat) anim;
3     cat.meow();
4 } else if (anim instanceof Dog) {
5     Dog dog = (Dog) anim;
6     dog.bark();
7 }
```

So if you are not sure about the original object type, use the `instanceof` operator to check the type before casting. This eliminates the risk of a `ClassCastException` thrown.

4. Why is Downcasting in Java?

Downcasting is used more frequently than upcasting. Use downcasting when we want to access specific behaviors of a subtype.

Consider the following example:

```
1
2 public class AnimalTrainer {
3     public void teach(Animal anim) {
4         // do animal-things
5         anim.move();
6         anim.eat();
7
8         // if there's a dog, tell it barks
9         if (anim instanceof Dog) {
10            Dog dog = (Dog) anim;
11            dog.bark();
12        }
13    }
14 }
```

Here, in the `teach()` method, we check if there is an instance of a `Dog` object passed in, downcast it to the `Dog` type and invoke its specific method, `bark()`.

Okay, so far you have got the nuts and bolts of upcasting and downcasting in Java.

Remember:

- Casting does not change the actual object type. Only the reference type gets changed.
- Upcasting is always safe and never fails.
- Downcasting can risk throwing a `ClassCastException`, so the `instanceof` operator is used to check type before casting.

Chapter 6: Abstract Class and Interface

Abstract Class in Java with example

A class that is declared using “**abstract**” keyword is known as abstract class. It can have abstract methods(methods without body) as well as concrete methods (regular methods with body). A normal class(non-abstract class) cannot have abstract methods. In this guide we will learn what is a abstract class, why we use it and what are the rules that we must remember while working with it in Java.

An abstract class can not be **instantiated**, which means you are not allowed to create an **object** of it. Why? We will discuss that later in this guide.

Why we need an abstract class?

Lets say we have a class `Animal` that has a method `sound()` and the subclasses(see **inheritance**) of it like `Dog`, `Lion`, `Horse`, `Cat` etc. Since the animal sound differs from one animal to another, there is no point to implement this method in parent class. This is because every child class must override this method to give its own implementation details, like `Lion` class will say “Roar” in this method and `Dog` class will say “Woof”.

So when we know that all the animal child classes will and should override this method, then there is no point to implement this method in parent class. Thus, making this method abstract would be the good choice as by making this method abstract we force all the sub classes to implement this method(otherwise you will get compilation error), also we need not to give any implementation to this method in parent class.

Since the `Animal` class has an abstract method, you must need to declare this class abstract.

Now each animal must have a sound, by making this method abstract we made it compulsory to the child class to give implementation details to this method. This way we ensures that every animal has a sound.

Abstract class Example

```
//abstract parent class
abstract class Animal{
    //abstract method
    public abstract void sound();
}
```

```
//Dog class extends Animal class
public class Dog extends Animal{

    public void sound(){
        System.out.println("Woof");
    }
    public static void main(String args[]){
        Animal obj = new Dog();
        obj.sound();
    }
}
```

Output:

Woof

Hence for such kind of scenarios we generally declare the class as abstract and later **concrete classes** extend these classes and override the methods accordingly and can have their own methods as well.

Abstract class declaration

An abstract class outlines the methods but not necessarily implements all the methods.

```
//Declaration using abstract keyword
abstract class A{
    //This is abstract method
    abstract void myMethod();

    //This is concrete method with body
    void anotherMethod(){
        //Does something
    }
}
```

Rules

Note 1: As we seen in the above example, there are cases when it is difficult or often unnecessary to implement all the methods in parent class. In these cases, we can declare the parent class as abstract, which makes it a special class which is not complete on its own.

A class derived from the abstract class must implement all those methods that are declared as abstract in the parent class.

Note 2: Abstract class cannot be instantiated which means you cannot create the object of it. To use this class, you need to create another class that extends this class and provides the implementation of abstract methods, then you can use the object of that child class to call non-abstract methods of parent class as well as implemented methods(those that were abstract in parent but implemented in child class).

Note 3: If a child does not implement all the abstract methods of abstract parent class, then the child class must need to be declared abstract as well.

Do you know? Since abstract class allows concrete methods as well, it does not provide 100% abstraction. You can say that it provides partial abstraction. Abstraction is a process where you show only “relevant” data and “hide” unnecessary details of an object from the user.

Interfaces on the other hand are used for 100% abstraction (See more about [abstraction](#) here).

You may also want to read this: [Difference between abstract class and Interface in Java](#)

Why can't we create the object of an abstract class?

Because these classes are incomplete, they have abstract methods that have no body so if java allows you to create object of this class then if someone calls the abstract method using that object then What would happen? There would be no actual implementation of the method to invoke.

Also because an object is concrete. An abstract class is like a template, so you have to extend it and build on it before you can use it.

Example to demonstrate that object creation of abstract class is not allowed

As discussed above, we cannot instantiate an abstract class. This program throws a compilation error.

```
abstract class AbstractDemo{
    public void myMethod(){
        System.out.println("Hello");
    }
    abstract public void anotherMethod();
}
public class Demo extends AbstractDemo{

    public void anotherMethod() {
        System.out.print("Abstract method");
    }
    public static void main(String args[])
    {
        //error: You can't create object of it
        AbstractDemo obj = new AbstractDemo();
        obj.anotherMethod();
    }
}
```

Output:

Unresolved compilation problem: Cannot instantiate the type AbstractDemo

Note: The class that extends the abstract class, have to implement all the abstract methods of it, else you have to declare that class abstract as well.

Abstract class vs Concrete class

A class which is not abstract is referred as **Concrete class**. In the above example that we have seen in the beginning of this guide, `Animal` is a abstract class and `Cat`, `Dog` & `Lion` are concrete classes.

Key Points:

1. An abstract class has no use until unless it is extended by some other class.
2. If you declare an **abstract method** in a class then you must declare the class abstract as well. you can't have abstract method in a concrete class. It's vice versa is not always true: If a class is not having any abstract method then also it can be marked as abstract.
3. It can have non-abstract method (concrete) as well.

I have covered the rules and examples of abstract methods in a separate tutorial, You can find the guide here: [Abstract method in Java](#)
For now lets just see some basics and example of abstract method.

- 1) Abstract method has no body.
- 2) Always end the declaration with a **semicolon(;)**.
- 3) It must be **overridden**. An abstract class must be extended and in a same way abstract method must be overridden.
- 4) A class has to be declared abstract to have abstract methods.

Note: The class which is extending abstract class must override all the abstract methods.

Example of Abstract class and method

```
abstract class MyClass{
    public void disp(){
        System.out.println("Concrete method of parent class");
    }
    abstract public void disp2();
}

class Demo extends MyClass{
    /* Must Override this method while extending
     * MyClas
     */
    public void disp2()
    {
```

```

        System.out.println("overriding abstract method");
    }
    public static void main(String args[]){
        Demo obj = new Demo();
        obj.disp2();
    }
}

```

Output:

```
overriding abstract method
```

Abstract method in Java with examples

A method without body (no implementation) is known as abstract method. A method must always be declared in an abstract class, or in other words you can say that if a class has an abstract method, it should be declared abstract as well. In the last tutorial we discussed Abstract class, if you have not yet checked it out read it here: [Abstract class in Java](#), before reading this guide. This is how an abstract method looks in java:

```
public abstract int myMethod(int n1, int n2);
```

As you see this has no body.

Rules of Abstract Method

1. Abstract methods don't have body, they just have method signature as shown above.
2. If a class has an abstract method it should be declared abstract, the vice versa is not true, which means an abstract class doesn't need to have an abstract method compulsory.
3. If a regular class extends an abstract class, then the class must have to implement all the abstract methods of abstract parent class or it has to be declared abstract as well.

Example 1: abstract method in an abstract class

```

//abstract class
abstract class Sum{
    /* These two are abstract methods, the child class
    * must implement these methods
    */
    public abstract int sumOfTwo(int n1, int n2);
    public abstract int sumOfThree(int n1, int n2, int n3);

    //Regular method
    public void disp(){
        System.out.println("Method of class Sum");
    }
}

```

```

    }
}
//Regular class extends abstract class
class Demo extends Sum{

    /* If I don't provide the implementation of these two methods, the
    * program will throw compilation error.
    */
    public int sumOfTwo(int num1, int num2){
        return num1+num2;
    }
    public int sumOfThree(int num1, int num2, int num3){
        return num1+num2+num3;
    }
    public static void main(String args[]){
        Sum obj = new Demo();
        System.out.println(obj.sumOfTwo(3, 7));
        System.out.println(obj.sumOfThree(4, 3, 19));
        obj.disp();
    }
}

```

Output:

10

26

Method of class Sum

Example 2: abstract method in interface

All the methods of an [interface](#) are public abstract by default. You cannot have concrete (regular methods with body) methods in an interface.

```

//Interface
interface Multiply{
    //abstract methods
    public abstract int multiplyTwo(int n1, int n2);

    /* We need not to mention public and abstract in interface
    * as all the methods in interface are
    * public and abstract by default so the compiler will
    * treat this as
    * public abstract multiplyThree(int n1, int n2, int n3);
    */
    int multiplyThree(int n1, int n2, int n3);

    /* Regular (or concrete) methods are not allowed in an interface
    * so if I uncomment this method, you will get compilation error
    * public void disp(){
    *     System.out.println("I will give error if u uncomment me");
    * }
    */
}

class Demo implements Multiply{
    public int multiplyTwo(int num1, int num2){
        return num1*num2;
    }
}

```

```

    public int multiplyThree(int num1, int num2, int num3){
        return num1*num2*num3;
    }
    public static void main(String args[]){
        Multiply obj = new Demo();
        System.out.println(obj.multiplyTwo(3, 7));
        System.out.println(obj.multiplyThree(1, 9, 0));
    }
}

```

Output:

```

21
0

```

Interface in java with example programs

BY CHAITANYA SINGH | FILED UNDER: [OOPS CONCEPT](#)

In the last tutorial we discussed [abstract class](#) which is used for achieving partial abstraction. Unlike abstract class an interface is used for full abstraction. Abstraction is a process where you show only “relevant” data and “hide” unnecessary details of an object from the user(See: [Abstraction](#)). In this guide, we will cover **what is an interface in java**, why we use it and what are rules that we must follow while using interfaces in [Java Programming](#).

What is an interface in Java?

Interface looks like a class but it is not a class. An interface can have methods and variables just like the class but the methods declared in interface are by default abstract (only method signature, no body, see: [Java abstract method](#)). Also, the variables declared in an interface are public, static & final by default. We will cover this in detail, later in this guide.

What is the use of interface in Java?

As mentioned above they are used for full abstraction. Since methods in interfaces do not have body, they have to be implemented by the class before you can access them. The class that implements interface must implement all the methods of that interface. Also, java programming language does not allow you to extend more than one class, However you can implement more than one interfaces in your class.

Syntax:

Interfaces are declared by specifying a keyword “interface”. E.g.:

```

interface MyInterface
{
    /* All the methods are public abstract by default

```



```

    * As you see they have no body
    */
    public void method1();
    public void method2();
}

```

Example of an Interface in Java

This is how a class implements an interface. It has to provide the body of all the methods that are declared in interface or in other words you can say that class has to implement all the methods of interface.

Do you know? class implements interface but an interface extends another interface.

```

interface MyInterface
{
    /* compiler will treat them as:
    * public abstract void method1();
    * public abstract void method2();
    */
    public void method1();
    public void method2();
}
class Demo implements MyInterface
{
    /* This class must have to implement both the abstract methods
    * else you will get compilation error
    */
    public void method1()
    {
        System.out.println("implementation of method1");
    }
    public void method2()
    {
        System.out.println("implementation of method2");
    }
    public static void main(String arg[])
    {
        MyInterface obj = new Demo();
        obj.method1();
    }
}

```

Output:

```
implementation of method1
```

You may also like to read: [Difference between abstract class and interface](#)

Interface and Inheritance

As discussed above, an interface can not implement another interface. It has to extend the other interface. See the below example where we have two

interfaces Inf1 and Inf2. Inf2 extends Inf1 so If class implements the Inf2 it has to provide implementation of all the methods of interfaces Inf2 as well as Inf1.

Learn more about inheritance here: [Java Inheritance](#)

```
interface Inf1{
    public void method1();
}
interface Inf2 extends Inf1 {
    public void method2();
}
public class Demo implements Inf2{
    /* Even though this class is only implementing the
     * interface Inf2, it has to implement all the methods
     * of Inf1 as well because the interface Inf2 extends Inf1
     */
    public void method1(){
        System.out.println("method1");
    }
    public void method2(){
        System.out.println("method2");
    }
    public static void main(String args[]){
        Inf2 obj = new Demo();
        obj.method2();
    }
}
```

In this program, the class Demo only implements interface Inf2, however it has to provide the implementation of all the methods of interface Inf1 as well, because interface Inf2 extends Inf1.

Tag or Marker interface in Java

An empty interface is known as tag or marker interface. For example Serializable, ActionListener, Remote(java.rmi.Remote) are tag interfaces. These interfaces do not have any field and methods in it. Read more about it [here](#).

Nested interfaces

An interface which is declared inside another interface or class is called **nested** interface. They are also known as inner interface. For example Entry interface in collections framework is declared inside Map interface, that's why we don't use it directly, rather we use it like this: Map.Entry.

Key points: Here are the key points to remember about interfaces:

1) We can't instantiate an interface in java. That means we cannot create the object of an interface

2) Interface provides full abstraction as none of its methods have body. On the other hand abstract class provides partial abstraction as it can have abstract and concrete(methods with body) methods both.

3) `implements` keyword is used by classes to implement an interface.

4) While providing implementation in class of any method of an interface, it needs to be mentioned as `public`.

5) Class that implements any interface must implement all the methods of that interface, else the class should be declared abstract.

6) Interface cannot be declared as `private`, `protected` or `transient`.

7) All the interface methods are by default **abstract and public**.

8) Variables declared in interface are **public, static and final** by default.

```
interface Try
{
    int a=10;
    public int a=10;
    public static final int a=10;
    final int a=10;
    static int a=0;
}
```

All of the above statements are identical.

9) Interface variables must be initialized at the time of declaration otherwise compiler will throw an error.

```
interface Try
{
    int x;//Compile-time error
}
```

Above code will throw a compile time error as the value of the variable x is not initialized at the time of declaration.

10) Inside any implementation class, you cannot change the variables declared in interface because by default, they are public, static and final. Here we are implementing the interface “Try” which has a variable x. When we tried to set the value for variable x we got compilation error as the variable x is public static **final** by default and final variables can not be re-initialized.

```
class Sample implements Try
{
```

```

public static void main(String args[])
{
    x=20; //compile time error
}

```

11) An interface can extend any interface but cannot implement it. Class implements interface and interface extends interface.

12) A **class** can implement any **number of interfaces**.

13) If there are **two or more same methods** in two interfaces and a class implements both interfaces, implementation of the method once is enough.

```

interface A
{
    public void aaa();
}
interface B
{
    public void aaa();
}
class Central implements A,B
{
    public void aaa()
    {
        //Any Code here
    }
    public static void main(String args[])
    {
        //Statements
    }
}

```

14) A class cannot implement two interfaces that have methods with same name but different return type.

```

interface A
{
    public void aaa();
}
interface B
{
    public int aaa();
}

class Central implements A,B
{
    public void aaa() // error
    {
    }
    public int aaa() // error
    {
    }
    public static void main(String args[])
    {
    }
}

```

```
}  
}
```

15) Variable names conflicts can be resolved by interface name.

```
interface A  
{  
    int x=10;  
}  
interface B  
{  
    int x=100;  
}  
class Hello implements A,B  
{  
    public static void Main(String args[])  
    {  
        /* reference to x is ambiguous both variables are x  
        * so we are using interface name to resolve the  
        * variable  
        */  
        System.out.println(x);  
        System.out.println(A.x);  
        System.out.println(B.x);  
    }  
}
```

Advantages of interface in java:

Advantages of using interfaces are as follows:

1. Without bothering about the implementation part, we can achieve the security of implementation
2. In java, **multiple inheritance** is not allowed, however you can use interface to make use of it as you can implement more than one interface.

Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

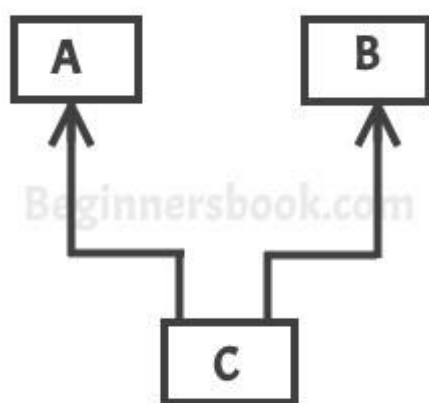
```
1. class A{
2. void msg(){System.out.println("Hello");}
3. }
4. class B{
5. void msg(){System.out.println("Welcome");}
6. }
7. class C extends A,B{//suppose if it were
8.
9. public static void main(String args[]){
10. C obj=new C();
11. obj.msg();//Now which msg() method would be invoked?
12.}
13.}
```

Compile Time Error

Does Java support Multiple inheritance?

BY CHAITANYA SINGH | FILED UNDER: [OOPS CONCEPT](#)

When one class extends more than one classes then this is called **multiple inheritance**. For example: Class C extends class A and B then this [type of inheritance](#) is known as multiple inheritance. Java doesn't allow multiple inheritance. In this article, we will discuss why java doesn't allow multiple inheritance and how we can use interfaces instead of classes to achieve the same purpose.



Multiple Inheritance

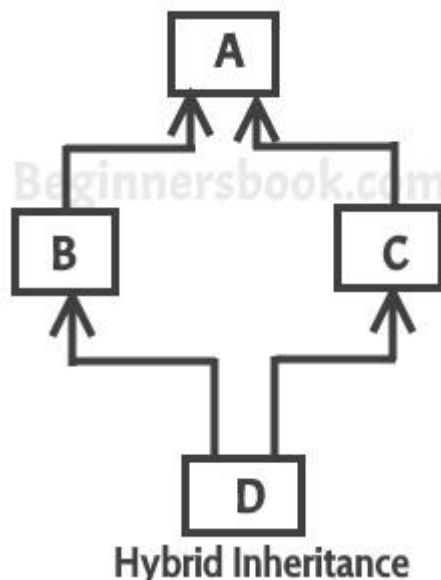
Why Java doesn't support multiple inheritance?

C++ , Common lisp and few other languages supports multiple inheritance while java doesn't support it. Java doesn't allow multiple inheritance to **avoid the ambiguity** caused by it. One of the example of such problem is the **diamond problem** that occurs in multiple inheritance.

To understand the basics of inheritance, refer this main guide: [Inheritance in Java](#)

What is diamond problem?

We will discuss this problem with the help of the diagram below: which shows multiple inheritance as Class D extends both classes B & C. Now lets assume we have a method in class A and class B & C overrides that method in their own way. **Wait!! here the problem comes** – Because D is extending both B & C so if D wants to use the same method which method would be called (the overridden method of B or the overridden method of C). Ambiguity. That's the main reason why Java doesn't support multiple inheritance.



Can we implement more than one interfaces in a class

Yes, we can implement more than one interfaces in our program because that doesn't cause any ambiguity(see the explanation below).

```
interface X
{
    public void myMethod();
}
interface Y
{
    public void myMethod();
}
```

```

class JavaExample implements X, Y
{
    public void myMethod()
    {
        System.out.println("Implementing more than one interfaces");
    }
    public static void main(String args[]){
        JavaExample obj = new JavaExample();
        obj.myMethod();
    }
}

```

Output:

Implementing more than one interfaces

As you can see that the class implemented two interfaces. A class can implement any number of interfaces. In this case there is no ambiguity even though both the interfaces are having same method. Why? Because methods in an interface are always abstract by default, which doesn't let them give their implementation (or method definition) in interface itself.

Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

```

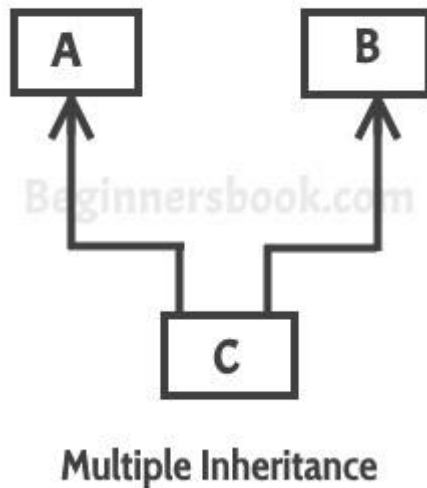
14. class A{
15. void msg(){System.out.println("Hello");}
16. }
17. class B{
18. void msg(){System.out.println("Welcome");}
19. }
20. class C extends A,B{//suppose if it were
21.
22. public static void main(String args[]){
23. C obj=new C();
24. obj.msg();//Now which msg() method would be invoked?
25. }
26. }

```


Does Java support Multiple inheritance?

BY CHAITANYA SINGH | FILED UNDER: [OOPS CONCEPT](#)

When one class extends more than one classes then this is called **multiple inheritance**. For example: Class C extends class A and B then this [type of inheritance](#) is known as multiple inheritance. Java doesn't allow multiple inheritance. In this article, we will discuss why java doesn't allow multiple inheritance and how we can use interfaces instead of classes to achieve the same purpose.



Why Java doesn't support multiple inheritance?

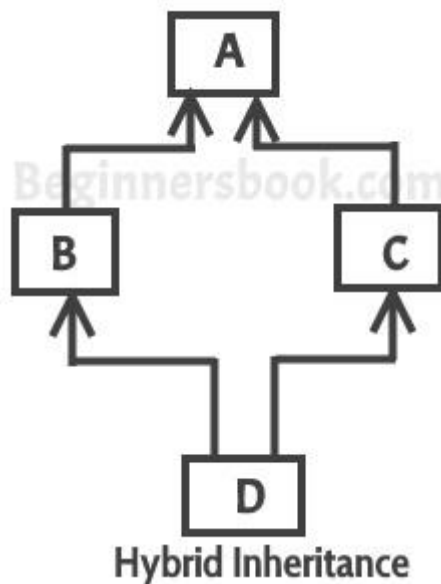
C++ , Common lisp and few other languages supports multiple inheritance while java doesn't support it. Java doesn't allow multiple inheritance to **avoid the ambiguity** caused by it. One of the example of such problem is the **diamond problem** that occurs in multiple inheritance.

To understand the basics of inheritance, refer this main guide: [Inheritance in Java](#)

What is diamond problem?

We will discuss this problem with the help of the diagram below: which shows multiple inheritance as Class D extends both classes B & C. Now lets assume we have a method in class A and class B & C overrides that method in their own way. **Wait!! here the problem comes** – Because D is extending both B & C so if D wants to use the same method which method would be called (the

overridden method of B or the overridden method of C). Ambiguity. That's the main reason why Java doesn't support multiple inheritance.



Can we implement more than one interfaces in a class

Yes, we can implement more than one interfaces in our program because that doesn't cause any ambiguity(see the explanation below).

```
interface X
{
    public void myMethod();
}
interface Y
{
    public void myMethod();
}
class JavaExample implements X, Y
{
    public void myMethod()
    {
        System.out.println("Implementing more than one interfaces");
    }
    public static void main(String args[]){
        JavaExample obj = new JavaExample();
        obj.myMethod();
    }
}
```

Output:

Implementing more than one interfaces

As you can see that the class implemented two interfaces. A class can implement any number of interfaces. In this case there is no ambiguity even though both the interfaces are having same method. Why? Because methods

in an interface are always [abstract](#) by default, which doesn't let them give their implementation (or method definition) in interface itself.

Difference Between Abstract Class and Interface in Java

In this article, we will discuss the **difference between Abstract Class and Interface in Java with examples.**

1. [Abstract class in java](#)
2. [Interface in Java](#)

	Abstract Class	Interface
1	An abstract class can extend only one class or one abstract class at a time	An interface can extend any number of interfaces at a time
2	An abstract class can extend another concrete (regular) class or abstract class	An interface can only extend another interface
3	An abstract class can have both abstract and concrete methods	An interface can have only abstract methods
4	In abstract class keyword "abstract" is mandatory to declare a method as an abstract	In an interface keyword "abstract" is optional to declare a method as an abstract

5	An abstract class can have protected and public abstract methods	An interface can have only have public abstract methods
6	An abstract class can have static, final or static final variable with any access specifier	interface can only have public static final (constant) variable

Each of the above mentioned points are explained with an example below:

Abstract class vs interface in Java

Difference No.1: Abstract class can extend only one class or one abstract class at a time

```

class Example1{
    public void display1(){
        System.out.println("display1 method");
    }
}
abstract class Example2{
    public void display2(){
        System.out.println("display2 method");
    }
}
abstract class Example3 extends Example1{
    abstract void display3();
}
class Example4 extends Example3{
    public void display3(){
        System.out.println("display3 method");
    }
}
class Demo{
    public static void main(String args[]){
        Example4 obj=new Example4();
        obj.display3();
    }
}

```

Output:

```
display3 method
```

Interface can extend any number of interfaces at a time

```

//first interface
interface Example1{

```

```

        public void display1();
    }
    //second interface
    interface Example2 {
        public void display2();
    }
    //This interface is extending both the above interfaces
    interface Example3 extends Example1,Example2{
    }
    class Example4 implements Example3{
        public void display1(){
            System.out.println("display2 method");
        }
        public void display2(){
            System.out.println("display3 method");
        }
    }
    class Demo{
        public static void main(String args[]){
            Example4 obj=new Example4();
            obj.display1();
        }
    }

```

Output:

display2 method

Difference No.2: Abstract class can be extended(inherited) by a class or an abstract class

```

class Example1{
    public void display1(){
        System.out.println("display1 method");
    }
}
abstract class Example2{
    public void display2(){
        System.out.println("display2 method");
    }
}
abstract class Example3 extends Example2{
    abstract void display3();
}
class Example4 extends Example3{
    public void display2(){
        System.out.println("Example4-display2 method");
    }
    public void display3(){
        System.out.println("display3 method");
    }
}
class Demo{
    public static void main(String args[]){
        Example4 obj=new Example4();
        obj.display2();
    }
}

```

Output:

Example4-display2 method

Interfaces can be extended only by interfaces. Classes has to implement them instead of extend

```
interface Example1{
    public void display1();
}
interface Example2 extends Example1{
}
class Example3 implements Example2{
    public void display1(){
        System.out.println("display1 method");
    }
}
class Demo{
    public static void main(String args[]){
        Example3 obj=new Example3();
        obj.display1();
    }
}
```

Output:

display1 method

Difference No.3: Abstract class can have both abstract and concrete methods

```
abstract class Example1 {
    abstract void display1();
    public void display2(){
        System.out.println("display2 method");
    }
}
class Example2 extends Example1{
    public void display1(){
        System.out.println("display1 method");
    }
}
class Demo{
    public static void main(String args[]){
        Example2 obj=new Example2();
        obj.display1();
    }
}
```

Interface can only have abstract methods, they cannot have concrete methods

```
interface Example1{
    public abstract void display1();
}
class Example2 implements Example1{
    public void display1(){
        System.out.println("display1 method");
    }
}
```

```

class Demo{
    public static void main(String args[]){
        Example2 obj=new Example2();
        obj.display1();
    }
}

```

Output:

display1 method

Difference No.4: In abstract class, the keyword ‘abstract’ is mandatory to declare a method as an abstract

```

abstract class Example1{
    public abstract void display1();
}

class Example2 extends Example1{
    public void display1(){
        System.out.println("display1 method");
    }
    public void display2(){
        System.out.println("display2 method");
    }
}

class Demo{
    public static void main(String args[]){
        Example2 obj=new Example2();
        obj.display1();
    }
}

```

In interfaces, the keyword ‘abstract’ is optional to declare a method as an abstract because all the methods are abstract by default

```

interface Example1{
    public void display1();
}

class Example2 implements Example1{
    public void display1(){
        System.out.println("display1 method");
    }
    public void display2(){
        System.out.println("display2 method");
    }
}

class Demo{
    public static void main(String args[]){
        Example2 obj=new Example2();
        obj.display1();
    }
}

```

Difference No.5: Abstract class can have protected and public abstract methods

```

abstract class Example1{

```

```

    protected abstract void display1();
    public abstract void display2();
    public abstract void display3();
}
class Example2 extends Example1{
    public void display1(){
        System.out.println("display1 method");
    }
    public void display2(){
        System.out.println("display2 method");
    }
    public void display3(){
        System.out.println("display3 method");
    }
}
class Demo{
    public static void main(String args[]){
        Example2 obj=new Example2();
        obj.display1();
    }
}

```

Interface can have only public abstract methods

```

interface Example1{
    void display1();
}
class Example2 implements Example1{
    public void display1(){
        System.out.println("display1 method");
    }
    public void display2(){
        System.out.println("display2 method");
    }
}
class Demo{
    public static void main(String args[]){
        Example2 obj=new Example2();
        obj.display1();
    }
}

```

Difference No.6: Abstract class can have static, final or static final variables with any access specifier

```

abstract class Example1{
    private int numOne=10;
    protected final int numTwo=20;
    public static final int numThree=500;
    public void display1(){
        System.out.println("Num1="+numOne);
    }
}
class Example2 extends Example1{
    public void display2(){
        System.out.println("Num2="+numTwo);
        System.out.println("Num2="+numThree);
    }
}

```



```

class Demo{
    public static void main(String args[]){
        Example2 obj=new Example2();
        obj.display1();
        obj.display2();
    }
}

```

Interface can have only public static final (constant) variable

```

interface Example1{
    int numOne=10;
}
class Example2 implements Example1{
    public void display1(){
        System.out.println("Num1="+numOne);
    }
}
class Demo{
    public static void main(String args[]){
        Example2 obj=new Example2();
        obj.display1();
    }
}

```

Chapter 7:Exception Handling

Exception handling in java with examples

Exception handling is one of the most important feature of java programming that allows us to handle the runtime errors caused by exceptions. In this guide, we will learn what is an exception, types of it, exception classes and how to handle exceptions in java with examples.

What is an exception?

An Exception is an unwanted event that interrupts the normal flow of the program. When an exception occurs program execution gets terminated. In such cases we get a system generated error message. The good thing about exceptions is that they can be handled in Java. By handling the exceptions we can provide a meaningful message to the user about the issue rather than a system generated message, which may not be understandable to a user.

Why an exception occurs?

There can be several reasons that can cause a program to throw exception. For example: Opening a non-existing file in your program, Network connection problem, bad input data provided by user etc.

Exception Handling

If an exception occurs, which has not been handled by programmer then program execution gets terminated and a system generated error message is shown to the user. For example look at the system generated exception below:

An exception generated by the system is given below

```
Exception in thread "main" java.lang.ArithmeticException: / by zero at
ExceptionDemo.main(ExceptionDemo.java:5)
ExceptionDemo : The class name
main : The method name
ExceptionDemo.java : The filename
java:5 : Line number
```

This message is not user friendly so a user will not be able to understand what went wrong. In order to let them know the reason in simple language, we handle exceptions. We handle such conditions and then prints a user friendly

warning message to user, which lets them correct the error as most of the time exception occurs due to bad data provided by user.

Advantage of exception handling

Exception handling ensures that the flow of the program doesn't break when an exception occurs. For example, if a program has bunch of statements and an exception occurs mid way after executing certain statements then the statements after the exception will not execute and the program will terminate abruptly.

By handling we make sure that all the statements execute and the flow of program doesn't break.

Difference between error and exception

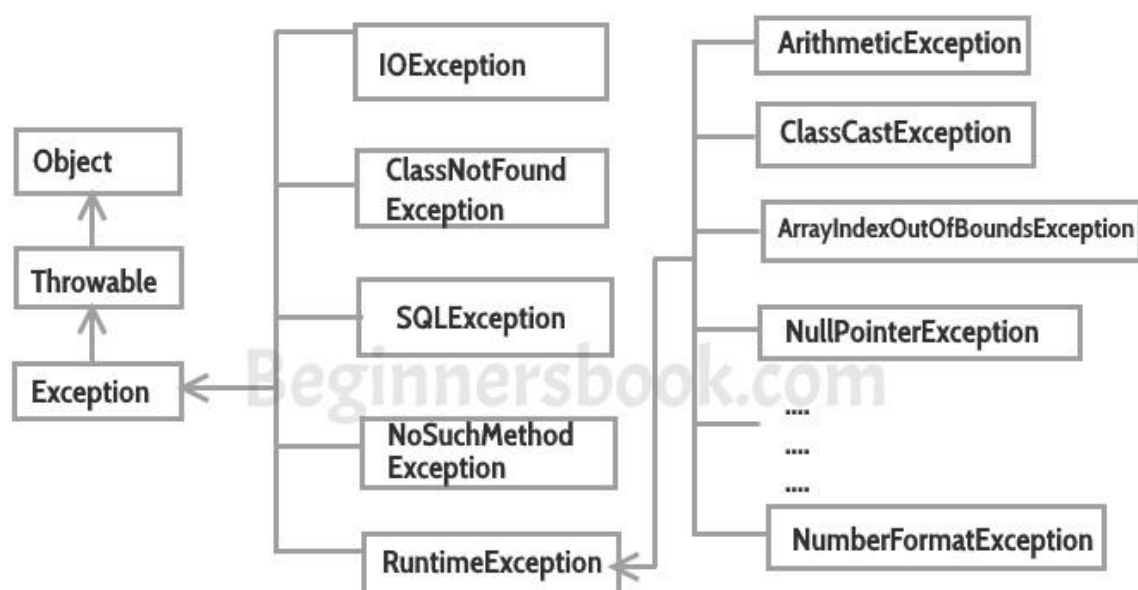
Errors indicate that something severe enough has gone wrong, the application should crash rather than try to handle the error.

Exceptions are events that occurs in the code. A programmer can handle such conditions and take necessary corrective actions. Few examples:

`NullPointerException` – When you try to use a reference that points to null.

`ArithmeticException` – When bad data is provided by user, for example, when you try to divide a number by zero this exception occurs because dividing a number by zero is undefined.

`ArrayIndexOutOfBoundsException` – When you try to access the elements of an array out of its bounds, for example array size is 5 (which means it has five elements) and you are trying to access the 10th element.



Types of exceptions

There are two types of exceptions in Java:

- 1)Checked exceptions
- 2)Unchecked exceptions

I have covered this in detail in a separate tutorial: [Checked and Unchecked exceptions in Java](#).

Checked exceptions

All exceptions other than Runtime Exceptions are known as Checked exceptions as the compiler checks them during compilation to see whether the programmer has handled them or not. If these exceptions are not handled/declared in the program, you will get compilation error. For example, SQLException, IOException, ClassNotFoundException etc.

Unchecked Exceptions

Runtime Exceptions are also known as Unchecked Exceptions. These exceptions are not checked at compile-time so compiler does not check whether the programmer has handled them or not but it's the responsibility of the programmer to handle these exceptions and provide a safe exit. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.

Compiler will never force you to catch such exception or force you to declare it in the method using throws keyword.

Try Catch in Java - Exception handling

In the previous topic we discussed what is exception handling and why we do it. In this we will see try-catch block which is used for exception handling.

Try block

The try block contains set of statements where an exception can occur. A try block is always followed by a catch block, which handles the exception that occurs in associated try block. A try block must be followed by catch blocks or finally block or both.

Syntax of try block

```
try{
    //statements that may cause an exception
}
```

While writing a program, if you think that certain statements in a program can throw an exception, enclosed them in try block and handle that exception

Catch block

A catch block is where you handle the exceptions, this block must follow the try block. A single try block can have several catch blocks associated with it. You can catch different exceptions in different catch blocks. When an exception occurs in try block, the corresponding catch block that handles that particular exception executes. For example if an arithmetic exception occurs in try block then the statements enclosed in catch block for arithmetic exception executes.

Syntax of try catch in java

```
try
{
    //statements that may cause an exception
}
catch (exception(type) e(object))
{
    //error handling code
}
```

Example: try catch block

If an exception occurs in try block then the control of execution is passed to the corresponding catch block. A single try block can have multiple catch blocks associated with it, you should place the catch blocks in such a way that the generic exception handler catch block is at the last(see in the example below).

The generic exception handler can handle all the exceptions but you should place it at the end, if you place it before all the catch blocks then it will display the generic message. You always want to give the user a meaningful message for each type of exception rather than a generic message.

```
class Example1 {
    public static void main(String args[]) {
        int num1, num2;
        try {
            /* We suspect that this block of statement can throw
             * exception so we handled it by placing these statements
             * inside try and handled the exception in catch block
             */
            num1 = 0;
            num2 = 62 / num1;
            System.out.println(num2);
            System.out.println("Hey I'm at the end of try block");
        }
```

```

    }
    catch (ArithmeticException e) {
        /* This block will only execute if any Arithmetic exception
        * occurs in try block
        */
        System.out.println("You should not divide a number by zero");
    }
    catch (Exception e) {
        /* This is a generic Exception handler which means it can handle
        * all the exceptions. This will execute if the exception is not
        * handled by previous catch blocks.
        */
        System.out.println("Exception occurred");
    }
    System.out.println("I'm out of try-catch block in Java.");
}
}

```

Output:

```

You should not divide a number by zero
I'm out of try-catch block in Java.

```

Multiple catch blocks in Java

The example we seen above is having multiple catch blocks, lets see few rules about multiple catch blocks with the help of examples. To read this in detail, see [catching multiple exceptions in java](#).

1. As I mentioned above, a single try block can have any number of catch blocks.
2. A generic catch block can handle all the exceptions. Whether it is `ArrayIndexOutOfBoundsException` or `ArithmeticException` or `NullPointerException` or any other type of exception, this handles all of them. To see the examples of `NullPointerException` and `ArrayIndexOutOfBoundsException`, refer this article: [Exception Handling example programs](#).

```

catch(Exception e){
    //This catch block catches all the exceptions
}

```

If you are wondering why we need other catch handlers when we have a generic that can handle all. This is because in generic exception handler you can display a message but you are not sure for which type of exception it may trigger so it will display the same message for all the exceptions and user may not be able to understand which exception occurred. Thats the reason you should place is at the end of all the specific exception catch blocks

3. If no exception occurs in try block then the catch blocks are completely ignored.
4. Corresponding catch blocks execute for that specific type of exception: `catch(ArithmeticException e)` is a catch block that can handle `ArithmeticException`

catch(NullPointerException e) is a catch block that can handle NullPointerException

5. You can also throw exception, which is an advanced topic and I have covered it in separate tutorials: [user defined exception](#), [throws keyword](#), [throw vs throws](#).

Example of Multiple catch blocks

```
class Example2{
    public static void main(String args[]){
        try{
            int a[]=new int[7];
            a[4]=30/0;
            System.out.println("First print statement in try block");
        }
        catch(ArithmeticException e){
            System.out.println("Warning: ArithmeticException");
        }
        catch(ArrayIndexOutOfBoundsException e){
            System.out.println("Warning: ArrayIndexOutOfBoundsException");
        }
        catch(Exception e){
            System.out.println("Warning: Some Other exception");
        }
        System.out.println("Out of try-catch block...");
    }
}
```

Output:

```
Warning: ArithmeticException
Out of try-catch block...
```

In the above example there are multiple catch blocks and these catch blocks executes sequentially when an exception occurs in try block. Which means if you put the last catch block (catch(Exception e)) at the first place, just after try block then in case of any exception this block will execute as it can handle all exceptions. This catch block should be placed at the last to avoid such situations.

Finally block

I have covered this in a separate topic here: [java finally block](#). For now you just need to know that this block executes whether an exception occurs or not. You should place those statements in finally blocks, that must execute whether exception occurs or not.

Java Finally block - Exception handling

In the previous tutorials I have covered [try-catch block](#) and [nested try block](#). In this guide, we will see finally block which is used along with try-catch.

A **finally block** contains all the crucial statements that must be executed whether exception occurs or not. The statements present in this block will always execute regardless of whether exception occurs in try block or not such as closing a connection, stream etc.

Syntax of Finally block

```
try {  
    //Statements that may cause an exception  
}  
catch {  
    //Handling exception  
}  
finally {  
    //Statements to be executed  
}
```

A Simple Example of finally block

Here you can see that the exception occurred in try block which has been handled in catch block, after that finally block got executed.

```
class Example  
{  
    public static void main(String args[]) {  
        try{  
            int num=121/0;  
            System.out.println(num);  
        }  
        catch(ArithmeticException e){  
            System.out.println("Number should not be divided by zero");  
        }  
        /* Finally block will always execute  
        * even if there is no exception in try block  
        */  
        finally{  
            System.out.println("This is finally block");  
        }  
        System.out.println("Out of try-catch-finally");  
    }  
}
```

Output:

```
Number should not be divided by zero  
This is finally block  
Out of try-catch-finally
```

Few Important points regarding finally block

1. A finally block must be associated with a try block, you cannot use finally without a try block. You should place those statements in this block that must be executed always.

2. Finally block is optional, as we have seen in previous tutorials that a try-catch block is sufficient for [exception handling](#), however if you place a finally block then it will always run after the execution of try block.
3. In normal case when there is no exception in try block then the finally block is executed after try block. However if an exception occurs then the catch block is executed before finally block.
4. An exception in the finally block, behaves exactly like any other exception.
5. The statements present in the **finally block** execute even if the try block contains control transfer statements like return, break or continue.
Let's see an example to see how finally works when return statement is present in try block:

Another example of finally block and return statement

You can see that even though we have return statement in the method, the finally block still runs.

```
class JavaFinally
{
    public static void main(String args[])
    {
        System.out.println(JavaFinally.myMethod());
    }
    public static int myMethod()
    {
        try {
            return 112;
        }
        finally {
            System.out.println("This is Finally block");
            System.out.println("Finally block ran even after return statement");
        }
    }
}
```

Output of above program:

```
This is Finally block
Finally block ran even after return statement
112
```

To see more examples of finally and return refer: [Java finally block and return statement](#)

.

Cases when the finally block doesn't execute

The circumstances that prevent execution of the code in a finally block are:

- The death of a Thread
- Using of the System. exit() method.
- Due to an exception arising in the finally block.

Finally and Close()

close() statement is used to close all the open streams in a program. Its a good practice to use close() inside finally block. Since finally block executes even if exception occurs so you can be sure that all input and output streams are closed properly regardless of whether the exception occurs or not.

For example:

```
....
try{
    OutputStream osf = new FileOutputStream( "filename" );
    OutputStream osb = new BufferedOutputStream(opf);
    ObjectOutput op = new ObjectOutputStream(osb);
    try{
        output.writeObject(writableObject);
    }
    finally{
        op.close();
    }
}
catch(IOException e1){
    System.out.println(e1);
}
...
```

Finally block without catch

A try-finally block is possible without catch block. Which means a try block can be used with finally without having a catch block.

```
...
InputStream input = null;
try {
    input = new FileInputStream("inputfile.txt");
}
finally {
    if (input != null) {
        try {
            in.close();
        }catch (IOException exp) {
            System.out.println(exp);
        }
    }
}
...
```

Finally block and System.exit()

System.exit() statement behaves differently than **return statement**. Unlike return statement whenever System.exit() gets called in try block then **Finally block** doesn't execute. Here is a code snippet that demonstrate the same:

```
....
try {
    //try block
    System.out.println("Inside try block");
    System.exit(0)
}
catch (Exception exp) {
    System.out.println(exp);
}
finally {
    System.out.println("Java finally block");
}
....
```

In the above example if the **System.exit(0)** gets called without any exception then finally won't execute. However if any exception occurs while calling **System.exit(0)** then finally block will be executed.

try-catch-finally block

- Either a try statement should be associated with a catch block or with finally.
- Since catch performs exception handling and finally performs the cleanup, the best approach is to use both of them.

Syntax:

```
try {
    //statements that may cause an exception
}
catch (...) {
    //error handling code
}
finally {
    //statements to be executed
}
```

Examples of Try catch finally blocks

Example 1: The following example demonstrate the working of finally block when no exception occurs in try block

```
class Example1{
    public static void main(String args[]){
        try{
            System.out.println("First statement of try block");
            int num=45/3;
            System.out.println(num);
        }
    }
}
```

```

    catch(ArrayIndexOutOfBoundsException e){
        System.out.println("ArrayIndexOutOfBoundsException");
    }
    finally{
        System.out.println("finally block");
    }
    System.out.println("Out of try-catch-finally block");
}
}

```

Output:

```

First statement of try block
15
finally block
Out of try-catch-finally block

```

Example 2: This example shows the working of finally block when an exception occurs in try block but is not handled in the catch block:

```

class Example2{
    public static void main(String args[]){
        try{
            System.out.println("First statement of try block");
            int num=45/0;
            System.out.println(num);
        }
        catch(ArrayIndexOutOfBoundsException e){
            System.out.println("ArrayIndexOutOfBoundsException");
        }
        finally{
            System.out.println("finally block");
        }
        System.out.println("Out of try-catch-finally block");
    }
}

```

Output:

```

First statement of try block
finally block
Exception in thread "main" java.lang.ArithmeticException: / by zero
at beginnersbook.com.Example2.main(Details.java:6)

```

As you can see that the system generated exception message is shown but before that the finally block successfully executed.

Example 3: When exception occurs in try block and handled properly in catch block

```

class Example3{
    public static void main(String args[]){
        try{
            System.out.println("First statement of try block");
            int num=45/0;
            System.out.println(num);
        }
        catch(ArithmeticException e){

```

```

        System.out.println("ArithmeticException");
    }
    finally{
        System.out.println("finally block");
    }
    System.out.println("Out of try-catch-finally block");
}
}

```

Output:

```

First statement of try block
ArithmeticException
finally block
Out of try-catch-finally block

```

How to throw exception in java with example

In Java we have already defined exception classes such as `ArithmeticException`, `NullPointerException`, `ArrayIndexOutOfBoundsException` exception etc. These exceptions are set to trigger on different-2 conditions. For example when we divide a number by zero, this triggers `ArithmeticException`, when we try to access the array element out of its bounds then we get `ArrayIndexOutOfBoundsException`.

We can define our own set of conditions or rules and throw an exception explicitly using `throw` keyword. For example, we can throw `ArithmeticException` when we divide number by 5, or any other numbers, what we need to do is just set the condition and throw any exception using `throw` keyword. `Throw` keyword can also be used for throwing custom exceptions, I have covered that in a separate tutorial, see [Custom Exceptions in Java](#).

Syntax of throw keyword:

```
throw new exception_class("error message");
```

For example:

```
throw new ArithmeticException("dividing a number by 5 is not allowed in this program");
```

Example of throw keyword

Lets say we have a requirement where we we need to only register the students when their age is less than 12 and weight is less than 40, if any of the condition is not met then the user should get an `ArithmeticException` with the warning message “Student is not eligible for registration”. We have

implemented the logic by placing the code in the method that checks student eligibility if the entered student age and weight doesn't met the criteria then we throw the exception using throw keyword.

```
/* In this program we are checking the Student age
 * if the student age<12 and weight <40 then our program
 * should return that the student is not eligible for registration.
 */
public class ThrowExample {
    static void checkEligibility(int stuage, int stuweight){
        if(stuage<12 && stuweight<40) {
            throw new ArithmeticException("Student is not eligible for
registration");
        }
        else {
            System.out.println("Student Entry is Valid!!");
        }
    }

    public static void main(String args[]){
        System.out.println("Welcome to the Registration process!!");
        checkEligibility(10, 39);
        System.out.println("Have a nice day..");
    }
}
```

Output:

```
Welcome to the Registration process!!Exception in thread "main"
java.lang.ArithmeticException: Student is not eligible for registration
at beginnersbook.com.ThrowExample.checkEligibility(ThrowExample.java:9)
at beginnersbook.com.ThrowExample.main(ThrowExample.java:18)
```

In the above example we have throw an unchecked exception, same way we can throw unchecked and user-defined exception as well.

Throws clause in java - Exception handling

As we know that there are two types of exception checked and unchecked.

Checked exception (compile time) force you to handle them, if you don't handle them then the program will not compile.

On the other hand unchecked exception (Runtime) doesn't get checked during compilation. **Throws keyword** is used for handling checked exceptions . By using throws we can declare multiple exceptions in one go.

What is the need of having throws keyword when you can handle exception using try-catch?

Well, thats a valid question. We already know we can handle exceptions using try-catch block.

The throws does the same thing that try-catch does but there are some cases

where you would prefer throws over try-catch. For example:
Let's say we have a method `myMethod()` that has statements that can throw either `ArithmeticException` or `NullPointerException`, in this case you can use try-catch as shown below:

```
public void myMethod()
{
    try {
        // Statements that might throw an exception
    }
    catch (ArithmeticException e) {
        // Exception handling statements
    }
    catch (NullPointerException e) {
        // Exception handling statements
    }
}
```

But suppose you have several such methods that can cause exceptions, in that case it would be tedious to write these try-catch for each method. The code will become unnecessary long and will be less-readable.

One way to overcome this problem is by using throws like this: declare the exceptions in the method signature using throws and handle the exceptions where you are calling this method by using try-catch.

Another advantage of using this approach is that you will be forced to handle the exception when you call this method, all the exceptions that are declared using throws, must be handled where you are calling this method else you will get compilation error.

```
public void myMethod() throws ArithmeticException, NullPointerException
{
    // Statements that might throw an exception
}

public static void main(String args[]) {
    try {
        myMethod();
    }
    catch (ArithmeticException e) {
        // Exception handling statements
    }
    catch (NullPointerException e) {
        // Exception handling statements
    }
}
```

Example of throws Keyword

In this example the method `myMethod()` is throwing two **checked exceptions** so we have declared these exceptions in the method signature using **throws** Keyword. If we do not declare these exceptions then the program will throw a compilation error.

```

import java.io.*;
class ThrowExample {
    void myMethod(int num)throws IOException, ClassNotFoundException{
        if(num==1)
            throw new IOException("IOException Occurred");
        else
            throw new ClassNotFoundException("ClassNotFoundException");
    }
}

public class Example1{
    public static void main(String args[]){
        try{
            ThrowExample obj=new ThrowExample();
            obj.myMethod(1);
        }catch(Exception ex){
            System.out.println(ex);
        }
    }
}

```

Output:

```
java.io.IOException: IOException Occurred
```

User defined exception in java

BY CHAITANYA SINGH | FILED UNDER: [EXCEPTION HANDLING](#)

In java we have already defined, exception classes such as `ArithmeticException`, `NullPointerException` etc. These exceptions are already set to trigger on pre-defined conditions such as when you divide a number by zero it triggers `ArithmeticException`, In the last tutorial we learnt how to throw these exceptions explicitly based on your conditions using [throw keyword](#).

In java we can create our own exception class and throw that exception using `throw` keyword. These exceptions are known as **user-defined** or **custom** exceptions. In this tutorial we will see how to create your own custom exception and throw it on a particular condition.

To understand this tutorial you should have the basic knowledge of [try-catch block](#) and [throw in java](#).

Example of User defined exception in Java

```

/* This is my Exception class, I have named it MyException
 * you can give any name, just remember that it should
 * extend Exception class
 */
class MyException extends Exception{
    String str1;
    /* Constructor of custom exception class

```



```

    * here I am copying the message that we are passing while
    * throwing the exception to a string and then displaying
    * that string along with the message.
    */
    MyException(String str2) {
        str1=str2;
    }
    public String toString(){
        return ("MyException Occurred: "+str1) ;
    }
}

class Example1{
    public static void main(String args[]){
        try{
            System.out.println("Starting of try block");
            // I'm throwing the custom exception using throw
            throw new MyException("This is My error Message");
        }
        catch(MyException exp){
            System.out.println("Catch Block") ;
            System.out.println(exp) ;
        }
    }
}

```

Output:

```

Starting of try block
Catch Block
MyException Occurred: This is My error Message

```

Explanation:

You can see that while throwing custom exception I gave a string in parenthesis (throw new MyException("This is My error Message");). That's why we have a [parameterized constructor](#) (with a String parameter) in my custom exception class.

Notes:

1. User-defined exception must extend Exception class.
2. The exception is thrown using throw keyword.

Another Example of Custom Exception

In this example we are throwing an exception from a method. In this case we should use throws clause in the method signature otherwise you will get compilation error saying that “unhandled exception in method”. To understand how throws clause works, refer this guide: [throws keyword in java](#).

```

class InvalidProductException extends Exception
{
    public InvalidProductException(String s)
    {
        // Call constructor of parent Exception
    }
}

```

```

        super(s);
    }
}

public class Example1
{
    void productCheck(int weight) throws InvalidProductException{
        if(weight<100){
            throw new InvalidProductException("Product Invalid");
        }
    }

    public static void main(String args[])
    {
        Example1 obj = new Example1();
        try
        {
            obj.productCheck(60);
        }
        catch (InvalidProductException ex)
        {
            System.out.println("Caught the exception");
            System.out.println(ex.getMessage());
        }
    }
}

```

Output:

```

Caught the exception
Product Invalid

```

Difference between throw and throws in java

BY CHAITANYA SINGH | FILED UNDER: [EXCEPTION HANDLING](#)

In this guide, we will discuss the difference between throw and throws keywords. Before going through the difference, refer my previous tutorials about [throw](#) and [throws](#).

Throw vs Throws in java

1. **Throws clause** is used to declare an exception, which means it works similar to the try-catch block. On the other hand **throw** keyword is used to throw an exception explicitly.

2. If we see syntax wise than **throw** is followed by an instance of Exception class and **throws** is followed by exception class names.

For example:

```
throw new ArithmeticException("Arithmetic Exception");  
and
```

```
throws ArithmeticException;
```

3. Throw keyword is used in the method body to throw an exception, while throws is used in method signature to declare the exceptions that can occur in the statements present in the method.

For example:

Throw:

```
...  
void myMethod() {  
    try {  
        //throwing arithmetic exception using throw  
        throw new ArithmeticException("Something went wrong!!");  
    }  
    catch (Exception exp) {  
        System.out.println("Error: "+exp.getMessage());  
    }  
}  
...
```

Throws:

```
...  
//Declaring arithmetic exception using throws  
void sample() throws ArithmeticException{  
    //Statements  
}  
...
```

4. You can throw one exception at a time but you can handle multiple exceptions by declaring them using throws keyword.

For example:

Throw:

```
void myMethod() {  
    //Throwing single exception using throw  
    throw new ArithmeticException("An integer should not be divided by zero!!");  
}  
..
```

Throws:

```
//Declaring multiple exceptions using throws  
void myMethod() throws ArithmeticException, NullPointerException{  
    //Statements where exception might occur  
}
```

These were the main **differences between throw and throws in Java**. Lets see complete examples of throw and throws keywords.

Throw Example

To understand this example you should know what is throw keyword and how it works, refer this guide: [throw keyword in java](#).

```
public class Example1{
    void checkAge(int age){
        if(age<18)
            throw new ArithmeticException("Not Eligible for voting");
        else
            System.out.println("Eligible for voting");
    }
    public static void main(String args[]){
        Example1 obj = new Example1();
        obj.checkAge(13);
        System.out.println("End Of Program");
    }
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException:
Not Eligible for voting
at Example1.checkAge(Example1.java:4)
at Example1.main(Example1.java:10)
```

Throws Example

To understand this example you should know what is throws clause and how it is used in method declaration for exception handling, refer this guide: [throws in java](#).

```
public class Example1{
    int division(int a, int b) throws ArithmeticException{
        int t = a/b;
        return t;
    }
    public static void main(String args[]){
        Example1 obj = new Example1();
        try{
            System.out.println(obj.division(15,0));
        }
        catch(ArithmeticException e){
            System.out.println("You shouldn't divide number by zero");
        }
    }
}
```

Output:

```
You shouldn't divide number by zero
```

Checked and unchecked exceptions in java with examples

There are two types of exceptions: checked exception and unchecked exception. In this guide, we will discuss them. The main **difference between checked and unchecked exception** is that the checked exceptions are checked at compile-time while unchecked exceptions are checked at runtime.

What are checked exceptions?

Checked exceptions are checked at compile-time. It means if a method is throwing a checked exception then it should handle the exception using **try-catch block** or it should declare the exception using **throws keyword**, otherwise the program will give a compilation error.

Lets understand this with the help of an **example**:

Checked Exception Example

In this example we are reading the file `myfile.txt` and displaying its content on the screen. In this program there are three places where a checked exception is thrown as mentioned in the comments below. `FileInputStream` which is used for specifying the file path and name, throws `FileNotFoundException`. The `read()` method which reads the file content throws `IOException` and the `close()` method which closes the file input stream also throws `IOException`.

```
import java.io.*;
class Example {
    public static void main(String args[])
    {
        FileInputStream fis = null;
        /*This constructor FileInputStream(File filename)
        * throws FileNotFoundException which is a checked
        * exception
        */
        fis = new FileInputStream("B:/myfile.txt");
        int k;

        /* Method read() of FileInputStream class also throws
        * a checked exception: IOException
        */
        while(( k = fis.read() ) != -1)
        {
            System.out.print((char)k);
        }

        /*The method close() closes the file input stream
        * It throws IOException*/
    }
}
```

```
        fis.close();
    }
}
```

Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problems:
Unhandled exception type FileNotFoundException
Unhandled exception type IOException
Unhandled exception type IOException
```

Why this compilation error? As I mentioned in the beginning that checked exceptions gets checked during compile time. Since we didn't handled/declared the exceptions, our program gave the compilation error.

How to resolve the error? There are two ways to avoid this error. We will see both the ways one by one.

Method 1: Declare the exception using throws keyword.

As we know that all three occurrences of checked exceptions are inside main() method so one way to avoid the compilation error is: Declare the exception in the method using throws keyword. You may be thinking that our code is throwing FileNotFoundException and IOException both then why we are declaring the IOException alone. The reason is that IOException is a parent class of FileNotFoundException so it by default covers that. If you want you can declare them like this `public static void main(String args[]) throws IOException, FileNotFoundException.`

```
import java.io.*;
class Example {
    public static void main(String args[]) throws IOException
    {
        FileInputStream fis = null;
        fis = new FileInputStream("B:/myfile.txt");
        int k;

        while(( k = fis.read() ) != -1)
        {
            System.out.print((char)k);
        }
        fis.close();
    }
}
```

Output:

File content is displayed on the screen.

Method 2: Handle them using try-catch blocks.

The approach we have used above is not good at all. It is not the best **exception handling** practice. You should give meaningful message for each exception type so that it would be easy for someone to understand the error. The code should be like this:

```
import java.io.*;
```

```

class Example {
    public static void main(String args[])
    {
        FileInputStream fis = null;
        try{
            fis = new FileInputStream("B:/myfile.txt");
        }catch(FileNotFoundException fnfe){
            System.out.println("The specified file is not " +
                               "present at the given path");
        }
        int k;
        try{
            while(( k = fis.read() ) != -1)
            {
                System.out.print((char)k);
            }
            fis.close();
        }catch(IOException ioe){
            System.out.println("I/O error occurred: "+ioe);
        }
    }
}

```

This code will run fine and will display the file content.

Here are the few other Checked Exceptions –

- SQLException
- IOException
- ClassNotFoundException
- InvocationTargetException

What are Unchecked exceptions?

Unchecked exceptions are not checked at compile time. It means if your program is throwing an unchecked exception and even if you didn't handle/declare that exception, the program won't give a compilation error. Most of the times these exception occurs due to the bad data provided by user during the user-program interaction. It is up to the programmer to judge the conditions in advance, that can cause such exceptions and handle them appropriately. All Unchecked exceptions are direct sub classes of **RuntimeException** class.

Lets understand this with an example:

Unchecked Exception Example

```

class Example {
    public static void main(String args[])
    {
        int num1=10;
        int num2=0;
    }
}

```

```

        /*Since I'm dividing an integer with 0
        * it should throw ArithmeticException
        */
        int res=num1/num2;
        System.out.println(res);
    }
}

```

If you compile this code, it would compile successfully however when you will run it, it would throw `ArithmeticException`. That clearly shows that unchecked exceptions are not checked at compile-time, they occurs at runtime. Lets see another example.

```

class Example {
    public static void main(String args[])
    {
        int arr[] ={1,2,3,4,5};
        /* My array has only 5 elements but we are trying to
        * display the value of 8th element. It should throw
        * ArrayIndexOutOfBoundsException
        */
        System.out.println(arr[7]);
    }
}

```

This code would also compile successfully since `ArrayIndexOutOfBoundsException` is also an unchecked exception.

Note: It **doesn't mean** that compiler is not checking these exceptions so we shouldn't handle them. In fact we should handle them more carefully. For e.g. In the above example there should be a exception message to user that they are trying to display a value which doesn't exist in array so that user would be able to correct the issue.

```

class Example {
    public static void main(String args[]) {
        try{
            int arr[] ={1,2,3,4,5};
            System.out.println(arr[7]);
        }
        catch(ArrayIndexOutOfBoundsException e){
            System.out.println("The specified index does not exist " +
                               "in array. Please correct the error.");
        }
    }
}

```

Output:

```
The specified index does not exist in array. Please correct the error.
```

Importance of Exception Handling

Exception handling is used when the frequency of occurrence of an exception cannot be predicted.

Real world examples:

1. you provide a web form for users to fill in and submit. but in case there are a lot of conditions to be handled and the conditions keeps changing periodically, you use exception handling to simplify the process
2. database connectivity uses exception handling(why???) this is because the reason for database connectivity failure cannot be predicted and handled as it can be caused by many variables such as power failure, unreachable server, failure at client front/back end and so on.
3. internet communication
4. arithmetic exceptions such as division by zero and so on.
5. operating systems use exception handling to resolve deadlocks, recover from crash and so forth

Chapter – 4

Architecture of Multiprocessor Systems

○ **Parallel Processing**

Parallel processing can be described as a class of techniques which enables the system to achieve simultaneous data-processing tasks to increase the computational speed of a computer system.

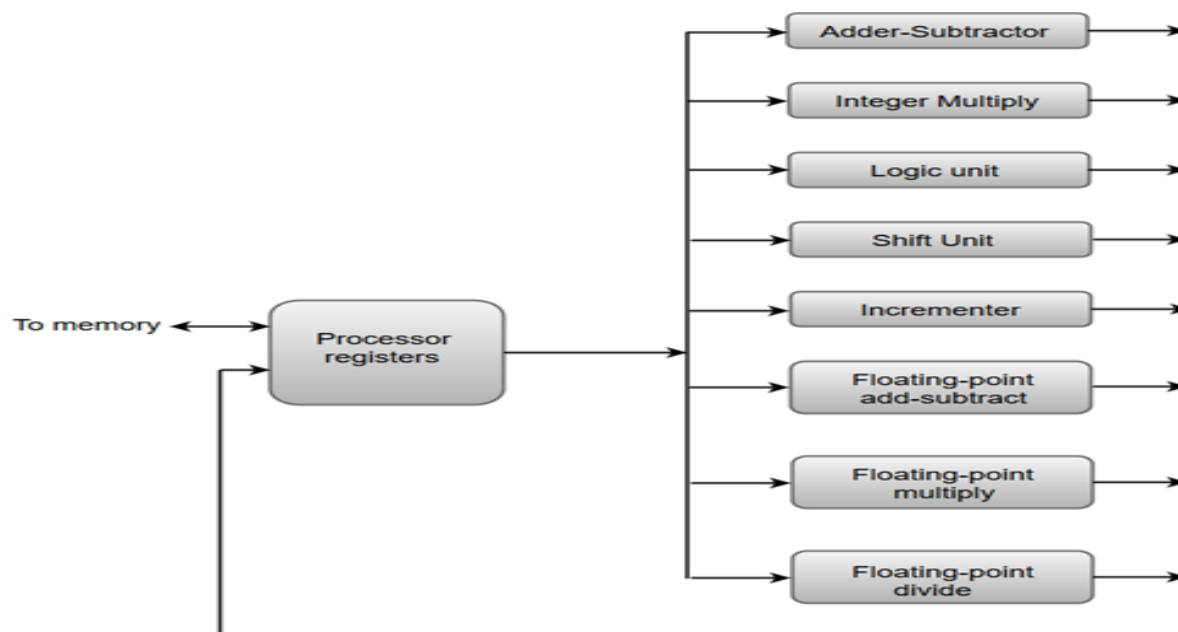
A parallel processing system can carry out simultaneous data-processing to achieve faster execution time. For instance, while an instruction is being processed in the ALU component of the CPU, the next instruction can be read from memory.

The primary purpose of parallel processing is to enhance the computer processing capability and increase its throughput, i.e. the amount of processing that can be accomplished during a given interval of time.

A parallel processing system can be achieved by having a multiplicity of functional units that perform identical or different operations simultaneously. The data can be distributed among various multiple functional units.

The following diagram shows one possible way of separating the execution unit into eight functional units operating in parallel.

The operation performed in each functional unit is indicated in each block of the diagram:



- The adder and integer multiplier performs the arithmetic operation with integer numbers.
- The floating-point operations are separated into three circuits operating in parallel.

- The logic, shift, and increment operations can be performed concurrently on different data. All units are independent of each other, so one number can be shifted while another number is being incremented.

Parallel Processing Systems are designed to speed up the execution of programs by dividing the program into multiple fragments and processing these fragments simultaneously. Such systems are multiprocessor systems also known as tightly coupled systems. Parallel systems deal with the simultaneous use of multiple computer resources that can include a single computer with multiple processors, a number of computers connected by a network to form a parallel processing cluster or a combination of both. Parallel systems are more difficult to program than computers with a single processor because the architecture of parallel computers varies accordingly and the processes of multiple CPUs must be coordinated and synchronized. Several models for connecting processors and memory modules exist, and each topology requires a different programming model. The three models that are most commonly used in building parallel computers include synchronous processors each with its own memory, asynchronous processors each with its own memory and asynchronous processors with a common, shared memory. Flynn has classified the computer systems based on parallelism in the instructions and in the data streams. These are:

1. Single instruction stream, single data stream (SISD).
2. Single instruction stream, multiple data stream (SIMD).
3. Multiple instruction streams, single data stream (MISD).
4. Multiple instruction stream, multiple data stream (MIMD).

The above classification of parallel computing system is focused in terms of two independent factors: the number of data streams that can be simultaneously processed, and the number of instruction streams that can be simultaneously processed. Here 'instruction stream' we mean an algorithm that instructs the computer what to do whereas 'data stream' (i.e. input to an algorithm) we mean the data that are being operated upon.

Even though Flynn has classified the computer 'systems into four types based on parallelism but only two of them are relevant to parallel computers. These are SIMD and MIMD computers.

Single Instruction, Single Data (SISD):

A serial (non-parallel) computer

Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle

Single data: only one data stream is being used as input during any one clock cycle

Deterministic execution

This is the oldest and even today, the most common type of computer

Examples: older generation mainframes, minicomputers and workstations; most modern day PCs.

Single Instruction, Multiple Data (SIMD):

A type of parallel computer

Single instruction: All processing units execute the same instruction at any given clock cycle

Multiple data: Each processing unit can operate on a different data element

Best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing.

Synchronous (lockstep) and deterministic execution

Two varieties: Processor Arrays and Vector Pipelines

Examples:

Processor Arrays: Connection Machine CM-2, MasPar MP-1 & MP-2, ILLIAC IV

Vector Pipelines: IBM 9000, Cray X-MP, Y-MP & C90, Fujitsu VP, NEC SX-2, Hitachi S820, ETA10

Most modern computers, particularly those with graphics processor units (GPUs) employ SIMD instructions and execution units.

Multiple Instruction, Single Data (MISD):

A single data stream is fed into multiple processing units.

Each processing unit operates on the data independently via independent instruction streams.

Few actual examples of this class of parallel computer have ever existed. One is the experimental Carnegie-Mellon C.mmp computer (1971).

Some conceivable uses might be: Multiple frequency filters operating on a single signal stream. Multiple cryptography algorithms attempting to crack a single coded message.

Multiple Instruction, Multiple Data (MIMD):

Currently, the most common type of parallel computer. Most modern computers fall into this category.

Multiple Instruction: every processor may be executing a different instruction stream

Multiple Data: every processor may be working with a different data stream

Execution can be synchronous or asynchronous, deterministic or non-deterministic

Examples: most current supercomputers, networked parallel computer clusters and "grids", multi-processor SMP computers, multi-core PCs.

Note: many MIMD architectures also include

SIMD execution sub-component

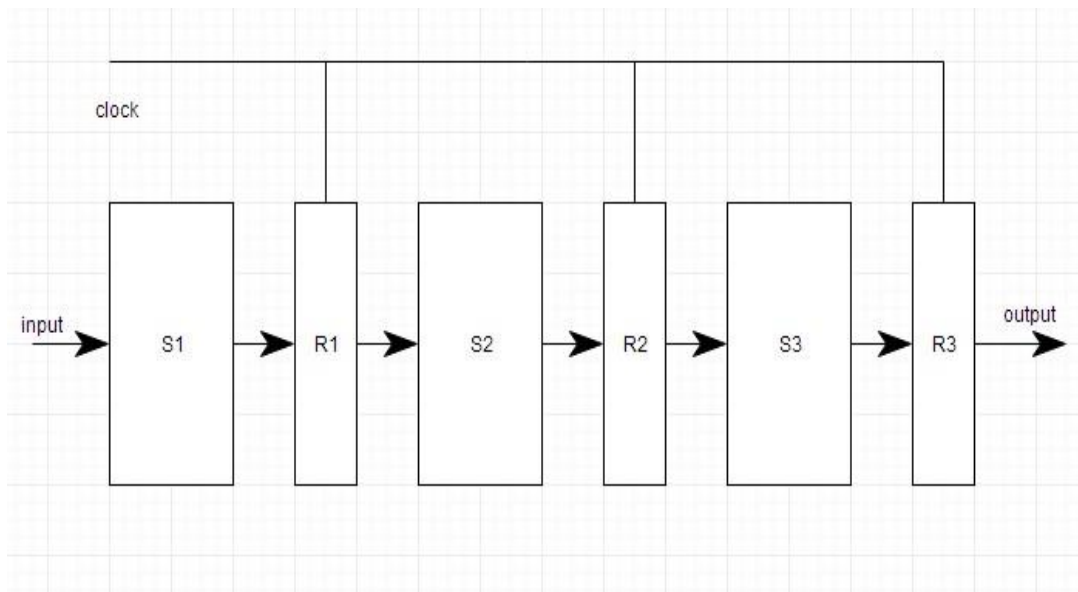
What is Pipelining?

Pipelining is the process of accumulating instruction from the processor through a pipeline. It allows storing and executing instructions in an orderly process. It is also known as **pipeline processing**.

Pipelining is a technique where multiple instructions are overlapped during execution. Pipeline is divided into stages and these stages are connected with one another to form a pipe like structure. Instructions enter from one end and exit from another end.

Pipelining increases the overall instruction throughput.

In pipeline system, each segment consists of an input register followed by a combinational circuit. The register is used to hold data and combinational circuit performs operations on it. The output of combinational circuit is applied to the input register of the next segment.



Pipeline system is like the modern day assembly line setup in factories. For example in a car manufacturing industry, huge assembly lines are setup and at each point, there are robotic arms to perform a certain task, and then the car moves on ahead to the next arm.

Types of Pipeline

It is divided into 2 categories:

1. Arithmetic Pipeline
2. Instruction Pipeline

Arithmetic Pipeline

Arithmetic pipelines are usually found in most of the computers. They are used for floating point operations, multiplication of fixed point numbers etc. For example: The input to the Floating Point Adder pipeline is:

$$X = A * 2^a$$

$$Y = B * 2^b$$

Here A and B are mantissas (significant digit of floating point numbers), while **a** and **b** are exponents.

The floating point addition and subtraction is done in 4 parts:

1. Compare the exponents.
2. Align the mantissas.
3. Add or subtract mantissas
4. Produce the result.

Registers are used for storing the intermediate results between the above operation

Instruction Pipeline

In this a stream of instructions can be executed by overlapping *fetch*, *decode* and *execute* phases of an instruction cycle. This type of technique is used to increase the throughput of the computer system.

An instruction pipeline reads instruction from the memory while previous instructions are being executed in other segments of the pipeline. Thus we can execute multiple instructions simultaneously. The pipeline will be more efficient if the instruction cycle is divided into segments of equal duration.

Advantages of Pipelining

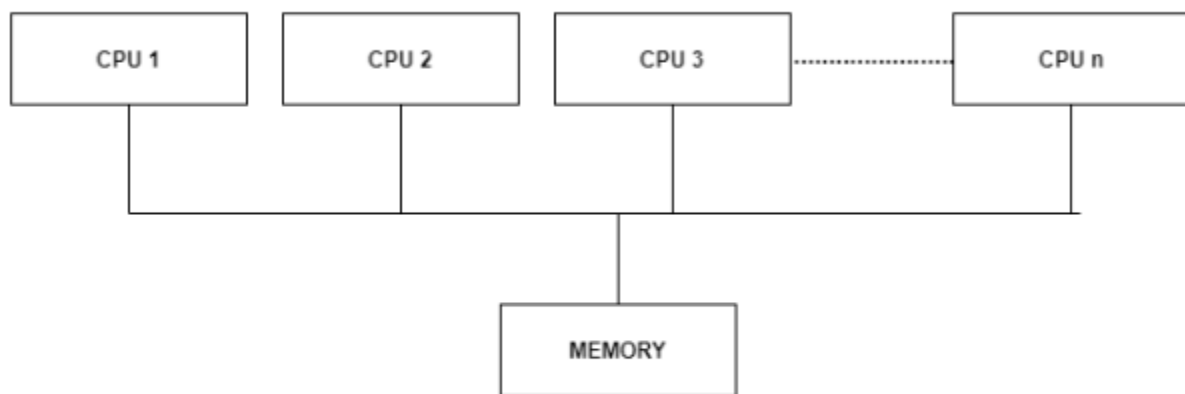
1. The cycle time of the processor is reduced.
2. It increases the throughput of the system
3. It makes the system reliable.

Disadvantages of Pipelining

1. The design of pipelined processor is complex and costly to manufacture.
2. The instruction latency is more.

Introduction: Multiprocessor systems

Most computer systems are single processor systems i.e they only have one processor. However, multiprocessor or parallel systems are increasing in importance nowadays. These systems have multiple processors working in parallel that share the computer clock, memory, bus, peripheral devices etc. An image demonstrating the multiprocessor architecture is:



Multiprocessing Architecture

Classification Multiprocessors are classified by the way their memory is organized.

There are two main kinds of multiprocessing systems:-

- Tightly Coupled Systems
- Loosely Coupled Systems



Computer Organization and Architecture

Tightly Coupled Systems → A multiprocessor system with common shared memory is classified as a shared memory or tightly coupled multiprocessor. This does not prevent each processor from having its own local memory. → In fact, most commercial tightly coupled multiprocessors provide a cache memory with each CPU. In addition, there is a global common memory that all CPUs can access. Information can be therefore be shared among the CPUs by placing it in the common global memory. → Symmetric multiprocessing (SMP) involves a multiprocessor system architecture where two or more identical processors connect to a single, shared main memory, have full access to all I/O devices, and are controlled by a single operating system instance that treats all processors equally, reserving none for special purposes.

Loosely Coupled Systems → An alternative model of microprocessor is the distributed memory or loosely coupled system. Each processor element in a loosely coupled system has its own private local memory. → The processors are tied together by a switching scheme designed to route information from one processor to another through a message-passing scheme. → Loosely coupled systems are most efficient when the interaction between tasks is minimal unlike tightly coupled systems can tolerate a higher degree of interaction between tasks.

Types of Multiprocessors

There are mainly two types of multiprocessors i.e. symmetric and asymmetric multiprocessors. Details about them are as follows:

1. Symmetric Multiprocessors

In these types of systems, each processor contains a similar copy of the operating system and they all communicate with each other. All the processors are in a peer to peer relationship i.e. no master - slave relationship exists between them.

An example of the symmetric multiprocessing system is the Encore version of Unix for the Multimax Computer.

2. Asymmetric Multiprocessors

In asymmetric systems, each processor is given a predefined task. There is a master processor that gives instruction to all the other processors. Asymmetric multiprocessor system contains a master slave relationship.

Asymmetric multiprocessor was the only type of multiprocessor available before symmetric multiprocessors were created. Now also, this is the cheaper option.

Tightly Coupled Systems vs Loosely Coupled Systems :

1. Tightly-coupled systems perform better and are physically smaller than loosely coupled systems, but have historically required greater initial investments and may depreciate rapidly. On the other hand, nodes in a loosely coupled system are usually inexpensive commodity computers and can be recycled as independent machines upon retirement from the cluster.

2. Tightly coupled systems tend to be much more energy efficient than clusters. Considerable economy can be realized by designing components to work together from the



Computer Organization and Architecture

beginning in tightly coupled systems Loosely coupled systems use components that were not necessarily intended specifically for use in such systems.

3. Advantages of Multiprocessor Systems

There are multiple advantages to multiprocessor systems. Some of these are:

More reliable Systems

In a multiprocessor system, even if one processor fails, the system will not halt. This ability to continue working despite hardware failure is known as graceful degradation. For example: If there are 5 processors in a multiprocessor system and one of them fails, then also 4 processors are still working. So the system only becomes slower and does not ground to a halt.

Enhanced Throughput

If multiple processors are working in tandem, then the throughput of the system increases i.e. number of processes getting executed per unit of time increase. If there are N processors then the throughput increases by an amount just under N.

More Economic Systems

Multiprocessor systems are cheaper than single processor systems in the long run because they share the data storage, peripheral devices, power supplies etc. If there are multiple processes that share data, it is better to schedule them on multiprocessor systems with shared data than have different computer systems with multiple copies of the data.

4. Disadvantages of Multiprocessor Systems

There are some disadvantages as well to multiprocessor systems. Some of these are:

Increased Expense

Even though multiprocessor systems are cheaper in the long run than using multiple computer systems, still they are quite expensive. It is much cheaper to buy a simple single processor system than a multiprocessor system.

Complicated Operating System Required

There are multiple processors in a multiprocessor system that share peripherals, memory etc. So, it is much more complicated to schedule processes and impart resources to processes than in single processor systems. Hence, a more complex and complicated operating system is required in multiprocessor systems.

Large Main Memory Required

All the processors in the multiprocessor system share the memory. So a much larger pool of memory is required as compared to single processor systems.

Characteristics of multiprocessors

- ‡ A multiprocessor system is an interconnection of two or more CPUs with memory and input-output equipment.
- ‡ The term “processor” in multiprocessor can mean either a central processing unit (CPU) or an input-output processor (IOP).
- ‡ Multiprocessors are classified as *multiple instruction stream*, *multiple data stream*



Computer Organization and Architecture

(MIMD) systems

- ¶ The similarity and distinction between multiprocessor and multicomputer are
 - Similarity
 - Both support concurrent operations
 - Distinction
 - The network consists of several autonomous computers that may or may not communicate with each other.
A multiprocessor system is controlled by one operating system that provides interaction between processors and all the components of the system cooperate in the solution of a problem.
- ¶ Multiprocessing improves the reliability of the system.
- ¶ The benefit derived from a multiprocessor organization is an improved system performance.
 - Multiple independent jobs can be made to operate in parallel.
 - A single job can be partitioned into multiple parallel tasks.
- ¶ Multiprocessing can improve performance by decomposing a program into parallel executable tasks.
 - The user can explicitly declare that certain tasks of the program be executed in parallel.
This must be done prior to loading the program by specifying the parallel executable segments.
 - The other is to provide a compiler with multiprocessor software that can automatically detect parallelism in a user's program.
- ¶ Multiprocessor are classified by the way their memory is organized.
 - A multiprocessor system with *common shared memory* is classified as a *shared-memory* or *tightly coupled multiprocessor*.
Tolerate a *higher degree* of interaction between tasks.
 - Each processor element with its own *private local memory* is classified as a *distributed-memory* or *loosely coupled system*.
Are most efficient when the interaction between tasks is *minimal*



Interconnection Structures

- ‡ The components that form a multiprocessor system are CPUs, IOPs connected to input-output devices, and a memory unit.
- ‡ The interconnection between the components can have different physical configurations, depending on the number of transfer paths that are available
 - Between the processors and memory in a shared memory system
 - Among the processing elements in a loosely coupled system
- ‡ There are several physical forms available for establishing an interconnection network.
 - Time-shared common bus
 - Multiport memory
 - Crossbar switch
 - Multistage switching network
 - Hypercube system

Time Shared Common Bus

- ‡ A common-bus multiprocessor system consists of a number of processors connected through a common path to a memory unit.
- ‡ *Disadv.:*
 - Only one processor can communicate with the memory or another processor at any given time.
 - As a consequence, the total overall transfer rate within the system is limited by the speed of the single path
- ‡ A more economical implementation of a dual bus structure is depicted in Fig. below.
- ‡ Part of the local memory may be designed as a *cache memory* attached to the CPU.

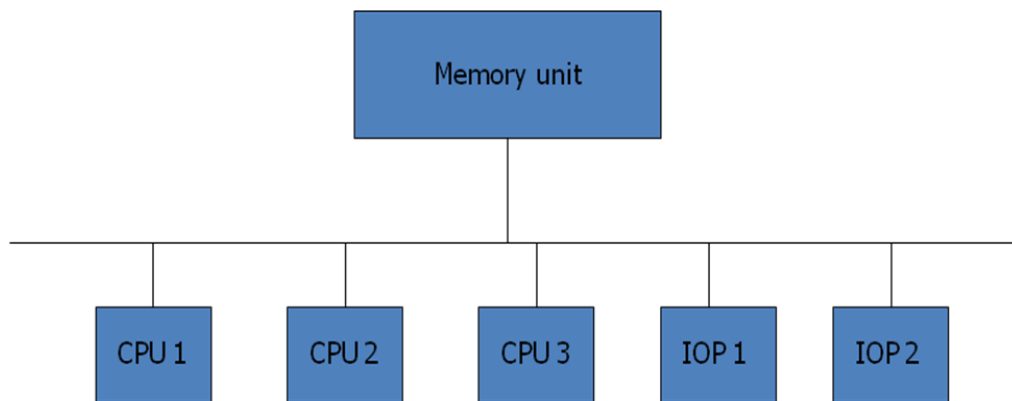


Fig: Time shared common bus organization

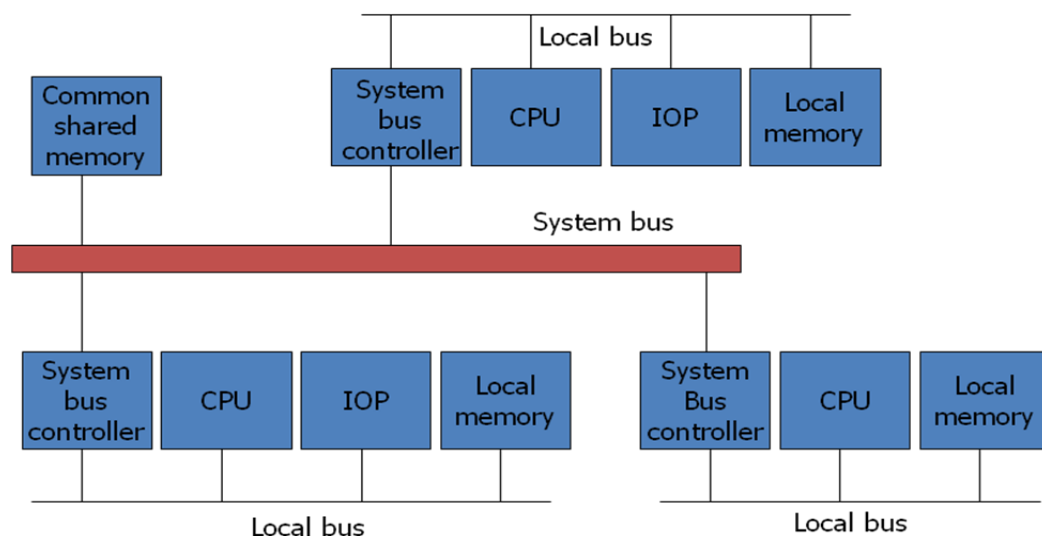


Fig: System bus structure for multiprocessors

Multiport Memory

- A multiport memory system employs separate buses between each memory module and each CPU.
- The module must have internal control logic to determine which port will have access to memory at any given time.
- Memory access conflicts are resolved by assigning fixed priorities to each memory port.
- *Adv.:*
 - The high transfer rate can be achieved because of the multiple paths.
- *Disadv.:*
 - It requires expensive memory control logic and a large number of cables and connections

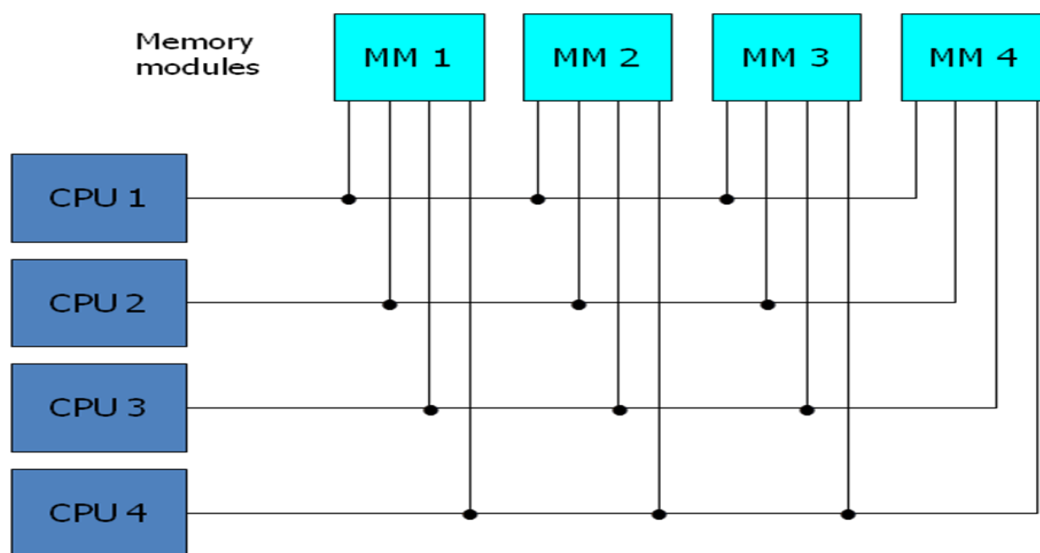


Fig: Multiport memory organization

Crossbar Switch

- ‡ Consists of a number of *crosspoints* that are placed at intersections between processor buses and memory module paths.
- ‡ The small square in each crosspoint is a *switch* that determines the path from a processor to a memory module.
- ‡ Adv.:
 - Supports simultaneous transfers from all memory modules
- ‡ Disadv.:
 - The hardware required to implement the switch can become quite large and complex.
- ‡ Below fig. shows the functional design of a crossbar switch connected to one memory module.

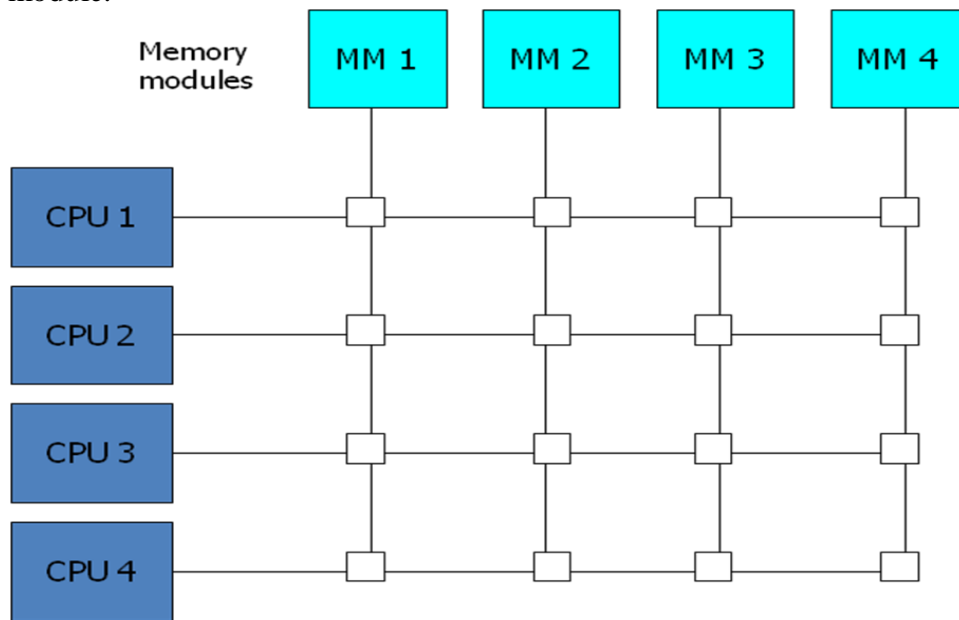


Fig: Crossbar switch

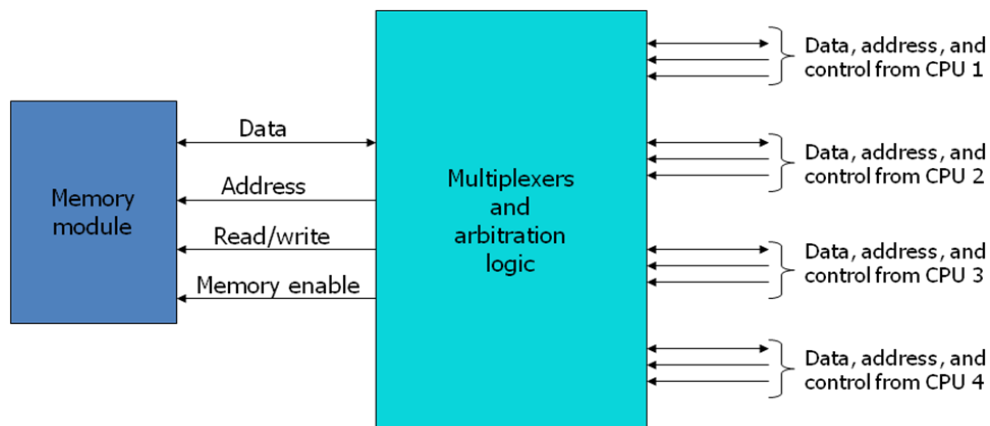
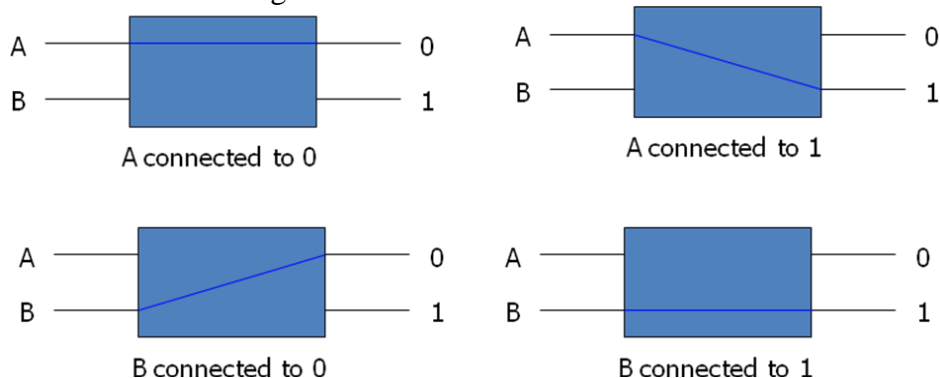


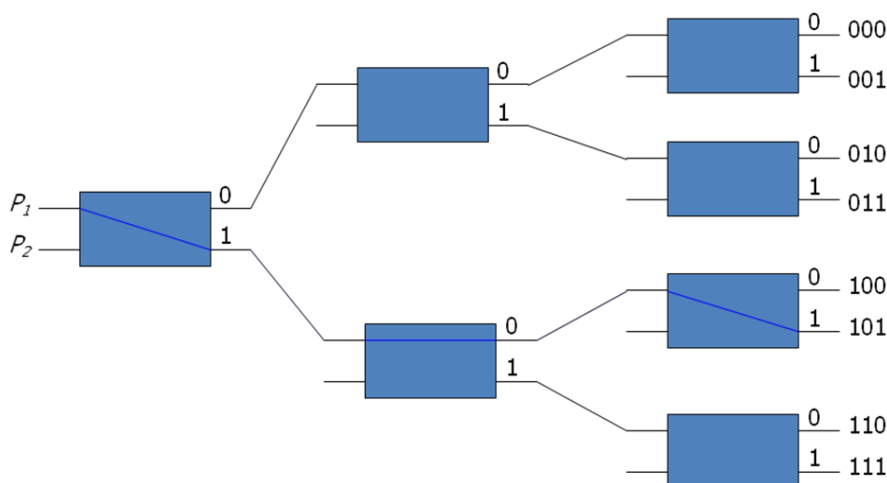
Fig: Block diagram of crossbar switch

Multistage Switching Network

- ¶ The basic component of a multistage network is a two-input, two-output interchange switch as shown in Fig. below.



- ¶ Using the 2x2 switch as a building block, it is possible to build a multistage network to control the communication between a number of sources and destinations.
 - To see how this is done, consider the binary tree shown in Fig. below.
 - Certain request patterns cannot be satisfied simultaneously. i.e., if P_1 000~011, then P_2 100~111



- ¶ One such topology is the omega switching network shown in Fig. below

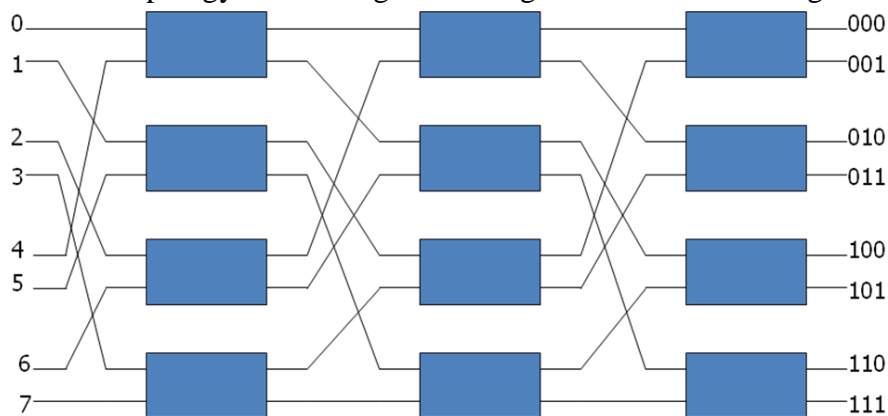


Fig: 8 x 8 Omega Switching Network

- ⌋ Some request patterns cannot be connected simultaneously. i.e., any two sources cannot be connected simultaneously to destination 000 and 001
- ⌋ In a tightly coupled multiprocessor system, the source is a processor and the destination is a memory module.
- ⌋ Set up the path transfer the address into memory transfer the data
- ⌋ In a loosely coupled multiprocessor system, both the source and destination are processing elements.

Hypercube System

- ⌋ The hypercube or binary n -cube multiprocessor structure is a loosely coupled system composed of $N=2^n$ processors interconnected in an n -dimensional binary cube.
 - Each processor forms a node of the cube, in effect it contains not only a CPU but also local memory and I/O interface.
 - Each processor address differs from that of each of its n neighbors by exactly one bit position.
- ⌋ Fig. below shows the hypercube structure for $n=1, 2$, and 3 .
- ⌋ Routing messages through an n -cube structure may take from one to n links from a source node to a destination node.
 - A routing procedure can be developed by computing the exclusive-OR of the source node address with the destination node address.
 - The message is then sent along any one of the axes that the resulting binary value will have 1 bits corresponding to the axes on which the two nodes differ.
- ⌋ A representative of the hypercube architecture is the Intel iPSC computer complex.
 - It consists of 128 ($n=7$) microcomputers, each node consists of a CPU, a floating-point processor, local memory, and serial communication interface units.

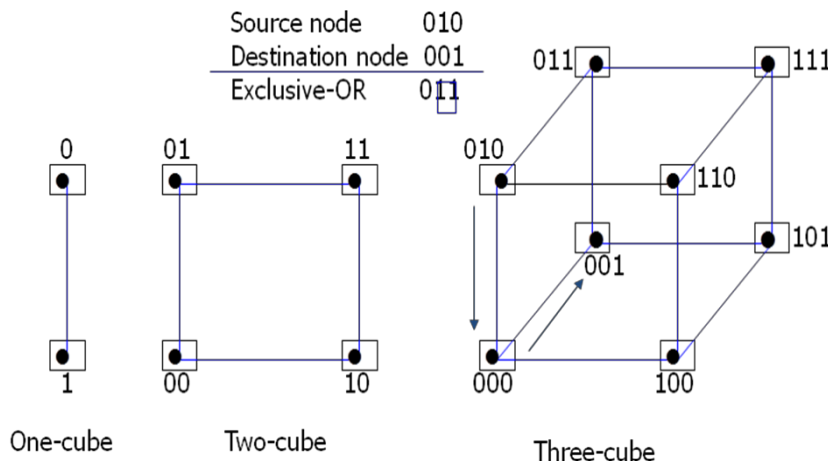


Fig: Hypercube structures for $n=1,2,3$

