1 . Introduction to data structure

Introduction: Data structures are used in almost every program or software system. Specific data structures are essential ingredients of many efficient algorithms, and make possible the management of huge amounts of data, such as large databases and internet indexing services. Some formal design methods and programming languages emphasize data structures, rather than algorithms, as the key organizing factor in software design.

1.1 Data: Data are simply values or set of values. A data item refers to a single unit of values. Data item that are divided into sub items are called group items; those which are not are called elementary items. Computer science is concerned with study of data which involves

- Machine that hold the data.
- Technologies used for processing data.
- Methods for obtaining information from data.
- Structure for representing data.

1.2 Information: Raw data is of little value in its present form unless it is organized into a meaningful format. If we organize data or process data so that it reflect some meaning then this meaningful or processed data is called information.

1.3 Data type: Data type is a term used to describe information type that can be processed by a computer system and which is supported by a programming language. More formally we define the data type as " a term which refers to the kind of data that a variable may take in ".

Several different types of data can be processed by a computer system e.g. Numeric data, text data, video data, audio data, spatial data etc.

A brief classification of data types is as shown in fig.

Data type

Built in or primary	**Derived**	**User defined**	**Abstract**	**other**
int,	array,	struct,	class	set of files
real,	enum	union		
char,				
boolean,				
void,				
pointer				

1.4.1 Data Structure: Data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently. When we define a data structure we are in fact creating a new data type of our own i.e. using predefined types or previously user defined types. The basic types of data structure includes : files, lists, arrays, trees etc.

Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. The choice of a particular data structure for an application depends upon two factors:

1. Structure must reflect the actual relationship of the data in the real world.
2. The structure should be simple enough so that one can efficiently process the data when necessary.

1.4.2 Data Structure operation: Depending upon the application we choose a data structure and perform specific operation. Following are some frequently performed operation with any data structure:

1. Traversing: To process the data we need to access each record exactly once so that certain records may be processed
2. Searching: Searching is an operation which finds the location of given records or finding records which satisfy one or more conditions.
3. Inserting: Adding new record to structure.
4. Deleting: Removing a record from structure .

1.4.3 Classification of Data Structure: Data structures are normally classified into two categories:

1. Primitive Data Structure: Primitive data structure are native to machine's hardware. Some primitive data structure are
   - Integer
   - Character
   - Real Number
   - Logical Number
   - Pointer
2. Non-primitive Data Structure: These type of data structure are derived from the primitive data structures. Some non primitive data structures are

   - Array
   - List
   - Files
   - Tree
   - Graph

Non-primitive data structures are further classified into two types

1. Linear Data Structure : If elements of the structure form a sequence then it is called linear data structure

e.g. array, stack, queue, linked list etc.

2. Non-linear Data Structure: If elements of the structure do not form a sequence then it is called non-linear data structure.
   e.g. tree, graph etc.

1.4.4 Need of Data Structure:

1. Data structures study how data are stored in a computer so that operations can be implemented efficiently
2. Data structures are especially important when you have a large amount of data
3. Conceptual and concrete ways to organize data for efficient storage and manipulation.

1.5 Abstract Data Types : Abstract data type is a mathematical model ,along with set of values and operations that can be performed on it with some rules. ADT is not concerned with storage representation & implementation part of data type rather it helps in making correct use of data type.
Abstract Data type is a type of a variable which specifies three sets

1. A set of Values
2. A set of rules or conditions
3. A set of operations

For example, when we declare a variable x of type integer we know that x represents an integer in the range of 16 bit and we can perform operation on x such as addition, subtraction, multiplication, and division. In case of data type integer we are not concerned with storage representation of it and how operations are implemented with it.

Array as an Abstract Data Type (ADT) can be defined as follows:

Set of Values : A set of values for an array are index and value at specified index i.e.(index, value) , where for each value in an array there is an index.

Set of Operations:

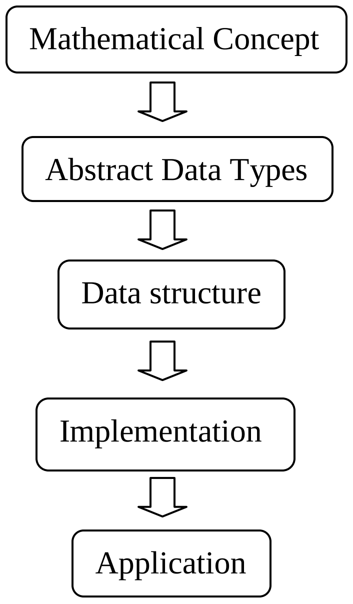
1. Item Retrieve(Arr, i)
2. Store (Arr,i,x)

1.6 Atomic Types:
It is set of values which is treated as single entity only and can not be subdivided. Data consists of a set of atoms. An atom usually consists of single elements(viz int, char etc) which hold certain information.

1.7 Structured Types:

A data object that is constructed as an aggregate of other data object is termed as "structured data object".
It is a set of values which has two ingredients:
1. It is made up of 'Component' elements. Component may be elementary or it may be another data structure.
2. There is a structure, i.e. a set of rules for putting the components together.

1.8 Refinement Stages:
Stages from any mathematical concept to its implementation are shown in fig .



ADT is mathematical concept which help in understanding the logical properties of a data type. So in initial stages of refinement we construct ADT of an object. We use Data structure to represent the mathematical model described by ADT. After defining the data structure we implement it using some technologies. There are number of alternatives to implement the data structure. Efficiency is the main factor concerned during the implementation of data structure. We can also implement a data structure using another data structure. After implementation of data structure we can use it in various applications. For example we can use data structure stack in evaluation of postfix expressions.

1.8 Difference between data type, data structure and abstract data type:
Data type is a term used to describe information type that can be processed by a computer system and which is supported by a programming language. More formally we define the data type as " a term which refers to the kind of data that a variable may take in ".
Abstract data type is a mathematical model, along with set of values and operations that can be performed on it with some rules. ADT is not concerned with storage

representation & implementation part of data type rather it helps in making correct use of data type. We use Data structure to represent the mathematical model described by ADT. Data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently. When we define a data structure we are in fact creating a new data type of our own i.e. using predefined types or previously user defined types.

<u>Questions asked in BTE</u>

Q. Define data structure. Enlist any two types of non-linear data structure along with examples of any one.
Q. Define data structure. What are different types of data structure.
Q. Define data type. Enlist different data types used in C programming language.
Q. Enlist the operations performed on data structure and explain any four operations with suitable example.
Q. Describe the term "Primitive data structures". Enlist four types of primitive data structure.
Q. Declare the structure in C for the following members.

Roll no	int
Name	char[10]
Marks	float

Calculate the size of memory required to store the above structure.

<u>2. Principles of Programming and Analysis of Algorithms</u>

Introduction:
The word "algorithm" has special significance in computer science, where "algorithm" refers to the method that can be used by a computer for solving a particular problem. This is what makes algorithm different from words such as process, technique. In short algorithm is outline, the essence of a computational procedure.

2.1 Algorithms:
Definition: Algorithm is finite set of instructions that can be followed to find a solution for given problem. An algorithm must have following characteristics

1. Input: An algorithm must accept input if supplied externally.
2. Output: An algorithm must give at least one output after processing input data.
3. Definiteness: Each instruction in algorithm must be lucid and unambiguous.
4. Effectiveness: Each instruction in an algorithm must be practicable. Any person either expert or novice user must be able to perform the operation with only pencil and paper.
5. Finiteness: Algorithm must terminate after a finite number of steps.

2.2 Different approaches for designing an algorithms:

An Algorithm is a hierarchy of component, the highest level components corresponding to the total algorithm. To design such a hierarchy there are two different possible approaches:

1. Top-down
2. Bottom-up

A top-down design approach starts by identifying the major component of the system, decomposing them into their lower level components, iterating until the desired level of details are achieved.

A bottom-up design approach starts with designing the most basic or primitive components that use these lower level components.

For the bottom up approach to be successful, we must have a good notion of the top where the design should be heading.

2.4 Algorithm analysis:

An algorithm can be analyzed by determining

1. The running time of a program as a function of its inputs;
2. The total or maximum memory space needed for program data;
3. The total size of the program code;
4. Does it work correctly accordingly to original specification of the task;
5. The complexity of the program i.e. how easy is it to read, understand, and modify;
6. The robustness of the program i.e. how well does it deal with the unexpected inputs?

However primarily the analysis of algorithm is concerned with the running time and the memory space needed to execute the program. An algorithm is said to be efficient if it has small running time and takes less memory space. The time and space requirements, referred as *time and space complexity* of an algorithm which enable us to measure how efficient an algorithm is.

2.5 Complexity of Algorithms:

The analysis of algorithm is a major task in Computer Science. In order to compare algorithms, we must have some criteria to measure the efficiency of our algorithms.

The total time needed by any algorithm in execution and memory space required by the algorithm are the two main measures for the efficiency of any algorithm.

To determine the execution time the following information is required:

1. Speed of the computer.
2. Language used.
3. The time required by each machine instruction.
4. What optimization compiler performs on the executable code etc.

While analyzing an algorithm time required to execute it is determined but time units are not useful since while analyzing algorithms our main concern is the relative efficiency of the different algorithms. Do also note that an algorithm cannot be termed as better because it takes less time or worse because it takes more time to execute. A worse algorithm may take less time to execute if we move it to a faster computer, or use efficient language. Hence while comparing two algorithms it is assumed that all other things like speed of the computer and the language used are same for both the algorithms.

Another approach to measure the efficiency of an algorithm is Frequency Count approach. In this approach the time is measured by counting the number of key operations performed by an algorithm on specific size of input data. Key operations are operations or steps which are not to be excluded from the algorithm. We count only key operations because time for the other operations is much less than or at the most proportional to the time for the key operations. The space is measured by counting the maximum of memory needed by the algorithm.
Consider the following examples:

Example1: Consider the following code segment

```
For( i=1; i<=n ; i++ )
      For( j=1 ; j<=n ; j++)
       X++;
```

Frequency cont for first for loop is n.
Frequency cont for second for loop is n.
Total frequency count for instruction X++ = n.n = n^2
Therefore Order of Complexity = 2.

Example2: Consider the following code segment

```
For( i=1; i<=n ; i++ )
      For( j=1; j<=n ;  j++ )
           For( k=1; k<=n ; k++ )
            X++;
```

Frequency cont for first for loop is (n-1).
Frequency cont for second for loop is (n-1)
Frequency cont for third for loop is (n-1)
Total frequency count for instruction X++ = (n-1) (n-1) (n-1) = n^3-3n^2+3n-1
Therefore Order of Complexity = 3.

2.5.1 Time Complexity:

Time complexity can be defined as running time of the program. There are many factors that affect the running time of a program which can be the algorithm itself, instance of input, and the computer system used to run the program. Time complexity cannot be measured by calculating the time using the computer clock, because: Program may also wait for I/O or other resources.
While running a program, a computer may performs many other computations. So, we must use some abstract notation to calculate time complexity. Generally the running time of an algorithm is proportional to the number of steps it takes to execute the algorithm. In such theoretical analysis of algorithms it is common to estimate their complexity in asymptotic sense, i.e. running time can be defined as a function of the size of the input data which is denoted as “Big-O-Notation”.

2.5.2 Space Complexity:

Space complexity can be defined as amount of computer memory required during the program execution. The space complexity also can not be exactly measured, but can be calculated by considering data and its size.

So, space complexity can also be calculated exactly like time complexity i.e. using "Big-O-Notation" of the size of the items which require maximum storage in given data.

2.6 Big-O Notation

There are infinitely many correct algorithm for the same algorithmic problem. So we need to choose a good algorithm among all the algorihms. A good algorithm is a one which is efficient means which has small running time & takes less memory. It is clear that if we have small input size algorithm takes less running time and as the input size increases running time algorithm also increases. Two compare algorithm we need to use same hardware and software platforms because different machine can also make a difference. In order to evaluate efficiency of any algorithm we need a way that is independent of the hardware and software environment. To analyze the algorithm generally we use asymptotic notation. Asymptotic notation simplifies the analysis of algorithm by getting rid of "details", which may be affected by specific implementation and hardware. Asymptotic notation is used to make meaningful statements about the efficiency of a program. It describes the behavior of the time or space complexity for large instance characterstics.

Big 0-Notation

Efficiency of particular algorithm can be measured for a particular size of input(N), but when N increases and it becomes larger, then same algorithm behaves differently. So Big O-Notation defines order of growth.

Definition

$f(n) = O(g(n))$, if there exist constant C and n_0

s.t. $f(n) < Cg(n)$ for all $n > n_0$

where $f(n)$ and $g(n)$ are functions over non-negative integers.

The statement $f(n) = O(g(n))$ read as "*f of n is big oh of g of n*" .

Here $f(n)$ represents the computing time of some algorithm $O(g(n))$ which takes time no more than constant $g(n)$. Big O-Notation provides an upper bound for the function.

We write $O(1)$ to mean a computing time that is constant. $O(n)$ is called linear, $O(n^2)$ is called quadratic, $O(n^3)$ is called cubic and $O(2^n)$ is called exponential. If algorithm takes time *$n \log(n)$*, it is faster for sufficiently large n, than if it had taken $O(n)$.

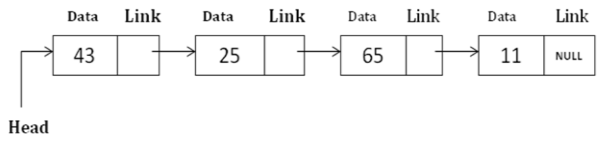
Similarly, *$O(n \log n)$* is better than *$O(n^2)$* but not as good as $O(n)$.

The statement $f(n) = O(g(n))$ states only that $g(n)$ is upper bound on the value of $f(n)$ for all $n > n_0$. It say nothing about how good this bound is.

Linked List

Linked list

Linked list is a linear data structure used to store similar data. It can also be defined as collection of elements called nodes, each consisting of data and link part. Linear linked list is most commonly used data structure of linked list.



In the figure above , **head** is pointer variable which contains the location of the first node of the list. Each node consist of data part & link part. The data part consist of data which is to be stored & link part consist of location of successor node in the list. The last node of the list does not have a successor node, and consequently, no actual address is stored in the link field. In such case, a **NULL** value is stored as the address. This is also called as *grounded header list* because the last node contains NULL pointer. The term grounded come from the fact that many texts use the electrical ground symbol to indicate NULL. The primitive operations of linear linked list include insert, delete, search and travels.

Linked List

Terminologies

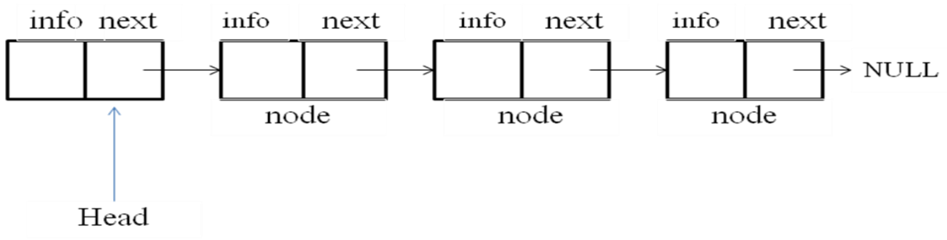


Fig. Linear Linked List

Node

Each element in the list is called a *node*. Linked list is collection of nodes. Node contains two fields, information field and a next address field.

Address

Address is *physical location number* of the node in computer memory. The next field of node contains address of successor node. If successor node is absent then next address field contains NULL value.

Pointer
Pointer is a reference to a data structure. We need to maintain address of each node in the list explicitly, The addressing mode, where explicit addressing of element is done, is called “pointer addressing”

Information
The information field holds the actual element of the list.

Next
Next is a pointer or an address that indicates explicitly the location of the successor node in the linked list.

Null Pointer
NULL pointer is a pointer, which doesn’t point anywhere. The NULL in the last node indicates that this is last node in the list.

Empty List
List with no nodes is called an empty list or NULL list.

Overflow
When we try to insert a data element when list is full this situation is called as *overflow* situation. If free memory is not available to reserve space for new node this condition occurs.

Underflow
When we try to delete a node from empty list this situation is called as *underflow* situation. If list is empty we can’t delete a node from list.

Representation of Linked List
Each node of the list contains two parts data part and link part, we have to represent each node through a structure.
While defining linked list we must have recursive definitions:

```
struct node
 {
   int          data ;
   struct node  *link;
 }
```

Link is a pointer of *struct node typ*e i.e. it can hold the address of variable of struct node type. Pointers permit the referencing of structure in a uniform way, regardless of the organization of the structure being referenced. Pointers are capable of representing a much more complex relationship between elements of a structure than a linear order.

Implementation of Linked List
Linked list can be implemented with following primitive operations:

1. **Insertion**: In this operation we add a new node to list. Node can be added at the beginning , at the end or in between two nodes.
2. **Deletion**: Removing a node from the list.
3. **Searching**: Searching a location of a particular node in the list.
4. **Traversing**: Visiting each node in the list.

Important Facts:
Linked List is:
1. A collection of nodes that form a linear ordering.
2. It is not a data type, its a representation.
3. Each "node" consist of data and a link to the successor node in the list.
4. A linked list can have additional pointers.
5. First item in list is called the *head*.
6. Last item in list is called the *tail*.
7. Other kinds of linked lists have additional pointers.

Advantages over Arrays
1. They are not fixed in size and hence can grow as elements are added.
2. No need to know beforehand how many elements are going to be stored.
3. Better storage space requirements.
4. When keeping sorted items, we never have to shift elements.

Disadvantages over Arrays
1. Overhead to extra storage of pointers.
2. May be less efficient when accessing or modifying data.

Types of Link List
1. Linearly-linked List
   - Singly-linked list
   - Doubly-linked list
2. Circularly-linked list
   - Singly-circularly-linked list
   - Doubly-circularly-linked list

3.3 **Operation on linked list**

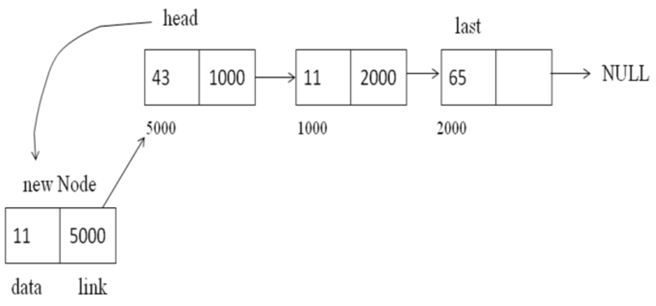
The operation that can be frequently performed on linked list are insert, delete & search. Several other operations can also be performed on list such as sorting list, reversing list etc. Now we will discuss some basic operations that can be performed on a linear linked list.

3.3.1. Inserting node

3.3.1.1 Inserting node at the beginning of the list

Algorithm for inserting a node to the beginning of the List
1. allocate memory space for a *newNode.*
2. copy the data into data part of the *newNode*.
3. If list is *empty* make the external pointer *head* to point to *newNode* and set the link of *newNode* to point to NULL.
4. If list is not *empty* make the *newNode ->link* point to *head node* and make the external pointer *head* to point to *newNode*.

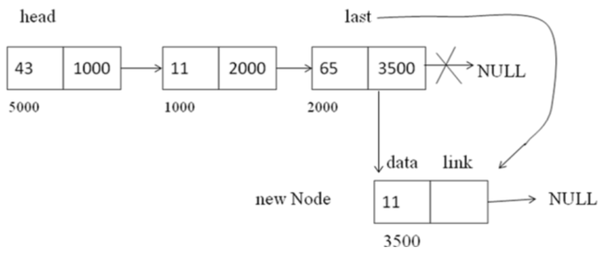
5. RETURN.



3.3.1.2 Append a node at the end of the list

Algorithm for appending a node at the end of the list

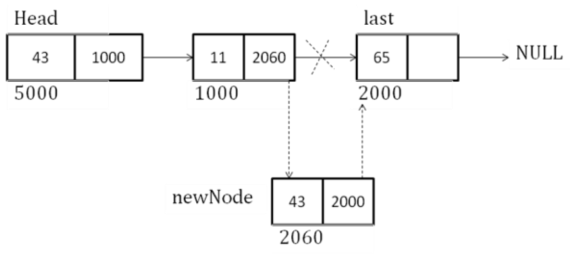
1. Allocate memory space for a *newNode.*
2. Copy the data into data part of the *newNode* and make link of *newNode* to point to NULL.
3. If list is *empty* make the external pointer *head* point to *newNode*.
4. If list is *not empty* mark the last node of list as R. Set the R-> *link* to point to *newNode*.
5. RETURN.



3.3.1.3 Inserting node at the specified position :

Algorithm for inserting a node to the specified position

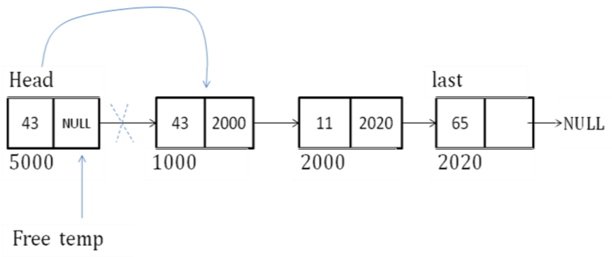
1. Allocate memory space for a *newNode*.
2. Copy the data into data part of the *newNode*.
3. Mark the node as R after which *newNode* is to be inseted. Set the *newNode-> link* to point to R->*link* and set R-> *link* to point to *newNode*.
4. RETURN.



3.3.2 Deleting node

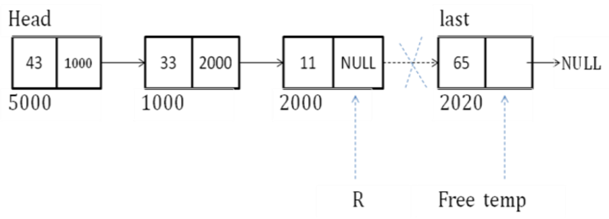
3.3.2.1 Deleting first node of the list
Algorithm

1. Mark the first node as *temp.*
2. Then, make the *head* node to point to the next node.
3. Make link of *temp* to point to NULL.
4. Free *temp.*
5. RETURN.



3.3.2.2 Deleting last node of the list
Algorithm

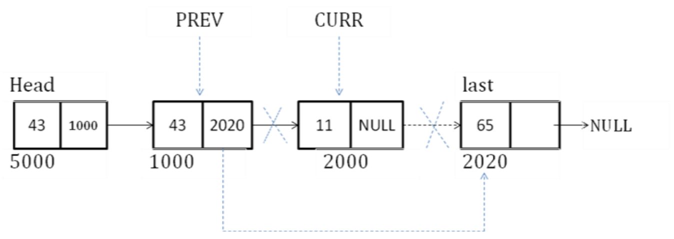
1. Mark the last node as *temp* and last but one node as R
2. Then, make the link field of R to point to NULL.
3. Free *temp*



3.3.2.3 Deleting specified node of the list
Algorithm

1. If list is *empty* print the message “list is empty” and go to step 6.
2. If list is *not empty* mark the node to be deleted as CURR and mark the previous node as PREV .

3. Make the PREV-> *link* to point to CURR-> *link*.
4. Make the CURR-> *link* to point to NULL.
5. Free CURR.
6. RETURN.



3.3.3.1 Searching specified node in the list

Algorithm for searching a specified node in the list

- If list is *empty* print the message " list is empty " .
- Otherwise take *temp* node which point to the *head* node of the list.
- Go on comparing value to be searched with data part of *temp.*
- If data not found make the *temp* point to next node, repeat till we reach end of the list.
- If value found print the message " node is found " .
- If we reach end of the list then print the message " node is not present in the list".

Circular Linked list:

A singly linked circular list is a linked list where the *last node* in the list points to the *first node* in the list. A circular list does not contain NULL pointer. This is also called as *circular header list* because last node points back to *header node*.

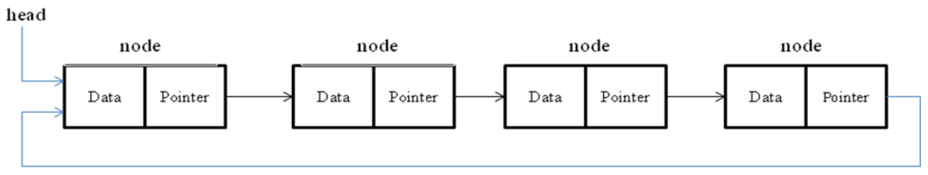


Fig. Circular Linked List

A good example of an application where circular linked list should be used is a timesharing problem solved by the operating system. In a timesharing environment, the operating system must maintain a list of present users and must alternately allow each user to use a small slice of CPU time, one user at a time. The operating system will pick a user, let him/her use a small amount of CPU time and then move on to the next user, etc. For this application, there should be no NULL pointers unless there is absolutely no one requesting CPU time.

The *insert*, *delete* and *search* operations are similar to the singly linked list. In circular linked list, we should always maintain that the next node of the tail is linked to the *head* after the insertion and deletion operations:

Advantages:

- Each node is accessible from any node.
- Address of the first node is not needed.
- Certain operations, such as concatenation and splitting of string, is more efficient with circular linked list.

Disadvantage:

- Danger of an infinite loop !

Implementation of circular list

A *circular linked list* does not have a first or last node. We must, therefore, establish a first and last node by convention. A circular list can be implemented as *stack* and *queue*.

Queue as a circular linked list

Insert a node

Algorithm

1. Allocate a memory space for *newNode*.
2. Copy the data into data part of the *newNode*.
3. If list is *empty* then make the *next* link of *newNode* to point to *itself*. Then make external pointer *front* and *rear* to point to *newNode*.
4. If list is *not empty* then make ther rear->*link* to point to *newNode* and then make external pointer *rear* to point to *newNode*.
   Now make *rear->link* to point to *front*.
5. RETURN.

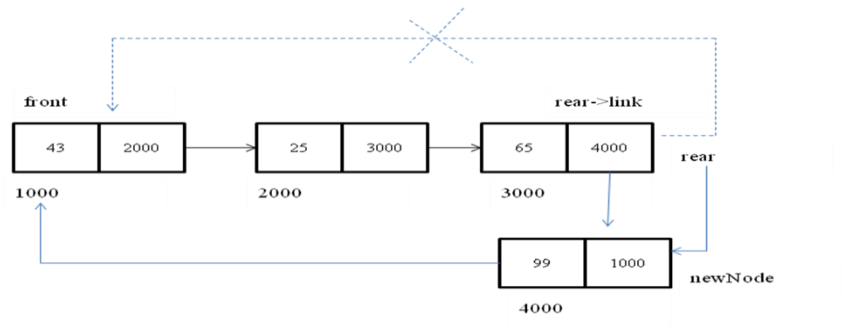


Fig. Circular Linked List as a queue

Delete a node

Algorithm

1. Make the *temp* to point to front node.
2. Make external pointer *front* to point to *front->link*.
3. Make *temp->next* to point to NULL.
4. Make *rear->next* to point to *front*.
5. free *temp*.
6. RETURURN.

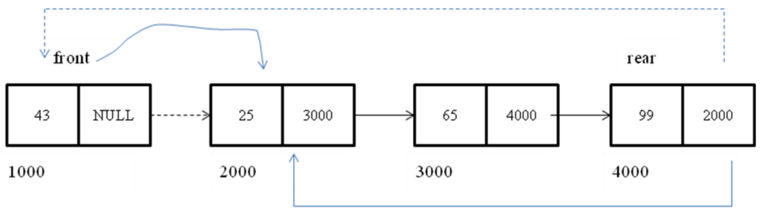
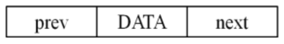


Fig. Delete operation in circular linked list .

Doubly Linked list:
In the linear linked list, we can only traverse the linked list in one direction. But sometimes, it is very desirable to traverse a linked list in either a forward or reverse manner. This property of a linked list implies that each node must contain two link fields instead of one. The links are used to denote the predecessor and successor of a node. The link ***prev*** denoting the predecessor of a node is called the left link, and the link ***next*** denoting the successor of a node is called the right link. A node in a doubly linked list has three fields

- Data
- prev
- next

The Figure shows the structure of the node.



list containing this type of node is called a *doubly linked linear list*. This list is also called as *two way header list* as we can traverse in both direction forward or backward.

In doubly linked list the previous link of first node and next link of last node points to NULL. As doubly linked list is having two pointers, insertion and deletion require more operations but the doubly linked list can be displayed in both directions i.e. from first node to last node following next link and also from last node to first node following previous link.

Advantages:

- A doubly linked list can be traversed in both directions (forward and backward).
- Searching and deletion operation are efficient.

Disadvantage:

- Overhead to extra storage of pointers.

3.3 **Operation on doubly linked list**

3.3.1.1 Inserting node at the beginning of the doubly linked list

Algorithm for inserting a node to the beginning of doubly liked list

1. Allocate the memory space for a newNode.
2. Copy the data into data part of the newNode.
3. If list is empty make the external pointer *head* to point to newNode and set the *prev* and *next* link of newNode to point to NULL.
4. If list is not empty then make *prev* link of *head* node to point to *newNode*. Then make *prev* link of *newNode* to point to NULL and *next* link to point to head node of the list. Make the external pointer *head* to point to *newNode*.
5. RETURN.

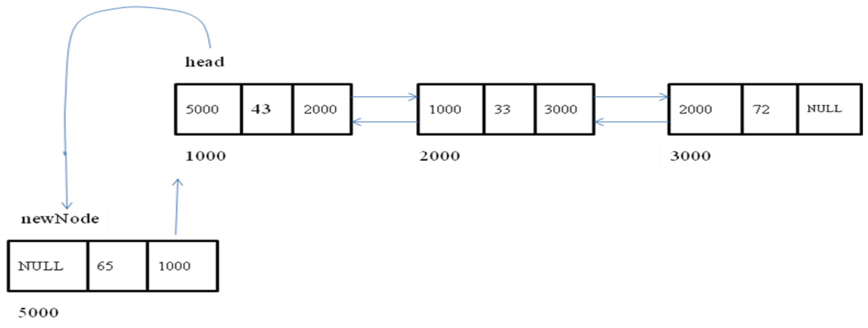


Fig. Insertion of a node at the beginning o

3.3.1.2 Append a node at the end of the list

Algorithm for appending a node at the end of doubly linked list

1. Allocate the memory space for a *newNode*.
2. Copy the data into data part of the *newNode*.
3. If list is empty then make the external pointer *head* to point to *newNode* and set the *prev* and *next* link of *newNode* to point to NULL.
4. If list is not empty mark the last node as *temp*. Make *next* link of temp to point to *newNode*. Make *prev* link of *newNode* to point to *temp* and *next* link to point to NULL.
5. RETURN.

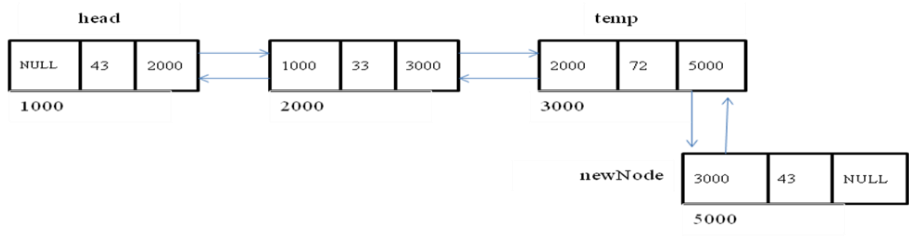


Fig. Insertion of a node at last position

3.3.1.3 Inserting node at the specified position :

Algorithm for inserting a node to the specified position

1.Allocate the memory space for a *newNode*.
2. Copy the data into data part of the *newNode*.
3. Mark the node as *temp* after which a *newNode* is to be inserted.
4. Make *next* link of *newNode* to point to *temp->next* and *prev* link to point to *temp*.

5. Make *temp->next->prev* link to point *to newNode* and *temp->next* to point to newNode.
6. RETURN.

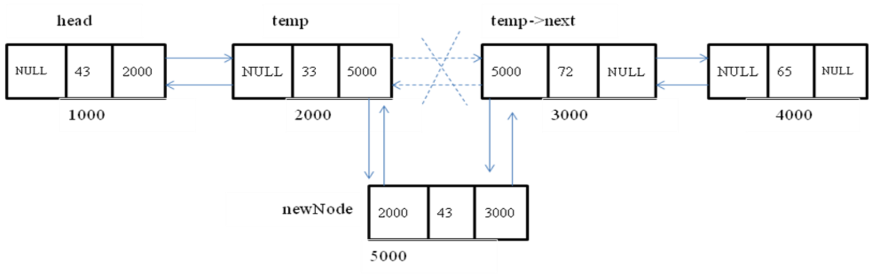


Fig. Insertion of a node at specified position

3.3.2 Deleting node

3.3.2.1 Deleting first node of doubly linked list

Algorithm

1. Mark the first node as *temp*.
2. Then, make the external pointer *head* to point to next node.
3. Make the *temp->next->prev* link to point to NULL.
4. Make the *temp->next* link to point to NULL.
5. Free *temp*.
6. RETURN.

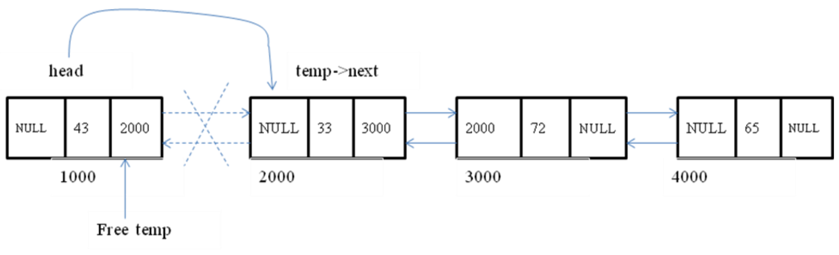


Fig. Deletion of first node

3.3.2.2 Deleting last node of doubly linked list

Algorithm

1. Mark the last node as *temp.*
2. Make *temp->prev->next* link to point to NULL.
3. Make the *temp->prev* link to point to NULL.
4. Free *temp.*
5. RETURN.

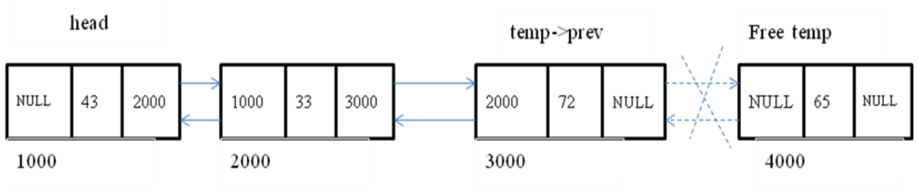


Fig. Deletion of last node

3.3.2.3 Deleting in between node of doubly linked list

Algorithm

1. Mark the node to be deleted as *temp*
2. Make *temp->next->prev* link to point to *temp->prev*.
3. Make *temp->prev->next* link to point to *temp->next*.
4. Make the *next* and *prev* link of *temp* to point to NULL.
5. Free *temp*.
6. RETURN

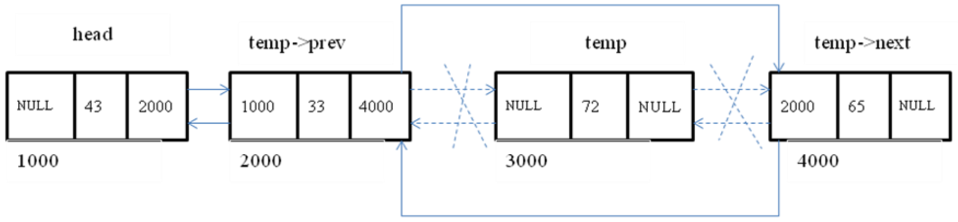


Fig. Deletion of in between node

<u>Menu driven program to perform various operation on linked list.</u>

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

struct node            /* Structure for node in the list */
 {
 int info;
 struct node *next;
 };
void add_last(struct node **,int); /* Prototype declaration */
void display(struct node *);
void add_begin(struct node**,int);
void delet(struct node**,int);

void main()
{
 clrscr();
 struct node *head=NULL;         /* External pointer to the
list*/
 int   option,data;
 char   choice1,choice2;
 do
 {
 clrscr();
 printf("\n----------------LINKED LIST------------------");
 printf("\n  1: CREATE A LIST       ");
 printf("\n  2: DISPLAY LIST       ");
 printf("\n  3: ADD NODE AT FIRST POSITION");
 printf("\n  4: DELETE NODE FROM LIST");
 printf("\n\n  Enter the option(1/2/3/4):\t ") ;
 scanf("%d",&option);
 switch(option)
  {
  case 1:
        do
        {
        printf("\n Enter the data to add  :\t");
        scanf("%d",&data);
        add_last(&head,data);
        printf("Do you want to add anoter node(Y/N):\t");
        choice1=getch();
        }while(choice1=='y' || choice1=='Y');
        break;

  case 2:   display(head);
        break;

  case 3:
        do
        {
        printf("\n Enter the data to add  :\t");
        scanf("%d",&data);
        add_begin(&head,data);
        printf("Do you want to add another node at first
position(Y/N):\t");
        choice1=getch();
        }while(choice1=='y' || choice1=='Y');
        break;
  case 4:
        do
        {
        printf("\n Enter the data to be deleted :\t");
        scanf("%d",&data);
        delet(&head,data);
        printf("\nDo you want to delete another node(Y/N):\t");
        choice1=getch();
        }while(choice1=='y' || choice1=='Y');
        break;
 }
 printf("\n\ndo you want to try other option(Y/N):\t ");
 choice2=getch();
}while(choice2=='y'||choice2=='Y');
getch();
 }

void add_last(struct node **head,int data)
 {
 struct node *temp,*newNode;

 newNode =((struct node*)malloc(sizeof(struct node)));      /*
Reserve space for newNode */
 newNode->info= data;
 newNode->next= NULL;

 if(!*head)                                         /* if list is
empty */
  {
  *head=newNode;      /* make head node to point to newNode*/
  return;
  }
  else                                                  /* if
list is not empty */
   {
     temp= *head;
     while(temp->next!=NULL)
     temp= temp->next;
     temp->next = newNode;             /*make last node to point
to newNode*/
   }
 }

void display(struct node *list)
{
 if(!list)
 {
 printf("\nSorry list is empty ");
 return;
 }
 printf("\n Node in the list are:\n");
 while(list)
  {
  printf("\t%d",list->info);
  list= list->next;
  }
}
void add_begin(struct node **head,int data)
{
 struct node *newNode;
 newNode =((struct node*)malloc(sizeof(struct node)));
 newNode->info= data;
 newNode->next= *head;
/* make newNode to point to head node */
 *head=newNode;                        /* make head to point to
new node */
}

void delet(struct node **head,int data)
 {
 struct node *curr=NULL,*prev=NULL;
 curr=*head;
 if(!*head)
  {
  printf("\n Sorry can not delete node from list");
  return;
  }
 while(curr && curr->info!=data )  /*search node in the list*/
    {
    prev=curr;
    curr=curr->next;
    }
 if(curr->info==data)             /*if node to be deleted found*/
  {
   if(curr==*head)                  /* if it is first node */
    *head=curr->next;
   else
    prev->next=curr->next;     /*if it is intermediate node*/
  free(curr);
  }
 else
 printf("\n Data is not found in the list");
 }
```

STACK

Stack is an ordered list in which all insertion and deletions are take place at one end called the top. The operations of a stack suggest that if the elements A, B, C, D and E are inserted into a stack, in that order, then the first element to be removed must be E. Equivalently we say that stack follows the principle of Last In First Out (LIFO).

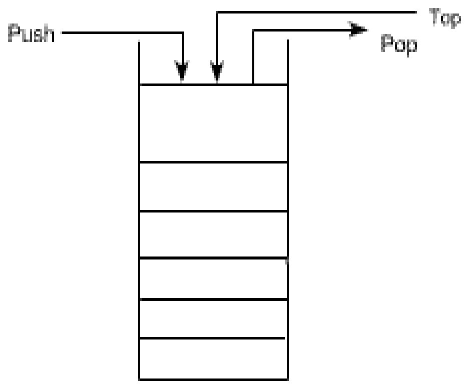


Fig: Stack

Basic Terminology

Special terminology is used for two basic operation associated with stack :

1. "Push" is the term used to insert an element into a stack.
2. "Pop" is the term used to delete an element from a stack.

The terminology associated with stacks comes from the spring loaded plate containers common in dining halls. When a new plate is washed it is *pushed on the stack.* When someone is hungry, a clean plate is *popped off the stack.* A stack is an appropriate data structure for this task since the plates don't care about when they are used!

Primitive operation on STACK

Stack can be implemented with following primitive operations

Opration	Description
Push	Adds element at the top of stack
Pop	Removes element from top of the stack
Isempty	Check whether stack is empty or not
Isfull	Check whether stack is full or not

STACK as an ADT

Set of values for STACK

Stack S is a finite collection of elements having same data type and all operation are carried out at one end called top.

Operation on STACK

1. **PUSH(STACK S, element x) -> STACK S$^{|}$;**

If stack is not full then, this function insert an element x into top of stack and returns new stack S’ with top pointing to the new element x.

2. **POP(STACK S) -> STACK** $S^{|;}$
   If stack is not empty then, this function deletes the element x pointed by the top of the stack and returns new stack $S^{|}$ with top pointing to the element down to the deleted element.
3. **isEmpty(STACK S) -> TRUE / FALSE;**
   This function returns TRUE when S is empty else FALSE.
4. **isFull(STACK S) -> TRUE / FALSE;**
   This function returns TRUE when S is empty else FALSE.

Implementation of STACK

Stack can be created using static implementation and dynamic implementation. In static implementation of stack we use array. It would be very easy to manage stack if we use array to represent a stack. However problem with an array is that we are required to declare the size of an array before using it in a program. This means size of array is fixed. Stack on other hand does not have fixed size. It keeps on changing as the elements in stack are popped or pushed. If there are too few elements to be stored in the stack the memory will be wasted. On the other hand, if there are more number of elements to be stored in the stack we will not be able to store, as we cannot increase the size of array once it is declared.

Dynamic implementation allocates memory at run time so, we don’t have to specify the size in advance. We can allocates memory as and when required, also there is no wastage of memory.

Representation of STACK through ARRAYS

Stack is a collection of elements having same data type. Hence for simplicity, we use array for representation of stack. As we know the operation on the stack are carried out at the end known as top, it is represented by an integer denoting index of the location of top most element of the stack. Hence stack is combination of array and top, it can be defined using structure as shown below

```
#define SIZE 100
struct stack
{
  int  top ;
  <data type>  s[SIZE];
}
```

Where <data type> indicates that the data can be of any type such as *int, float, char* etc

Representation of STACK through Linked List

When we implement stack as an array it suffers from basic limitation of an arry-that is size cannot be increased or decresed once it is declared. As result, one ends up reserving either too much space or too less space for an array and in turn for a stack. This problem can be overcome if we implement a stack using a linked list. In case of linked list stack shall push and pop nodes from one end of linked list. The stack as linked list is represented as a singly connected list. Each node in the list contains the data and a pointer to preceded node in the list. The node in the list can be represented as

```
struct node
{
  <data type>  data;
   node        *link;
}
```

Where <data type> indicates that the data can be of any type such as *int, float, char etc* and *link* is a pointer to the preceded node in the list. The pointer to the top node of the list serves the purpose of the *top* of the *stack.*

Operation on STACK

To push() an element to the stack

Elements can be added to the stack at top position only. Before adding an element we have to check Isfull() condition i.e. stack overflow condition. If *top* is equal SIZE-1 then stack is full.

Algorithm

```
1. Chek whether stack is full or not.
2. Otherwise, increment the top by 1.
3. Insert the element to the stack
   stack[top]=element.
```

To pop() an element from the stack.

Elements which is present at top of stack can be removed from stack. Before deleting an element from the stack underflow condition should be checked. Underflow condition occurs when *stack* is *empty*.

Algorithm

```
1. Chek whether stack is empty or not.
2. Otherwise, decrement the top by 1.
```

Application of Stack

1. Stack can be used to evaluate recursive function call and subroutine calls.
2. Stack can be used in string reverse operation.
3. Stack can be used in evaluation of arithmetic expression.
4. Stack can be used to check whether the arithmetic expression is well parenthesized or not.

Evaluation of arithmetic expression

One of the biggest challenge faced by computer scientist when higher level language came into existence was to generate machine level language instruction that evaluate any arithmetic instruction. In order to evaluate an expression each language assigns each operator a priority. Even after assigning priorities how can a compiler produce correct code. Solution lies in polish notation. A polish mathematician suggested a notation called Polish notation, which gives two alternatives to represent an arithmetic expression. The fundamental property of Polish notation is that the order in which the operations are to be performed is completely determined by the position of the operator and operands in the expression. Hence, parenthesis are not required while writing expression in Polish notation.

While writing an arithmetic expression, the operator is usually placed between two operands. e.g. A+B. This way of representing arithmetic expression is called infix notation. We can also represent arithmetic expression in another two different notation i.e. prefix and postfix notation. In prefix notation, the operator comes before the operands e.g. +AB which is equivalent to the infix expression A+B. The same expression in postfix form would be represented as AB+. In postfix notation, the operator follows the two operands.

Conversion of infix expression to postfix expression using stack

For the sake of simplicity we will assume that the string representing the infix expression, contains only arithmetic operators, parenthesis and operands. Also we will assume an arbitrary number associated with priority of operator.

Operator	Priority
^ ($)	3
* /	2
+ -	1

Rules for converting infix expression to postfix expression

1. The expression is to be read from left to right.
2. Scan one character at a time from infix expression.
3. Make use of STACK to store operators.

Algorithm

1. Scan the character from infix expression.
2. If scanned character is an opening parenthesis ‘(‘ then PUSH it to the STACK.
3. If scanned character is an operand then PUSH it to the postfix expression.
4. If scanned character is an operator then,
   a. POP the topmost element from the STACK. If the popped element has higher or same priority than the scanned character then add it to the postfix expression. Repeat step 4(a) till the popped element has higher or same priority than the scanned character.
   b. Otherwise PUSH back the popped element and character scanned to STACK respectively.
5. If scanned character is a closing parenthesis ‘)‘ then

a. POP the topmost element from the STACK and add it to the postfix expression. Repeat step 5(a) till the opening parenthesis is not popped. When popped element is '(' then discard both opening and closing parenthesis.

6. Repeat the step from 1 to 5 till end of the infix expression.
7. Till STACK is not empty, POP the element from STACK and add it to the postfix expression.
8. Finally print the postfix expression.

Example

Input infix expression: 5 $ 3 * 2 – 5 + 6 / 3 / (1 + 1)

Scanned Character	STACK	Postfix Expression	Comment
5	Empty	5	If operand add it to the postfix expression
$	$	5	STACK is empty. If operator add to the STACK
3	$	53	If operand add it to the postfix expression
*	*	53$	While priority(popped element) >= priority(scanned character) add popped element to postfix expression
2	*	53$2	If operand add it to the postfix expression
-	-	53$2*	While priority(popped element) >= priority(scanned character) add popped element to postfix expression
5	-	53$2*5	If operand add it to the postfix expression
+	+	53$2*5-	While priority(popped element) >= priority(scanned character) add popped element to postfix expression
6	+	53$2*5-6	If operand add it to the postfix expression
/	+ /	53$2*5-6	While priority(popped element) >= priority(scanned character) add popped element to postfix expression
3	+ /	53$2*5-63	If operand add it to the postfix expression
/	+ /	53$2*5-63/	While priority(popped element) >= priority(scanned character) add

			popped element to postfix expression
(	+ / (	53$2*5-63/	If '(' PUSH it to the STACK
1	+ / (	53$2*5-63/1	If operand add it to the postfix expression
+	+ / (+	53$2*5-63/1	While priority(popped element) >= priority(scanned character) add popped element to postfix exprssion
1	+ / (+	53$2*5-63/11	If operand add it to the postfix expression
)	+ / (+	53$2*5-63/11+	While not found '(' add popped operator to postfix expression
	+ /	53$2*5-63/11+	While STACK is not empty add operator to postfix expression
	Empty	53$2*5-63/11+/+	Final postfix expression

Evaluation of postfix expression
STACK can be used for evaluation of postfix expression. Here we will use STACK to store operands rather than operators.

Rules for evaluation of postfix expression

1. The expression is to be read from left to right.
2. Scan one character at a time from infix expression.
3. Make use of STACK to store operands.
4. Operands are taken as numeric values.

Algorithm

1. Scan the character from postfix expression.
2. If the scanned character is an operand then PUSH it to the STACK.
3. If the scanned character is an operator then POP two operands from the STACK and perform the arithmetic operation.
4. PUSH the result of arithmetic operation back to the STACK.
5. Repeat the step from 1 to 4 till the end of postfix expression.

Example

Input postfix expression: 53$2*5-63/11+/+

Scanned Character	STACK	Comment
5	5	If operand add to the STACK
3	5, 3	If operand add to the STACK

$	125	If operator pop two element from the STACK and perform operation then push result back to the STACK.
2	125,2	If operand add to the STACK
*	250	If operator pop two element from the STACK and perform operation then push result back to the STACK.
5	250,5	If operand add to the STACK
-	245	If operator pop two element from the STACK and perform operation then push result back to the STACK.
6	245,6	If operand add to the STACK
3	245,6,3	If operand add to the STACK
/	245,2	If operator pop two element from the STACK and perform operation then push result back to the STACK.
1	245,2,1	If operand add to the STACK
1	245,2,1,1	If operand add to the STACK
+	245,2,2	If operator pop two element from the STACK and perform operation then push result back to the STACK.
/	245,1	If operator pop two element from the STACK and perform operation then push result back to the STACK.
+	246	If operator pop two element from the STACK and perform operation then push result back to the STACK.
	246	Display result from STACK

Conversion of postfix expression to infix expression using stack
STACK can be used to convert postfix expression into infix expression. Here we will use STACK to store operands.

Rules for converting postfix to infix expression

1. The expression is to be read from left to right.
2. Scan one character at a time from infix expression.
3. Make use of STACK to store operands.

Algorithm

1. Scan the character from *postfix expression*.
2. If the scanned character is an *operand* then PUSH it to the STACK.
3. If the scanned character is an *operator* then POP topmost two operands from the STACK and form the temporary string containing operand-operator-operand. PUSH the temporary string formed to STACK.
4. Repeat the step from 2 to 3 till the end of infix expression.

Example

Input postfix expression: 53$2*5-63/11+/+

Scanned Character	STACK	Comment
5	5	If operand add to the STACK
3	5, 3	If operand add to the STACK
$	5$3	If operator pop two element from the STACK and perform operation then push result back to the STACK.
2	5$3,2	If operand add to the STACK
*	5$3*2	If operator pop two element from the STACK and perform operation then push result back to the STACK.
5	5$3*2,5	If operand add to the STACK
-	5$3*2-5	If operator pop two element from the STACK and perform operation then push result back to the STACK.
6	5$3*2-5,6	If operand add to the STACK
3	5$3*2-5,6,3	If operand add to the STACK
/	5$3*2-5,6/3	If operator pop two element from the STACK and perform operation then push result back to the STACK.
1	5$3*2-5,6/3,1	If operand add to the STACK
1	5$3*2-5,6/3,1,1	If operand add to the STACK
+	5$3*2-5,6/3,1+1	If operator pop two element from the STACK and perform operation then push result back to the STACK.
/	5$3*2-5,6/3/1+1	If operator pop two element from the STACK and perform operation then push result back to the STACK.
+	5$3*2-5+6/3/1+1	If operator pop two element from the STACK and perform operation then push result back to the STACK.
		Display result from STACK

BTE questions asked in Exam:

Q. State the principal of stack with basic operations. 2 mark
[S-09]
Q. Write down the application of stack 4 mark
[S-09]
Q. Define the terms 'overflow' and 'underflow' with respect to stack . 2 mark
[S-08]
Q. Write a 'C/C++' program to generate fibonacci series using recursion. 4 mark
[S-08]
Q. Write a menu driven program for implementing stack using array 6 mark
[S-08]

```
/* Program for static implementation of stack */

#include<stdio.h>
#include<conio.h>
#define SIZE 10

void push(int);                  //Prototype declaration
void pop();
int stack[SIZE],top=-1;   //Global defination

void main()
 {
  clrscr();
  push(43);
  push(74);
  push(33);
  push(25);
  pop();
  pop();
  getch();
 }

void push(int data)
 {
  if(top==SIZE-1)
  {
   printf("\n Stack is overflowing");
   return;
  }
  top++;
  stack[top]=data;
 }

void pop()
 {
  int data;
  if(top==-1)
```

```
   {
    printf("\n Stack is uderflowing");
    return;
   }
   data=stack[top];
   stack[top]=0;
   top--;
   printf("\n Element %d is popped from stack ",data);
  }

/*
 OUTPUT :

 Element 25 is popped from stack
 Element 33 is popped from stack                */
```

/* Program for dynamic implementation of stack */

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

struct node
 {
 int     data;
 struct  node *link;
 };

void push(struct node**,int data);  //Prototype declaration
void  pop(struct node**);

void main()
{
 clrscr();
 struct node *top=NULL;  //Pointer for top of element of stack.
 push(&top,43);
 push(&top,74);
 push(&top,33);
 push(&top,25);
 pop(&top);
 pop(&top);
 getch();
}

void push(struct node **top ,int data)
{
 struct node *newNode;

 newNode=(struct node*)malloc(sizeof(struct node));   //create
new node
```

```
 if(newNode==NULL)         //Checking overflow condition
 {
  printf("\n Stack is overflowing");
  return;
 }
 newNode->data = data;
 newNode->link = *top;
 *top = newNode;
}

void pop(struct node **top)
{
 struct node *temp;
 int data;

 if(*top==NULL)           //checking  uderflow condition
 {
  printf("\n Stack is undeflowing ");
  return;
 }
 temp = *top;
 data = temp->data;
 *top = (*top)->link;
 free(temp);
 printf("\n Element %d is popped from stack",data);
}
```

OUTPUT :

```
 Element 25 is popped from stack
 Element 33 is popped from stack
```

/* Program for converting infix expression into postfix expression */

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<ctype.h>
#define SIZE 30

char stack[SIZE];
int  top= -1;

int priority(char);
void conversion(char*,char*);
void push(char);
char pop();

void main()
```

```
{
 clrscr();
 char infix[SIZE],postfix[SIZE];

 printf("\n\n Enter infix expression : ");
 gets(infix);
 conversion(infix,postfix);
 printf("\n\n    Posfix expression is : %s ",postfix);

 getch();
}

void push(char c)
{
 stack[++top]=c;
}
char pop()
{
 if(top==-1)
 return -1;
 else
 return(stack[top--]);
}

void conversion(char *infix, char *postfix)
{
 char opr;
 while(*infix)
   {
     if(*infix=='(')
      {
      push(*infix);
      infix++;
      }

     if(isdigit(*infix)|| isalpha(*infix))
      {
       while(isdigit(*infix) || isalpha(*infix))
         {
           *postfix=*infix;
           infix++;
           postfix++;
         }
      }

     if(*infix=='*'|| *infix=='/'|| *infix=='%'||
       *infix=='+'|| *infix=='-'|| *infix=='$'   )
      {
       if(top!=-1)
        {
          opr=pop();
          while(priority(opr) >= priority(*infix))
```

```
           {
            *postfix=opr;
            postfix++;
            opr=pop();
           }
          push(opr);
          push(*infix);
        }
       else
       push(*infix);
       *infix++;
      }

     if(*infix==')')
      {
           opr=pop();
      while(opr!='(')
               {
             *postfix=opr;
               postfix++;
               opr=pop();
         }
       infix++;
      }
  }
  while(top!=-1)
   {
       opr=pop();
      *postfix=opr;
        postfix++;
   }
   *postfix='\0';
}
int priority(char c)
{
  if(c=='$')
  return 3;
  if(c=='*'||c=='%'||c=='/')
  return 2;
  if(c=='+'||c=='-')
  return 1;
  else
  return 0;
}
```

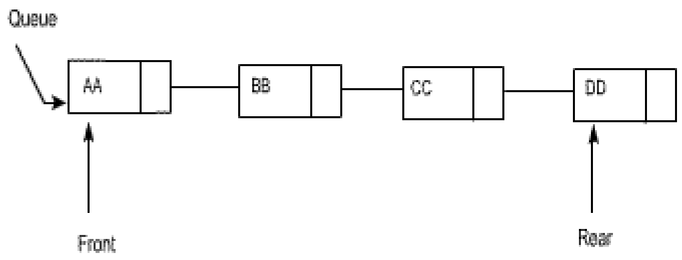
OUTPUT :

```
Enter infix expression : 4$2*3-3+8/4/(1+1)

Posfix expression is    : 42$3*3-84/11+/+ÿ
```

```
/*  Program to evaluate postfix expression */

#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<ctype.h>
#include<process.h>
#define  SIZE 30

int stack[SIZE];
int top=-1;

void evaluate(char*);
void push(int data)
{
 stack[++top]=data;
}
int pop()
{
return(stack[top--]);
}

void main()
{
 clrscr();
 char postfix[30];
 printf("\n Enter postfix expression : ");
 gets(postfix);
 evaluate(postfix);
 getch();
}

void evaluate(char postfix[])
{
   int num1,num2,num3;
    while(*postfix)
     {
       if(isdigit(*postfix))
              push((int)*postfix-'0');
       else
            {
              num1=pop();
              num2=pop();
              switch(*postfix)
               {
                case '+' : num3=num2+num1;
                           break;
                case '-' : num3=num2-num1;
                           break;
                case '/' : num3=num2/num1;
                           break;
                case '%' : num3=num2%num1;
                           break;
                case '*' : num3=num2*num1;
                           break;
                case '$' : num3=pow(num2,num1);
                           break;
                default :  printf("\n Wrong expression !");
                           exit(1);
               }
               push(num3);
             }
             postfix++;
        }
 printf("\n Result of evaluation is : %d",pop());
}

/*
OUTPUT :
 Enter postfix expression : 42$3*3-84/11+/+

 Result of evaluation is : 46                    */
```

Queue
Queue is an ordered list in which deletions can take place only at one end, called the “front” and insertion can take place only at the other end, called "*rear* ".
The operations of a queue suggest that if the elements A, B, C, D and E are inserted into a queue, in that order, then the first element to be removed must be A. Equivalently we say that queue follows the principle of First In First Out (FIFO). The real life example: the people waiting at bus stop in queue. Each new person who comes takes his or her place at

end of the queue, and when bus comes, the people at the front of the queue board first. An important example of a queue in computer science occurs in timesharing system, in which programs with the same priority form a queue while waiting to be executed.



Queue as an Abstract data type
Set of values for queue
Queue Q is a finite collection of elements having same data type and all operations are carried out at two ends called as front and rear, where front is used to remove the elements and rear is used to insert the elements.
Operation on Queue

1. **Enqueue(Queue Q, element x) -> Queue Q $^{|}$**
   If queue is not full then this function inserts an element x into rear of the queue and returns new queue Q $^{|}$ with rear pointing to the position of x.

2. **Dequeue(Queue Q) -> Queue Q $^{|}$**
   If queue is not empty then this function deletes an element x pointed by the front of the queue and returns new queue Q $^{|}$ with front pointing to the element up to the deleted element.

3. **isEmpty(Queue Q) -> TRUE / FALSE;**
   This function returns TRUE when Q is empty else FALSE.

4. **isFull(Queue S) -> TRUE / FALSE;**
   This function returns TRUE when Q is full else FALSE.

Representation of Queue
Representation of QUEUE through ARRAYS
Queue is a collection of elements having same data type. Hence for simplicity, we use array for representation of queue. Each queue is maintained by an array and two integer variable front and rear containing the location of the front element of the queue and the location of the rear element of the queue. Hence queue is combination of array and two integer variable front and rear, it can be defined using structure as shown below

```
#define SIZE 100
struct queue
```

```
{
   int            front, rear ;
<data type>  q[SIZE];
}
```

Where <data type> indicates that the data can be of any type like *int, float, char* etc

Representation of QUEUE As A Linked-List

Queue can also be represented using a linked list. We refer it as dynamic implementation of queue because as in array there is no limitation on number of elements to be inserted in queue. Space for the elements in a linked list is allocated dynamically, hence it can grow as long as there is enough memory available for dynamic allocation. The node in the queue can be represented as follows

```
struct node
{
<data type> data;
struct node *link;
};
```

Where <data type> indicates that the data can be of any type such as *int, float, char etc* and *link* is a pointer to the successor node in the list. The pointer to the beginning of the list serves the purpose of the front of the queue and pointer to the last node in the list serves the purpose of the rear of the queue.

Operation on queue

Operation on the queue are carried out at the ends called front and rear of queue. Elements are inserted in queue at rear end and that are deleted from the queue at front end. Front and rear ends are represented by an integers denoting index of the location of first and last element of the queue respectively.

Inserting element in the Queue

As stated earlier elements are inserted in queue at rear end only. . Before inserting an element we have to check Isfull() condition i.e. queue overflow condition. If *rear* is equal SIZE-1 then queue is full.

Algorithm

```
4. Chek whether queue is full or not.
5. Otherwise, increment the rear by 1.
6. Insert the element to the queue
           queue[rear]=element.
```

Deleting an element from the Queue

Elements are deleted from queue at front end. Before deleting an element from the queue underflow condition should be checked. Underflow condition occurs when *queue* is *empty*.

Algorithm

```
3. Chek whether queue is empty or not.
4. Set queue[front]=0.
```

5. Increment the front by 1.

Difference between Stack and Queue

Stack	Queue
A stack is a linear data structure in which elements can be inserted or deleted at one end called top of the stack	A queue is a linear data structure in which elements can be inserted at one end called rear and elements are deleted from the other end called front of the queue
Stack follows the principal Last-in-First out	Queue follows the principal First-in-First out
It maintain one pointer called top of the stack	It maintain two pointers called front and rear
e.g. Stack of plates	e.g. People waiting at bus stop
Stack is frequently used in subroutine calls	Queue is used in time sharing system

Circular queue

A circular queue overcomes the problem of utilizing space in linear queue implemented using arrays. It also has a front and rear to keep the track of items to be deleted and inserted and therefore to maintain the unique characteristics of the queue.

Suppose if we insert 5 elements in the queue of size 5. Now remove the first element . Now if we try to insert an element, program will give the message as "Queue overflow" even though there are only 4 elements in the queue of size 5. This condition occurs because we are making front and rear to move in straight or linear direction instead of using previous empty slots. Hence in normal queue if "rear" is already at Queue's maximum position and though we delete few elements from the "front" and release some space, we are not able to add anymore elements. This is major drawback of linear queue. We can overcome this drawback of linear queue by implementing queue as circular queue. In circular queue if "rear" is at Queue's maximum position and if there are empty slots at the beginning of the queue then new elements are added at these empty slots. In short just because we have reached the maximum position of the queue , queue is not reported as full. The queue is reported full only when all the slots in queue are occupied.

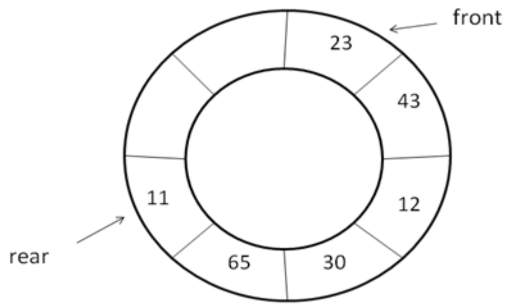


Fig : Circular queue

Priority Queue

A priority queue is different from the normal queue because it does not follow the principal of First-in-First-out. The order in which elements are inserted and deleted is determined by the priority of the elements. Following rules are applied to maintain a priority queue.

1. The elements with higher priority is processed before any elements of lower priority.
2. If there are elements with the same priority, then the elements added first in the queue would get processed.

Priority queues are used for implementing job scheduling by the operating system where jobs with higher priorities are to be processed first.

There are two types of priority queue

1. Ascending priority queue
2. Descending priority queue

A collection of elements into which elements can be inserted arbitrarily and from which only the smallest element can be removed is called Ascending priority queue.
In descending priority queue only the largest element is deleted.

Deque Queue

The word deque is short form of double-ended queue and defines a data structure in which items can be added or deleted at either the front or rear end, but no changes can be made elsewhere in the list. Thus a deque is generalization of both stack and a queue. Fig shows the representation of a deque.

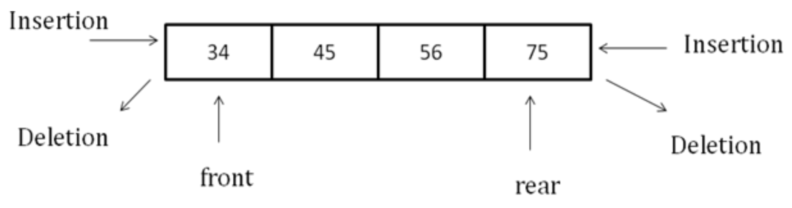


Fig : Representation of a deque

Application of queue

1. Queue can be used by operating system in Process scheduling.
2. Queue can be used by operating system in Batch programming.
3. Queue can be used in radix sort.
4. Queue can be used in a time-sharing computer system where many user can share computer system simultaneously.
5. Round robin technique for processor scheduling is implemented using circular queue.

BTE questions asked in Exam:

Q. Describe the concept of queue with the help of suitable example. 2 mark [S-07]
Q. Explain the types of queue with example. 8 mark [S-07]
Q. Distinguish between stack and queue with suitable example . 4 mark [W-06]
Q. What is Queue ? Explain Priority queue with suitable example 4 mark [W-06]
Q. Write a procedure to insert an element into a queue and to delete an element from a queue. Also explain DEQUES 8 mark [S-09]
Q. Define Priority Queue. Describe the one-way list representation of a priority Queue with suitable example and diagram.

```
/* Program for static implementation queue */
#include<stdio.h>
#include<conio.h>
#define SIZE 10

int queue[SIZE];
int front=-1,rear=-1;
void enqueue(int data);
void dequeue();
void main()
{
 clrscr();
 dequeue();
 enqueue(43);
 enqueue(65);
 enqueue(12);
 enqueue(74);
 dequeue();
 dequeue();
 getch();
}
```

```
void enqueue(int data)
{
 if(rear==SIZE-1)
  {
   printf("\n Queue is overflowing");
   return;
  }
 rear++;
 queue[rear]=data;
 if(front==-1)
 front=0;
}

void dequeue()
{
 int data;
 if(front==-1)
  {
   printf("\n Queue is underflowing");
   return;
  }
  data=queue[front];
  queue[front]=0;
  if(front==rear)
  front=rear=-1;
  else
  front++;
  printf("\n Element %d is deleted from queue ",data);
}
/*
 OUTPUT:
 Queue is underflowing
 Element 43 is deleted from queue
 Element 65 is deleted from queue                    */

  /*Program for dynamic implementation of queue*/
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>

struct node
{
 int data;
 struct node *link;
};
```

```
void enqueue(struct node**,struct node**,int);
void dequeue(struct node**, struct node**);

void main()
{
 clrscr();
 struct node *front=NULL, *rear=NULL;
 dequeue(&front,&rear);
 enqueue(&front,&rear,43);
 enqueue(&front,&rear,65);
 enqueue(&front,&rear,12);
 enqueue(&front,&rear,74);
 dequeue(&front,&rear);
 dequeue(&front,&rear);
 getch();
}

void enqueue(struct node **front,struct node **rear,int data)
{
 struct node *newNode;
 newNode=(struct node*)malloc(sizeof(struct node));

  if(newNode==NULL)
   {
    printf("\n Queue is overflowing");
    return;
   }

 newNode->data=data;
 newNode->link=NULL;

 if(*front==NULL)
 *front=newNode;
 else
 (*rear)->link=newNode;
 *rear=newNode;
}
void dequeue(struct node **front,struct node **rear)
{
 struct node *temp;
 int data;

 if(*front==NULL)
  {
  printf("\n Queue is underflowing");
  return;
  }

 temp=*front;
 data=temp->data;
 *front=(*front)->link;
 free(temp);

 if(*front==NULL)
 *rear=NULL;
 printf("\n Element %d is deleted from queue",data);
}
/*
 OUTPUT :

 Queue is underflowing
 Element 43 is deleted from queue
 Element 65 is deleted from queue        */

/* Program for circular queue */
#include<stdio.h>
#include<conio.h>
#define SIZE 10

int cqueue[SIZE];
int front=-1,rear=-1;

void enqueue(int);
void dequeue();

void main()
{
 clrscr();
 enqueue(12);
 enqueue(43);
 enqueue(74);
 enqueue(65);
 dequeue();
 dequeue();
 getch();
}

void enqueue(int data)
{
 if((rear+1==front) || (front==0&&rear==SIZE-1))
  {
    printf("\n Queue is overflowing ");
    return;
   }
  if(rear==SIZE-1)
  rear=0;
  else
  rear++;
  cqueue[rear]=data;
  if(front==-1)
  front++;
}
void dequeue()
{
 int data;
 if(front==-1)
 {
  printf("\n Queue is underflowing");
  return;
 }
 data=cqueue[front];
 cqueue[front]=0;

 if(front==rear)
  front=rear=-1;
  else
   {
     if(front==SIZE-1)
            front=0;
       else
            front++;
   }
   printf("\n Element %d is deleted from queue",data);

}

/*
OUPUT ;

 Element 12 is deleted from queue
 Element 43 is deleted from queue                */
```

Tree

A tree is a connected acyclic graph in which one vertex is singled out as the root & several other vertices connected to it by an edge inherit a parent child relationship. Again each node is further connected to their child nodes making set of trees

Basic Tree Terminology

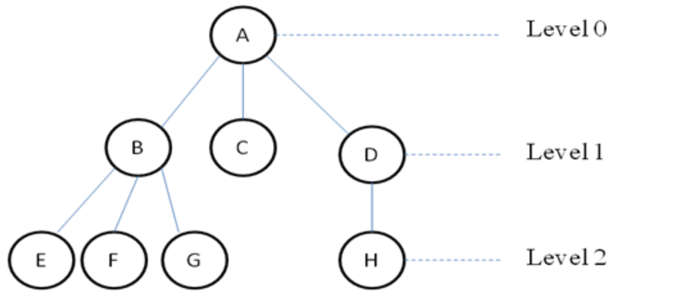


Fig. A tree

Root

The node having no parent is called root of the tree. This is unique node in the tree to which sub-tree are attached. In Fig above the root of the tree is node A.

Parent node

Node A is said to be parent of node B if there exist an edge going from A down to B.

Child node

Node B is said to be child of node A if there exist an edge going from b up to A.

Siblings

Node that share a common parent are known as siblings.

Leaf node

Nodes which have no children are called leaf nodes. These are also referred as non-terminals.

Ancestor

All the nodes that exist on path from a particular node to the root node are called as an ancestor of that particular node.

Decendant

All the nodes that exist on a path from a particular node to the leaf node are called as descendant of that particular node.

In-degree

In-degree of node is number of edges arriving at that node. In-degree of node B in above tree is 1. Root node in the tree is having in-degree 0.

Out-degree

Out-degree of node is the number of edges leaving that node. Out-degree of node B in above tree is 3. Leaf node in the tree is having out-degree 0.

Level of node

The level of node n is the length of path from the root to that node n. The root node is at level 0.

Height of the tree

The height of the tree is the length of path from the root to the deepest node in the tree. In above example leaf nodes are at level 2 and the height of the tree is 2.

Size of node
The size of node is the number of descendant it has including itself. In above example, the size of B is 4.
Forest
Collection of disjoint tree is called forest.

Types of tree

Binary Tree
In computer science a most important type of tree structure is a binary tree. In this type of tree any node will have atmost two branches i.e. degree of each node will be atmost two. We distinguish the two branches of tree as left subtree and right subtree.

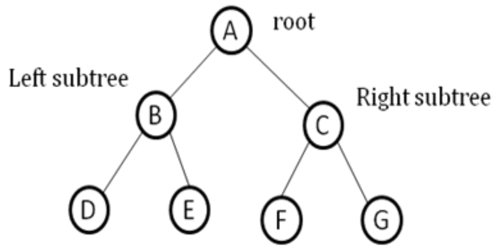


Fig : Binary tree

Fig above shows a binary tree with root as A, a left subtree with root B and right subtree with root C. We can define a binary tree as an empty tree or it is finite set of nodes with root and two disjoint binary tree called left subtree and right subtree.

Skewed Binary Tree
Binary tree having only left or right branches are known as skewed binary tree. Left skewed binary trees has only left branches whereas right skewed binary tree has only right braches.

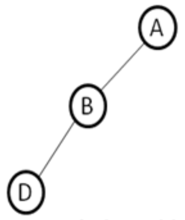


Fig : Left skewed binary tree

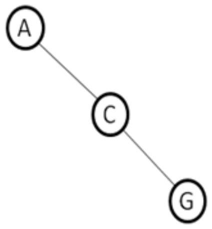


Fig : Right skewed binary tree

Complete Binary Tree
A complete binary tree is defined as a binary tree in which
1. All the leaf nodes are present at level n-1 or level n.
2. All the internal nodes present at level 0 through n-2 have two child nodes.
3. Leaf nodes present at level n are as far to the left as possible.

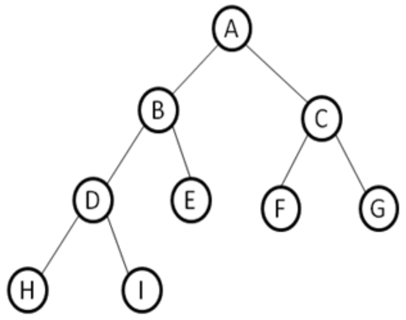


Fig : Complete binary tree

Strictly Binary Tree

A strictly binary tree is defined as binary tree in which every node other than leaf nodes has two children. It is also referred as full binary tree, proper binary tree or 2-tree.

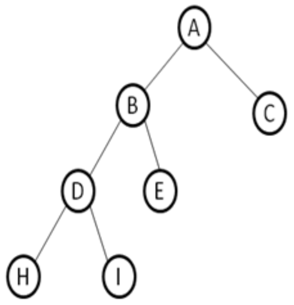


Fig : Strictly binary tree

Binary Search Tree

Binary search tree is defined as binary tree in which all elements in the right subtree of a node n are larger than or equal to the content of n and all elements in the left subtree of node n are less than the content of n. If a binary tree is traversed in inorder & content of each node are displayed as the node is visited, the elements get displayed in ascending order. This tree has major application in searching technique. If nodes are inserted in a tree as

50 24 63 11 43 75 9 18 70 88 then the binary tree produced is as follows

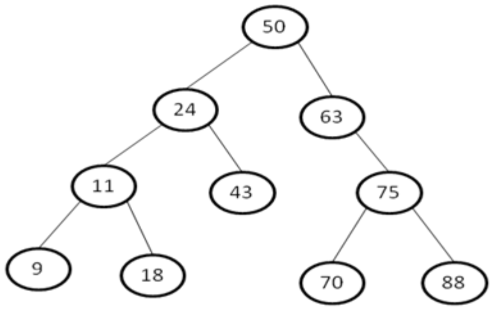


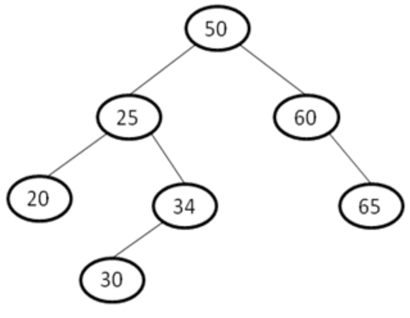
Fig : Binary search tree

Height balanced tree

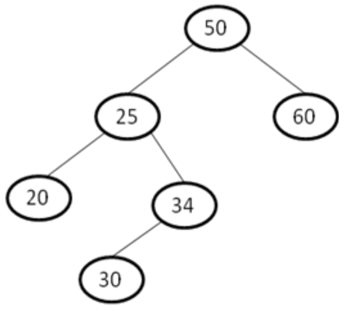
Height balanced tree is special type of binary tree in which heights of the left and right subtree of the node differ by at the most 1. In order to prevent a tree from becoming unbalanced we associate a ‘balance factor’ with each node of the height balanced tree. This indicator will contain one of the three values -1, 0 or 1. If it is other than these three values then the tree is not balanced .

- If the value of balance factor of any node is -1, then the height of the right sub-tree of that node is one more than the height of its left sub-tree.
- If the value of balance factor of any node is 0 then the height of its left and right sub-tree is exactly same.
- If the value of balance factor of any node is 1 then the height of the left sub-tree of that node is one more than the height of its right of sub-tree.

Searching in a binary search tree is efficient if the heights of both left and right sub-tree of any node are equal. So height balanced tree has major application in searching technique.
Follwing fig shows balanced and unbalanced tree.

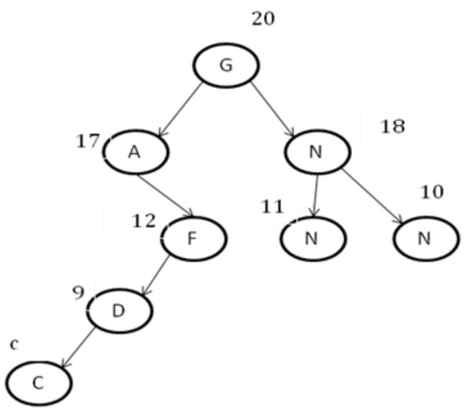


Height balanced tree



Unbalanced tree

Weight balanced tree
A weight-balanced binary tree is a binary tree which is balanced based on knowledge of the probabilities of searching for each individual node. Within each subtree, the node with the higest weight appears at the root. This can result in more efficient searching performance.



Weight balanced tree

In the diagram above, the letters represent node *values* and the numbers represent node *weights.* Values are used to order the tree, as in a general binary search tree. The weight may be thought of as probability or activity count associated with the node. In the diagram, the root is G because its weight is the greatest in the tree. The left subtree begins with A because,
out of all nodes with values that come before G, A has the highest weight. Similarly, N is the higest-weight node that comes before G.

Representation of A Binary Tree In Memory
Binary tree can be represented in two ways :

1. Sequential representation.
2. Linked representation.

Sequential Representation
Tree can be represented sequentially using array. When a binary tree is represented by arrays three separate arrays are required, One array stores the data fields of the trees. The other two arrays lc and rc represents the left and right child of the nodes.

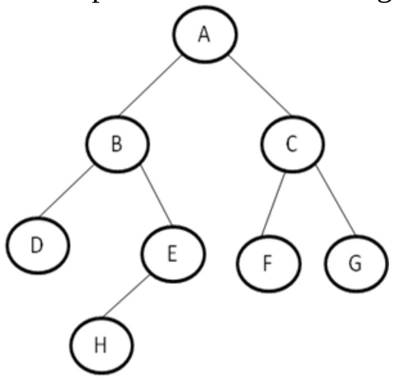
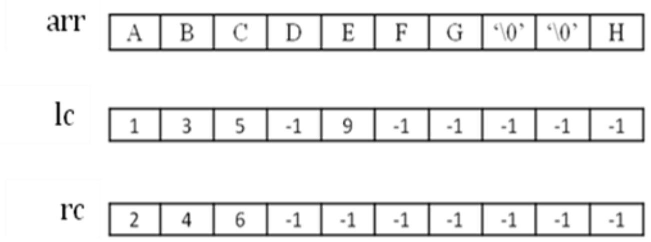


Fig : Binary tree

Above tree can be represented using array as follow



arr	A	B	C	D	E	F	G	'\0'	'\0'	H
lc	1	3	5	-1	9	-1	-1	-1	-1	-1
rc	2	4	6	-1	-1	-1	-1	-1	-1	-1

The array lc and rc contains the index of the array **arr** where the data is present. If the node does not have any left child or right child then the element of the array **lc** or **rc** contains value -1. The 0^{th} element of the array **arr** that contains the data is always the root node of the tree. Some elements of the array **arr** contain '\0' which represent an empty child.
Let us understand this with the help of an example. Suppose we want to find the left and right child of the node E. Then we need to find the value present at index 4 in array **lc** and **rc** since E is present at index 4 in the array **arr.** The value present at index 4 in the **lc** is 9, which is the index position of node H in the array **arr**. So the left child of the node E is

H. The right child of the node E is empty because the value present at index 4 in the array **rc** is -1.

Linked representation

Binary tree can be represented by links. Where each node contains the address of the left child and right child. These addresses are nothing but links to the left and right child respectively. A node that does not have a left or a right child contains a NULL value in its link field. Figure below shows the linked representation of a binary tree.

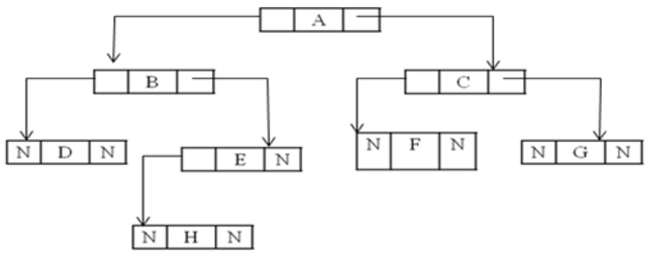


Fig : Linked representation of a binary tree

In figure above the link field of node C contain the address of the nodes F and G. The left field of node E contain the address of the node H. Similarly the right link contain a NULL value as there is only one child of E. The node D, F, G and H contain NULL value in both their link fields, as these are the leaf nodes.

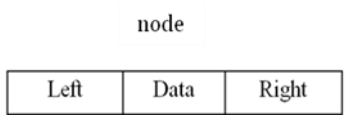
Operations on tree

Many operations can be performed on binary tree that are creation of tree, traversing tree, deletion of node from tree and searching an element in a tree. These are most basic operations
that are required to perform on tree to maintain a tree. Let us see these operation in more detail.

Creation of binary search tree

As we know the binary search tree has the property that all elements in the left subtree of a node *n* are less than the content of *n* and all the elements in the right subtree of *n* are greater than or equal to the content of *n*. While creating a tree we need to declare structure and creating pointer variable to perform further operations.

The structure of each node of a binary tree contains the data field, a pointer to the left child and a pointer to right child. Fig shows the structure.



The structure can be defined as follows

```
        struct  node
                {
                  struct node *left;
                  int              data;
                  struct node  *right;
                }
```

With so much background knowledge of binary search tree, we will directly see the algorithm for creation of binary search tree.

Algorithm to create binary search tree

```
1. Read key value and store it in newNode.
2. SET temp = ROOT
3. IF temp = NULL then
   Assign newNode to temp . Goto step 5.
4. IF key < temp → data
   SET temp = temp → left
   ELSE
   SET temp = temp → right
   Goto step 3.
5. Repeat step 1 to step 4 for all nodes.
6. STOP.
```

Traversal of A Binary Tree

Traversing a binary tree means visiting each node in the tree exactly once in a specific order. According to this order there are three types of traversals. All traversal algorithms are described with following diagram.

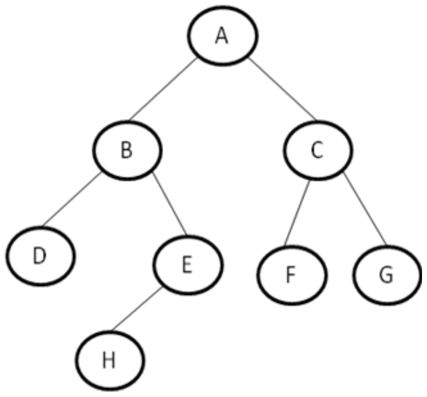


Fig : Binary tree

Preorder traversal

"Visit the node first then go to left subtree and then go to right subtree". The algorithm is

1. Visit the node
2. Traverse the left subtree in preorder.
3. Traverse the right subtee in preorder.

So when we traverse the above tree in preorder result is : **A B D E H C F G**

Following is the code to traverse a tree in preorder

```
void preorder (struct node *root )
  {
    if ( root != NULL )
     {
```

```
                printf( " %d ", root→ data) ;
                preorder(root→ left );
                preorder(root→ right );

            }

        }
```

Inorder traversal
The algorithm is

1. Traverse the left subtree in inorder.
2. Visit the node
3. Traverse the right subtee in inorder.

So when we traverse the above tree in inorder result is : **D B H E A F C G**

Following is the code to traverse a tree in inorder

```
   void inorder (struct node *root )
      {
        if ( root != NULL )
         {
              inorder(root  → left );
              printf( " %d ", root→data) ;
             inorder(root →right );
          }
      }
```

Postorder traversal
The algorithm is

1. Traverse the left subtree in inorder.
2. Traverse the right subtee in inorder.
3. Visit the node

So when we traverse the above tree in postorder result is : **D H E B F G C A**

Following is the code to traverse a tree in inorder

```
   void postorder (struct node *root )
      {
        if ( root != NULL )
         {
              postorder(root  →left );
              postorder(root→ right );
              printf( " %d ", root→data) ;
```

```
            }
        }
```

Searching node in binary search tree
A binary search tree is itself a special kind of binary tree. A binary tree is a tree which is either empty or consist of a node called the root, together with two children called the left subtree and the right subtree of the root. Each of these children is itself a binary tree.

Algorithm

1. Compare the given key value first to the root node of the tree.
2. If the value of the key is less than the value of the root node.
   If the left subtree is not empty, the search continues down the left subtree
   Else display the message data not found and quit.
3. If the value of the key is greater than the value of the root node.
   If the right subtree is not empty, the search continues down the right subtree
   Else display the message data not found and quit.
4. If the key value not exist in the tree, then terminate the procedure.

The code to search node in binary tree is as follows

```
struct node* search( struct node *root, int key)
{
   while(root != NULL)
       {
           if (root  data = = key)
              return n;
           if (root data > key)
              root = root →left ;
           else
               root = root→ right;
       }
    return NULL;
}
```

Depth First Search
In depth first search traversal we follow a path in the tree as deeply as we can go. When there no node, then we proceed backward go to right child of node, wherein we can repeat the process. Hence we need to examine the depths of the tree first. We can maintain stack to keeptrack of the visited node, so that this can help in backtracking. Following is the algorithm for depth first search.
Algorithm

1. START.

```
2. SET  temp = root.
3. IF  temp is not NULL
   PUSH it to STACK.
   ELSE goto step 5.
4. Make temp to point to left child. Goto step 3.
5. IF STACK is  EMPTY
   Goto step 9.
   ELSE
   POP the node from STACK
   Display its value on screen.
6. Make temp to point to node's right child and goto step 3.
7. STOP.
```

Breath First Search
In breath first search traversal we traverse tree acrros the breath. Tree nodes are displayed levelwise from left to right. To traverse tree levelwise we need to make use of queue.

Algorithm

```
1. START
2. SET temp = root.
3. Enqueue temp.
4. IF QUEUE is EMPTY goto step 9
   ELSE
   Dequeue node
5. Display data of node on screen.
6. IF node has left child, Enqueue it.
7. If  node has right child, Enqueue it.
8. Goto step 4.
9. STOP.
```

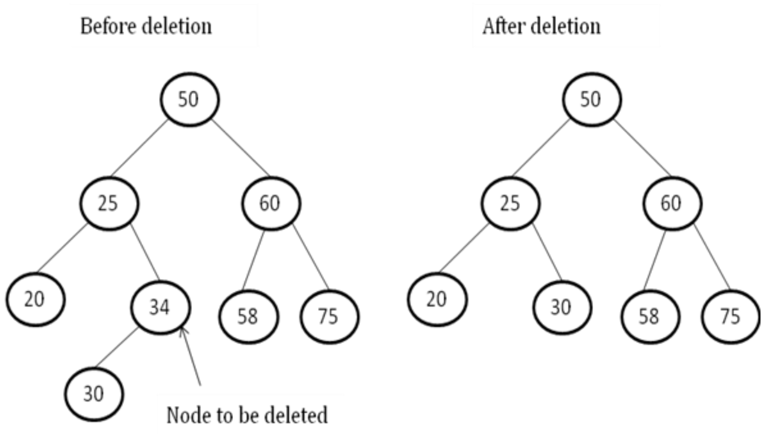
Deletion of a node from binary search tree
The algorithm to delete a node from a tree has a two distinct operations, searching and deletion. Once the node to be deleted has been determined by searching algorithm, it can be deleted from the tree. The algorithm must ensure that when the node is deleted from the tree, the ordering of the binary tree should be maintained. There are some special cases when the node is to be deleted from the tree is to be considered.
Special Cases

1. No node contains data to be deleted.
2. The node containing data having no children.
3. The node containing data having exactly one child.
4. The node containing data having two children.

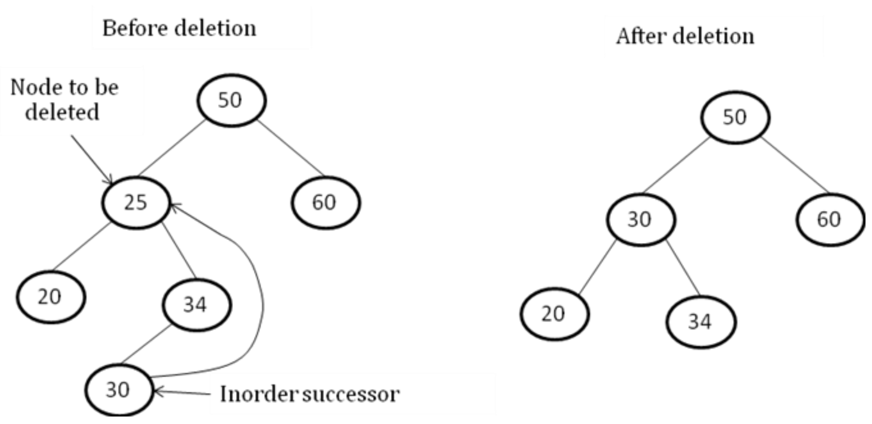
Case 1: In such a case we just need to print the message that data is not present in the tree.

Case 2: In such a case as node has no children we just delete the node and set the left link or right link of parent of this node to NULL. If node is left child of parent node then set left link of parent to NULL else set right link to NULL.

Case 3 : In such a case as node to be deleted has one child we need to adjust link of parent node such that after deleting it parent node link point to child of the node being deleted.



Case 4 : In such a case as node to be deleted as two children it is more complex than the previous cases because the order of the tree must be maintained. In this case it is important to consider which node should be used in place of the node to be deleted. Use the inorder successor from right subtree of the node to be deleted.

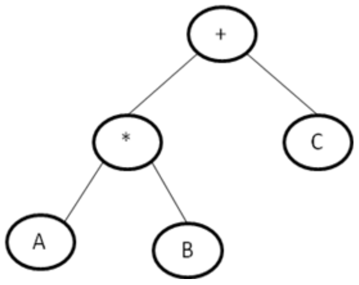


Application of binary tree

There are many applications of a binary tree. As we already know binary search tree is used in searching technique. Binary can also be used to represent expressions. Let us delve in expression tree.

Expression tree

The arithmetic expression represented as binary trees are known as expression trees. In expression tree root node contains an operator that is to be applied to the result of evaluation of the expressions represented by the left and right subtree. A node representing an operator is nonleaf, where as node representing an operand is a leaf. In case of unary operator the left child is absent and right child is the operand. For example expression A*B+C can be represented as follows.



When the expression tree is traversed in pre-order then prefix form of expression is obtained. Similarly, when the expression tree is traversed in post-order then the postfix form of the expression is obtained.
Let us see how to convert an expression into binary expression tree stepwise.

(A-B) * (C-D) + (F-G)

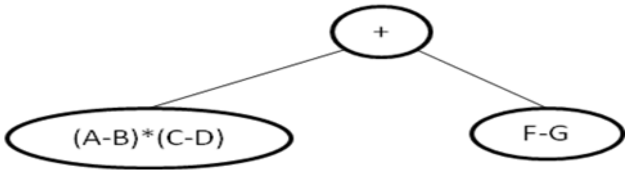
Step 1 : Read preorder of expression, first element becomes root.

Step 2 : Now scan the infix expression, till you get an element found in step1. Place all the elements left of this element(of infix expression) to the left of root and others to right.

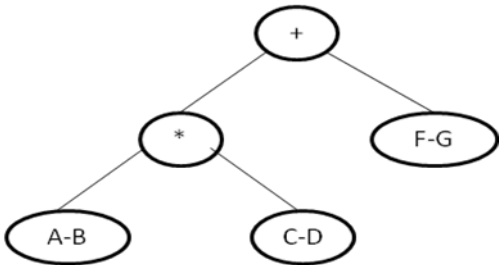
Step3: Repeat step 2 and 3 till all the elements from infix sequence gets placed in tree.

INFIX   : (A-B) * (C-D) + (F-G)
PREFIX   : + * - AB-CD-FG

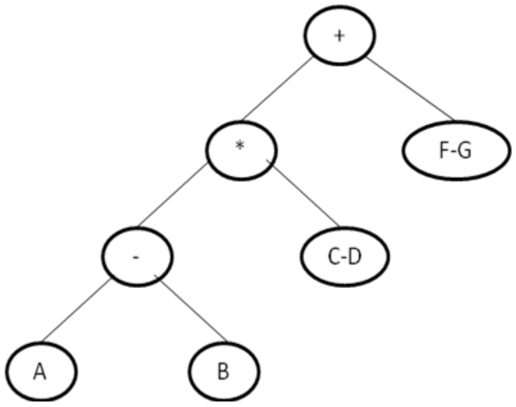
Place all elements left of + from infix expression to root's left and others to the right.



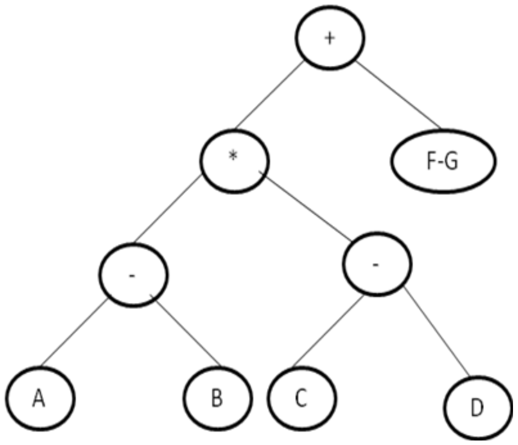
Now scan the next element from prefix i.e. *, which becomes root for left subtree.



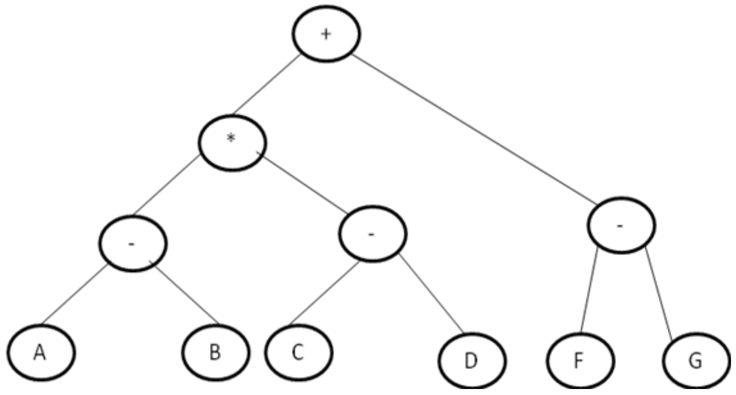
Now scan the next element from prefix i.e. -, which becomes root for left subtree.



Now scan next element from prefix i.e. -, which becomes root for right subtree of *.



Now scan next element from prefix i.e. -, which becomes root for right subtree.

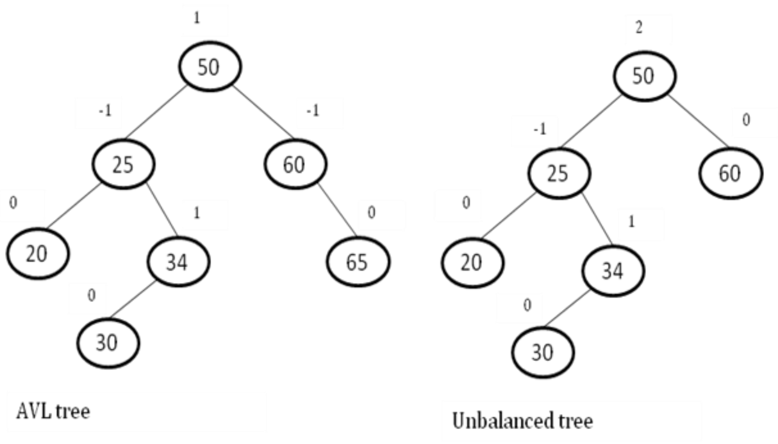


AVL Tree

Searching in a binary search tree is efficient if the heights of both left and right sub-tree of any node are equal. The height of the binary tree is the length of path from the root to deepest node in the tree. An AVL tree is a binary search tree in which the difference between the height of the right and left subtrees is never more than one. However, frequent insertion and deletion in tree make it unbalanced. In order to prevent a tree from becoming unbalanced we associate a 'balance factor' with each node of the height balanced tree. This indicator will contain one of the three values -1, 0 or 1. If it is other than these three values then the tree is not balanced .

- If the value of balance factor of any node is -1, then the height of the right sub-tree of that node is one more than the height of its left sub-tree.
- If the value of balance factor of any node is 0 then the height of its left and right sub-tree is exactly same.
- If the value of balance factor of any node is 1 then the height of the left sub-tree of that node is one more than the height of its right of sub-tree.

Note that a balance factor of -1 means that the left-subtree is heavy, a balance factor of +1 means that the right-subtree is heavy and a balance factor of 0 means that left-subtree and right-subtree are having same height.



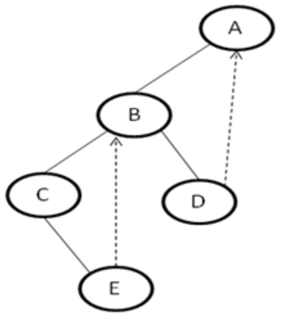
A node with any other balance factor than 1,0,-1 is considered unbalanced and requires rebalancing the tree. The balance factor is either stored directly at each node or computed from the height of the subtrees. Whenever we insert or delete an item, the AVL tree can be “violated”. We must then restore it by performing a set of manipulation called “rotations” on the tree. These rotation are having two types : single rotation and double rotations.

Threaded Binary Tree

Binary tree has following drawbacks

1. A binary tree has more NULL links(N+1) than actrual pointers(N-1) i.e. 2N total links.
2. There is no way of back tracking.(i.e. we can not traverse the node above the node to which we have reached.)

If we make use of these NULL links the we can remove all the drawbacks of a binary tree. To make use of these links, the NULL links can be replaced by pointers, called threads to other nodes in the tree. If right child of any node p is NULL then pointer to this right child will be replaced by a pointer to the inorder successor of the node. A NULL leftchild link at p is replaced by a pointer to the inorder predecessor of the node.

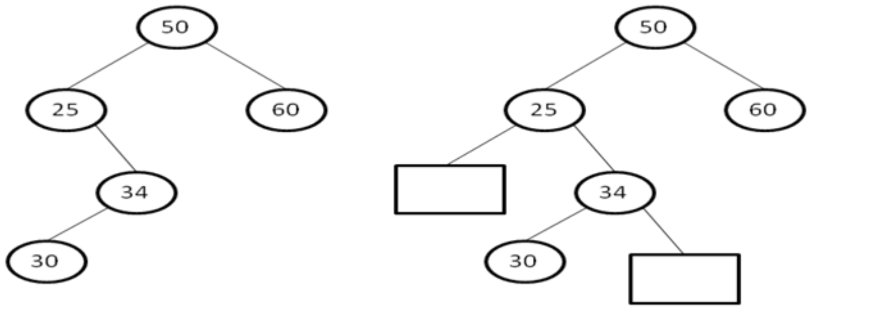


Threaded binary tree

Threads suffers from one problem that we need to know whether a pointer is a normal pointer to a child or a thread that points back to an in-order successor or in-order predecessor of the node. One solution to this problem is to add to the data in each tree node two fields that indicate whether the left and right pointers in that node are normal pointers or threads. So we need to maintain two boolean variables left and right. The variable right is true if the right pointer is a thread and false if it is pointer to right child. Likewise, the variable left is true if the left pointer is thread and false if it is a pointer to left child. If we add these to boolean variables to each tree node, we would make the following structure declaration for a node.

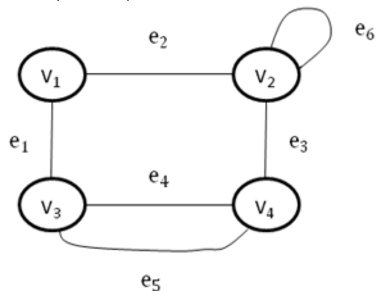
```
struct thtree
  {
   enum boolean  left;
    struct thtree     *lc;
    int data;
    struct thtree    *rc;
   enum boolean  right;
}
```

Extended Binary Tree
A binary tree can be converted to an extended binary tree by adding new nodes to its leaf nodes and to the nodes that have only one child. These new nodes are added in such a way that all the nodes in the resultant tree have either zero or two children. The extended tree is also known as a 2-tree. The nodes of the original tree are called internal nodes and the new nodes that are added to binary tree, to make it an extended binary tree are called external nodes.

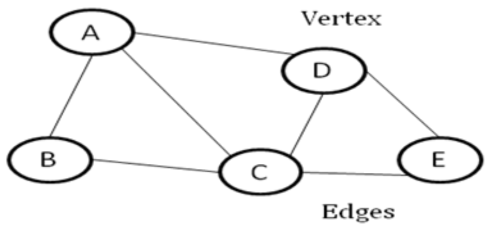


Extended binary tree

Graph
Introduction
Trees provide a very useful way of representing relationship in which a hierarchy exists. If we remove the restriction that each node may be pointed by atmost one node, we come across another data structure called graphs. Graph is used in many areas like Networking, Geographical information system(GIS) for finding the route between the source and destination etc.
Defination : A graph consist of two sets V & E. Set V is a finite, nonempty set of vertices & E is a set of edges that link vertices.
G = (V, E)



Set of vertices = { v1, v2, v3, v4 }
Set of edges = { e1, e2, e3, e4, e5, e6 }
Basic Terminology



Graph G

Adjacent vertices

Vertex $\mathbf{v_1}$ is said to be adjacent to vertex $\mathbf{v_2}$ if there exist an edge (v_1,v_2) or(v_2,v_1). In above figure vertex adjacent to D are A, C, E.

Path

A path from verted v is a sequence of vertices, each adjacent to the next. A simple path is a path in which each vertex is distinct. Path A-C-D-E is simple path.

Cycle

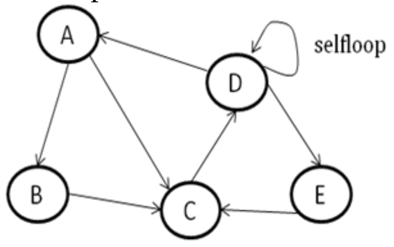
A cycle is a path that begins and ends at the same vertex. In other words it's a path from u to u. In above example A-C-B-A is a cycle.

Subgraphs

A subgraph of (V,E) is a pair ($V^{|}$,$E^{|}$), where $V^{|}$ is a subset of V, and $E^{|}$ consists of all edges (u,v) of E between two nodes of $V^{|}$. For example in above figure, subgraph for A,B, and C has three edges.

Directed graphs

A directed graph is one in which every edge (u,v) has a direction, so that edge(u,v) is different from edge(v,u). Directed graphs are often called *digraphs*. We denote an edge from vertex **u** to vertex **v** in a digraph ~~by u~~$\rightarrow$ u. Formally, the edges in a directed graph are ordered pairs of vertices rather than sets of two vertices. One can also allow *selfloops,* edges with both endpoints at one vertex. Here is an example of a graph with selfloops



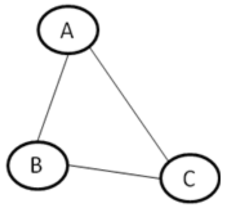
Directed graph

Undirected graph

If edges of the graph has no direction then it is called undirected graph. In undirected graph there is no distinction between edge (u,v) and edge (v,u). So in undirected graph the order of pair of vertices is unimportant.

Complete graph

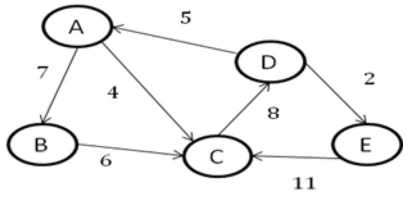
A graph is said to be complete or fully connected if there is an edge from every vertex to every other vertex in a graph. A complete graph with *n* vertices will have *n(n-1)* edges.



Complete graph

Weighted graph

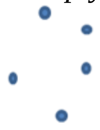
A graph is said to be weighted graph if every edge in the graph is assigned some weight or value. The weight of the edge is a non-negative value that may be representing the cost of moving along the edge or distance between the vertices.



Weighted graph

Empty Graph

The *empty graph* has no edges at all. Here is the empty graph on 5 vertices:

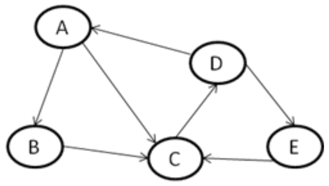


Empty graph

Degree

The degree of any vertex in undirected graph is number of adjacent node to the vertex.

In directed graph, the vertex is having two types of degrees i.e. indegree and outdegree.



directed graph

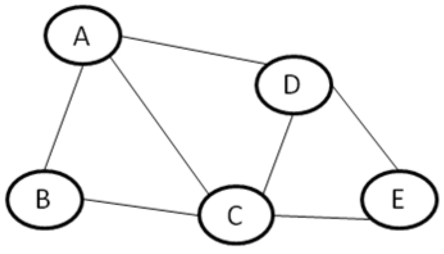
1. Indegree : Indegree of vertex can be defined as number of incident edges on that vertex from other vertices. Indegree of vertex D in above graph is 1.
2. Outdegree: Outdegree of vertex can be defined as number of out going edges from vertex to other vertex. Outdegree of vertex D in above graph is 2.

Representation of Graph

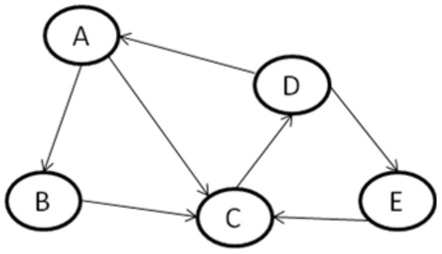
There are two ways to represent a graph in computer memory

1. Sequential representation of graph using adjacency matrix.
2. Linked representation of graph using adjacency list.

Adjacency matrix
Adjacency matrix is representation of both directed and undirected graph using two dimensional array of n$\times$ n elements where n is the number of vertices. This array has property that a[i][j] = 1 if the edge (v_i , v_j) is in the set of edges, and a[i][j] = 0 if there is no such edge.



G1



G2

The adjacency matrix for undirected graph G1 is as shown below

	A	B	C	D	E
A	0	1	1	1	0
B	1	0	1	0	0
C	1	1	0	1	1
D	1	0	1	0	1
E	0	0	1	1	0

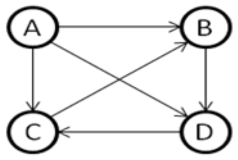
The adjacency matrix for directed graph G2 is as shown below

	A	B	C	D	E
A	0	1	1	0	0
B	0	0	1	0	0
C	0	0	0	1	0
D	1	0	0	0	1
E	0	0	1	0	0

As seen from above, the adjacency matrix for an undirected graph is symmetric. The adjacency matrix for a directed graph need not be symmetric. The space needed to represent a graph using its adjacency matrix is n^2 locations. About half this space can be saved in the case of undirected graphs by storing only the upper or lower triangle of the matrix.
Path Matrix
Adjacency matrix does not give any idea about existence of path between two vertices of graph through intermediate nodes if exists. Path matrix shows the existence of path within the diagraph using boolean operators and matrices.
Let G be a simple directed graph with *n* nodes V_1, V_2,- - - V_N. The path matrix or reachability matrix of G is the n-square matrix.

$$P_{ij} = \begin{cases} 1 \text{ if there is path from } \mathbf{V_i} \text{ to } \mathbf{V_j} \\ 0 \text{ otherwise} \end{cases}$$

In adjacency matrix A $_{\mathbf{i\,j}}$ **=** 1 if there is exist edge from V_i to V_j otherwise 0. But in path matrix A $_{\mathbf{i\,j}}$ **=** 1 if there is exist path from V_i to V_j through any number of intermediate nodes otherwise 0.
The path matrix for graph G with n vertices is as follows
$B_n = A + A^2 + - - - + A^n$.
Following example shows how to create path matrics using powers of the adjacency matrix.



$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 1 & 1 & 1 \\ B & 0 & 0 & 0 & 1 \\ C & 0 & 1 & 0 & 0 \\ D & 0 & 0 & 1 & 0 \end{array}$$

Adjacency matrix for graph

$$A^1 = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$A^2 = A\,.\,A = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$A^2 = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

$$A^3 = A^2\,.\,A = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$A^3 = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A^4 = A^3 . A = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$A^4 = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

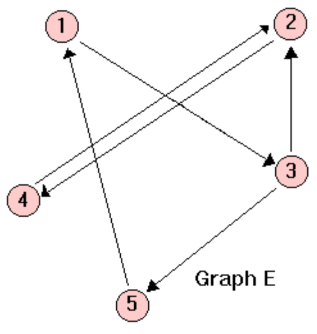
$$B_4 = A^1 + A^2 + A^3 + A^4$$

$$B_4 = \begin{bmatrix} 0 & 4 & 4 & 4 \\ 0 & 1 & 1 & 2 \\ 0 & 2 & 1 & 1 \\ 0 & 1 & 2 & 1 \end{bmatrix}$$

Path Matrix is
$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix}$$

Warshall's Algorithm

In it's simplest form, *Warshall's Algorithm* is used to compute the existence of paths within a digraph using Boolean operators and matrices. Begin by creating an adjacency matrix **A** for Graph as before, with one notable difference - instead of using weights we will use Boolean operators. That is to say, if there is a path, enter a 1 in matrix **A**, and enter 0 if no path exists.



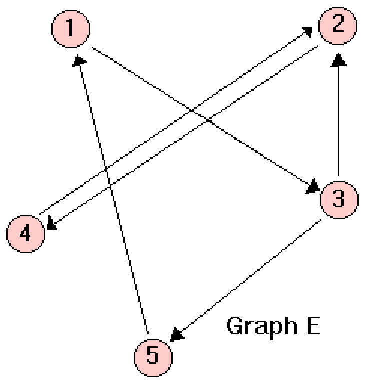
A	1	2	3	4	5
1	0	0	1	0	0
2	0	0	0	1	0
3	0	1	0	0	1
4	0	1	0	0	0
5	1	0	0	0	0

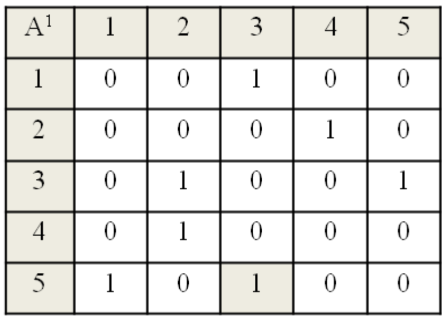
This matrix tells us whether or not there is a path ***p*** of length 1 between two adjacent nodes. Building upon matrix **A**, we will create a new matrix $\mathbf{A^1}$, for which we will choose one vertex to act as a *pivot* -- an intermediate point between 2 other vertices. Initially, we will chose vertex 1 as our pivot for $\mathbf{A^1}$. The value we are seeking is that of $p^{(1)}_{i\,j}$. For vertices v_i and v_j, $p^{(1)}_{i\,j}$ is one of the following:

1, if there exists an edge between vertices $\boldsymbol{v}_i$ and $\boldsymbol{v}_j$, or if there is a path of length $\geq$ 2 from $\boldsymbol{v}_i$ to $\boldsymbol{v}_1$ and from $\boldsymbol{v}_1$ to $\boldsymbol{v}_j$; else

0, if there is no path.

Begin by scanning *column 1* of matrix **A**; the only $\boldsymbol{v}_i$ which connects to $\boldsymbol{v}_1$ is vertex 5. Now scan *row 1*; the only path from $\boldsymbol{v}_1$ to $\boldsymbol{v}_j$ is to vertex 3. Since we have established that a path of length 2 lies between $\boldsymbol{v}_5$ and $\boldsymbol{v}_3$, we update matrix $\mathbf{A}^1$ accordingly.





A^1	1	2	3	4	5
1	0	0	1	0	0
2	0	0	0	1	0
3	0	1	0	0	1
4	0	1	0	0	0
5	1	0	1	0	0

MATRIX $\mathbf{A}^2$

Next create matrix $\mathbf{A}^2$, using vertex 2 as the pivot point.
Begin by scanning *column 2* of matrix **A**; the $\boldsymbol{v}_i$ which connect to $\boldsymbol{v}_2$ are vertices 3 and 4.
Now scan *row 2*; only 1 path from $\boldsymbol{v}_2$ exists to $\boldsymbol{v}_j$ = vertex 4.
We have now established paths between the following vertices:

$\boldsymbol{v}_3$ to $\boldsymbol{v}_4$ and $\boldsymbol{v}_4$ to $\boldsymbol{v}_4$.

Newly added paths have been highlighted in gray. Notice that each new path created is being built upon previously existing paths.

A^2	1	2	3	4	5
1	0	0	1	0	0
2	0	0	0	1	0
3	0	1	0	1	1
4	0	1	0	1	0
5	1	0	0	0	0

MATRIX $\mathbf{A}^3$
Matrix $\mathbf{A}^3$ will use vertex 3 as the pivot point.
As before, scan column 3 to see which vertices $\boldsymbol{v}_i$ connect to $\boldsymbol{v}_3$. In this case, vertices 1 and 5 have a path to 3.
Now, scanning row 3, $\boldsymbol{v}_3$ connects to vertices 2, 4, 5. We have now established paths between the following:

V_1 to V_2	*and*	*V_5 to V_2*
V_1 to V_4		*V_5 to V_4*
V_1 to V_5		*V_5 to V_5*

A^3	1	2	3	4	5
1	0	1	1	1	1
2	0	0	0	1	0
3	0	1	0	1	1
4	0	1	0	1	0
5	1	1	1	1	1

MATRICES A^4 and A^5

For $\mathbf{A}^4$, first scan column 4. At this point, all vertices now have a path to vertex 4. Scanning row 4, we see that 4 has a path only to vertex 2, indicating that all vertices have a path to 2. However, the only vertex which doesn't already have a path to vertex 2 is 2 itself, so we update the matrix accordingly.

A^4	1	2	3	4	5
1	0	1	1	1	1
2	0	1	0	1	0
3	0	1	0	1	1
4	0	1	0	1	0
5	1	1	1	1	1

Completing this process, scan column 5 to see that vertices 1, 3 and 5 all have paths to vertex 5. Scanning row 5 indicates that 5 has a path to all other vertices. Consequently, we add 1's to rows 1, 3 and 5 to reflect that vertices 1, 3 and 5 have paths to all other vertices.

A^5	1	2	3	4	5
1	1	1	1	1	1
2	0	1	0	1	0
3	1	1	1	1	1
4	0	1	0	1	0
5	1	1	1	1	1

This completes the path matrix for Graph E.

Warshall's Algorithm

Given the adjacency matrix A of *n* x *n* size this algorithm produces path matrix P.

```
1. [ initialize]
   P $\leftarrow$ A
```

```
2. [ perform a pass ]
   Repeat through step 3 for K = 1- - - n.
3. [ process rows ]
   Repeat through step 4 for I = 1,- - - n.
4. [ process column ]
   Repeat for j = 1,- - - n.
   Set P[ I, J ] = P[ I, J] OR ( P[ I, K ] AND P[ K, J] )
5. EXIT.
```

Adjacency List

Adjacency list is representation of both directed and undirected graph using an array of n lists of vertices. List i contains vertex j if there is an edge from vertex i to vertex j. An undirected graph may be represented by having vertex j in the list of vertex i and vertex i in the list of vertex j.

The adjacency list representation typically uses less space than the adjacency matrix.

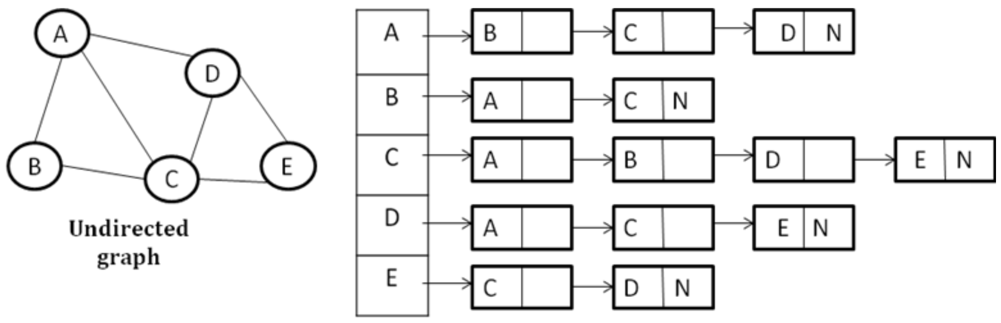


Fig : Adjacency list

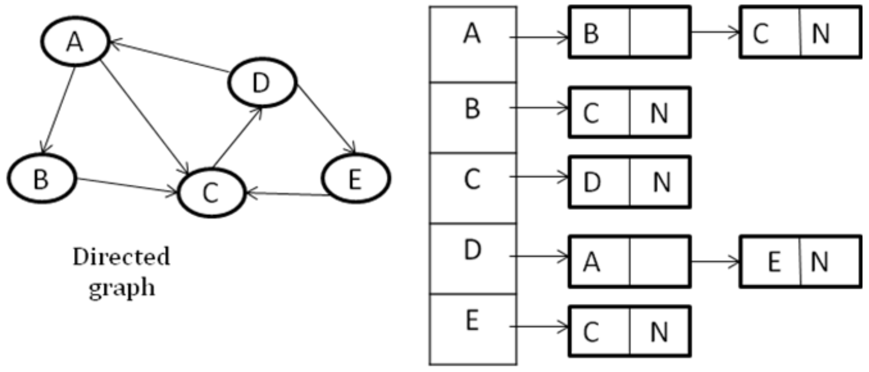


Fig : Adjacency list

Traversal of Graph

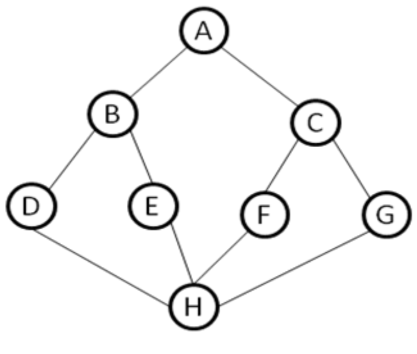
The traversal of graph means to visit the vertices of graph in some systematic order. There are two ways to traverse graph : Depth first search (DFS) and Breath First Search.

Depth First Search

In depth first search traversal we follow a path in the graph as deeply as we can go. When there no adjacent, non-visited nodes, then we proceed backward, wherein we can repeat the process. Hence we need to examine the depths of the graph first. We can maintain stack to keeptrack of the visited node, so that this can help in backtracking. Following is the algorithm for depth first search.

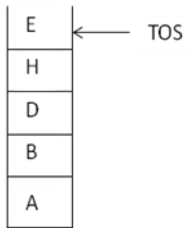
Algorithm for depth first search

1. Select any unvisited node in the graph. Mark this node as visited and then push it to the stack.
2. Find the adjacent node to the node on the top of the stack and which is not yet visited. Mark this new node as visited and push it to the stack.
3. Repeat step 2 till no new adjacent node to the top of the stack node can be found, pop the top of the stack.
4. Repeat step 2 and 3 till stack becomes empty.
5. Stop.



GRAPH

1. PUSH *A B D H E* nodes onto the stack and mark them visited

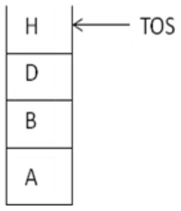


Visited = [*A B D H E*]

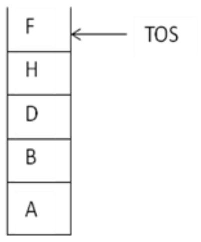
Starting with node A we pushed nodes onto the stack which are adjacent, non – visited.

2. Now, Backtrack

   Pop TOS i.e. E, as it has no unvisited adjacent node.

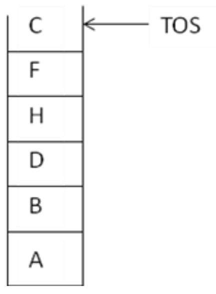


3. POP H, PUSH H and its adjacent F , visit F.



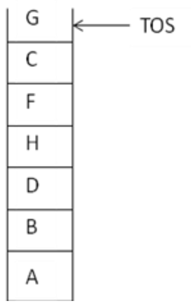
Visited = [*A B D H E F*]

4. POP F, PUSH F and its adjacent C, visit C.



Visited = [*A B D H E F C*]

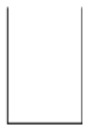
5. POP C, PUSH C and its adjacent G, visit G.



Visited = [*A B D H E F C G*]

6. POP G,C,F,H,D,B,A as they have no adjacent unvisited node.

Empty STACK



So, the visited nodes are

Visited = [*A B D H E F C G*]

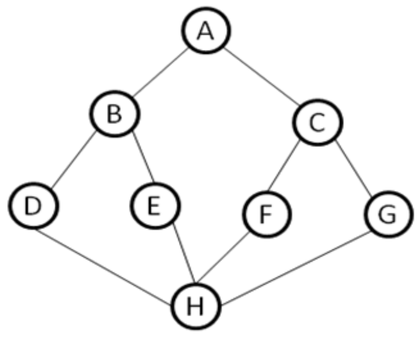
Breath First Search

In this method we visit the nodes across the breath of the graph, before going to the next leve. For breath first search we use queue. Here we first visit the node and place its unvisited adjacent nodes in the queue. In the next step deque the node from front of the queue and visit its adjacent unvisited node and place them into the queue. We go on dequing node from the queue and visit unvisited node till queue becomes empty. When queue becomes empty we stop searching.

Algorithm for breath first search

1. Select any unvisited node, visit it and place it into the queue.

2. Deque the node which is present at front of the queue. Visit unvisited adjacent node to the dequed node and place them into the queue.
3. Repeat step 2 till the queue becomes empty.
4. Stop.



GRAPH

1. Visit node A and place it into the queue.
Visited = [*A*]

queue : A

2. Deque : A
   Visit : B C
   Add : B C

queue : B , C

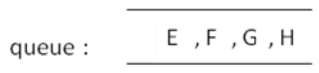
Visited = [*A B C*]
3. Deque : B
   Visit : D E
   Add : D E

queue : C , D , E

Visited = [*A B C D E*]
4. Deque : C
   Visit : F G
   Add : F G

queue : D , E , F , G

Visited = [*A B C D E F G*]
5. Deque : D
   Visit : H
   Add : H

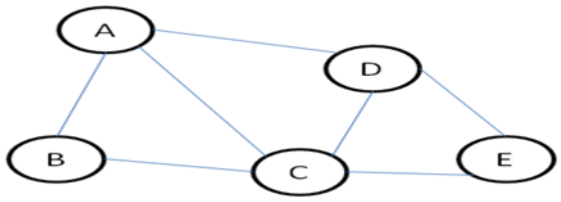


Visited = [*A B C D E F G H*]
6. Deque E, F, G, H they have no adjacent unvisited node.
Empty queue

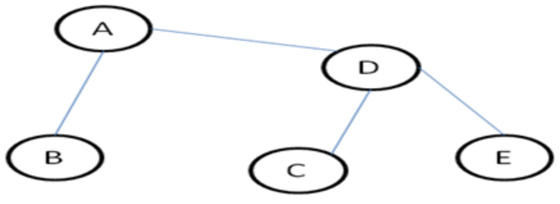
So, the visited nodes are
Visited = [*A B C D E F G H*]

Spanning Trees

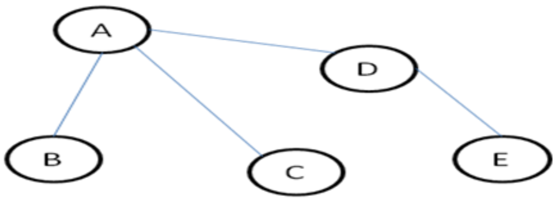
In mathematical field of graph theory, a spanning tree T of a connected undirected graph G is a tree composed of all the vertices and some of the edges of G. Informally, a spanning tree of G is a selection of edges of G that form a tree spanning every vertex of G. That is, every vertex lies in the tree, but no cycle (or loops) are formed.



Graph



Spanning tree 1



Spanning tree 2

A spanning tree of a connected graph G can also be defined as a maximal set of edges of G that contains no cycle, or minimal set of edges that connect all vertices.
In certain fields of graph theory it is often useful to find a minimal spanning tree of a weighted graph.

Shortest Path Algorithm

A minimal spanning tree gives no idea about shortest path possible between two vertices. For given source vertex in the graph shortest path algorithm finds the path with lowest cost between that vertex and every other vertex. It can also be used for finding cost of shortest paths from single vertex to a single destination vertex by stopping the algorithm once the shortest path to the destination vertex has been determined. For example, if the vertices of the graph represents cities and edges path costs represents driving distance between pairs of cities connected by a direct road, the following algorithm can be used to find the shortest route between one city and all other cities.

Algorithm

A weighted graph G with N node is maintained in memory by its weight matrix W. This algorithm finds a matrix P such that P[I , J] is length of a shortest path from node V_I to V_J .

```
1. Repeat for I, J = 1, 2 . . . N
   IF W[ I, J ] = 0 THEN
       SET  P[ I, J ] = INFINITY
   ELSE
       SET P[ I, J ] = W[ I, J ]
   [ END LOOP ]
2. Repeat for step 3 for 4 for K = 1, 2, . . . N
3. Repeat step 4 for I = 1, 2, . . . N
4. Repeat for J = 1, 2, . . . N
   SET P[ I, J ] = MIN { P[ I, J ] , P[ I, J ] + P[ K, J ]}
   [ END LOOP ]
   [ END of step 3 LOOP]
   [ END of step 2 LOOP]
5. Exit.
```

Application of Graph

1. Graph are used to represent Networks of cities, computers etc.
2. Graph is used to represent geographical database of Maps.
3. Graph is used to prepare Graphics : Geometrical Objects.
4. Graph is used in analysis of electrical circuits.
5. Graph is used to find route between two locations.
6. Graph is used to find shortest path in different applications.

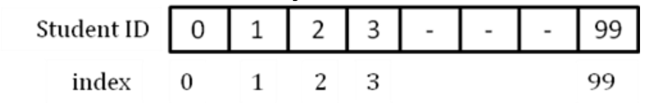
Hashing

Concept of Hashing

Hashing is a method to support constan time insertion, deletion and search in a table on an average i.e. data manipulation takes place with an efficiency of O(1). Any operation like searching a particular element should be happened in single attempt only.

Suppose we want to store information about student in an array, so that we can later retrieve information about any student. Simply information can be stored and retrieved using student ID. If student ID's are in the range 0 to 99, we can store the records in an following array, placing student ID k in location data[k]

int data[100] // array of 100 records



The record for student ID k can be retrieved immediately i.e. in a single attempt since we know it is at position data[k]. Hence such operations will be having efficiency O(1).

For hashing, every element requires a unique key which indicates the position or index of element in an array. The basic idea of hashing is not to search the correct position of an element with comparisons but to compute the position of an element within the array.

Hash Table

A hash table is a data structure that supports constant time insertion,deletion, and search on an average.

For example, a hash table for a dictionary contains all the alphabet and the page number where each letter is stored. Hence, instead of searching from start to end for a word, we can use the hash table to get right page number for a word to be searched. Hence *hash table* is a storage which provides constant time insertion, deletion and search on an average.

Some facts about hash table

- degenerative performance is possible, but unlikely
- it may waste some storage
- iteration order is not defined (and may even change over time)
- idea: data items are stored in a table, based on a key
- the key is mapped to an index in the table, where the data is stored/accessed

Hash Function

A function that maps the keys to indices of a hash table is called *hash function*. The hash function must be chosen so that it return value that is always a valid index for the hash table. A hash function is usually a combination to two maps, a hash code map and a compression map.

Hash code map maps the keys to an integer.

Hash code map : key -> integer

If keys are already an integer then there is no need to convert them into an integer. When keys are not integer then we need to convert them into an integer.

Compression map : Integer value generated from hash code map can be from arbitrary range. We need to bring it to the range of our size of the hash table to make it valid index of the hash table. If N is the size of the table then we need to bring it to the range 0 to N-1 so that it can be mapped to an index of the hash table. This map is called compression map.

Compression map : integer -> [0, N-1]

A good hash functions are very rare. Follwing are some characteristics of good hash function.

1. It should be as uncomplicated as possible and fast to compute
   e.g., a single mathematical or bitwise operation.
2. It should scatter the data evenly throughout the hash table .
3. It should avoid collisions.collisions are unavoidable except for the case of perfect hashing but they should be minimized.
3. The calculation of the index should involve the entire search key.
4. If the hash function uses modulo arithmetic then the base should be

prime

• e.g., index = key MODULO <base>

Perfect hash function

When hash function maps every key to a unique location in the hash table then this function is called perfect hash function. Typically it is not possible because all keys are not known in advance.

Some examples of hash functions

1. **Mid Square**

   The mid square hash function computes the square of the key and then use appropriate number of bits from the middle of the squared value to obtain an address of the bucket in hash table. The number of bits used to obtain the bucket address depends on the table size.

   Example

   Suppose key is 15 then hash function computes the square of the key i.e. 225 then take number of bits from the middle of the square value, in our case is 25. Thus 25 is hash address of key 15.

2. **Selecting Digit**

   This hash function selects portion of the key to use it as the address of the bucket in hash table. This function works only for keys that are positive integers.

   Example

   Suppose key is 403-210-9455 then hash function select the even position digits starting with the4th digit. Thus hash function generate the address 2045.

   In this mehod mapping of key to index is fast but it usually does not evenly distribute keys through the hash table

3. **Folding Method**

   To increase the size of the hash table (and increase the range of possible values generated by the hash function) groups of numbers can be added instead of individual numbers.

   Example

   Suppose key is 403-210-9455 then hash function add up the group of numbers from key and generate the hash address as 4 + 032 + 109 + 455 = 600. Here 600 is hash address for key 403-210-9455

   Other examples of hashing algorithms that employ folding could combine selecting certain digits to be "folded" into a key e.g. sum only the odd positioned digits.

   Other mathematical/bitwise operations could be employed e.g. multiplying digits together, bit shifting or bit rotating the numerical values.

   The quality of the hash function using folding will vary.

4. **Division**

   In this method we use the modulo operator. Here we divide the key by table size and generate the hash address. If N is the size of the table then hash function is as follow *index = (key) modulo N*

Generally N is chosen to be prime to ensure an even distribution of keys to the different parts of the table. The reason for it has to do with number theoy.

Example

Suppose key is 1250 and size of the table is 100 then

Hash function : 1250 modulo 100 = 50

5. **Length dependent Method**

In this method length of the key is taken into consideration. Hash function use length
of the key and some portion of the key to generate addresses. In this mehod mapping
of key to index is fast but it usually does not evenly distribute keys through the hash
table.

Some Terms related to Hashing

1. **Hash address**

   The index generated by hash function after mapping the key is called hash address of that key.

2. **Bucket**

   Hash table is partitioned into number of slots to hold the keys. These slots are called bucket. Bucket either contains actual key or an reference pointer to the list of keys.

3. **Synonym**

   When two identifiers are mapped to the same location in hash table by hash function then these two identifiers are said to be synonym. Distinct synonym can be entered into the same bucket as long as slots in the bucket are not used.

4. **Collision**

   A collision is said to occur when two non identical identifiers are hashed into the same bucket. When bucket size is 1, collision and overflow occurs simultaneously. Suppose we
   are having student ID 300 . The record with student ID 300 will be stored in data[3] as before, but student ID 399 will also be placed at data[3] as student ID 399 will be hashed
   by the function to data[399/100] = data[3]. So there are two different elements which belongs to same location. This situation is known as collision.

5. **Overflow**

   When we try to insert a record onto a full bucket overflow condition occurs. When bucket
   size is 1, collision and overflow occurs simultaneously.

6. **Load Factor**

The load factor (ά) is ratio of total number of identifiers to table size.

LF = ά = (Total number of identifiers) / (Table size)

Generally the load factor should be kept below 0.5 – 0.67 for closed hashing and for open addressing it should be kept around 1.0.

Overflow Handling

When situation of overflow occurs it should be resolved and the record must be placed somewhere else in the table. Let see some of the methods used for overflow resolution.

Overflow chaining

Despite allocation of a few more buckets than required, bucket overflow can still occur. We handle bucket overflow using Overflow Bucket. If a record has to be inserted into a bucket b, and b is already full, an overflow bucket is provided for b, and the record is inserted into the overflow bucket. If the overflow bucket is also full, another overflow bucket is provided and so on. All the overflow buckets of a given bucket are chained together in a linked list. Overflow handling using such a linked list is called Overflow Chaining.

Hash table is considered as an array of pointers to the record, that is, an array of linked list.

Advantages of Chaining

1. **Space Saving** Since the hash table is a contiguous array, enough space must be set side at compilation time to avoid overflow. On the other hand, if the hash table contains only pointers to the records, then the size of the hash table may be reduced.
2. **Collision Resolution** Chaining allows simple and efficient collision handling mechanism. To handle collision only a link field is to be added.
3. **Overflow** It is no longer necessary that the size of the hash table exceeds the number of records.

Deletion Deletion proceeds in exactly the same way as deletion from a simple linked list. So in a chained hash table deletion becomes a quick and easy task.

Open Hashing

In open hashing, the set of bucket is fixed and there is no overflow chains, instead if a bucket is full, records are inserted in some other bucket in the initial set of buckets B. Two policies are used when assigning a new bucket to a record.

Linear Probing

One policy is to use the next bucket in the hash table. This policy is called "linear probing". It starts with a hash address(the location where the collision occurred) and does a sequential search through a table for an empty location. Hence, this method searches in straight line, and it is therefore called linear probing. The table should be considered circular, so that when last location is reached, the search proceeds to the first location of the table.

Rehashing

An other policy, in which we compute another hash function for a new record, is used. If this positon is filled then some other method is needed to get another position and so on.

Clustering

The major drawback of linear probing is that as the table becomes about half full, there is a tendency towards clustering, that is records start to appear in long strings of adjacent positons with gaps between strings.

Searching and Sorting

Searching

Many times we need to find a record with the help of key. Key is the unique value associated with each record which distinguishes records from each other. Searching is operation that finds the given key value in the list of elements. Searching algorithm accepts two arguments the key and list in which key is to be found. There are two standard algorithm for searching – Linear Search and Binary search.

Linear Searching

This is the simplest method for searching a element into the list of elements. We start from the beginning of the list and search the element by examining each consequetive element in the list until the element found in the list or the list is finished.

Algorithm

1. Read N elements into an array A.
2. Read the element to be searched into KEY.
3. Repeat step 4 for I=0 to N-1 by 1.
4. IF A[I] = KEY THEN PRINT “Element found”.
5. IF element is not in the list PRINT “Element not found”.

Following code segment illustrates linear search.

```
#include<stdio.h>

#include<conio.h>

void main()

{

 int a[] = {11, 43, 75, 21, 78}, key, flag=0, i;

printf(“\n Enter the number to be searched : ”);

scanf(“ %d”,&key);

  for(i=0;i<5;i++)

       {

           If(a[i]==key)
```

```
            {

              flag = 1;

               break;

             }

      }

  If(flag==1)

    printf("\n Element found at position %d",i);

  else

   printf("\n Element is not found in the list");

}
```

Binary Searching

This is the fast and efficient method that work on sorted list. In this method element to be searched is compared with middle element of the list. If it matches then we stop the search. Otherwise we divide the list in two sublist list1 and list2. List1 is from first element to the middle element of the list and list2 is from middle element to the last element in the list. If element to be found is less than the middle element of the list then we will search the element in list1 otherwise in list2. Again repeat the same procedure until we find the element.

Algorithm

1. Read N element into an array A.
2. Read the element to be searched inot KEY.
3. SET   LOW = 0
         HIGH = N-1
4. SET MID = (LOW + HIGH)/2
5. IF KEY = A[MID] THEN PRINT "ELEMENT FOUND".
6. IF KEY < A[MID] THEN search for key in A[LOW] TO A[MID-1].
7. IF KEY . A[MID] THEN search for key in a[MID+1] TO A[HIGH].

Following code segment illustrates binary search.

```
#include<stdio.h>

#include<conio.h>
```

```
void main()

{

 int a[] = {11, 43, 75, 21, 78}, key, flag=0, low = 0, high = 4, mid;

printf("\n Enter the number to be searched : ");

scanf(" %d",key);

 while(low <= high)

   {

     mid = (low + high ) / 2;

      if(a[mid]==key)

         {

          printf("\n Element is found at position %d",mid);

         flag = 1;

         break;

         }

      if(key < a[mid])

        high = mid-1;

      else

        low = mid+1;

   }

If(flag==0)

printf("\n Element is not present in the list");
```

Sorting

Sorting is used to arrange the data in meaningful order. There are different methods that are used to sort the data in ascending or descending order. These methods are divided into two categories: They are as follows

Internal Sorting

If all the data that is to be sorted is present in primary memory at a time then internal sorting methods are used. Some of the internal sorting methods are bubble sort, selection sort, insertion sort etc.

External Sorting

When the data to be sorted is so large that some of the data is present in primary memory and some is kept in secondary memory then external sorting methods are used.

Bubble Sort

Bubble sort is one of the simplest method for internal sorting. This method is very inefficient method for a general data set. In bubble sort algorithm, successive iterations are made through the elements of the list to be sorted. During each iteration, the algorithm compares the adjacent pair like (0,1), (1,2), (2,3), (3,4) etc depending on number of elements in an array. So pair is having two elements to be considered as previous element and next element. While comparing these elements in pair, the element will be swapped if previous element is greater than the next element. Swapping occurs only among consecutive elements of the list. Only one element is placed in its sorted place after each iteration. The sorted elements are need not be considered during the next successive iteration.

Algorithm

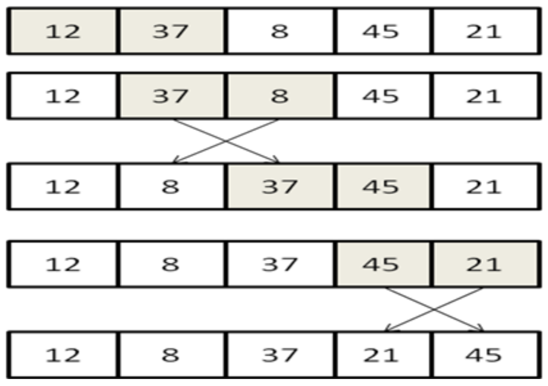
1. Consider the first pair of elements.
2. If previous element in pair is greater than next element, exchange the values of them.
3. Consider next pair and repeat step 2 for all remaining pair.
4. Repeat step no.1 equals to the number of element in an list.

Example

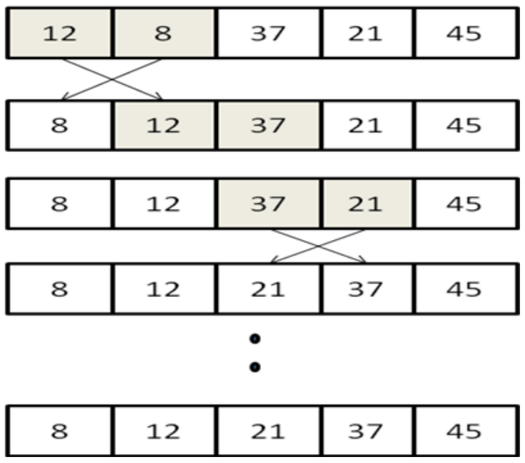
Consider the list given below

12 37 8 45 21

PASS



NEXT PASS



Complexity

Worst Case	Average Case	Best Case
$O(n^2)$	$O(n^2)$	$O(n^2)$

Selection Sort

This algorithm is simple but very inefficient because it does not consider partial or fully sorted lists. In other words, if you have partial or fully sorted list, the selection sort does the same number of comparison as it would do on completely random unsorted list and does not use any intelligence to improve the performance. In selection sort algorithm, successive iteration are made through the elements of the list to be sorted. During each iteration, successive position are taken into consideration. The element present at this selected position is compared with all elements present at successive position. If element present at selected position is found to be greater than any element present at successive position then they are exchanged. Only one element is placed in it's sorted place after each iteration. The sorted elements are need not to be considered in successive iteration.

Algorithm

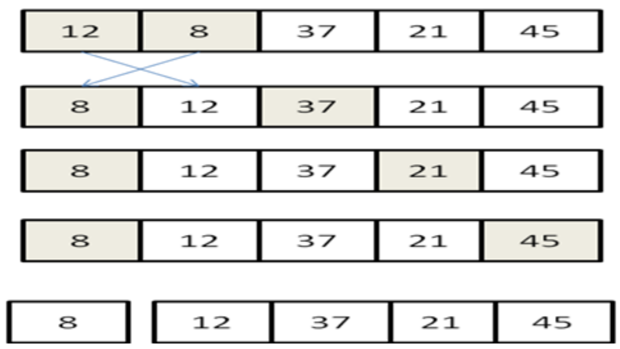
1. Select the element at first position.
2. Compare it with all remaining element at successive position.
3. If any element smaller than the element at selected position is found exchange the values of them.
4. Select the element at next position to the selected positon.
5. Repeat step 2 till all position are selected.

Example

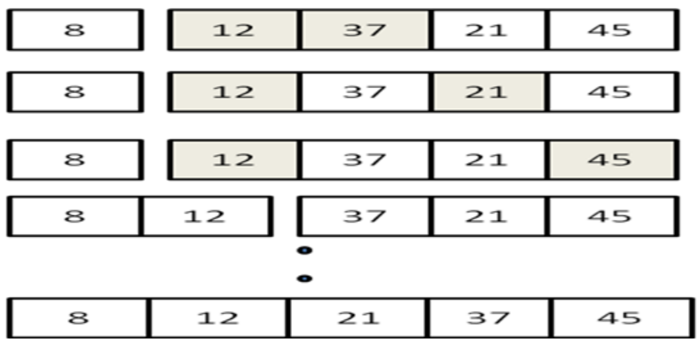
Consider the list given below

12 8 37 21 45

PASS



NEXT PASS



Complexity

Worst Case	Average Case	Best Case
$O(n^2)$	$O(n^2)$	$O(n^2)$

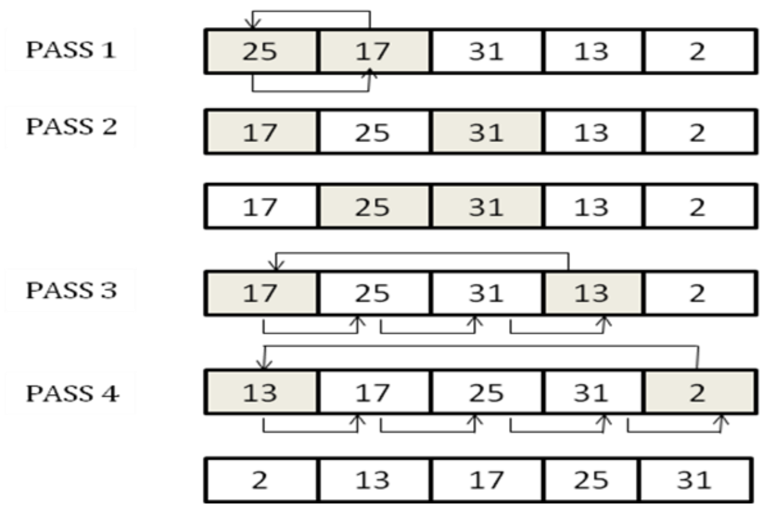
Insertion Sort

In this technique, the element at particular position will be compared with all the previous elements of it i.e. 1st element is compared with 0, 2 will be compared with 1 and 0 and so on depending upon number of elements in an array. The element which is to be compared with its previous element is considered a pivot element. While comparing these previous element, if any of them is greater than the pivot element, that element will be moved to its next positon and finally pivot element will be inserted at its appropriate position.

Algorithm

1. Consider the second element as pivot element.
2. Compare it with previous element.
3. If previous element is greater than pivot element place it at next position. Decrement the position of pivot element.
4. Repeat step 3 for all the previous elements.
5. Insert pivot element at its decremented position.
6. Consider next element as pivot element.
7. Repeat step 2 to the last element in an array.

Example
Consider the list given below
25 17 31 13 2



Complexity

Worst Case	Average Case	Best Case
O(n^2)	O(n^2)	O(n)

Quick Sort
The quick sort is the most efficient internal sort algorithm. Its performance is largely influenced by the choice of the pivot. The quick sort makes use of three strategies

- Split the array into smaller sub arrays
- Sort the sub arrays
- Merge the sorted sub arrays

A quick sort can be implemented in several ways, but the goal of each approach is to select a data element and place it in its proper positon so that all the elements on the left side of the pivot element are less than the pivot and all the elements on the right side of the pivot are greater than the pivot and the method used to split the array has a big influence on the overall performance.

Algorithm
The recursive algorithm consist of four steps

- If there are one or less elements in the array to be sorted, halt and exit.
- Pick left-most element in the array as a “pivot” element.
- Split the array into two parts where left part is having elements smaller than the pivot and the right with elements greater than the pivot.
- Recursively repeat the algorithm for both halves of the original array.

Example
If an initial array is given as :

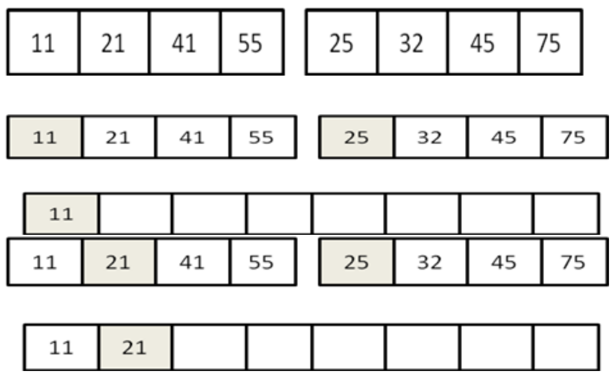
25 57 48 37 12 92 86 33
Make the element 25 as pivot element and after placing it at its proper position, the resulting array is
12 25 57 48 37 92 86 33
At this point, 25 is in its proper position in array (x[1]), each element at left side of the pivot element is less than or equal to 25, and each element at right side of the pivot element (57 48 37 92 86 33) are greater than the or equal to pivot element. Since 25 is in its final position the original problem has been splited into the problem of sorting the two subarrays. (12) and (57 48 37 92 86 33)
First subarray has one element so there is no need to sort it. Repeating the process on the subarray x[2] through x[7] yields:
12 25 (48 37 33) 57 (92 86)
And further repetitions yields

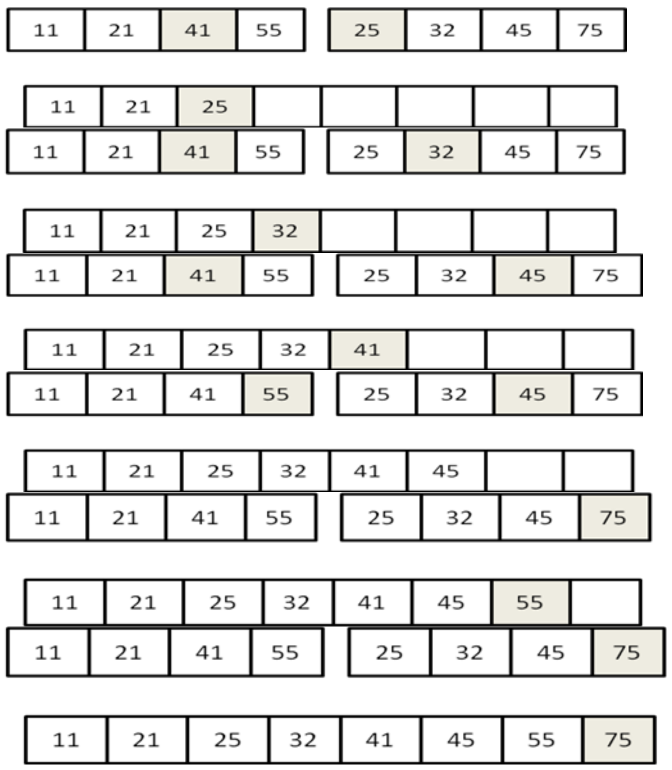
```
12   25    (37      33)    48      57     ( 92      86 )
12   25    (33)     37      48      57     ( 92      86 )
12   25    33       37      48      57     (86)     92
12   25    33       37      48      57     86       92
```

Complexity

Worst Case	Average Case	Best Case
$O(n^2)$	$O(\log_2 n)$	$O(\log_2 n)$

Merge Sort
In merge sort we merge two sorted list into third sorted list. For this we compare the elements from both the list and place smaller one of both the element in the third array. The sorting is complete when all the elements from both the list are placed in the third list.
If one of the list is exhausted during the process then the remaining elements from the other list are just copied to third list.
Example
Consider the two list given below
List1   11 23 41 55
List1 25 32 45 75





Complexity

Worst Case	Average Case	Best Case
O(n log n)	O(n log n)	O(n log n)

Shell Sort

Shell sort is one of the improvement over insertion sort. This method sorts separate subfiles of the original file. These subfiles contains every kth element of the original file. The value of k is called an increment. Suppose file contains 11 element and k is 5 then five subfiles, each containing one fifth element of the original array are created and sorted using insertion sort method. After sorting the first k subfile, a new smaller value of k is chosen and the array is again partitioned into a new set of subfiles and process is repeated till k becomes 1. The files that is to be created are in the following manner.

Subfile1 -> X[0] X[5] X[10]
Subfile2 -> X[1] X[6]
Subfile3 -> X[2] X[7]
Subfile4 -> X[3] X[8]
Subfile5 -> X[4] X[9]

Example

If the original file contains the following element
41 32 22 73 43 66 48 25 72 55 60

Let start with increment value k = 5
Pass 1
K = 5

Subfile1 -> X[0] = 41   X[5]= 66   X[10]= 60
Subfile2 -> X[1] = 32   X[6]= 48
Subfile3 -> X[2] = 22   X[7]= 25
Subfile4 -> X[3] = 73   X[8]= 72
Subfile5 -> X[4] = 43   X[9] = 55

After Sorting
Subfile1 -> X[0] = 41   X[5]= 66   X[10]= 60
Subfile2 -> X[1] = 32   X[6]= 48
Subfile3 -> X[2] = 22   X[7]= 25
Subfile4 -> X[3] = 72   X[8]= 73
Subfile5 -> X[4] = 43   X[9] = 55

Pass 2
K = 3
Subfile1 -> X[0] = 41   X[3]= 72   X[6]= 48   X[9] = 55
Subfile2 -> X[1] = 32   X[4]= 43   X[7]= 25   X[10] =60
Subfile3 -> X[2] = 22   X[5]= 66   X[8] = 73

After Sorting
Subfile1 -> X[0] = 41   X[3]= 48   X[6]= 55   X[9] = 72
Subfile2 -> X[1] = 25   X[4]= 32   X[7]= 43   X[10] =60
Subfile3 -> X[2] = 22   X[5]= 66   X[8] = 73

Pass 3
K = 1
Subfile1 -> X[0] = 41   X[1]= 25   X[2]= 22   X[3] = 48 X[4] = 32   X[5]= 66
X[6]= 55
X[7] = 43   X[8] = 73   X[9]= 72   X[10] = 60

After Sorting
Subfile1 -> X[0] = 22   X[1]= 25   X[2]= 32   X[3] = 41 X[4] = 43   X[5]= 48
X[6]= 55
X[7] = 60   X[8] = 66   X[9]= 72   X[10] = 73

Radix Sort
Radix sort or bucket sort is a method that can be used to sort the a list of numbers by its base i.e radix. It is based on the values of the actual digits in the positional representation of the numbers being sorted. For example the number 235 in decimal notation is written with a 2 in the hundreds position, a 3 in tens position and a 5 in the units position. To sort an array of decimal numbers, where the radix or base is 10, we required 10 buckets which can be numbered as 0, 1, 2, - - - ,9. Number of passes required to have a sorted array depends upon the number of digits in the largest element. Start with the least significant digit and ending with the most significant digit. Take each number in the order in which it appears in the list and place it in one of the 10 queues, depending upon the value of

digit currently being processed. Then restore each queue to the original list starting with the queue of numbers with the 0 digit and ending with the queue of numbers with the 9 digit.

Algorithm

Let A be a linear array of n element. Digit is the total number of digits in the largest element in array A.

1. Input n numbers into an array A.
2. Find the total number of digits in the largest element in the array.
3. Initialise i=1 and repeat the step 4 and 5 until (i <= Digit)
4. Initialise the queue j=0 and repeat step until(j<n)
5. Read the elements of the queue from 0th queue to 9th queue to generate new array A
6. Display the sorted array A.
7. Exit.

Example

1234 348 143 423 76 5

PASS 1 : Consider the unit digit of every numbers and put them into respective bucket.

0	1	2	3	4	5	6	7	8	9
5			143 423	1234	5	76		348	

Now list will be : 143, 423, 1234, 5, 76, 348

PASS 2 : Consider ten’s digit of every numbers and put them in respective bucket.

0	1	2	3	4	5	6	7	8	9
5		423	1234	143 348			76		

Now list will be : 5, 423,1234,143,348,76

PASS 3 : Consider hundred’s digit of every numbers and put them in respective bucket.

0	1	2	3	4	5	6	7	8	9
5 76	143	1234	348	423					

Now list will be : 5, 76, 143, 1234, 348, 423

PASS 4 : Consider 1000th digit of every numbers and put them in respective bucket.

0	1	2	3	4	5	6	7	8	9
5 76	1234								

143 348 423									

Now list will be : 5, 76, 143, 348, 423, 1234
Now list is sorted by radix sort.
Complexity

Worst Case	Average Case	Best Case
O(n^2)	O(n log n)	O(n log n)

Heap Sort
There are two phases in this sorting technique. In the first phase we construct a heap by adjusting the array elements and in the second phase sort the heap by repeatedly eliminating the root element of the heap by shifting it to the end of the of the array and then restore the heap structure with remaining elements.

Constructing a heap
A heap is a “complete binary tree”. There are two types of heaps. If the value present at any node is greater than all its children then such a tree is called as the max-heap or descending heap. In case of min-heap or ascending heap the value present in any node is smaller than all its children. It is not necessary that the two children must be in some order. Sometimes the value in the left child may be more than the value at right child and sometimes vice-versa.
Heap can be constructed from an array. To begin with a tree is constructed. If the tree doesn’t satisfy heap properties then it is converted to heap by adjusting nodes. Adjustment of nodes starts from the level one less than the maximaum level of the tree. Each sub-tree of that particular level is made a heap. Then all the subtree at the level two less than the maximum level of the tree are made heaps. This procedure is repeated till the root node. As a result the final tree becomes a heap.
Example
Consider the array elements given below
40 80 35 90 45 50 70
The tree that can be constructed from above array is as shown in figure.

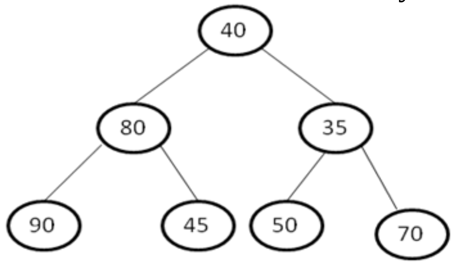
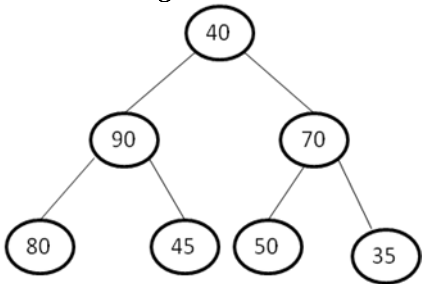


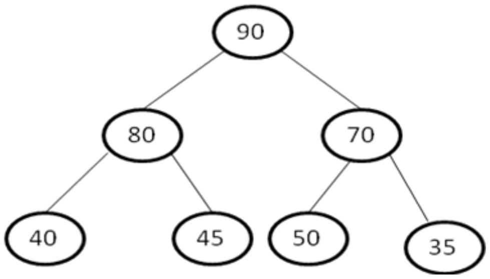
Fig : Equivalent binary tree of an array

To make it a heap initially the elements that are present at a level one less than the maximum level of the tree are taken into consideration. In our example , these are 80 and

35. They are converted to heap by comparing them with the children. If the value at node is smaller than it's children it is exchanged with child that is greater among them. The resultant tree is as shown in figure.



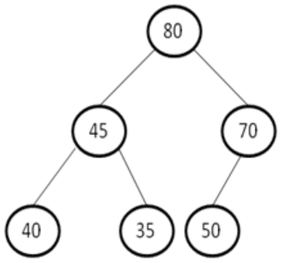
Now the elements that are present at a level two less than the maximum level of the tree are considered. In our case, it is the root node i.e. 40. This node is also made heap by the same procedure.

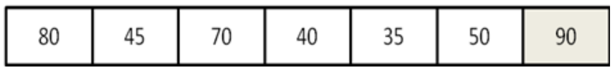


As seen from above, each time one level is decremented and all the sub-tree at that level are converted to heap. As result, finally entire tree gets converted to heap.

Sorting

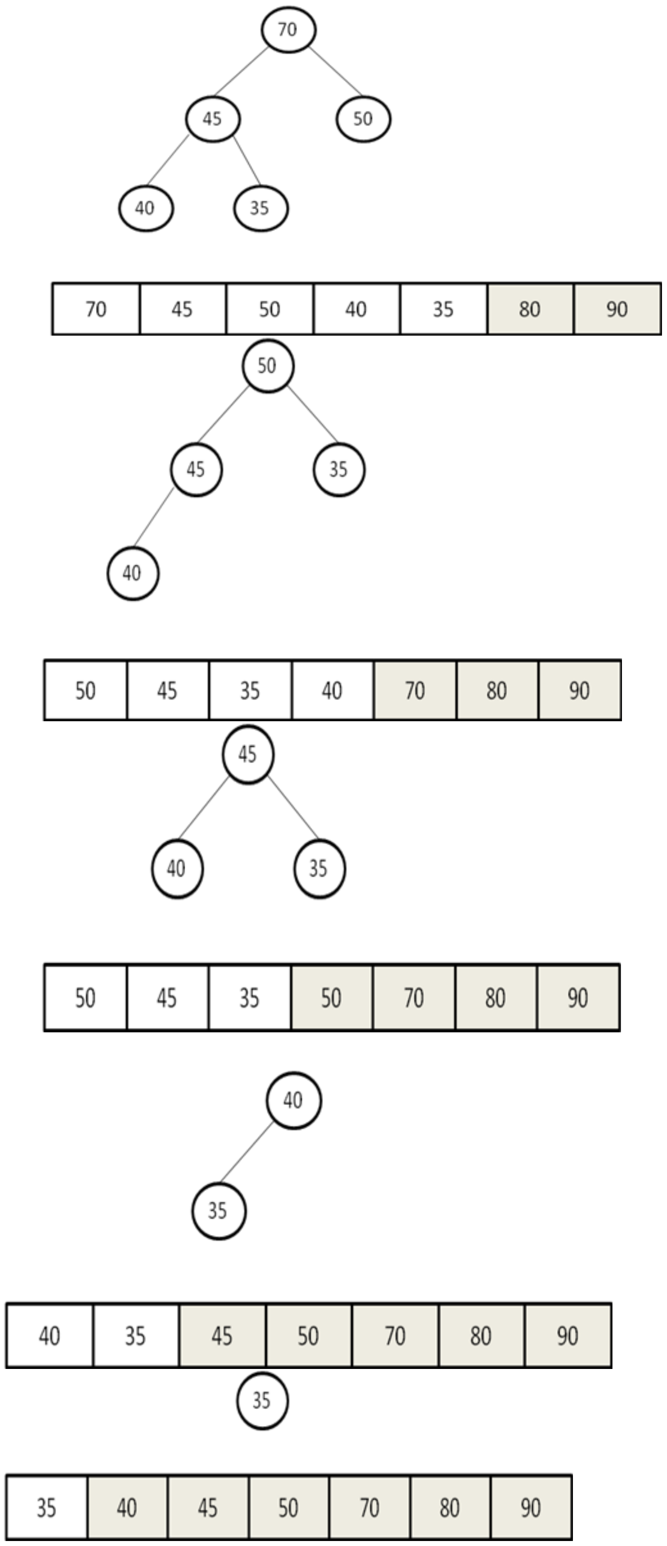
After constructing a heap now we can sort it . For sorting heap eliminate the root element of the heap by shifting it to the end of the array and then restore the heap structure with remaining elements. As shown in figure.





80	45	70	40	35	50	90

Similarly, one by one the root element of the heap is eliminated .Following figure shows the heap and the array after each elimination.



Complexity

Worst Case	Average Case	Best Case
O(n log n)	O(n log n)	O(n log n)

Binary tree Sort

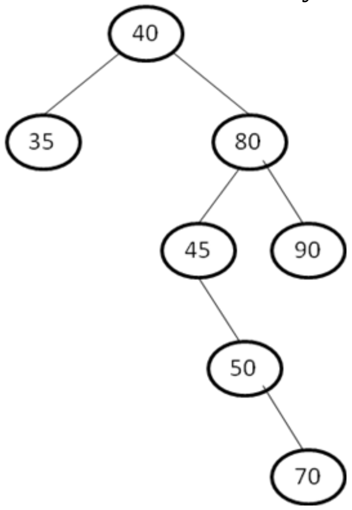
In this technique of sort we use binary search tree. There are two phases in this sorting technique. In first phase we construct a binary search tree from the array element. And in second phase we traverse the tree in inorder. In inorder access of tree node we get the elements in ascending order. To place a element in binary search tree, the elements is compared with the node. If this element is less than the content of node then it is placed in left branch and if it is greater than equal to content of node then it is placed in right branch.

Example

Consider the array elements given below

40 80 35 90 45 50 70

The tree that can be constructed from above array is as shown in figure.



We traverse the above binary search tree in inorder we get the list as

35 40 45 50 70 80 90.

Complexity

Worst Case	Average Case	Best Case
$O(n^2)$	O(n log n)	O(n log n)