MAR GREGORIOS COLLEGE OF ARTS & SCIENCE

Block No.8, College Road, Mogappair West, Chennai – 37

Affiliated to the University of Madras
Approved by the Government of Tamil Nadu
An ISO 9001:2015 Certified Institution

DEPARTMENT OF COMPUTER APPLICATION

SUBJECT NAME: DATA STRUCTURES

SEMESTER: III

PREPARED BY: PROF. A. HENCY JULIET

SYLLABUS

OBJECTIVES:

- To understand the concepts of ADTs
- To learn linear data structures-lists, stacks, queues
- To apply Tree and Graph structures
- To understand sorting, searching and hashing

OUTCOMES:

- Implement abstract data types for linear data structures.
- Apply the different linear and non linear data structures to problem solutions.
- Critically analyze the various sorting algorithms.

UNIT - I

Abstract Data Types (ADTs)- List ADT-array-based implementation-linked list implementation-singly linked lists-circular linked lists-doubly-linked lists-applications of lists-Polynomial Manipulation- All operations-Insertion-Deletion-Merge-Traversal.

UNIT - II

Stack ADT-Operations- Applications- Evaluating arithmetic expressions – Conversion of infix to postfix expression-Queue ADT-Operations-Circular Queue- Priority Queue- deQueue- applications of queues.

UNIT - III

Tree ADT-tree traversals-Binary Tree ADT-expression trees-applications of trees-binary search tree ADT- Threaded Binary Trees-AVL Trees- B-Tree- B+ Tree – Heap-Applications of heap.

UNIT - IV

Definition- Representation of Graph- Types of graph-Breadth first traversal – Depth first traversal-Topological sort- Bi-connectivity – Cut vertex- Euler circuits-Applications of graphs.

UNIT - V

Searching- Linear search-Binary search-Sorting-Bubble sort-Selection sort-Insertion sort-Shell sort-Radix sort-Hashing-Hash functions-Separate chaining- Open Addressing-Rehashing- Extendible Hashing.

TEXT BOOKS:

1. Mark Allen Weiss, "*Data Structures and Algorithm Analysis in C++",* Pearson Education 2014, 4th Edition.
2. Reema Thareja, "*Data Structures Using C",* Oxford Universities Press 2014, 2nd Edition.

REFERENCES:

1. Thomas H.Cormen,Chales E.Leiserson,Ronald L.Rivest, Clifford Stein, "*Introduction to Algorithms*", McGraw Hill 2009, 3rd Edition.
2. Aho, Hopcroft and Ullman, "*Data Structures and Algorithms*", Pearson Education 2003.

WEB REFERENCES:

- NPTEL & MOOC courses titled Data Structures
- https://nptel.ac.in/courses/106106127/

UNIT I

1.1 INTRODUCTION TO DATA STRUCTURE

The material contained in this part of the book provides the necessary foundation for the study of the Data Structures and Algorithms presented in subsequent chapters. Specially chapter one introduces the concept of data structures, abstract data and discusses the distinction between abstract data type and data types. This chapter also introduces this pseudo code that will be used throughout the book to specify that taken in an algorithm.

One of the most Brazil issues associated with the this One of the most Brazil issues associated with the Disney One of the most Brazil issues associated with the One of the most Brazil issues associated with the designer One of the most Brazil issues associated with the design One of the most Brazil issues associated with the design One of the most crucial issues associated with the design of a new system in wall managing the complexity of the in wall managing the complex city of the design process DESIG. Good designers type Achilles some form of a Good designers type Achilles some form of us Good designers type Achilles some form of Good designers type Achilles some form of extra Good designers type Achilles some form of a Good designers type Achilles some form of Good designers typically use some form of abstraction as a tool for dealing with this complexity. The term **abstraction** refers to the intellectual capability of considering an entity apart from any instance of the entity.

During the progress of the design process, the various types of data necessary, as well as the operations that must be performed on this data, become evident. At that point, special type of abstraction known as data abstraction can be employed this involves an abstract or logical description of both the data required by the software system and the operation that can be performed on this data.

1.2 ABSTRACT DATA TYPES

An **Abstract Data Type (ADT)** is defined as a mathematical model of the data objects that make up a data type, as well as the functions that operate on these objects. It is important to recognize that the operation manipulate the data objects are included in the specification of an ADT.

At this point it is useful to distinguish between ADTs, data type and data structures. The term data type refers to the implementation of mathematical model specified by an ADT. That is a data type is a computer representation of an ADT. A data type specifies the size and type of values that can be stored.

The term data type refers to the kinds of data that variable may hold in a programming language. The term Data Structure refers to a collection of computer variables that are connected in some specific manner.

The term data object refers to a set of elements say, D. For example the data object integers refers to D = {D,}. Thus D may be finite or infinite and if D is very large we may need to device special ways to representing its elements in computer.

1.3 DATA STRUCTURE

Data Structure was distinguished from a data object, that we want to describe not only the set of objects, but the way they are related. Another way to say this to describe the set of operations, which may be applied to elements of the data object. For integer, we have the arithmetic operations +, -, *, /, mod etc… Thus, the data object integer and some operations constitutes a data structure.

EXAMPLE TO DEFINE A DATA STRUCTURE

Consider a data structure natural number (abbreviated natno), natno = {0,1,2,3,.....} with the tree operations being real test for zero, addition and equality.

The following notations can be used

Structure NATNO

1. declare ZERO( ) → natno
2. ISZERO (natno) → Boolean
3. SUCC (natno) → natno
4. ADD (natno, natno) →natno.
5. EQ (natno, natno) → Boolean
6. For all x, y € natno let
7. ISZERO (ZERO) ::= TRUE;
8. ADD (ZERO, Y) ::= Y;
9. EQ (ZERO, SUCC (Y)) :: = FALSE;
10. End
11. End NATNO

In the declaration statement, five functions are defined. ZERO is a constant function which means it takes no input arguments and its result is the natural number zero. ISZERO is Boolean function who's result is gay the true or false. SUCC stands for successor using zero and SUCC we can define all the natural numbers as: ZERO, 1=SUCC (ZERO), 2 = SUCC(SUCC(ZERO)), 3=SUCC(SUCC(SUCC(ZERO))), etc…

If we want to add two and three we would get the following sequences expressions
ADD (SUCC(SUCC(ZERO)), SUCC(SUCC(SUCC(ZERO))).

DEFINITION

A data structure is a set of domains D, a designated domain d € D, a set of functions F and a set of axioms A. The triple (D,F,A) denotes the data structure the and it will usually be abbreviated by writing d.

In the previous example
D=natno, D = {natno, Boolean}
F = {ZERO, ISZERO, SUCC, ADD}
A= {lines 7 through 9 of the structure NATNO}

The set of axioms describe the matrix of the operations that is the form in which we chose to write the axioms is important. Also the functions are used to implement the data structure. Integers are represented by bit strings, bullying is represented by 0 and 1, an Aries represented by a set of consecutive words in memory.

The triple (D,F,A) is referred to as an abstract data type. An abstract data type is a data type that is organized in such a way that the specification of the objects and the specification of the operations on the objects is separated from the representation of the objects and implementations of the operation. It is called abstract because the axioms do not imply a form of representation.

1.4 PRIMITIVE AND COMPOSITE DATA TYPES

The term data type refers to the kind of data that variable main hold in a programming language. Specify the size and type of value that can be stored. There are two categories in data types.

1. Primitive data types
2. Composite data types

1.4.1 PRIMITIVE DATA TYPES

Primitive data types are known as primitive data types are known as built in types or basic type or primary data types. These are data type provided by a programming language as basic building blocks. Depending on the languages it may vary.

Depending on the language and implementation primitive types may or may not have a one to one correspondence with objects in the computer's memory. The actual range of primitive type is dependent upon the specific programming language that is being used. Examples for primitive data types in java is int, float, char, long in, double, short int.

1.4.2 COMPOSITE DATA TYPES

Composite type or data type which can be constructed in a programming language. The act of constructing a composite type is known as composition. Structure in C++ notation (STRUCT) is a composite type. A data type that composes a fixed set of labeled fields are members. It is so called because of the STRUCT keyword used in declaring them, which is short for structure or more preciously user defined data structure.

Composite data types are nothing but a user defined data type.

Example for composite data types are structure, union, array, enumerated data types.

```
Struct strname
{
 Data type member 1;
 data type member 2;
 data type member n;
};
```

1.5 ORDER LIST

One of the simple and most commonly found data object is the **ordered list** or **linear list**.

EXAMPLES

The days of the week

(Sunday Monday Tuesday Wednesday Thursday Friday Saturday)

or the values in a card deck

(2 3 4 5 6 7 8 9 10 Jack Queen King Ace)

or the floors of the building

(Basement, lobby, mezzanine, first, second, third)

or years of the United States fought in World War II

(1941 1942 1943 1944 1945)

If we consider an ordered list more abstractly, we say that it is either empty or it can be written as (a_1, a_2, a_3, ……,a_n)

Where a_i are atoms from some set S.

OPERATIONS ON LIST

There are a variety of operations that are performed on these lists. These operations include

1. Find the length of the list n
2. Read the list from left to right or right to left
3. Retrieve the i^{th} element $1 \leq i \leq n$
4. Store a new value into the ith position
5. Insert a new element at position i, $1 \leq i \leq n$ causing elements numbered i, i+1,....n to become numbered i + 1, i+ 2,.... n + 1
6. Delete the element at position i, $1 \leq i \leq n$ causing element numbered i, i+1,....n to become numbered i + 1, i+ 2,.... n - 1

The most common way to represent an ordered list is by an array where we associate the list element ai with the array index i. This will be referred to us as a sequential mapping, because using the conventional array representation we are storing ai and ai+1 in two consecutive locations i & i+1 of the array. This gives us the ability to describe or modify the values of random elements in the list in a constant amount of time, essentially because a computer memory has random access to any word. We can access the list elements in either direction by changing the subscript values in a controlled way.

ALGORITHM TO FIND THE LENGTH OF LIST L

Procedure List_Length(L,n)

// Let L is a list with n element

// Assume LEN is a function, it return the length of the list.

1. l = Length(L);
2. print l,n;
3. end List_Length

READING ELEMENTS FROM THE LIST

```
Procedure List_read(L,n)
// Let L is a list with n element
1. for i = 1 to n do
2.      read L(i); // Reading from left to right
3. end
4. for I = n to 1 do
5.      read L(i); // Reading from right to left
6. end
7. end List_read
```

RETRIEVE AN ELEMENT FROM THE LIST

```
Procedure List_Retrieve (L,n, i)
// Let L is a list consisting of  n element
// Retrieve the i^th element from the list. I ≤ n
1. for i = 1 to n do
2.      if ( i = j ) then
3.           print L (j);
4.           exit ;
5. end
6. end List_Retrieve
```

STORE AN ELEMENT INTO THE LIST

```
Procedure List_Store (L,n, i, x)
// Let L is a list consisting of n element
// Store a new element x into the i^th position of the list 1≤  i ≤ n
1. for j = 1 to n do
2.      if ( j = i ) then
3.           L ( i ) = x;
4. end
5. end List_Store
```

INSERT A NEW ELEMENT INTO THE LIST

```
Procedure List_Insert (L,n, i, x)
// Let L is a list consisting of n element
// Insert a new element x into the i^th position of the list 1≤ i ≤ n
1. for j = 1 to n step -1 do
2.      if ( j ≥ i ) then
3.              L ( j + 1 ) := L (j);
4. end
5. L (I ) := x;
6. for i = 1 to n + 1 do
7.      print L (i); // printing all the element, including the newly inserted element
8. end
9. end List_Insert
```

DELETE AN ELEMENT FROM THE LIST

```
Procedure List_Delete (L,n, i )
// Let L is a list with n element
// Delete an element at the position i
1. for j := 1 to n do
2.      if ( j ≥ i ) then
3.              L ( j ) := L ( j + 1);
4. end
```

ORDERED LIST APPLICATION POLYNOMIAL

Let us jump right into a problem requiring order lists, which we solve by using one dimensional arrays. This problem has become the classical example for motivating the use of list processing techniques. The problem calls for building a set of subroutine which are low for the manipulation of symbolic polynomials. Buy symbolic the mean the list ofBuy symbolic the mean the list of coefficients By **symbolic** we mean the list of coefficients and exponents which accompany a polynomial.

Example:

Two such polynomials are

$A(x) = 3x^2 + 2x + 4$ and $B(x) = x^4 + 10x^3 + 3x^2 + 1$

The first step is to consider how to define polynomial as input in computer structure. Mathematician a polynomial is a sum of terms for each term has form ax^e; x is the variable, a is the Coefficient and e is the exponent. However this is not an appropriate definition for our purposes. Panda finding a data object we must decide what functions will be available, what there input is, what their output is and exactly it is that they do

Let us see a problem for adding two polynomials using one dimensional array

Let $f(x) = a_m x^{em} + a_{m-1} x^{em-1} + \ldots\ldots\ldots + a_1 x^{e1}$

and

$g(x) = b_n x^{fn} + b_{n-1} x^{fn-1} + \ldots\ldots\ldots + b_1 x^{f1}$ be two polynomials with number of non-zero terms m and n respectively.

They are represented in memory using 1 – Dimensional array

A:

m	e_m	a_m	e_{m-1}	a_{m-1}	…..	e_1	a_1

B:

n	f_n	b_n	f_{n-1}	b_{n-1}	…..	f_1	b_1

To find the array which corresponds to A(x) + B (x). That is to add two given polynomials, compare the exponent of each polynomial, if they are equal we can add the coefficients. If the coefficient is not equal to zero then it can be stored with exponent.

If A polynomial exponent is greater than B polynomial, then just store T a polynomial into the result. If the B polynomial is Greater means store the B polynomial time into the result. If any of the polynomial time remaining then copy into the research without any comparison.

Let us now write a procedure in SPARKS for adding two polynomials

Procedure PADD (A, B, C)

//A(1 : 2m +1), B (1 : 2n +1), C(1 : 2 (m+n) + 1)

m ← A (1); n ← B(1);

p ← q ← r ← 2;

While $p \leq 2m$ and $q \leq 2n$ do

```
Case    // Compare Components //
        : A (p) = B (q) : C(r+1) ← A(p+1) + B (q+1)          // Add Coefficients//
                If C (r+1) ≠ 0 then
                        C (r) ← A (p); r ← r +2;              // Store Exponent //
                p := p +2; q:=q+2                 //Advance to next term //
        A (p) <  B (q)  : C(r+1) ←  B (q+1) ; C (r) ← B (q); // Store new Term //
                q:=q+2;  r ← r +2;      //Advance to next term //
        A (p) > B (q)   :  C(r+1) ←  A (p+1) ; C (r) ← A (p);          // Store new Term //
                p := p +2;  r ← r +2;   //Advance to next term //
end
end
While p ≤ 2m  do                  //Copy Remaining terms of A //
        C(r) ← A (p); C(r+1) ← A(p+1)
        p := p +2;  r ← r +2;
end
While q ≤ 2n  do                  //Copy Remaining terms of B //
        C(r) ← B (q); C(r+1) ← B(q+1)
        q:=q+2;  r ← r +2;
end
C ( 1) ←r/2 – 1          // Number of terms in the Sum //
End PADD
```

Time complexity of an algorithm

Line 1 & 2 needs 5 units → O (1)

Line 4 needs 4 units → O (1)

Line 3 needs m+n times of line 4

Line 6-8 needs O (1)

Total time complexity = O (1)+ O (m+n) + O (1)+ O (1)

= O (m+n)

Disadvantages of representing polynomials using array

1. There are some polynomials that are no longer needed such a space can be reused

2. By writing a function to compact the remaining polynomial leaving a large contiguous free space at one end
3. But this require much data movement also we must change its start and end pointers

BASIC DATA STRUCTURES - ARRAYS

INTRODUCTION

Data structures are classified as either linear or nonlinear. Data Structure is set to be linear if its elements or sequence or a linear list. There are two basic ways of representing such linear structures in memory. One way is to have the linear relationship between the elements represented by means of sequential memory locations; these linear structures are called arrays. The other way is to have the linear relationship between the elements are presented by means of pointer or links. These linear structures are called linked lists. Non linear structures are trees and graphs.

DEFINITION

An array is a set of consecutive memory locations. An arrays are always implemented by using constitutive memory, but not always, because there is a distinction between a data structure and its representation. Intuitively an array is a set of pairs, index and values. For each index which is defined, there is a value associated with that index.

An array is a data structure the operations performed on arrays are defined using functions.

Structure Array (value, index)

Declare CREATE ( ) → array;

RETRIEVE (array, index → value;

STORE (array, index, value) → array;

For all A € array, i,j, € index, x € value

Let STORE (A,i,x);

RETRIEVE (A, j) := if EQUAL (I,j) then x else RETRIEVE (A, j)

end

end Array

Explanation:

The function CREATE produces a new empty array. RETRIVE takes as input an array and an index and other returns the appropriate value or an error. STORE is used to enter new index value pairs.

The second axiom is read as to retry the Jth item by a has already been stored 8 index in a is equal and to checking if I and J are equal and if so x, aur search for the date value in the remaining array A,

also I N G need not necessarily be integers. If we restrict the index value to be integers then for implement store and retrieve so that they operate in a constant amount of time.

ARRAY REPRESENTATION

Are presentation means how the array elements are represented in memory. The memory may be regarded as one dimensional with words numbers from 1 to M. So we are concerned with the presenting n dimensional arrays in a one dimensional memory.

ONE DIMENSIONAL ARRAY REPRESENTATION

The elements in an array sort store sequentially in the memory location. Let a be the array with an elements. This is presented as a(n), the elements are stored in a(1), a(2), a(3),.... a(n). These elements are accessed through an index.

SCHEME 1

The polynomials are representation using array. A General polynomial a(x) can be written as

And the degree of A is n. This can be represented using one-dimensional array of length n + 2.

$A = (n, a_n, a_{n-1}, \ldots\ldots a_1, a_0)$.

The first element is the degree of a followed by the n + 1 coefficients. But some disadvantages are there in this representation, which is the large amount of waste at Storage for SAT and polynomial. In array representation the zero of easy and must be represented. But in conventional way you just need not be represented. Consider X + 1, for instance which will require a length 1002, while 999 of these values will be zero. Therefore we are led to consider an alternative scheme.

SCHEME 2

Suppose we take the polynomial A (x) and keep only its non zero coefficients.

Let $f(x) = a_0 x^n + \ldots\ldots\ldots a_n x + a_{n+1}$

We represent this by

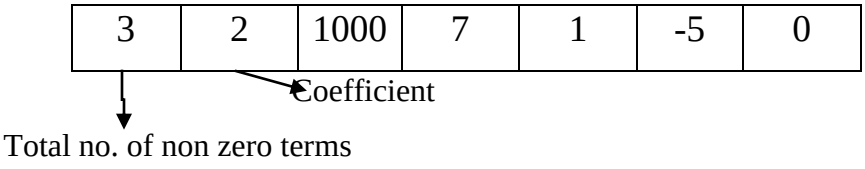
m	a_0	n	a_1	n-1	….	a_n	1	a_{n+1}	0
1	2	3						2m	2m+1

Where m is the number of non-zero terms. Suppose

$F(x) = 2x^{1000} + 7x - 5$

It contains 3 non zero terms. so that

Power



It requires 2 m + 1 memory locations in the above example m=3 (number of non zero terms), so it need 7 locations. But scheme one needs 1003 locations so that in general is the number of non zero element is less than or equal to half the degree, scheme 2 is best otherwise scheme 1 is better.

That is

$2m + 1 \leq n + 2$

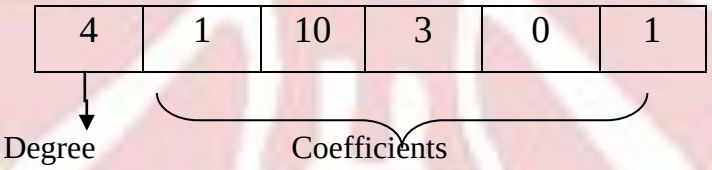
$2m \leq n + 1$

$m \leq (n+1) / 2$

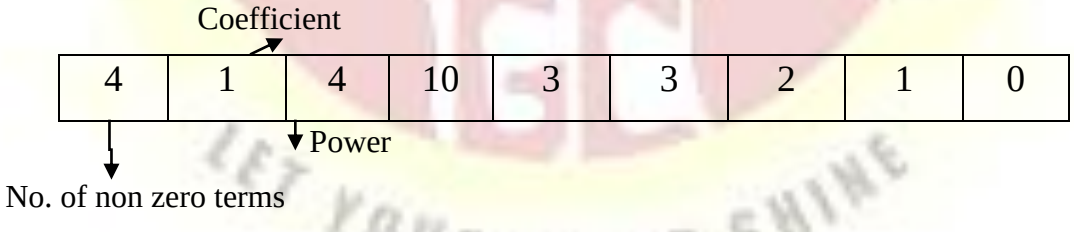
Therefore number of non zero terns $\leq$ degree/2

EXAMPLE 1:

$f(x) = x^4+10x^3 + 3x^2 +1$ This can be represented in memory using Scheme – 1



Using Scheme – 2



It requires 9 locations.

EXAMPLE 2:

$f(x) = x^{1000} +1$

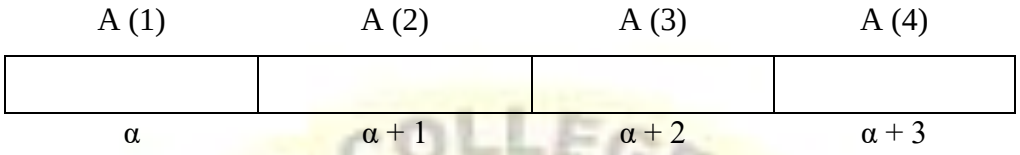
This can be represented in memory using scheme - 1 requires 1002 locations using scheme - 2 require 5 locations.

$2m + 1 \rightarrow 2 \text{ X } 2 + 1 \rightarrow 5$

REPRESENTATION OF 1- DIMENSIONAL ARRAY

Let A(1: n) be an one dimensional array. If sin infinity is the address of a(1) then the address of a(i) is

$$\alpha + (i - 1)$$



Address of i^{th} memory location is $\alpha + i - 1$

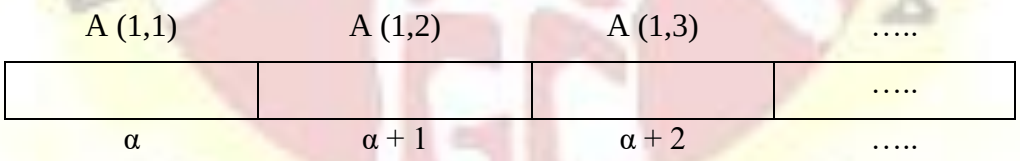
If i = 2 then second location address is $\alpha + 2 - 1 = \alpha + 1$

REPRESENTATION OF 2- DIMENSIONAL ARRAY

Let A(1 : m, 1 : n) is a two dimensional array with m rows and n columns. If α is the address of A(1,1) then the address of A(i,j) is $\alpha + (i - 1) n$ consequently the address of A(i,j) is

$$\alpha + (i - 1) n = j - 1$$

Where I is row and j is column, m → number of rows and n → number of columns



Consider 3 X 3 matrix with 3 rows (m=3) and 3 columns (3=columns). If i = 2, j = 2 that is A(1,2) is the 2^{nd} location. The address is

$\alpha + (1 - 1)$ X 3 + 2 – 1 = α + 1

REPRESENTATION OF 3 - DIMENSIONAL ARRAY

Let A(1:m, 1:n, 1:p) is a 3 – Dimensional array. Let α be the address of A(1,1,1). Then the address of A (i,j,k) is

$$\alpha + (i - 1) mn + (j - 1) m + (k - 1)$$

In general let A ($1 : u_1, 1 : u_2, \ldots\ldots 1: u_n$) be an n – dimensional array. Let α be the address of A (1,1,1,1,……1). Then the address of A(i1,i2,i3,…..in) is

$\alpha + (i_1\text{-}1)\, u_1\, u_2 \ldots\ldots\ u_{n-1}$

$+ (i_2 - 1)$

$+ \ldots\ldots\ldots$

$+ (i_{n-1} - 1)\, u_1 + (i_{n-1})$

OPERATIONS PERFORMED ON ARRAYS

There are two main operations performed on arrays.

1. Retrieve a value from an array
2. Store a value into an array

The operations which are normally performed on any linear structure, Whether it may be an array or a linked list includes the following:

1. **Traversal**: processing each element in the list
2. **Search**: finding the location of the element with the given value for the record with the given key.
3. **Insertion**: adding new element to the list
4. **Deletion**: removing an element from the list
5. **Sorting**: arranging the elements in order
6. **Merging**: combining two list into a single list

Traversing linear arrays:

Let A be Collection of data elements stored in the memory of the computer. Suppose we want to bring the contents of each element of A or suppose we want to call the number of elements of A with a given property. This can be accomplished by **traversing** A, that is by accessing and processing (frequently called visiting) each element of A exactly once.

The following and garden traverses an array A.

ALGORITHM FOR TRAVERSING AN ARRAY

Algorithm Traverse(A,n)

//A is an array with n elements A(1:n)

// This algorithm traverses A by applying an operation PROCESS to each Element of A

1. i := 1; // initialize the counter
2. Repeat step 3 and 4 while i ≤ n
3. Process A(i) // visit the element
4. i := i +1 // increase the counter

ALGORITHM FOR SEARCHING AN ELEMENT IN AN ARRAY

Algorithm search (A, n, x)

//A is an array with n elements A(1:n)

// This algorithm searches the element x in the array A

1. for i := 1 to do

2. if (x = A (i)) print ‘ Search Success’ ; exit;

3. end for

4. Print ‘ Search is not success”

End search

INSERTION AND DELETION

Let A be a collection of data elements in the memory of the computer. **Inserting** refers to the operation of adding another element to the collection A, and **deleting** refers to the operation of removing one of the element from A.

Inserting an element at the end of the Linear array can be easily done provided the memory space allocated for the array is large enough to accommodate the additional element. On the other hand suppose we need to insert an element in the middle of the array then on the average half of the elements must be moved down to new location to accommodate the new element and keep the order of the Other elements.

Similarly deleting an element at the end of an array present no difficulties, but deleting an element somewhere in the middle of the array would require that each subsequent element be moved one location upward in order to **fill up** the array.

ALGORITHM TO INSERT AN ITEM INTO AN ARRAY

Algorithm insert (A,n,k,item)

// A is an array with n element A(1: n)

// This algorithm insert a data item into an array in the k^{th} position

{

1. j :=n;
2. repeat steps 3 & 4 while j ≥ k
3. A(j+1) :=A(j) //Move the ith element downward
4. j : = j -1; // Decrease the counter
5. A (k) := item; // Insert Element
6. n:=n+1;

}

ALGORITHM TO DELETE ITEM FROM AN ARRAY

Algorithm delete (A,n,k,item)

//A is an array with n elements A(1:n), this algorithm to delete an item at the position k

{

1. item : = A(k);
2. Repeat for j:=k to n – 1
3. A(j) : = A(j+1); //Move the $j+1^{st}$ element upward
4. n := n-1;

}

SORTING

Let a be an array with n elements, sorting refers to the operation of rearranging the elements of A, so they are in increasing order, that is

For example suppose an array A contains

8 4 19 2 7 13 5 16

After sorting A is,

2 4 5 6 7 8 13 16 19

There are many different sorting algorithms, a very simple sorting algorithm known as the Bubble sort.

DEFINITION

Sorting refers to arranging numerical data in increasing order; sorting may also mean arranging numerical data in increasing order for arranging non numerical data in alphabetical order.

ALGORITHM FOR SORTING

```
Algorithm sort ( )
 // A is an array with n element A(1:n).
// This algorithm Sort the elements in ascending order.
{
1. for i := 1 to n do
2. for j := i+1 to n do
3. if (a[i] ≥ a[j])
4.        t := a[i];
5.        a [i] := a[j];
6.        a [j] := t;
7. end j
8. end i
9. for i := 1 to n do
10.       write a[i] ;
}
```

SEARCHING

Let A be an array with n elements a(1:n) in memory, and supports a specific item of information is given. Searching refers to the operation of finding the location of item in A, or printing some message that 'item does not appear there'. The starch is said to be successfully if item does appear in A and unsuccessful otherwise.

There are many different searching algorithms. One of the simple search algorithm is **Linear search**, another familiar search algorithm is **binary search**.

LINEAR SEARCH

Suppose a is a Linear array with n elements. The most Intuitive way to search for a given **item** in A is to compare **item** with each element of A one by one. That is first we test whether A[1]= item, and

then we test A[2]= item and so on. This method which traverses A sequentially to locate **item** is called a linear search or sequential search.

MERGING

The process of combining two arrays into a single array is known as merging. Let A be an array with 1 to n elements and B be an array with 1 to m elements. Dangerous accident array containing n + m elements. The resultant array size must be enough to hold the n + m elements.

```
Algorithm merge A,B,n,m)
//Let A is an array with n elements A(1:n) and B is an array with m elements B(1:m),
// Assume the size of array A is n + m
1. j :=n;
2. for i := 1 to m do
3.        a[j+1] := b[j];
4. for i := 1 to n + m do
5.        print a [i];
6. end merge
```

LINKED LIST IMPLEMENTATION

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

The disadvantages of arrays are:

- The size of the array is fixed. Most often this size is specified at compile time. This makes the programmers to allocate arrays, which seems "large enough" than required.
- Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.
- Deleting an element from an array is not possible. Linked lists have their own strengths and weaknesses, but they happen to be strong where arrays are weak.
- Generally array's allocates the memory for all its elements in one block whereas linked lists use an entirely different strategy. Linked lists allocate memory for each element separately and only when necessary.

Linked List Concepts:

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

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

Advantages of linked lists:

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

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

Disadvantages of linked lists:

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

Types of Linked Lists:

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

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

CIRCULAR DOUBLE LINKED LIST.

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

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

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

Comparison between array and linked list:

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

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

Trade offs between linked lists and arrays:

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

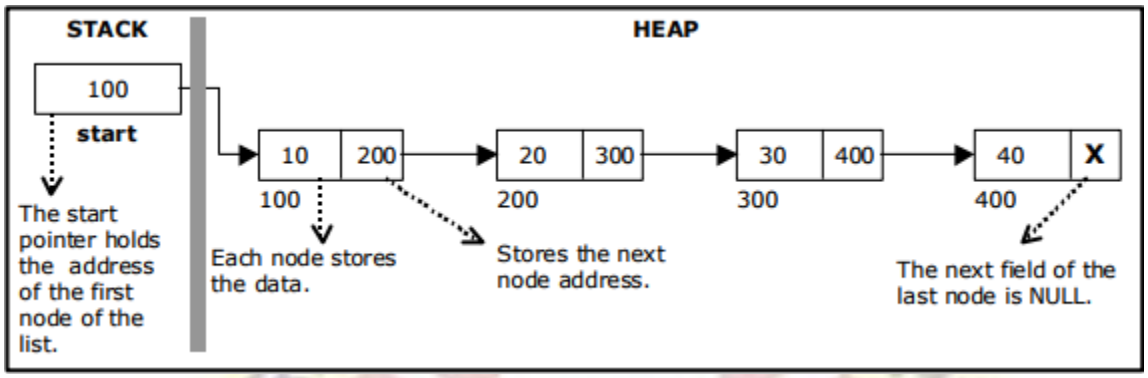
Applications of linked list:

1. Linked lists are used to represent and manipulate polynomial. Polynomials are expression containing terms with non zero coefficient and exponents. For example: P(x) = a0 Xn + a1 Xn-1 + …… + an-1 X + an
2. Represent very large numbers and operations of the large number such as addition, multiplication and division.
3. Linked lists are to implement stack, queue, trees and graphs. 4. Implement the symbol table in compiler construction.

Single Linked List:

A linked list allocates space for each element separately in its own block of memory called a "node". The list gets an overall structure by using pointers to connect all its nodes together like

the links in a chain. Each node contains two fields; a "data" field to store whatever element, and a "next" field which is a pointer used to link to the next node. Each node is allocated in the heap using malloc(), so the node memory continues to exist until it is explicitly de-allocated using free(). The front of the list is a pointer to the “start” node.



A single linked list

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

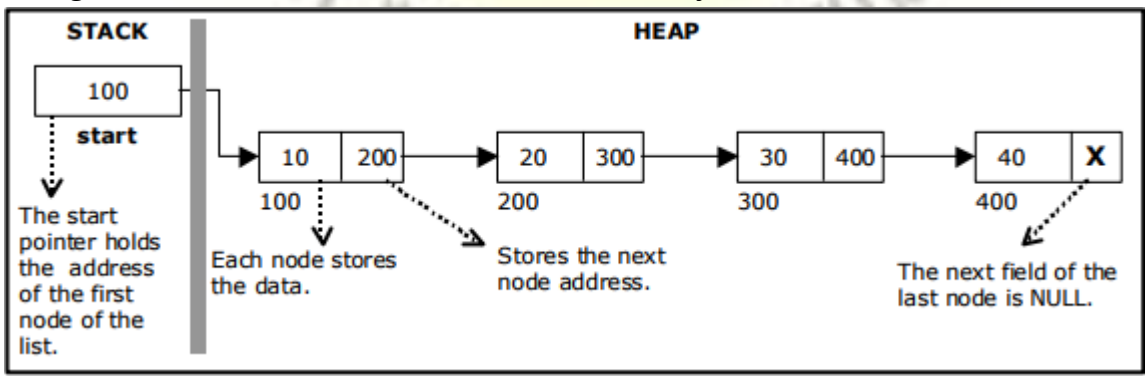
The start pointer is an ordinary local pointer variable, so it is drawn separately on the left top to show that it is in the stack. The list nodes are drawn on the right to show that they are allocated in the heap.

The basic operations in a single linked list are:

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

Creating a node for Single Linked List:

Creating a singly linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory.

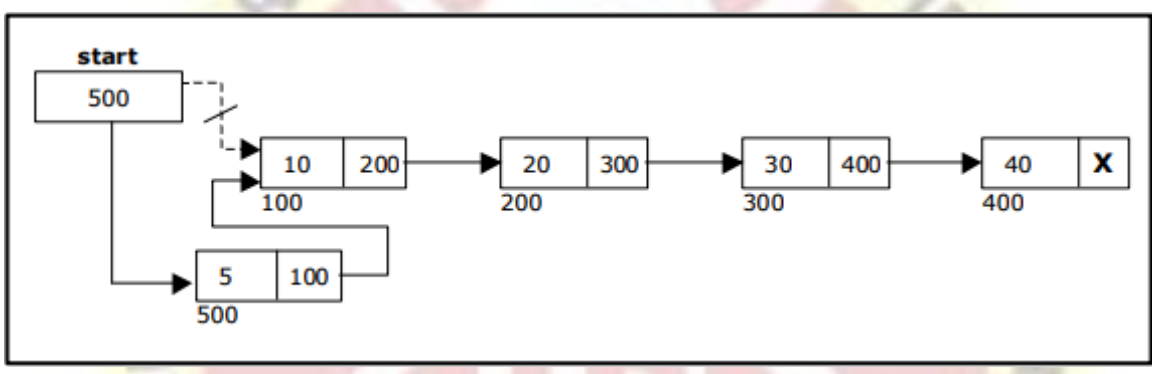


Insertion of a Node:

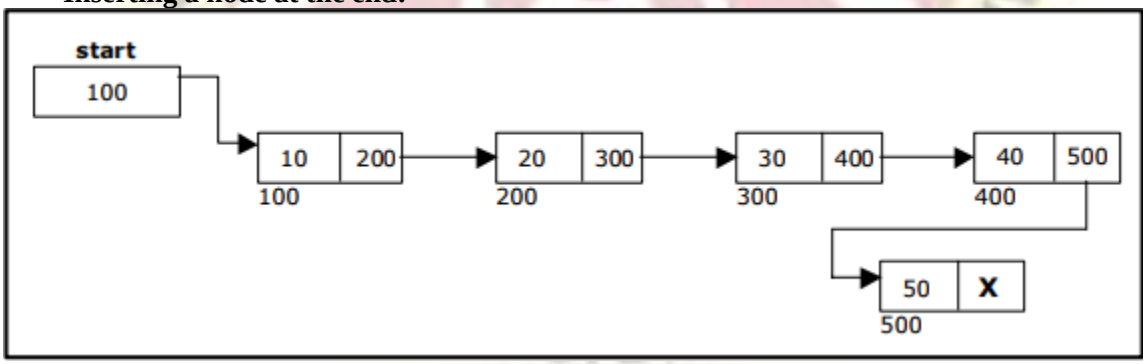
One of the most primitive operations that can be done in a singly linked list is the insertion of a node. Memory is to be allocated for the new node (in a similar way that is done while creating a list) before reading the data. The new node will contain empty data field and empty next field. The data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

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

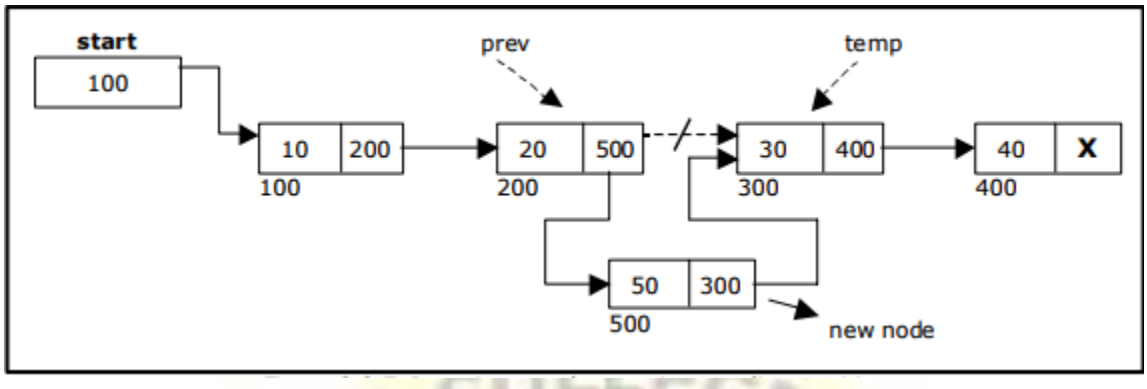
- ***Inserting a node at the beginning.***



- **Inserting a node at the end:**



- **Inserting a node into the single linked list at a specified intermediate position other than beginning and end.**

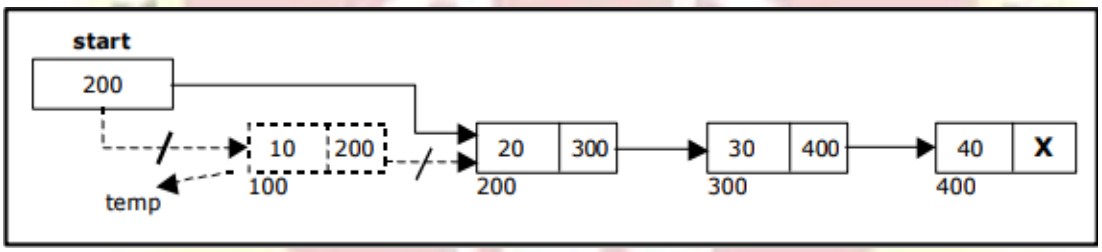


Deletion of a node:

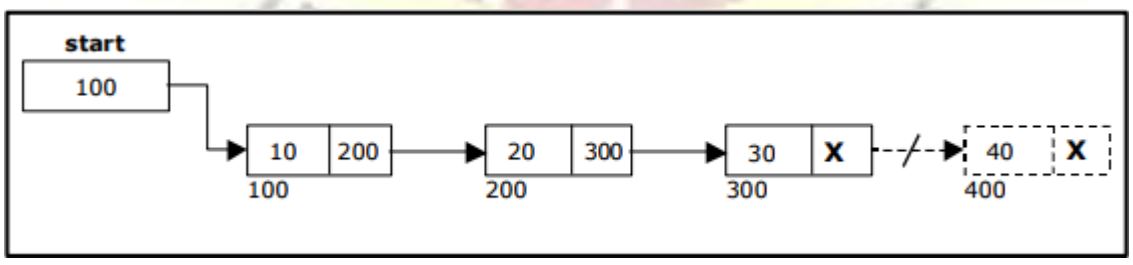
Another primitive operation that can be done in a singly linked list is the deletion of a node. Memory is to be released for the node to be deleted. A node can be deleted from the list from three different places namely.

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

Deleting a node at the beginning:

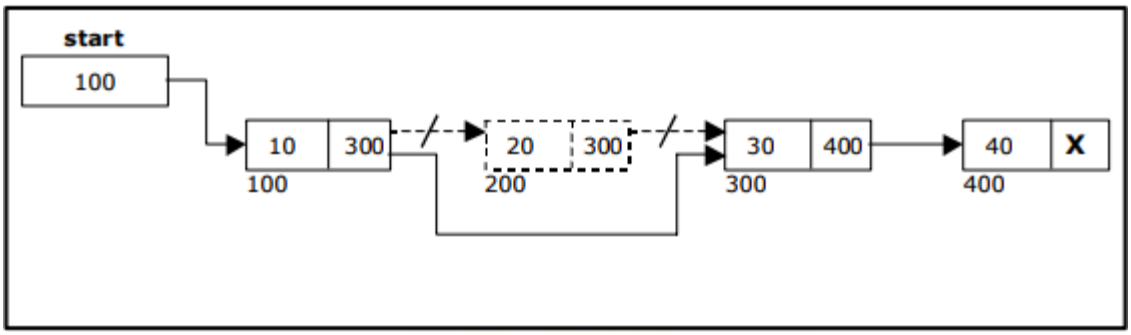


Deleting a node at the end:



Deleting a node at Intermediate position:

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



Traversal and displaying a list (Left to Right):

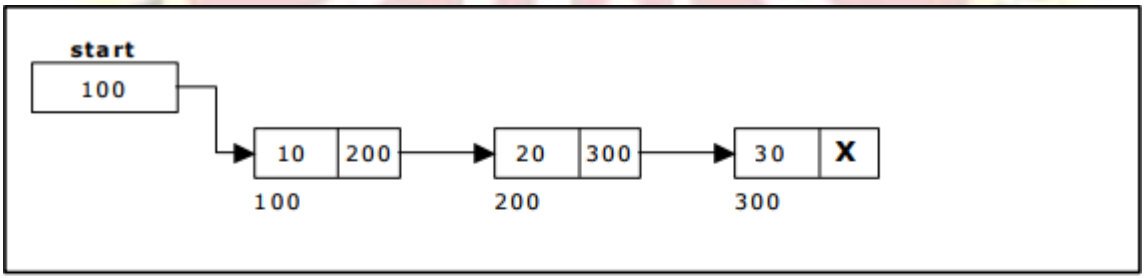
To display the information, you have to traverse (move) a linked list, node by node from the first node, until the end of the list is reached.

Traversing a list involves the following steps:

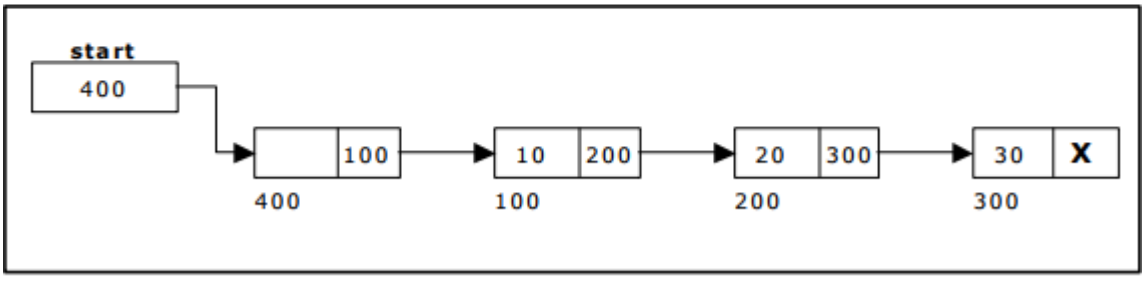
- Assign the address of start pointer to a temp pointer.
- Display the information from the data field of each node. The function traverse () is used for traversing and displaying the information stored in the list from left to right.

Using a header node:

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



Single Linked List without a header node



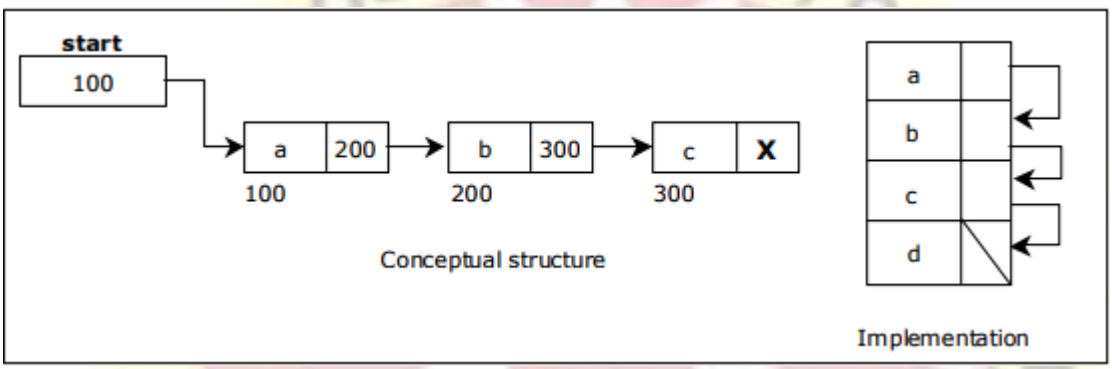
Single Linked List with header node

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

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

It is also useful when information other than that found in each node of the list is needed. For example, imagine an application in which the number of items in a list is often calculated. In a standard linked list, the list function to count the number of nodes has to traverse the entire list every time. However, if the current length is maintained in a header node, that information can be obtained very quickly. 3.5. Array based linked lists: Another alternative is to allocate the nodes in blocks. In fact, if you know the maximum size of a list a head of time, you can pre-allocate the nodes in a single array. The result is a hybrid structure – an array based linked list.

shows an example of null terminated single linked list where all the nodes are allocated contiguously in an array.



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

- Left link.
- Data.
- Right link.

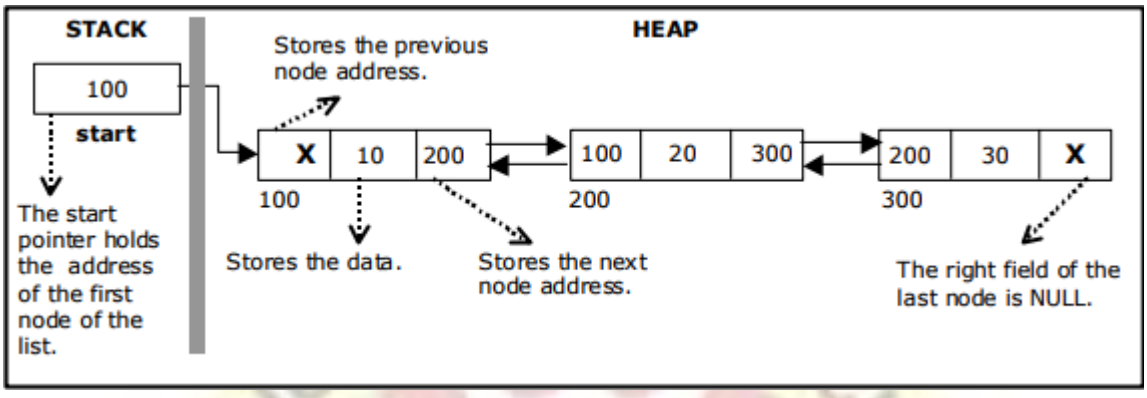
The left link points to the predecessor node and the right link points to the successor node. The data field stores the required data.

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

The basic operations in a double linked list are:

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

A double linked list is shown in figure

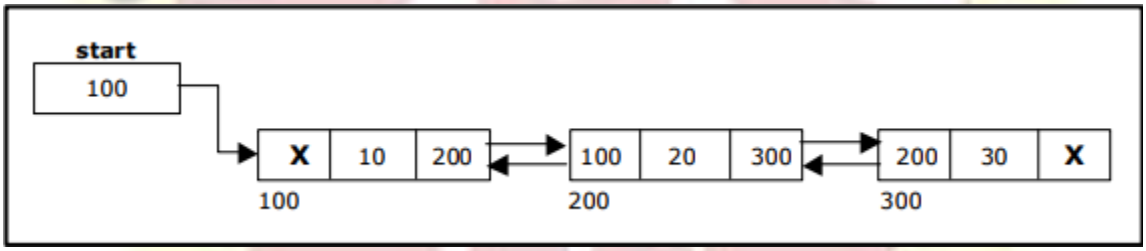


The beginning of the double linked list is stored in a "start" pointer which points to the first node. The first node's left link and last node's right link is set to NULL.

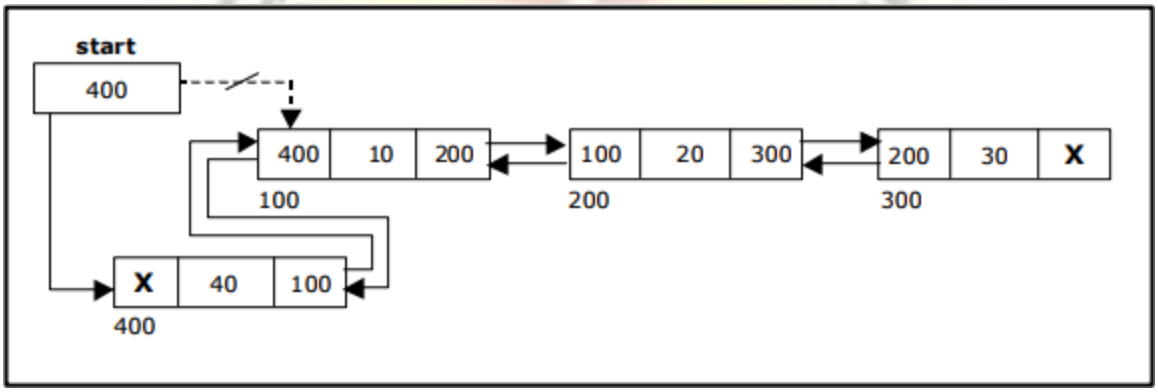
Creating a node for Double Linked List:

Creating a double linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory.

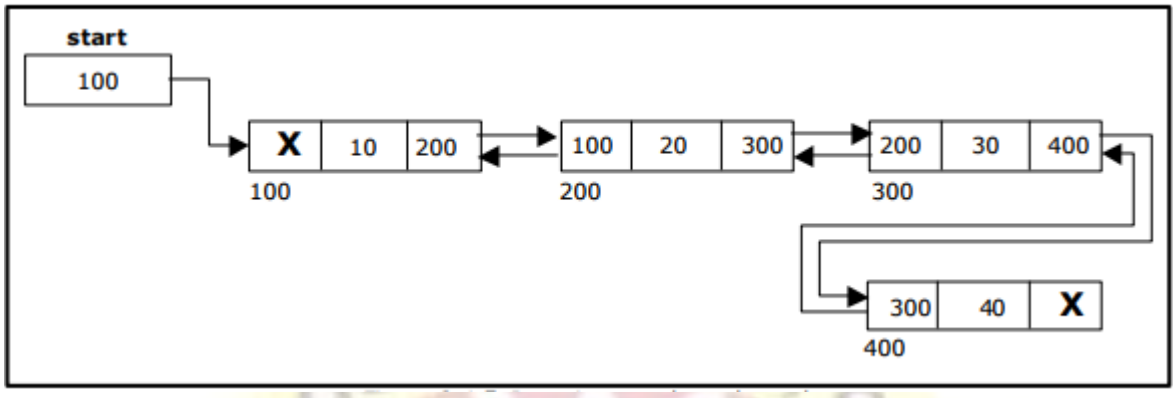
Double Linked List with 3 nodes:



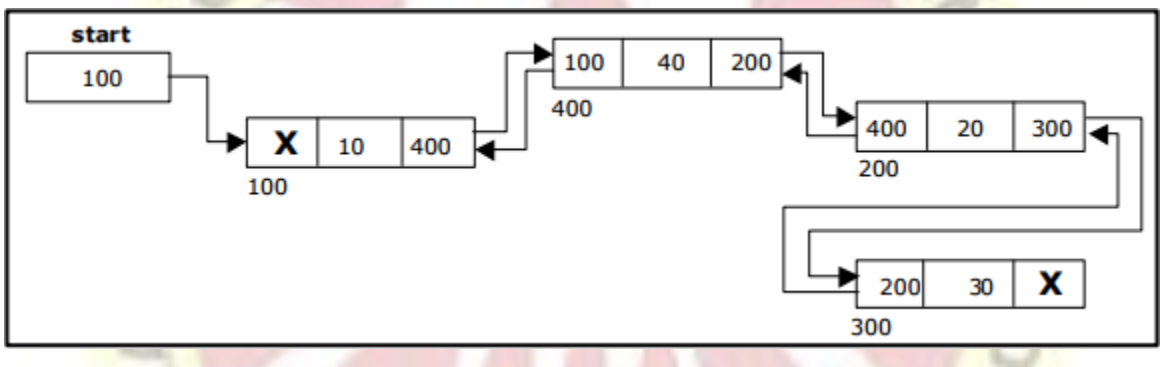
Inserting a node at the beginning:



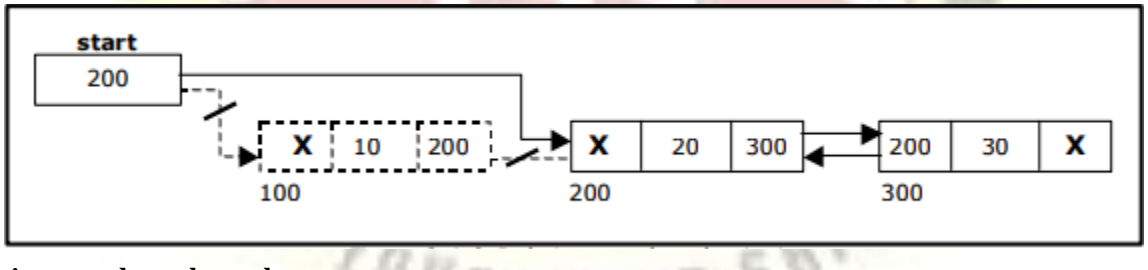
Inserting a node at the end:



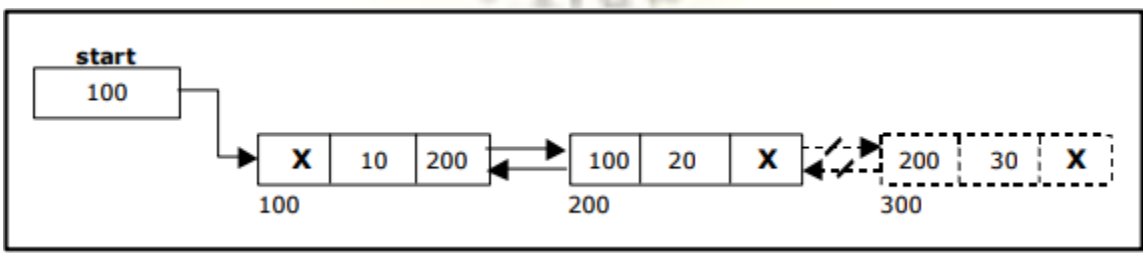
Inserting a node at an intermediate position:



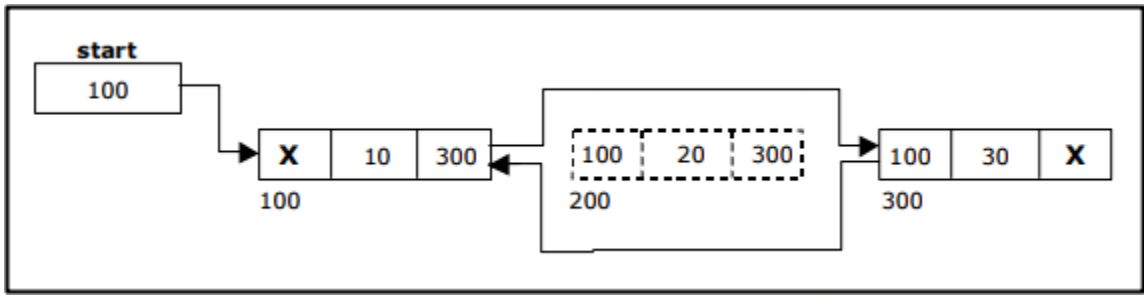
Deleting a node at the beginning:



Deleting a node at the end:



Deleting a node at Intermediate position:



Traversal and displaying a list (Left to Right):

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

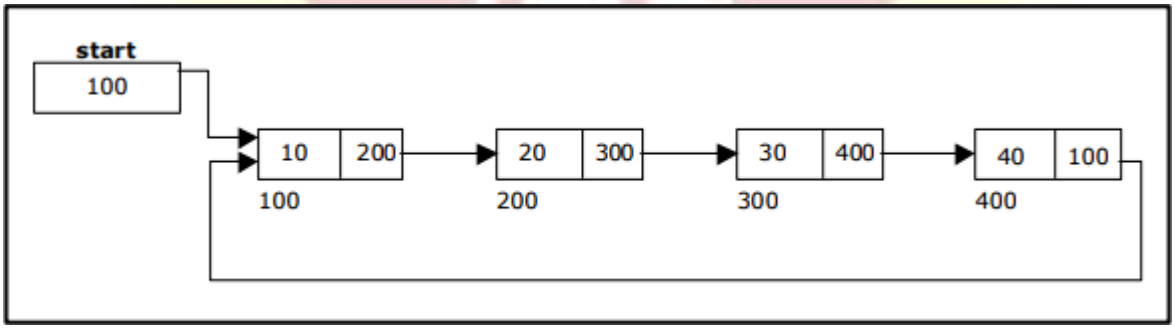
Traversal and displaying a list (Right to Left):

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

Circular Single Linked List:

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

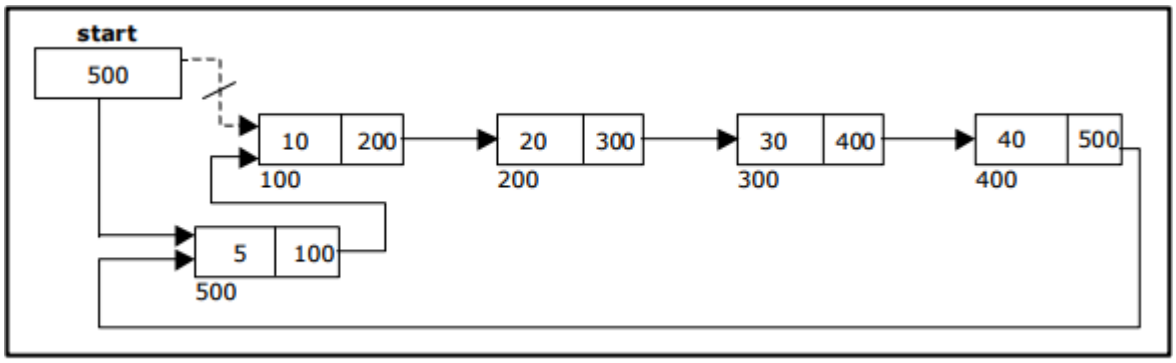
Creating a circular single Linked List with „n" number of nodes:



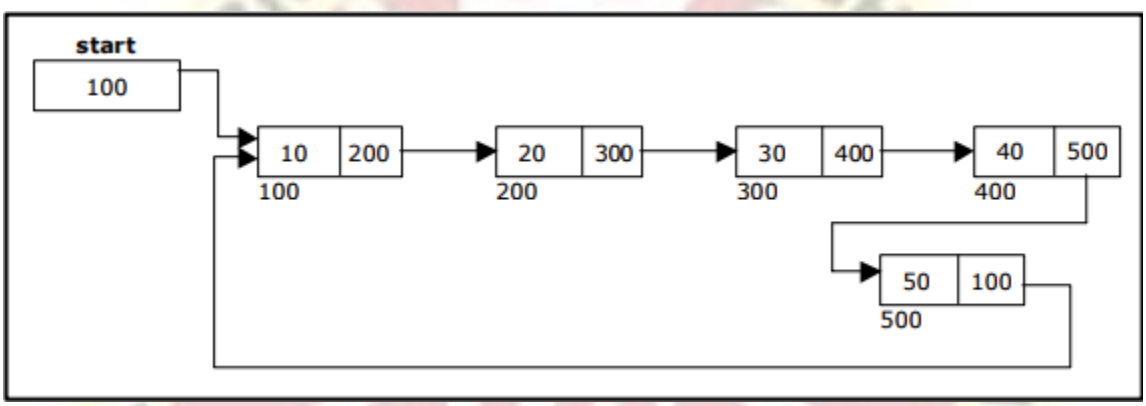
The basic operations in a circular single linked list are:

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

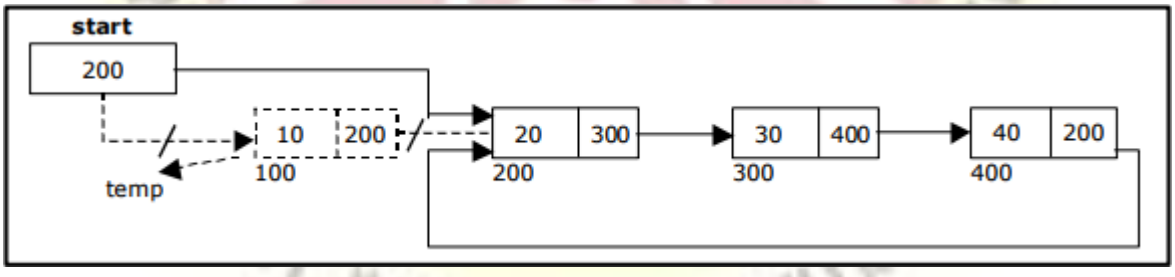
Inserting a node at the beginning:



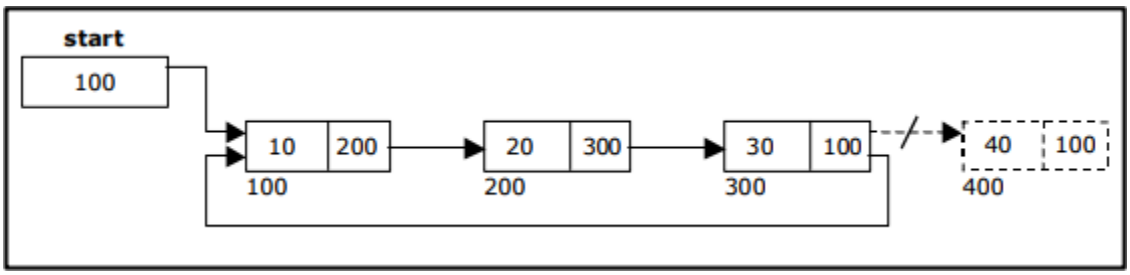
Inserting a node at the end:



Deleting a node at the beginning:



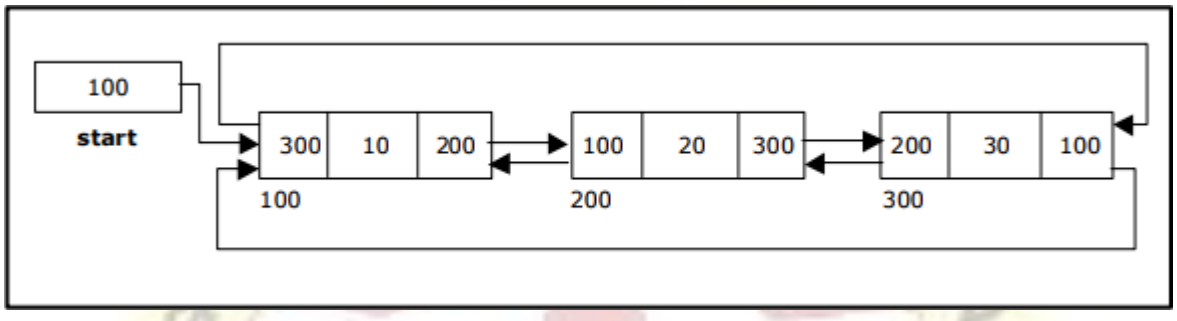
Deleting a node at the end:



Circular Double Linked List:

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

A circular double linked list is shown in figure

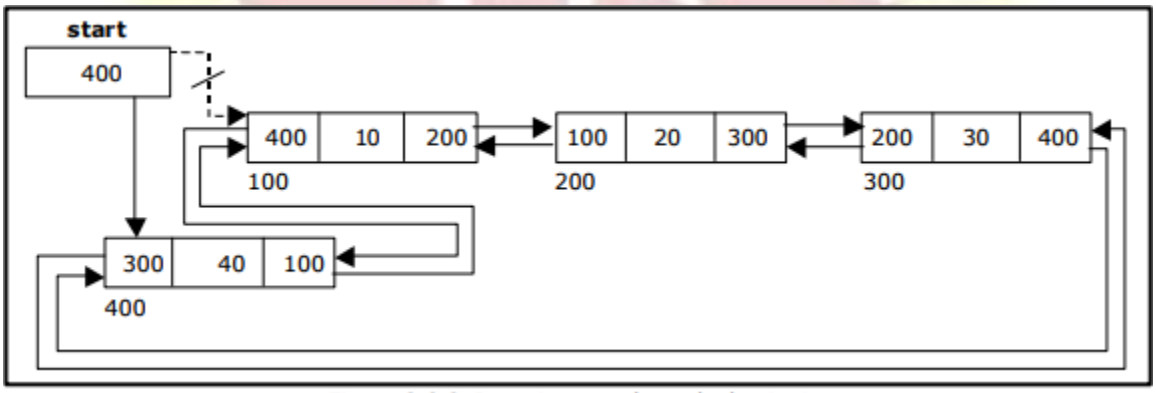


Creating a Circular Double Linked List with „n" number of nodes

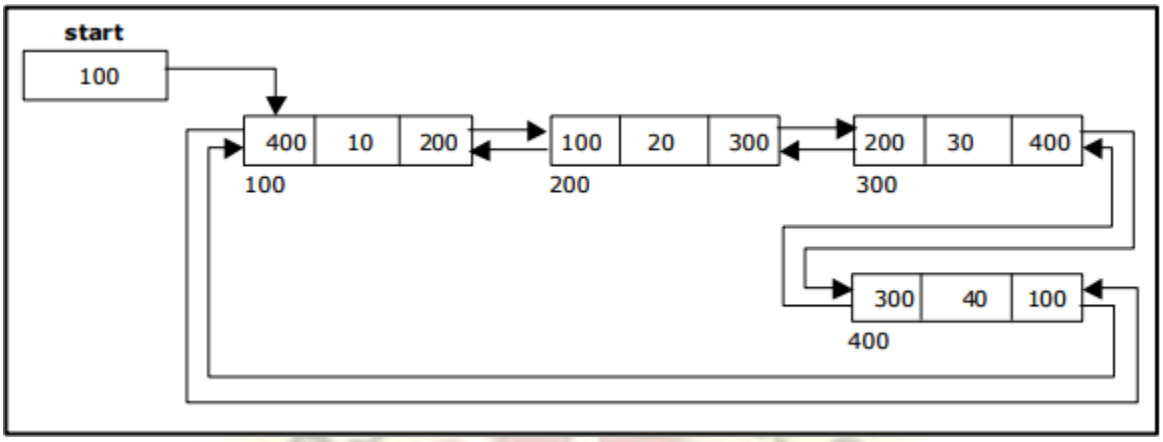
The basic operations in a circular double linked list are:

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

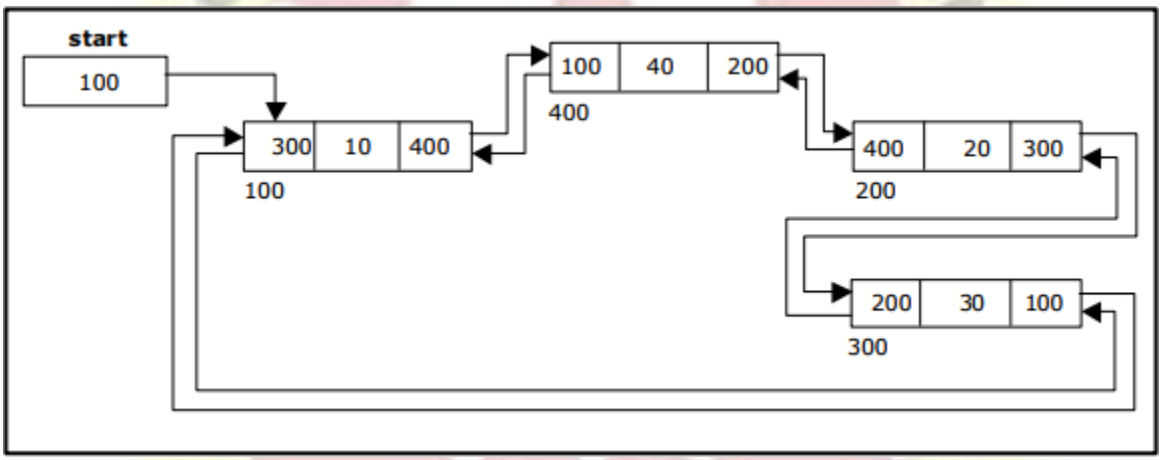
Inserting a node at the beginning:



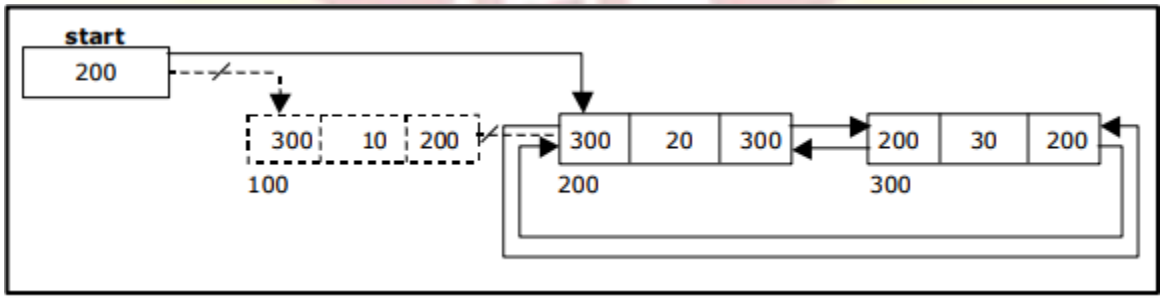
Inserting a node at the end:



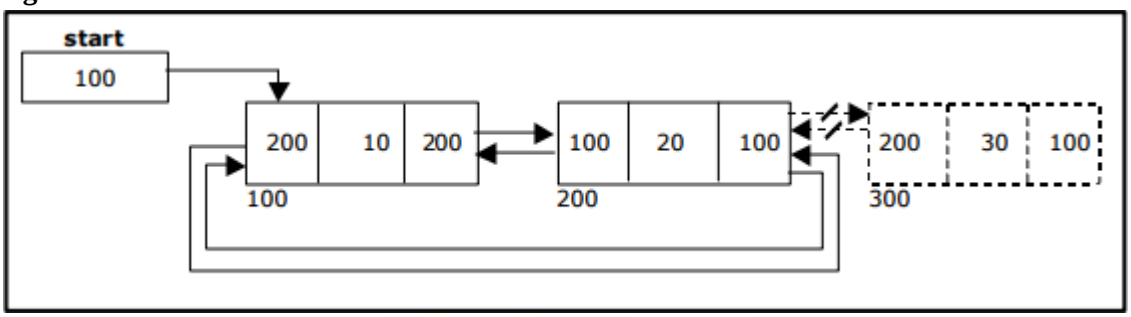
Inserting a node at an intermediate position:



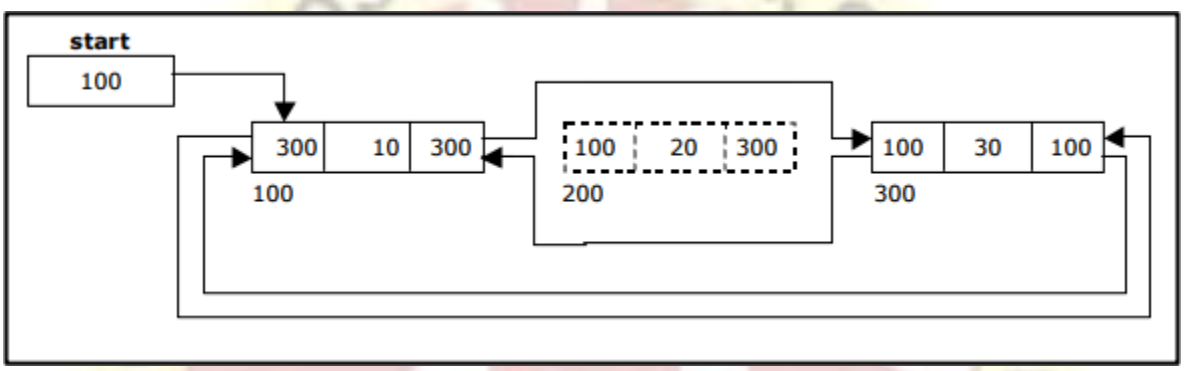
Deleting a node at the beginning:



Deleting a node at the end:



Deleting a node at Intermediate position:



Comparison of Linked List Variations:

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

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

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

Polynomials:

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

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

n is the degree of the polynomial Some examples are:

$5x^2 + 3x + 1$

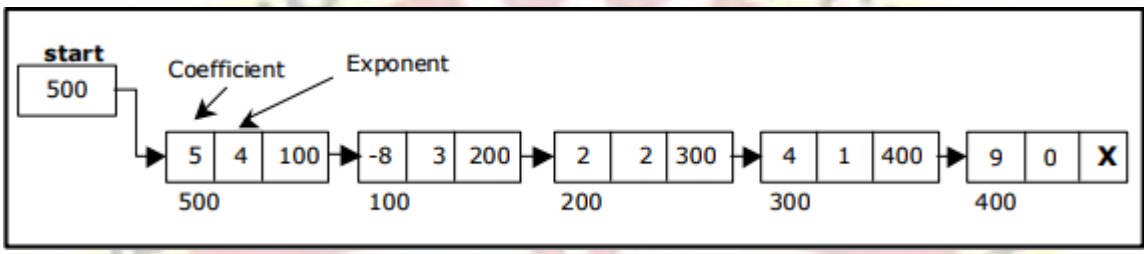
12x3 – 4x

5x4 – 8x3 + 2x2 + 4x1 + 9x0

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

The computer implementation requires implementing polynomials as a list of pairs of coefficient and exponent. Each of these pairs will constitute a structure, so a polynomial will be represented as a list of structures.

A linked list structure that represents polynomials 5x4 – 8x3 + 2x2 + 4x1 + 9x0



Addition of Polynomials:

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

To add two polynomials follow the following steps:

- Read two polynomials
- Add them.

Display the resultant polynomial.

UNIT II

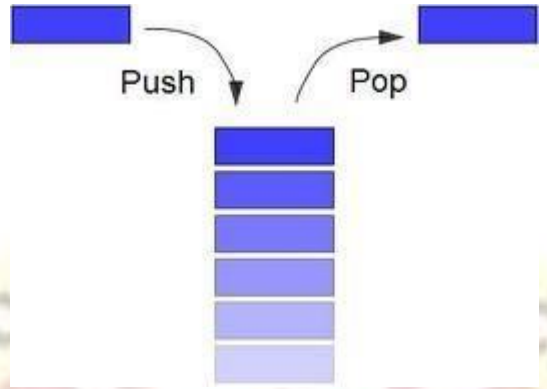
STACK ADT

Stacks Primitive Operations:

A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle. In the pushdown stacks only two operations are allowed: push the item into the stack, and pop the item out of the stack. A stack is a limited access data structure - elements can be added and removed from the stack only at the top. Push adds an item to the top of the stack, pop removes the item from the top. A helpful analogy is to think of a stack of books; you can remove only the top book, also you can add a new book on the top.

A stack may be implemented to have a bounded capacity. If the stack is full and does not

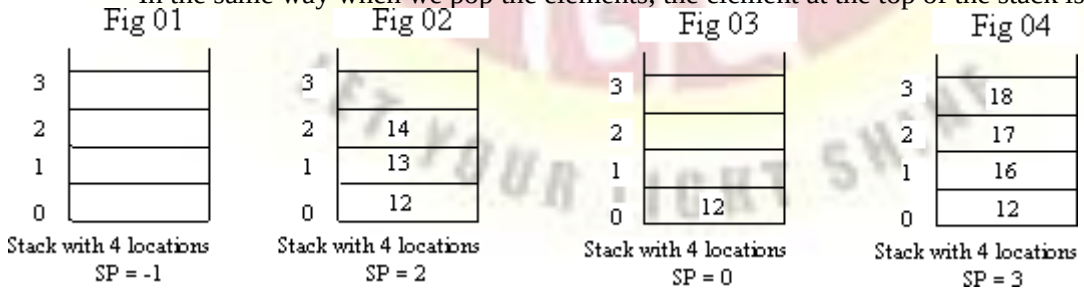
contain enough space to accept an entity to be pushed, the stack is then considered to be in an overflow state. The pop operation removes an item from the top of the stack. A pop either reveals previously concealed items or results in an empty stack, but, if the stack is empty, it goes into underflow state, which means no items are present in stack to be removed.



Stack (ADT) Data Structure:

Stack is an Abstract data structure (ADT) works on the principle Last In First Out (LIFO). The last element add to the stack is the first element to be delete. Insertion and deletion can be takes place at one end called TOP. It looks like one side closed tube.

- The add operation of the stack is called push operation
- The delete operation is called as pop operation.
- Push operation on a full stack causes stack overflow.
- Pop operation on an empty stack causes stack underflow.
- SP is a pointer, which is used to access the top element of the stack.
- If you push elements that are added at the top of the stack;
- In the same way when we pop the elements, the element at the top of the stack is deleted.



Operations of stack:

There are two operations applied on stack they are

1. push
2. pop.

While performing push & pop operations the following test must be conducted on the stack.

1) Stack is empty or not

2) Stack is full or not

Push:

Push operation is used to add new elements in to the stack. At the time of addition first check the stack is full or not. If the stack is full it generates an error message "stack overflow".

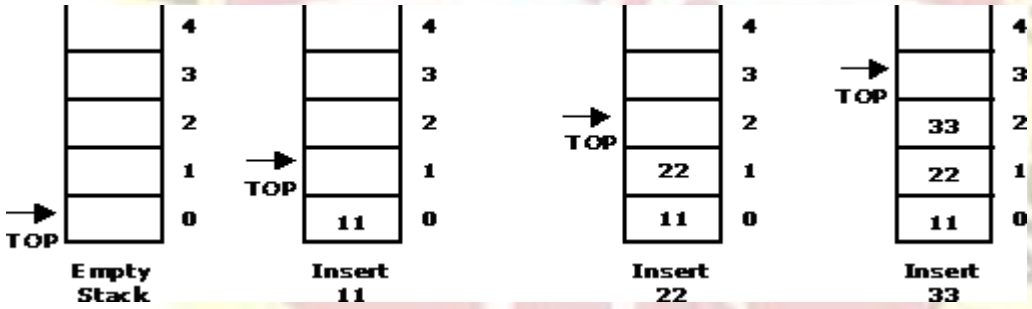
Pop:

Pop operation is used to delete elements from the stack. At the time of deletion first check the stack is empty or not. If the stack is empty it generates an error message "stack underflow".

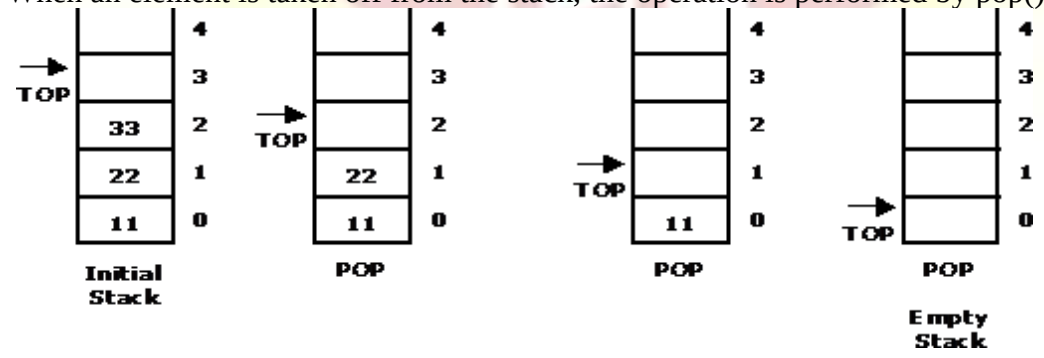
Representation of a Stack using Arrays:

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

When a element is added to a stack, the operation is performed by push().



When an element is taken off from the stack, the operation is performed by pop().



Source code for stack operations, using array:

STACK: Stack is a linear data structure which works under the principle of last in first out. Basic operations: push, pop, display.

1. **PUSH**: if (top==MAX), display **Stack overflow** else reading the data and making stack [top] =data and incrementing the top value by doing top++.

2. **POP:** if (top==0), display **Stack underflow** else printing the element at the top of the stack and decrementing the top value by doing the top.

3. **DISPLAY**: IF (TOP==0), display **Stack is empty** else printing the elements in the stack from stack [0] to stack [top].

Stack Applications:

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

In-fix- to Postfix Transformation:

Procedure:

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

1. Scan the infix expression from left to right.
2. a) If the scanned symbol is left parenthesis, push it onto the stack.

   b) If the scanned symbol is an operand, then place directly in the postfix expression (output).

   c) If the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.

   d) If the scanned symbol is an operator, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (or equal) to the precedence of the scanned operator and push the scanned operator onto the stack otherwise, push the scanned operator onto the stack.

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

Symbol	Postfix string	Stack	Remarks
A	A		
+	A	+	
B	A B	+	
*	A B	+ *	

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

Evaluating Arithmetic Expressions:

Procedure:

The postfix expression is evaluated easily by the use of a stack. When a number is seen, it is pushed onto the stack; when an operator is seen, the operator is applied to the two numbers that are popped from the stack and the result is pushed onto the stack.

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

Symbol	Operand 1	Operand 2	Value	Stack	Remarks
6				6	
5				6, 5	
2				6, 5, 2	
3				6, 5, 2, 3	The first four symbols are placed on the stack.
+	2	3	5	6, 5, 5	Next a „+" is read, so 3 and 2 are popped from the stack and their sum 5, is pushed
8	2	3	5	6, 5, 5, 8	Next 8 is pushed
*	5	8	40	6, 5, 40	Now a „*" is seen, so 8 and 5 are popped as 8 * 5 = 40 is pushed
+	5	40	45	6, 45	Next, a „+" is seen, so 40 and 5 are popped and 40 + 5 = 45 is pushed

3	5	40	45	6, 45, 3	Now, 3 is pushed
+	45	3	48	6, 48	Next, „+“ pops 3 and 45 and pushes 45 + 3 = 48 is pushed
*	6	48	288	**288**	Finally, a „*“ is seen and 48 and 6 are popped, the result 6 * 48 = 288 is pushed

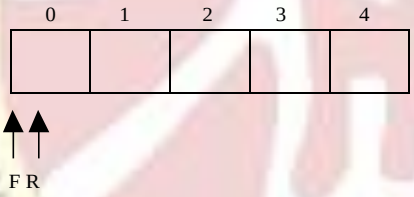
QUEUE

Basic Queue Operations:

A queue is a data structure that is best described as "first in, first out". A queue is another special kind of list, where items are inserted at one end called the rear and deleted at the other end called the front. A real world example of a queue is people waiting in line at the bank. As each person enters the bank, he or she is "enqueued" at the back of the line. When a teller becomes available, they are "dequeued" at the front of the line.

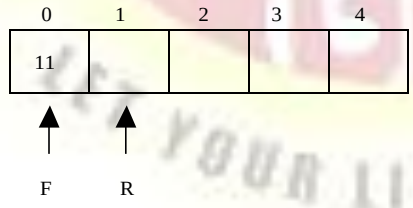
Representation of a Queue using Array:

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



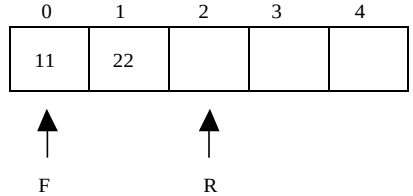
Queue Empty
F RO NT = RE A R = 0

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



RE A R = RE A R + 1 = 1 F RO
NT = 0

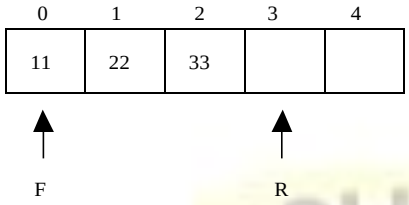
Next, insert 22 to the queue. Then the queue status is:



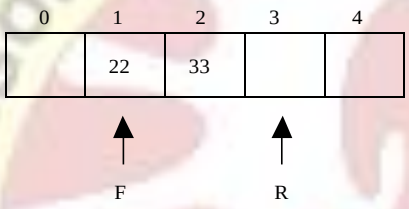
RE A R = RE A R + 1 = 2 F RO
NT = 0

RE A R = RE A R + 1 = 3 F RO NT = 0

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

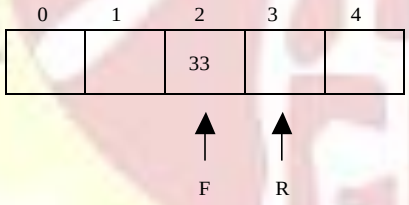


Now, delete an element. The element deleted is the element at the front of the queue. So the status of the queue is:



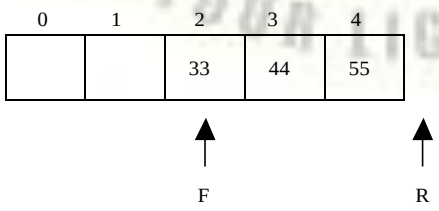
RE A R = 3
F RO NT = F R O NT + 1 = 1

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



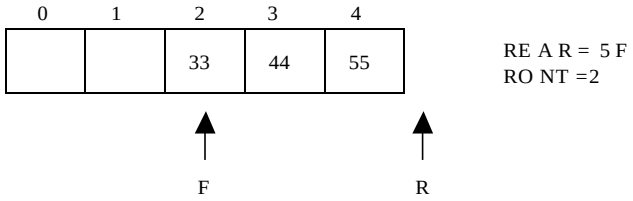
RE A R = 3
F RO NT = F R O NT + 1 = 2

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

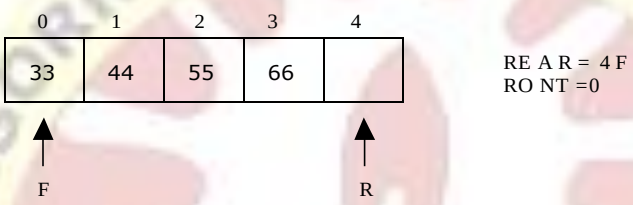


RE A R = 5 F RO NT =2

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



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



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

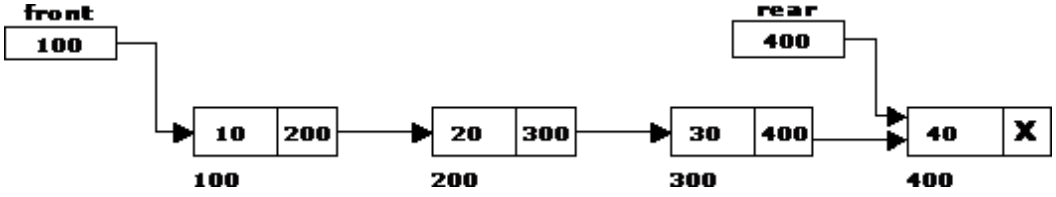
Procedure for Queue operations using array:

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

The various queue operations to perform creation, deletion and display the elements in a queue are as follows:

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

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



Applications of Queues:

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

Disadvantages of Linear Queue:

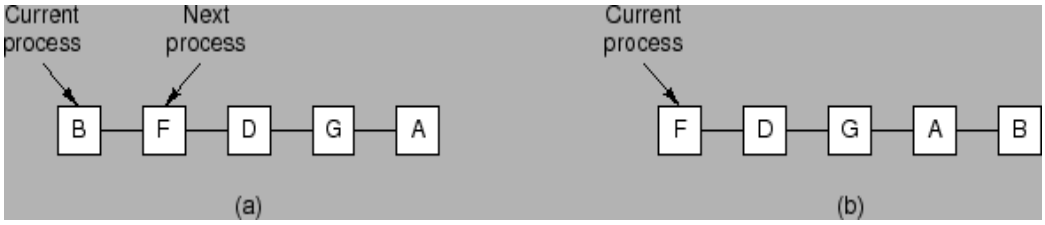
There are two problems associated with linear queue. They are:

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

Round Robin Algorithm:

The round-robin (RR) scheduling algorithm is designed especially for time-sharing systems. It is similar to FCFS scheduling, but pre-emption is added to switch between processes. A small unit of time, called a time quantum or time slices, is defined. A time quantum is generally from 10 to 100 milliseconds. The ready queue is treated as a circular queue. To implement RR scheduling

- We keep the ready queue as a FIFO queue of processes.
- New processes are added to the tail of the ready queue.
- The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.
- The process may have a CPU burst of less than 1 time quantum.
    - In this case, the process itself will release the CPU voluntarily.
    - The scheduler will then proceed to the next process in the ready queue.
- Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum,
    - The timer will go off and will cause an interrupt to the OS.
    - A context switch will be executed, and the process will be put at the tail of the ready queue.
    - The CPU scheduler will then select the next process in the ready queue.



The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds: (a time quantum of 4 milliseconds)

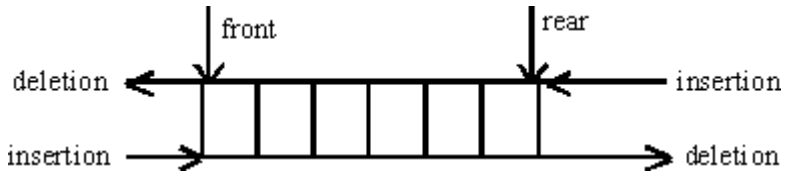
Process	Burst Time	Waiting Time	Turnaround Time
	24	6	30
	3	4	7
	3	7	10
Average	-	5.66	15.66

Using round-robin scheduling, we would schedule these processes according to the following chart:

DEQUE(Double Ended Queue):

A **double-ended queue** (**dequeue**, often abbreviated to **deque**, pronounced *deck*) generalizes a queue, for which elements can be added to or removed from either the front (head) or back (tail).It is also often called a **head-tail linked list.** Like an ordinary queue, a double-ended queue is a data structure it supports the following operations: enq_front, enq_back, deq_front, deq_back, and empty. Dequeue can be behave like a queue by using only enq_front and deq_front , and behaves like a stack by using only enq_front and deq_rear.

The DeQueue is represented as follows.



DEQUE can be represented in two ways they are

1) Input restricted DEQUE(IRD)

2) output restricted DEQUE(ORD)

The output restricted DEQUE allows deletions from only one end and input restricted DEQUE allow insertions at only one end. The DEQUE can be constructed in two ways they are

1) Using array

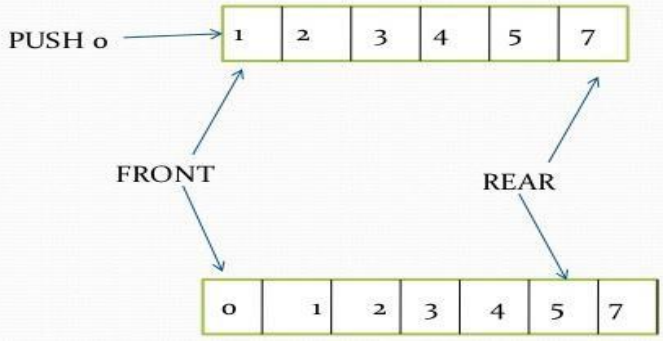
2)Using linked list

Operations in DEQUE

1. Insert element at back
2. Insert element at front
3. Remove element at front
4. Remove element at back

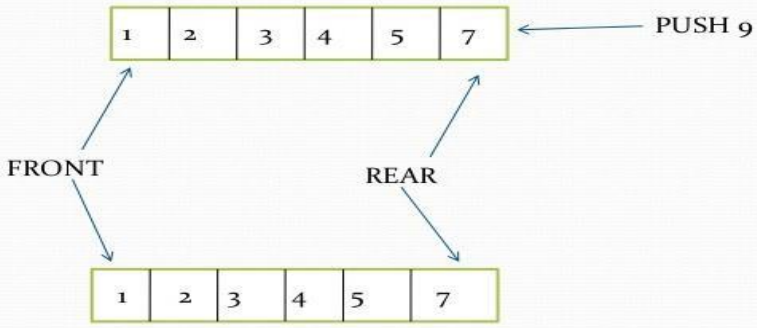
Insert_front

- insert_front() is a operation used to push an element into the front of the *Deque*.



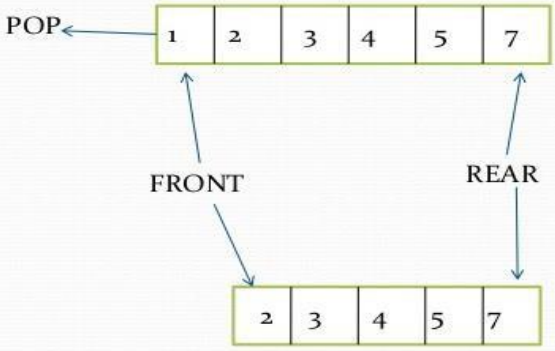
Insert_back

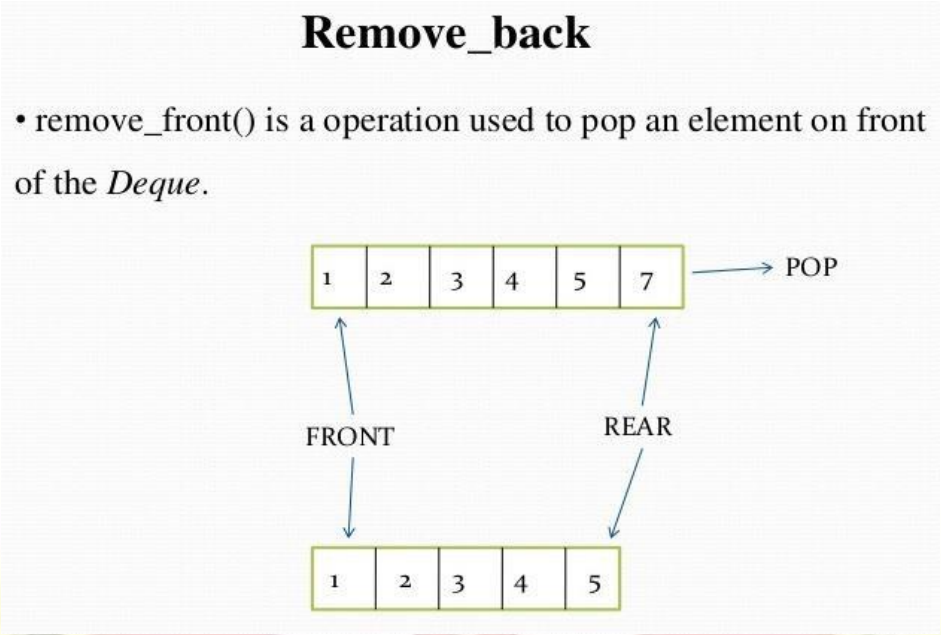
- insert_back() is a operation used to push an element at the back of a *Deque*.



Remove_front

- remove_front() is a operation used to pop an element on front of the *Deque*.





Applications of DEQUE:

1. The A-Steal algorithm implements task scheduling for several processors (multiprocessor scheduling).
2. The processor gets the first element from the deque.
3. When one of the processor completes execution of its own threads it can steal a thread from another processor.
4. It gets the last element from the deque of another processor and executes it.

Circular Queue:

Circular queue is a linear data structure. It follows FIFO principle. In circular queue the last node is connected back to the first node to make a circle.

- Circular linked list fallow the First In First Out principle
- Elements are added at the rear end and the elements are deleted at front end of the queue
- Both the front and the rear pointers points to the beginning of the array.
- It is also called as "Ring buffer".
- Items can inserted and deleted from a queue in O(1) time.

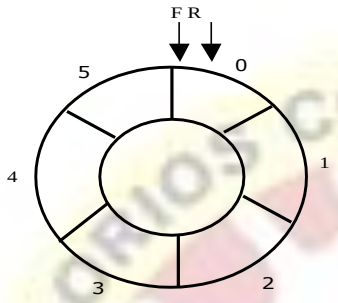
Circular Queue can be created in three ways they are

1. Using single linked list

2. Using double linked list
3. Using arrays

Representation of Circular Queue:

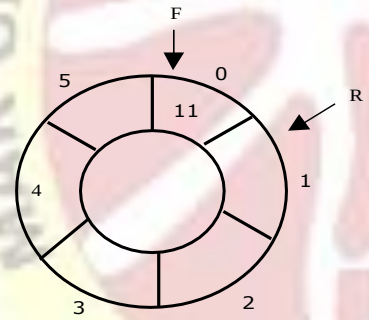
Let us consider a circular queue, which can hold maximum (MAX) of six elements. Initially the queue is empty.



Circular Queue

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

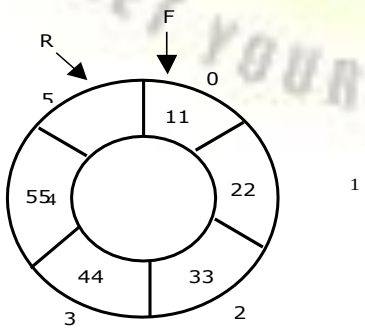
Now, insert 11 to the circular queue. Then circular queue status will be:



Circular Queue

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

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

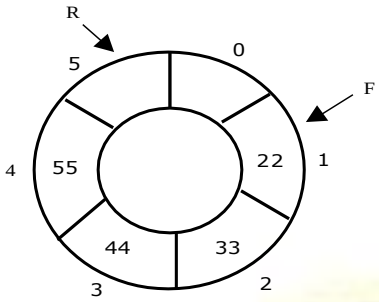


Circular Queue

FRONT = 0, REAR = 5
REAR = REAR % 6 = 5
COUNT = 5

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

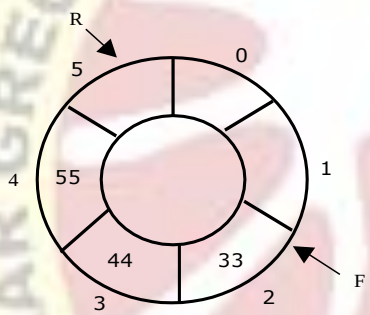
11 is deleted. The circular queue status is as follows:



Circular Queue

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

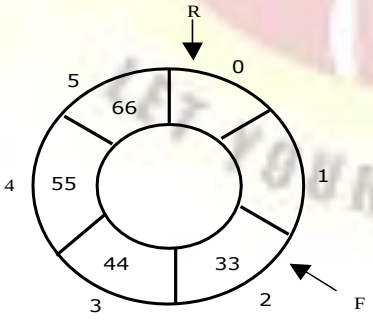
Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The circular queue status is as follows:



Circular Queue

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

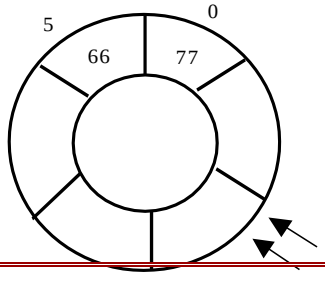
Again, insert another element 66 to the circular queue. The status of the circular queue is:



Circular Queue

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

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



UNIT III

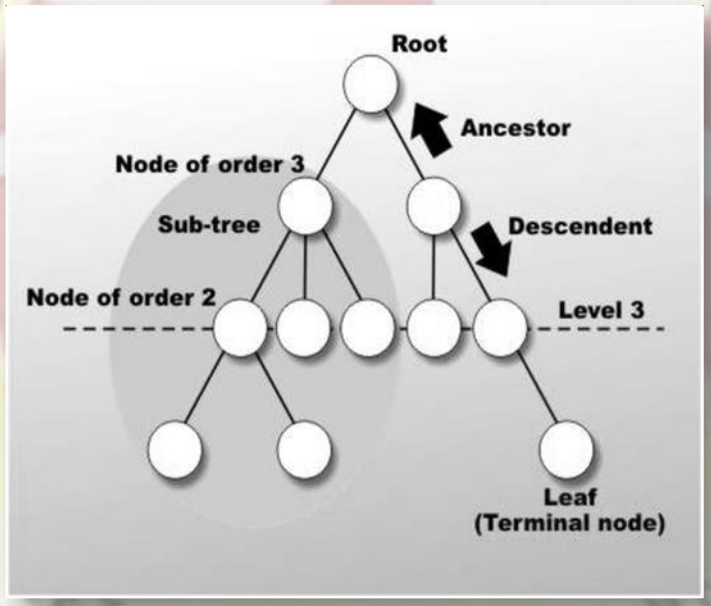
TREE ADT

Trees Basic Concepts:

A **tree** is a non-empty set one element of which is designated the root of the tree while the remaining elements are partitioned into non-empty sets each of which is a sub-tree of the root.

A tree T is a set of nodes storing elements such that the nodes have a parent-child relationship that satisfies the following

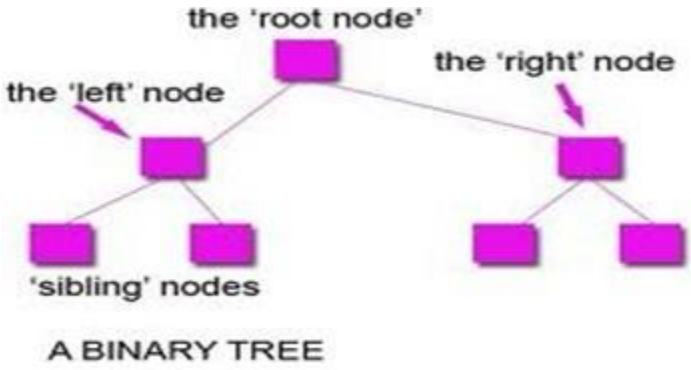
- If T is not empty, T has a special tree called the root that has no parent.
- Each node v of T different than the root has a unique parent node w; each node with parent w is a child of w.



Tree nodes have many useful properties. The **depth** of a node is the length of the path (or the number of edges) from the root to that node. The **height** of a node is the longest path from that node to its leaves. The height of a tree is the height of the root. A **leaf node** has no children -- its only path is up to its parent.

Binary Tree:

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



A BINARY TREE

Tree Terminology:

Leaf node

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

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

Siblings: The children of the same parent are called siblings.

Ancestor and Descendent If there is a path from node A to node B, then A is called an ancestor of B and *B* is called a descendent of A.

Subtree: Any node of a tree, with all of its descendants is a subtree.

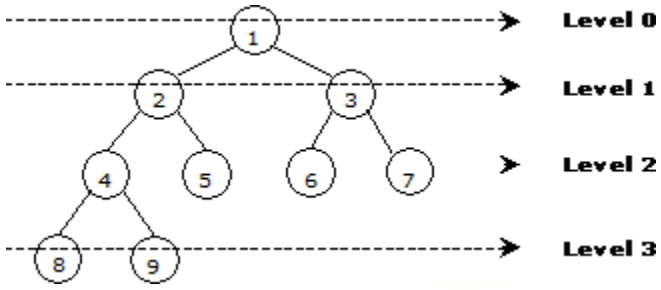
Level: The level of the node refers to its distance from the root. The root of the tree has level O, and the level of any other node in the tree is one more than the level of its parent.

The maximum number of nodes at any level is 2^n.

Height:The maximum level in a tree determines its height. The height of a node in a tree is the length of a longest path from the node to a leaf. The term depth is also used to denote height of the tree.

Depth: The depth of a node is the number of nodes along the path from the root to that node.

Assigning level numbers and Numbering of nodes for a binary tree: The nodes of a binary tree can be numbered in a natural way, level by level, left to right. The nodes of a complete binary tree can be numbered so that the root is assigned the number 1, a left child is assigned twice the number assigned its parent, and a right child is assigned one more than twice the number assigned its parent.



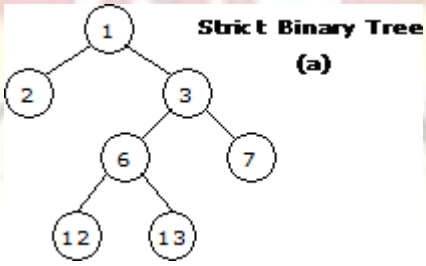
Properties of Binary Trees:

Some of the important properties of a binary tree are as follows:

1. If *h* = height of a binary tree, then
    a. Maximum number of leaves = 2^h
    b. Maximum number of nodes = $2^{h+1} - 1$
2. If a binary tree contains m nodes at level l, it contains at most 2m nodes at level l + 1.
3. Since a binary tree can contain at most one node at level 0 (the root), it can contain at most 2^l node at level l.
4. The total number of edges in a full binary tree with n node is n - 1.

Strictly Binary tree:

If every non-leaf node in a binary tree has nonempty left and right subtrees, the tree is termed a strictly binary tree. Thus the tree of figure 7.2.3(a) is strictly binary. A strictly binary tree with n leaves always contains 2n - 1 nodes.

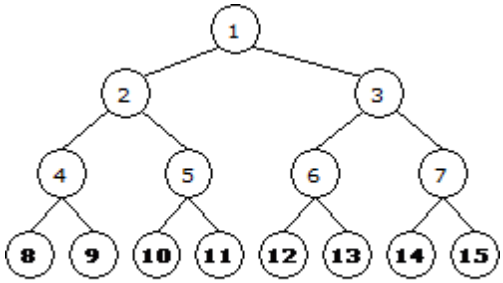


Full Binary Tree:

A full binary tree of height h has all its leaves at level h. Alternatively; All non leaf nodes of a full binary tree have two children, and the leaf nodes have no children.

A full binary tree with height *h* has $2^{h+1} - 1$ nodes. A full binary tree of height h is a *strictly*

binary tree all of whose leaves are at level h.



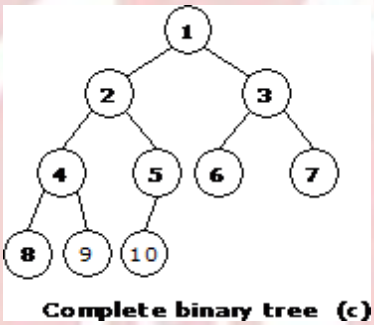
Full binary tree (d)

For example, a full binary tree of height 3 contains $2^{3+1} - 1 = 15$ nodes.

Complete Binary Tree:

A binary tree with *n* nodes is said to be **complete** if it contains all the first *n* nodes of the above numbering scheme.

A complete binary tree of height h looks like a full binary tree down to level h-1, and the level h is filled from left to right.



Complete binary tree (c)

Perfect Binary Tree:

A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at same level.
Following are examples of Perfect Binary Trees.

```
        18
      /    \
    15      30
   / \      / \
```

40 50 100 40

18

/ \

15 30

A Perfect Binary Tree of height h (where height is number of nodes on path from root to leaf) has $2^h - 1$ node.

Example of Perfect binary tree is ancestors in family. Keep a person at root, parents as children, parents of parents as their children.

Balanced Binary Tree:

A binary tree is balanced if height of the tree is O(Log n) where n is number of nodes. For Example, AVL tree maintain O(Log n) height by making sure that the difference between heights of left and right subtrees is 1. Red-Black trees maintain O(Log n) height by making sure that the number of Black nodes on every root to leaf paths are same and there are no adjacent red nodes. Balanced Binary Search trees are performance wise good as they provide O(log n) time for search, insert and delete.

Representation of Binary Trees:

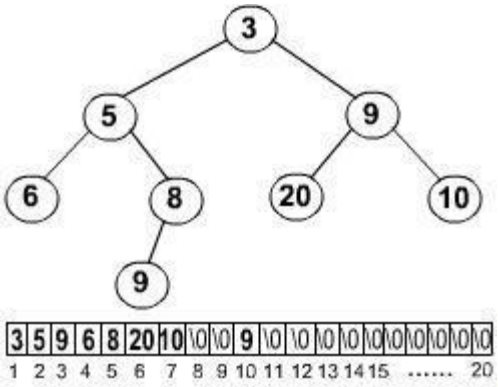
1. Array Representation of Binary Tree
2. Pointer-based.

Array Representation of Binary Tree:

A single array can be used to represent a binary tree.

For these nodes are numbered / indexed according to a scheme giving 0 to root. Then all the nodes are numbered from left to right level by level from top to bottom. Empty nodes are also numbered. Then each node having an index i is put into the array as its i^{th} element.

In the figure shown below the nodes of binary tree are numbered according to the given scheme.

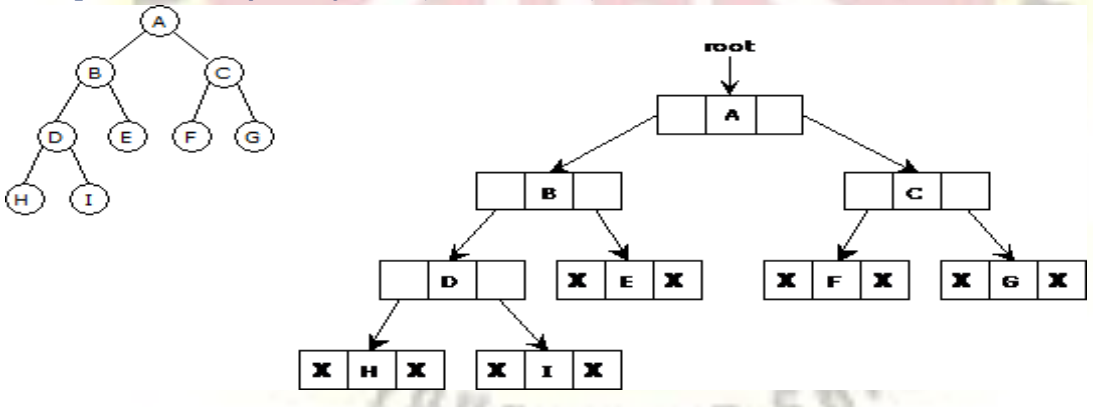


The figure shows how a binary tree is represented as an array. The root 3 is the 0^{th} element while its leftchild 5 is the 1 st element of the array. Node 6 does not have any child so its children i.e. 7 th and 8 th element of the array are shown as a Null value.

It is found that if n is the number or index of a node, then its left child occurs at $(2n + 1)^{th}$ position and right child at $(2n + 2)^{th}$ position of the array. If any node does not have any of its child, then null value is stored at the corresponding index of the array.

The following program implements the above binary tree in an array form. And then traverses the tree in inorder traversal.

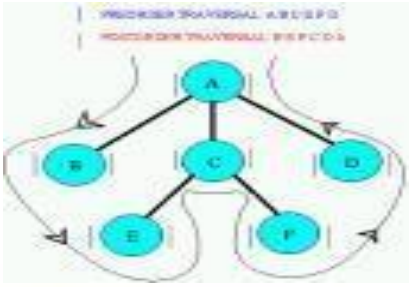
Linked Representation of Binary Tree (Pointer based):



Binary trees can be represented by links where each node contains the address of the left child and the right child. If any node has its left or right child empty then it will have in its respective link field, a null value. A leaf node has null value in both of its links.

Binary Tree Traversals:

Traversal of a binary tree means to visit each node in the tree exactly once. The tree traversal is used in all t it.



In a linear list nodes are visited from first to last, but a tree being a non linear one we need definite rules. Th ways to traverse a tree. All of them differ only in the order in which they visit the nodes.

The three main methods of traversing a tree are:

- Inorder Traversal
- Preorder Traversal
- Postorder Traversal

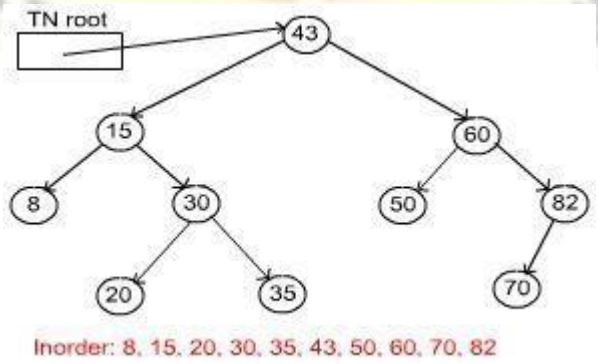
In all of them we do not require to do anything to traverse an empty tree. All the traversal methods are base functions since a binary tree is itself recursive as every child of a node in a binary tree is itself a binary tree.

Inorder Traversal:

To traverse a non empty tree in inorder the following steps are followed recursively.

- Visit the Root
- Traverse the left subtree
- Traverse the right subtree

The inorder traversal of the tree shown below is as follows.



Inorder: 8, 15, 20, 30, 35, 43, 50, 60, 70, 82

Preorder Traversal:

Algorithm Pre-order(tree)

1. Visit the root.
2. Traverse the left sub-tree, i.e., call Pre-order(left-sub-tree)
3. Traverse the right sub-tree, i.e., call Pre-order(right-sub-tree)

Post-order Traversal:

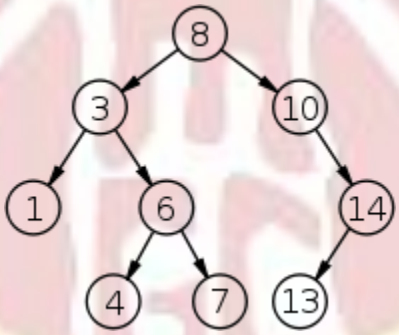
Algorithm Post-order(tree)

1. Traverse the left sub-tree, i.e., call Post-order(left-sub-tree)
2. Traverse the right sub-tree, i.e., call Post-order(right-sub-tree)
3. Visit the root.

Binary Search Tree:

Binary Search Tree, is a node-based binary tree data structure which has the following properties:

- The left sub-tree of a node contains only nodes with keys less than the node's key.
- The right sub-tree of a node contains only nodes with keys greater than the node's key.
- The left and right sub-tree each must also be a binary search tree.
  There must be no duplicate nodes.



The above properties of Binary Search Tree provide an ordering among keys so that the operations like search, minimum and maximum can be done fast. If there is no ordering, then we may have to compare every key to search a given key.

Searching a key

To search a given key in Binary Search Tree, we first compare it with root, if the key is present at root, we return root. If key is greater than root's key, we recur for right sub-tree of root node. Otherwise we recur for left sub-tree.

```
# A utility function to search a given key in
BST def search(root,key):
  # Base Cases: root is null or key is present at root
```

```
if root is None or root.val ==
    key: return root
 # Key is greater than root's
key if root.val < key:
    return search(root.right,key)
  # Key is smaller than root's
key return search(root.left,key)
```

Priority Queues

Priority Queue is an extension of queue with following properties.
1) Every item has a priority associated with it.
2) An element with high priority is dequeued before an element with low priority.
3) If two elements have the same priority, they are served according to their order in the queue.

A typical priority queue supports following operations.
insert(item, priority): Inserts an item with given priority.
getHighestPriority(): Returns the highest priority item.
deleteHighestPriority(): Removes the highest priority item.

Implementation priority queue

Using Array: A simple implementation is to use array of following structure.

insert() operation can be implemented by adding an item at end of array in O(1) time.

getHighestPriority() operation can be implemented by linearly searching the highest priority item in array. This operation takes O(n) time.

deleteHighestPriority() operation can be implemented by first linearly searching an item, then removing the item by moving all subsequent items one position back.

We can also use Linked List, time complexity of all operations with linked list remains same as array. The advantage with linked list is **deleteHighestPriority()** can be more efficient as we don't have to move items.

Applications of Priority Queue:

1) CPU Scheduling
2) Graph algorithms like Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree, etc
3) All queue applications where priority is involved.

Application of Trees:

1. One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:

file system

_____

```
    / <-- root
  /    \
...      home
    /        \
  ugrad       course
   /        /   |   \
  ...      cs101 cs112 cs113
```

2. If we organize keys in form of a tree (with some ordering e.g., BST), we can search for a given key in moderate time (quicker than Linked List and slower than arrays). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of $O(\log_n)$ for search.

3) We can insert/delete keys in moderate time (quicker than Arrays and slower than Unordered Linked Lists). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of $O(\log_n)$ for insertion/deletion.

4) Like Linked Lists and unlike Arrays, Pointer implementation of trees don't have an upper limit on number of nodes as nodes are linked using pointers.

The following are the common uses of tree.

1. Manipulate hierarchical data.
2. Make information easy to search (see tree traversal).
3. Manipulate sorted lists of data.
4. As a workflow for compositing digital images for visual effects.
5. Router algorithms

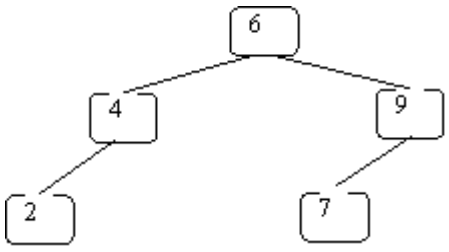
Binary Search Trees:

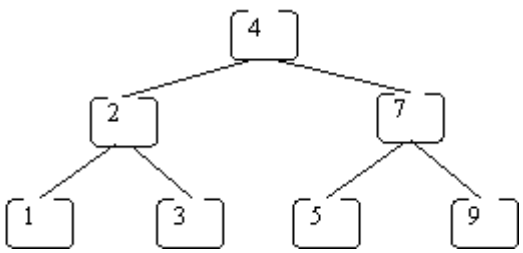
An important special kind of binary tree is the **binary search tree (BST)**. In a BST, each node stores some information including a unique **key value**, and perhaps some associated data. A binary tree is a BST iff, for every node n in the tree:

- All keys in n's left subtree are less than the key in n, and
- All keys in n's right subtree are greater than the key in n.

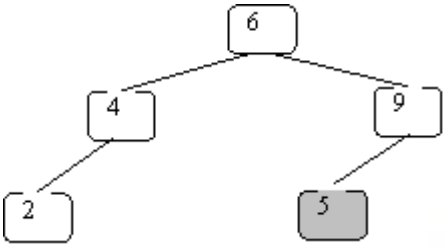
In other words, binary search trees are binary trees in which all values in the node's left subtree are less than node value all values in the node's right subtree are greater than node value.

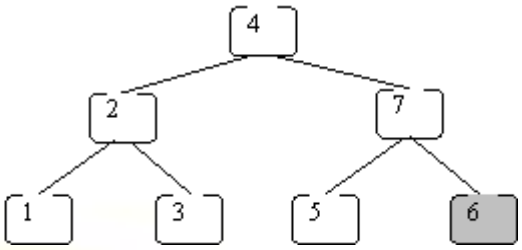
Here are some BSTs in which each node just stores an integer key:





These are not BSTs:



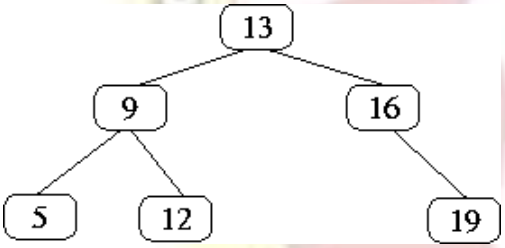


In the left one 5 is not greater than 6. In the right one 6 is not greater than 7.

The reason binary-search trees are important is that the following operations can be implemented efficiently using a BST:

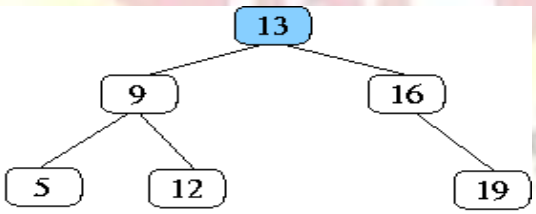
- insert a key value
- determine whether a key value is in the tree
- remove a key value from the tree
- print all of the key values in sorted order

Let's illustrate what happens using the following BST:

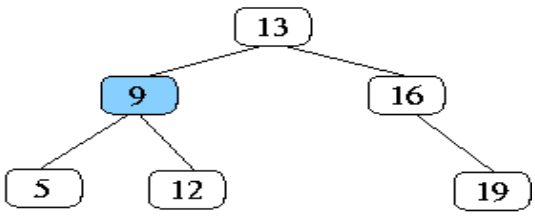


and searching for 12:

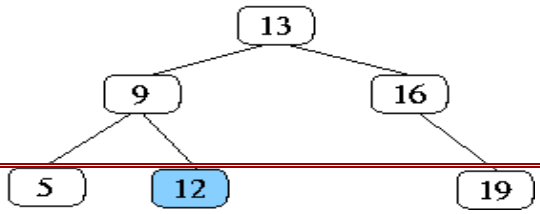
12 < 13 so go to left subtree



12 > 9 so go to right subtree.

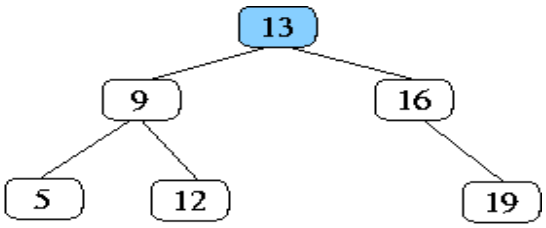


found!

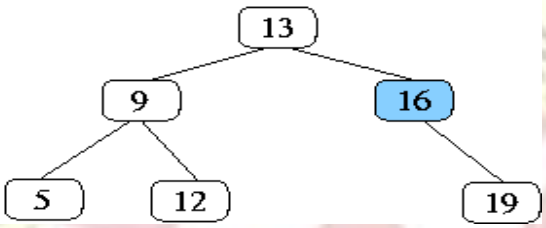


What if we search for 15:

15 > 13 so go to right subtree



15 < 16 so go to left subtree. It does not exist so search fails and it returns false



Properties and Operations:

A BST is a binary tree of nodes ordered in the following way:

1. Each node contains one key (also unique)
2. The keys in the left subtree are < (less) than the key in its parent node
3. The keys in the right subtree > (greater) than the key in its parent node
4. Duplicate node keys are not allowed.

Inserting a node

A naïve algorithm for inserting a node into a BST is that, we start from the root node, if the node to insert is less than the root, we go to left child, and otherwise we go to the right child of the root. We continue this process (each node is a root for some sub tree) until we find a null pointer (or leaf node) where we cannot go any further. We then insert the node as a left or right child of the leaf node based on node is less or greater than the leaf node. We note that a new node is always inserted as a leaf node. A recursive algorithm for inserting a node into a BST is as follows. Assume we insert a node N to tree T. if the tree is empty, the we return new node N as the tree. Otherwise, the problem of inserting is reduced to inserting the node N to left of right sub trees of T, depending on N is less or greater than T. A definition is as follows.

Insert(N, T) = N if T is empty
= insert(N, T.left) if N < T
= insert(N, T.right) if N > T

Searching for a node

Searching for a node is similar to inserting a node. We start from root, and then go left or right until we find (or not find the node). A recursive definition of search is as follows. If the node is

equal to root, then we return true. If the root is null, then we return false. Otherwise we recursively solve the problem for T.left or T.right, depending on N < T or N > T. A recursive definition is as follows.

Search should return a true or false, depending on the node is found or not. Search(N, T) = false if T is empty

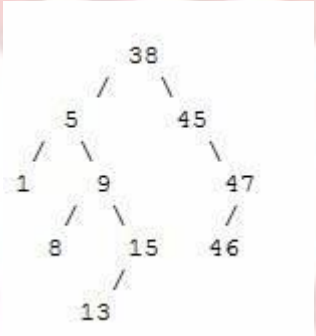
= true if T = N

= search(N, T.left) if N < T

= search(N, T.right) if N > T

Deleting a node

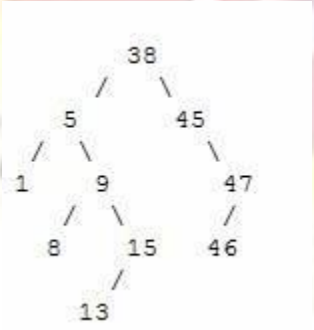
A BST is a connected structure. That is, all nodes in a tree are connected to some other node. For example, each node has a parent, unless node is the root. Therefore deleting a node could affect all sub trees of that node. For example, deleting node 5 from the tree could result in losing sub trees that are rooted at 1 and 9.



Hence we need to be careful about deleting nodes from a tree. The best way to deal with deletion seems to be considering special cases. What if the node to delete is a leaf node? What if the node is a node with just one child? What if the node is an internal node (with two children). The latter case is the hardest to resolve. But we will find a way to handle this situation as well.

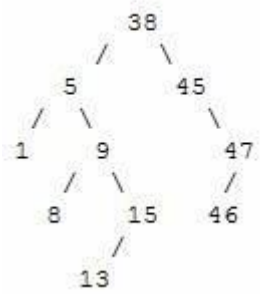
Case 1 : The node to delete is a leaf node

This is a very easy case. Just delete the node 46. We are done



Case 2 : The node to delete is a node with one child.

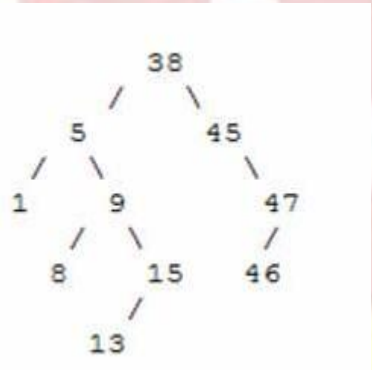
This is also not too bad. If the node to be deleted is a left child of the parent, then we connect the left pointer of the parent (of the deleted node) to the single child. Otherwise if the node to be deleted is a right child of the parent, then we connect the right pointer of the parent (of the deleted node) to single child.



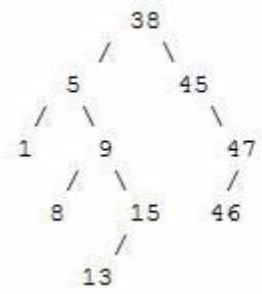
Case 3: The node to delete is a node with two children

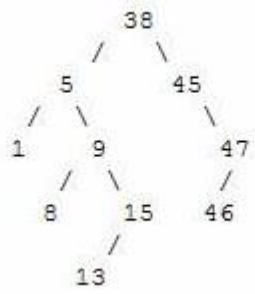
This is a difficult case as we need to deal with two sub trees. But we find an easy way to handle it. First we find a replacement node (from leaf node or nodes with one child) for the node to be deleted. We need to do this while maintaining the BST order property. Then we swap leaf node or node with one child with the node to be deleted (swap the data) and delete the leaf node or node with one child (case 1 or case 2)

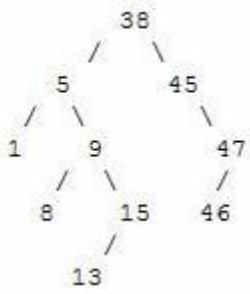
Next problem is finding a replacement leaf node for the node to be deleted. We can easily find this as follows. If the node to be deleted is N, the find the largest node in the left sub tree of N or the smallest node in the right sub tree of N. These are two candidates that can replace the node to be deleted without losing the order property. For example, consider the following tree and suppose we need to delete the root 38.



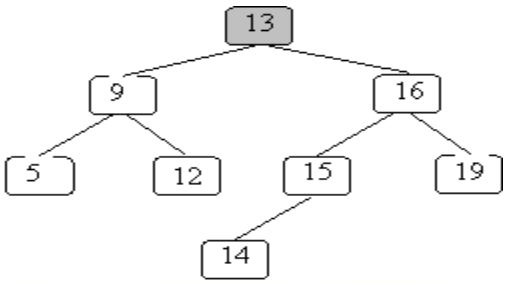
Then we find the largest node in the left sub tree (15) or smallest node in the right sub tree (45) and replace the root with that node and then delete that node. The following set of images demonstrates this process.



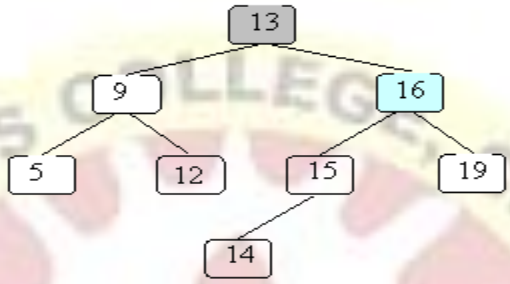




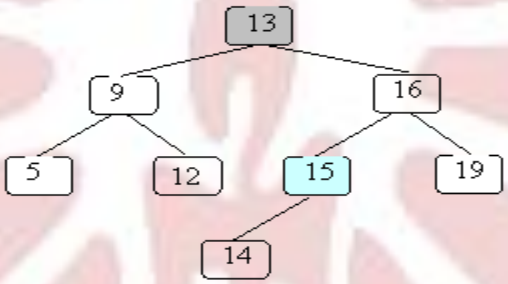
Let's see when we delete 13 from that tree.



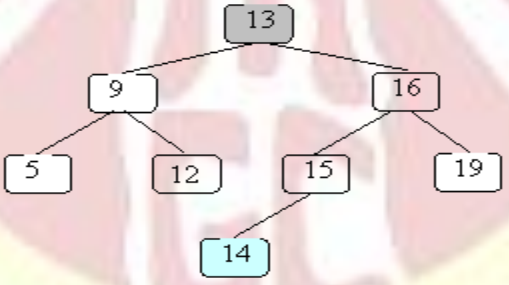
Original BST with 13 located



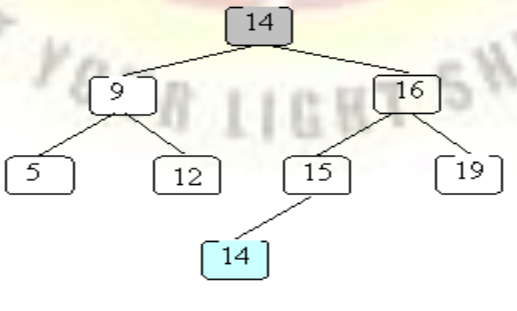
Step into right subtree.



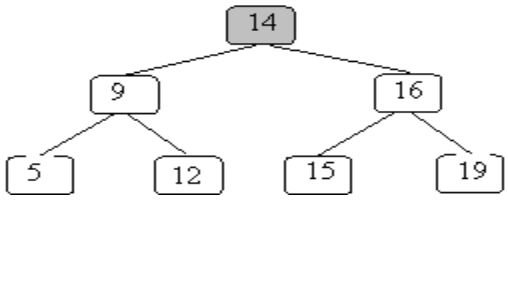
Go to left child.



Continue to left child. This is last one.



Replace node to delete with far left child of right subtree.

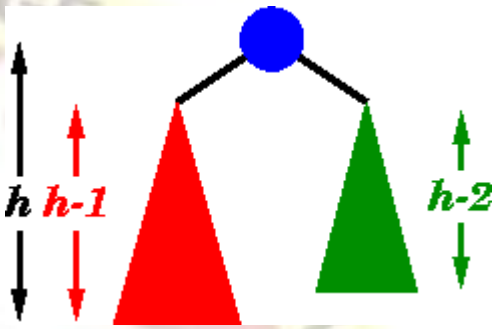


Remove far left child of right subtree.

An **AVL tree** is another balanced binary search tree. Named after their inventors, **A**delson-**V**elskii and **L**andis, they were the first dynamically balanced trees to be proposed. Like red-black trees, they are not perfectly balanced, but pairs of sub-trees differ in height by at most 1, maintaining an ***O(*log*n)*** search time. Addition and deletion operations also take ***O(*log*n)*** time.

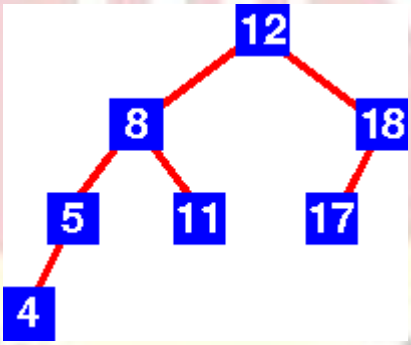
Definition of an AVL tree: An AVL tree is a binary search tree which has the following properties:

1. The sub-trees of every node differ in height by at most one.
2. Every sub-tree is an AVL tree.

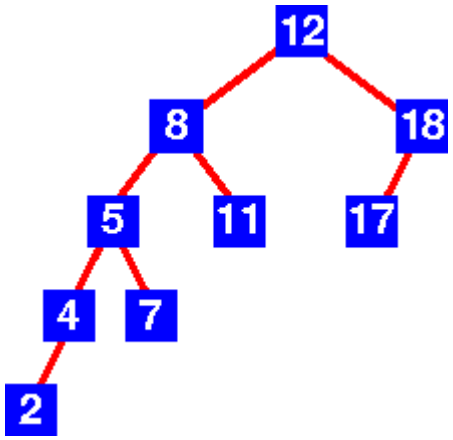


Balance requirement for an AVL tree: the left and right sub-trees differ by at most 1 in height.

For example, here are some trees:



Yes this is an AVL tree. Examination shows that *each* left sub-tree has a height 1 greater than



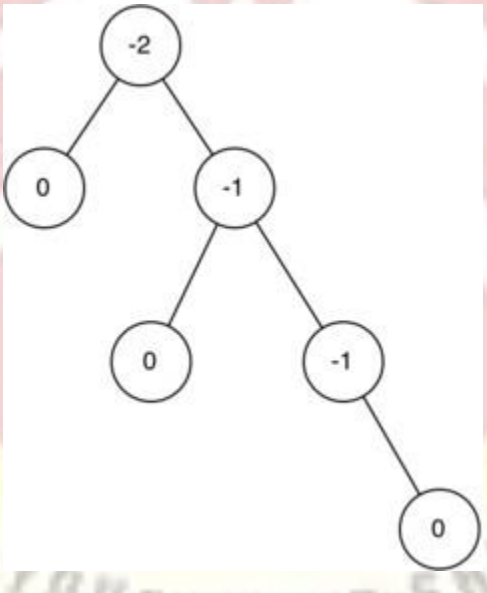
each right sub-tree.

No this is not an AVL tree. Sub-tree with root 8 has height 4 and sub-tree with root 18 has height 2.

An AVL tree implements the Map abstract data type just like a regular binary search tree, the only difference is in how the tree performs. To implement our AVL tree we need to keep track of a **balance factor** for each node in the tree. We do this by looking at the heights of the left and right subtrees for each node. More formally, we define the balance factor for a node as the difference between the height of the left subtree and the height of the right subtree.

$$balanceFactor = height(leftSubTree) - height(rightSubTree)$$

Using the definition for balance factor given above we say that a subtree is left-heavy if the balance factor is greater than zero. If the balance factor is less than zero then the subtree is right heavy. If the balance factor is zero then the tree is perfectly in balance. For purposes of implementing an AVL tree, and gaining the benefit of having a balanced tree we will define a tree to be in balance if the balance factor is -1, 0, or 1. Once the balance factor of a node in a tree is outside this range we will need to have a procedure to bring the tree back into balance. Figure shows an example of an unbalanced, right-heavy tree and the balance factors of each node.



Properties of AVL Trees

AVL trees are identical to standard binary search trees except that for every node in an AVL tree, the height of the left and right subtrees can differ by at most 1 (Weiss, 1993, p:108). AVL trees are HB-k trees (height balanced trees of order k) of order HB-1.

The following is the height differential formula:

$$|\text{Height}(T_L) - \text{Height}(T_R)| \leq k$$

When storing an AVL tree, a field must be added to each node with one of three values: 1, 0, or -1. A value of 1 in this field means that the left subtree has a height one more than the right subtree. A value of -1 denotes the opposite. A value of 0 indicates that the heights of both subtrees are the same. Updates of AVL trees require up to $O(\log n)$ rotations, whereas updating red-black trees can be done using only one or two rotations (up to $O(\log n)$ color changes). For this reason, they (AVL trees) are considered a bit obsolete by some.

Sparse AVL trees

Sparse AVL trees are defined as AVL trees of height h with the fewest possible nodes. Figure 3 shows sparse AVL trees of heights 0, 1, 2, and 3.

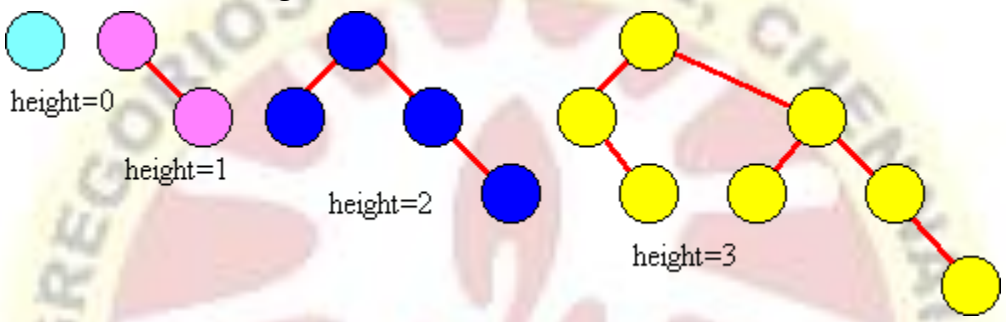


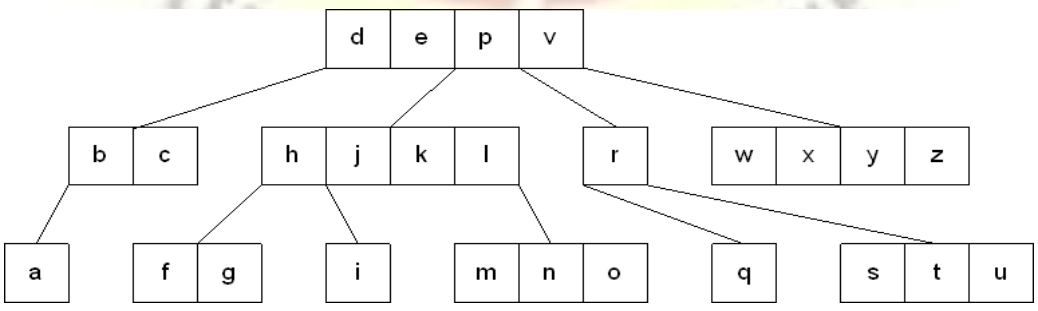
Figure Structure of an AVL tree

Introduction to M-Way Search Trees:

A **multiway tree** is a tree that can have more than two children. A **multiway tree of order m** (or an **m-way tree**) is one in which a tree can have m children.

As with the other trees that have been studied, the nodes in an m-way tree will be made up of key fields, in this case m-1 key fields, and pointers to children.

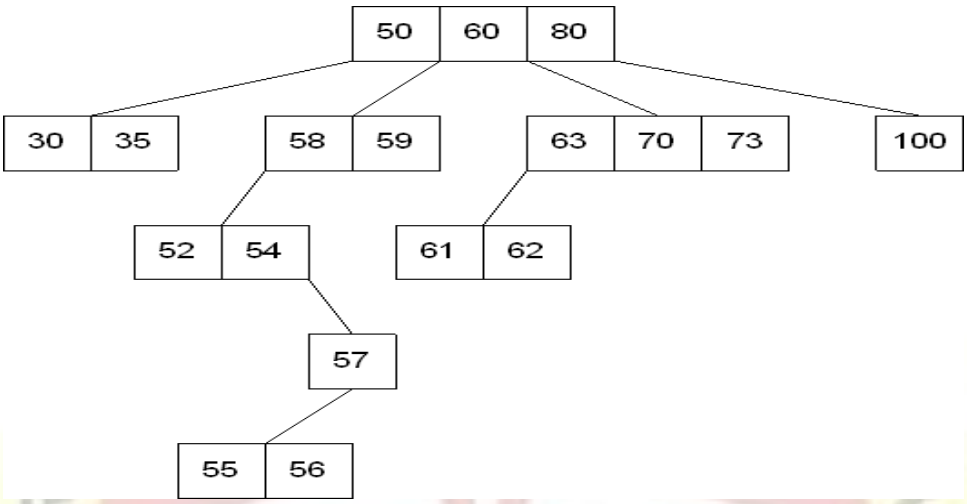
Multiday tree of order 5



To make the processing of m-way trees easier some type of order will be imposed on the keys within each node, resulting in a **multiway search tree of order m** (or an **m-way search tree**). By definition an m-way search tree is a m-way tree in which:

- Each node has m children and m-1 key fields
- The keys in each node are in ascending order.
- The keys in the first i children are smaller than the ith key
- The keys in the last m-i children are larger than the ith key

4-way search tree



M-way search trees give the same advantages to m-way trees that binary search trees gave to binary trees - they provide fast information retrieval and update. However, they also have the same problems that binary search trees had - they can become unbalanced, which means that the construction of the tree becomes of vital importance.

B Trees:

An extension of a multiway search tree of order m is a **B-tree of order m**. This type of tree will be used when the data to be accessed/stored is located on secondary storage devices because they allow for large amounts of data to be stored in a node.

A B-tree of order m is a multiway search tree in which:

1. The root has at least two subtrees unless it is the only node in the tree.
2. Each nonroot and each nonleaf node have at most m nonempty children and at least m/2 nonempty children.
3. The number of keys in each nonroot and each nonleaf node is one less than the number of its nonempty children.
4. All leaves are on the same level.

These restrictions make B-trees always at least half full, have few levels, and remain perfectly balanced.

Searching a B-tree

An algorithm for finding a key in B-tree is simple. Start at the root and determine which pointer to follow based on a comparison between the search value and key fields in the root node. Follow the appropriate pointer to a child node. Examine the key fields in the child node and continue to follow the appropriate pointers until the search value is found or a leaf node is reached that doesn't contain the desired search value.

Insertion into a B-tree

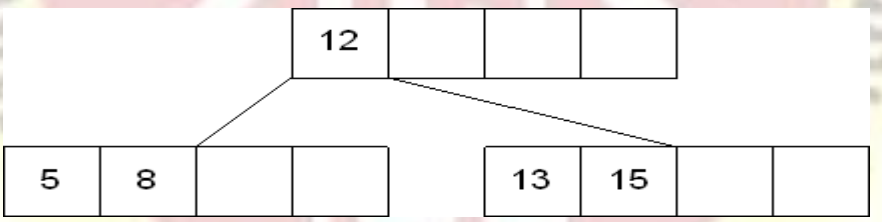
The condition that all leaves must be on the same level forces a characteristic behavior of B-trees, namely that B-trees are not allowed to grow at the their leaves; instead they are forced to grow at the root.

When inserting into a B-tree, a value is inserted directly into a leaf. This leads to three common situations that can occur:

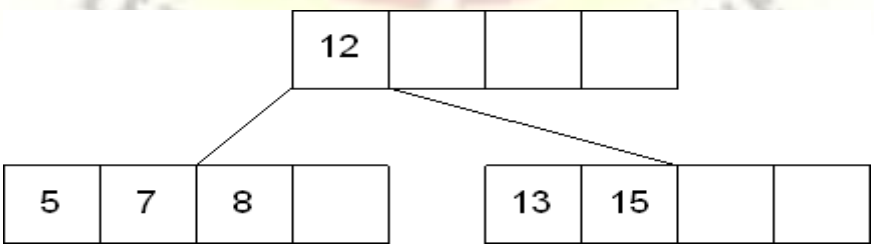
1. A key is placed into a leaf that still has room.
2. The leaf in which a key is to be placed is full.
3. The root of the B-tree is full.

Case 1: A key is placed into a leaf that still has room

This is the easiest of the cases to solve because the value is simply inserted into the correct sorted position in the leaf node.



Inserting the number 7 results in:

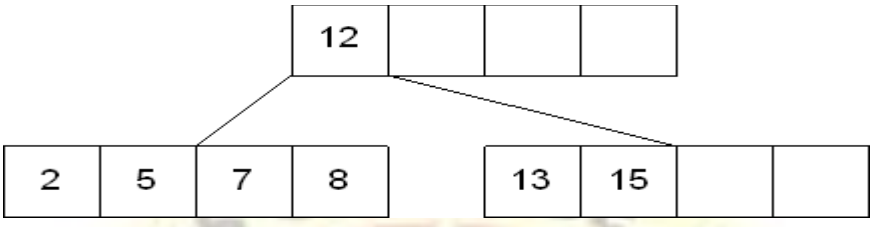


Case 2: The leaf in which a key is to be placed is full

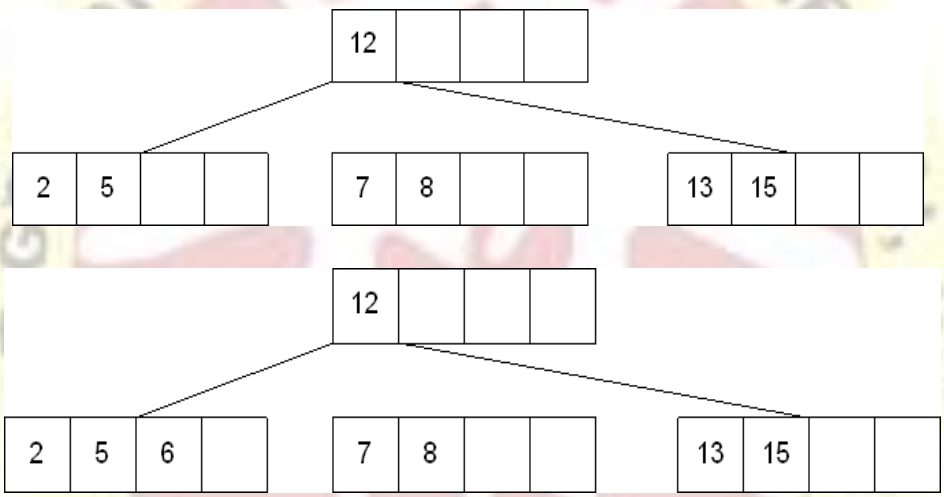
In this case, the leaf node where the value should be inserted is split in two, resulting in a new leaf node. Half of the keys will be moved from the full leaf to the new leaf. The new leaf is then incorporated into the B-tree.

The new leaf is incorporated by moving the middle value to the parent and a pointer to the new leaf is also added to the parent. This process is continues up the tree until all of the values have "found" a location.

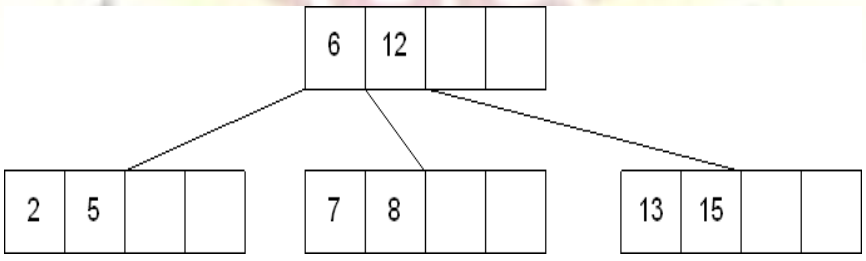
Insert 6 into the following B-tree:



results in a split of the first leaf node:



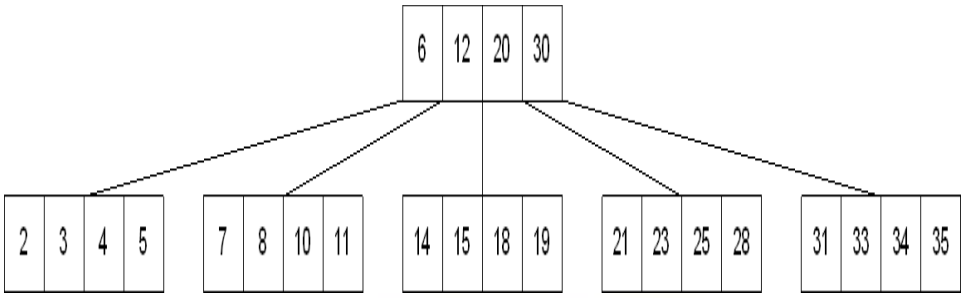
The new node needs to be incorporated into the tree - this is accomplished by taking the middle value and inserting it in the parent:



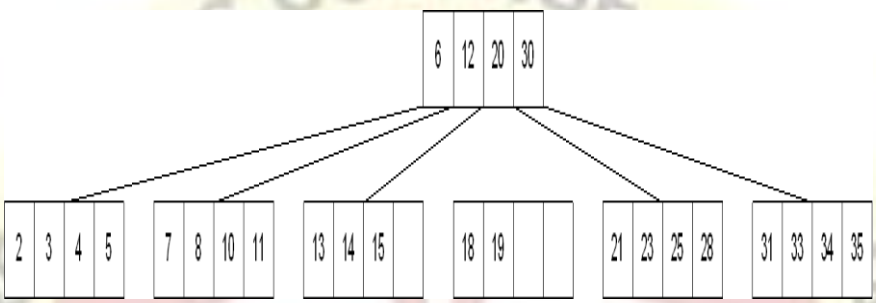
Case 3: The root of the B-tree is full

The upward movement of values from case 2 means that it's possible that a value could move up to the root of the B-tree. If the root is full, the same basic process from case 2 will be applied and a new root will be created. This type of split results in 2 new nodes being added to the B-tree.

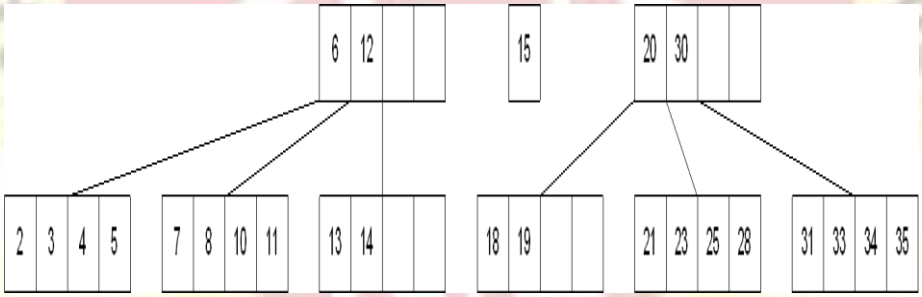
Inserting 13 into the following tree:



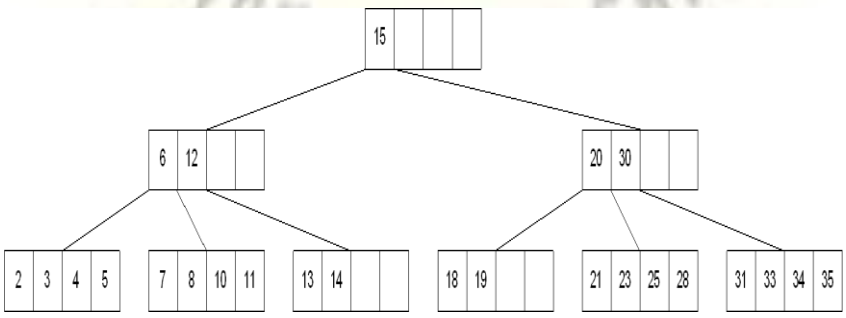
Results in:



The 15 needs to be moved to the root node but it is full. This means that the root needs to be divided:



The 15 is inserted into the parent, which means that it becomes the new root node:



Deleting from a B-tree

As usual, this is the hardest of the processes to apply. The deletion process will basically be a reversal of the insertion process - rather than splitting nodes, it's possible that nodes will be merged so that B-tree properties, namely the requirement that a node must be at least half full, can be maintained.

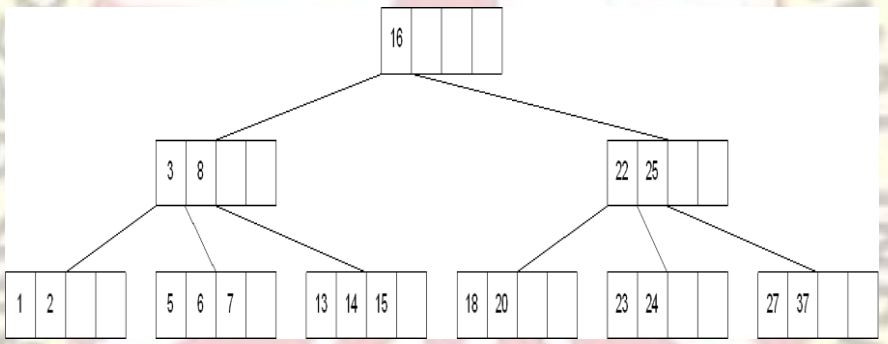
There are two main cases to be considered:

1. Deletion from a leaf
2. Deletion from a non-leaf

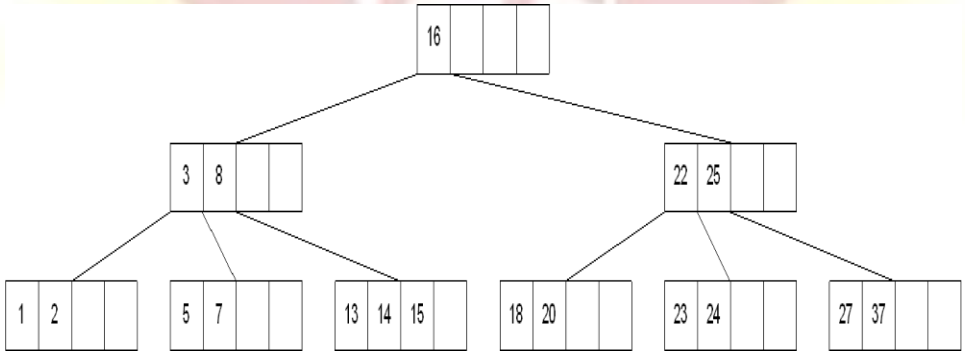
Case 1: Deletion from a leaf

1a) If the leaf is at least half full after deleting the desired value, the remaining larger values are moved to "fill the gap".

Deleting 6 from the following tree:

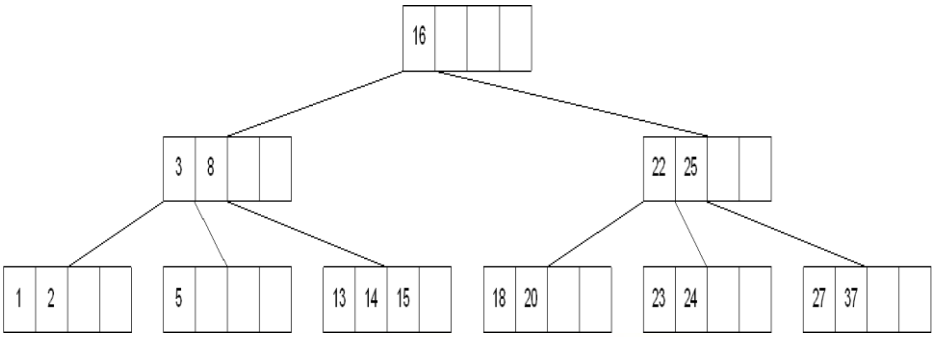


results in:

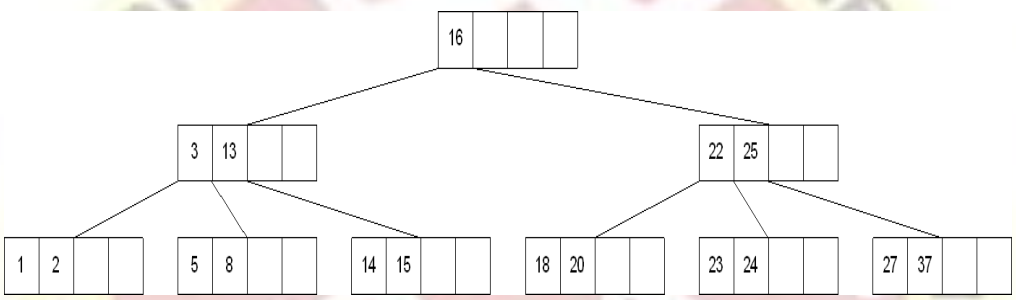


1b) If the leaf is less than half full after deleting the desired value (known as underflow), two things could happen:

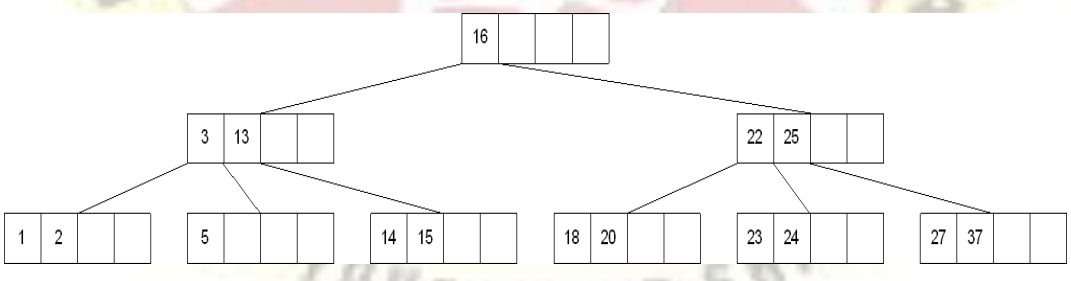
Deleting 7 from the tree above results in:



1b-1) If there is a left or right sibling with the number of keys exceeding the minimum requirement, all of the keys from the leaf and sibling will be redistributed between them by moving the separator key from the parent to the leaf and moving the middle key from the node and the sibling combined to the parent.

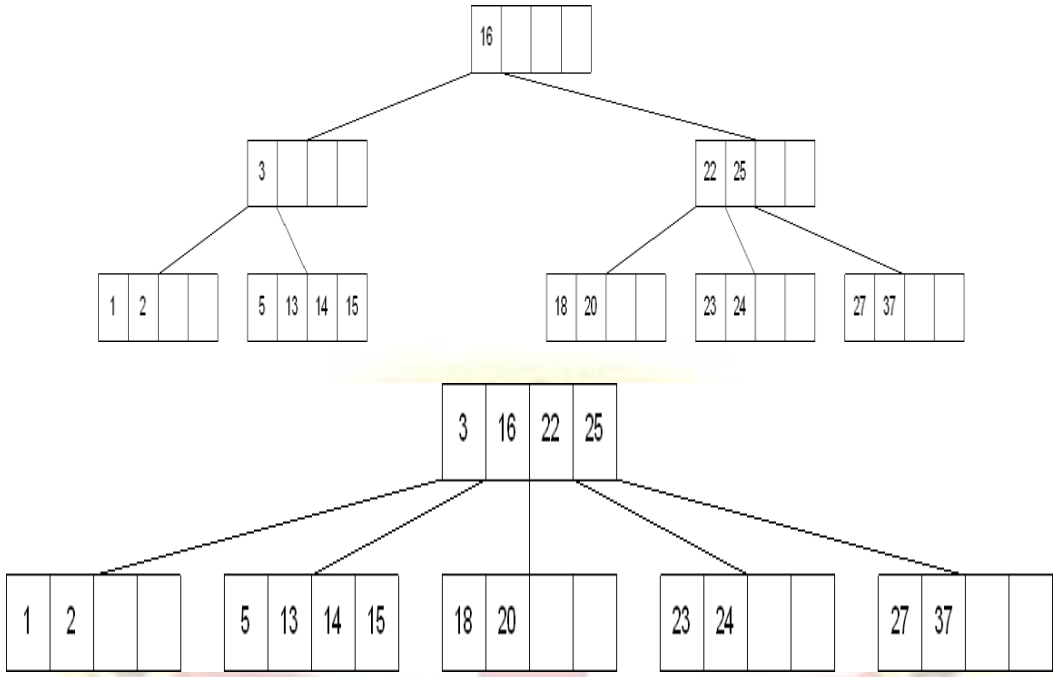


Now delete 8 from the tree:



1b-2) If the number of keys in the sibling does not exceed the minimum requirement, then the leaf and sibling are merged by putting the keys from the leaf, the sibling, and the separator from the parent into the leaf. The sibling node is discarded and the keys in the parent are moved to "fill the gap". It's possible that this will cause the parent to underflow. If that is the case, treat the parent as a leaf and continue repeating step 1b-2 until the minimum requirement is met or the root of the tree is reached.

Special Case for 1b-2: When merging nodes, if the parent is the root with only one key, the keys from the node, the sibling, and the only key of the root are placed into a node and this will become the new root for the B-tree. Both the sibling and the old root will be discarded.

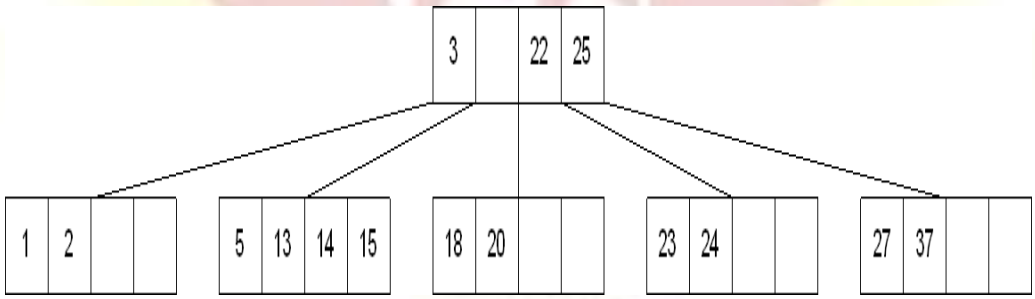


Case 2: Deletion from a non-leaf

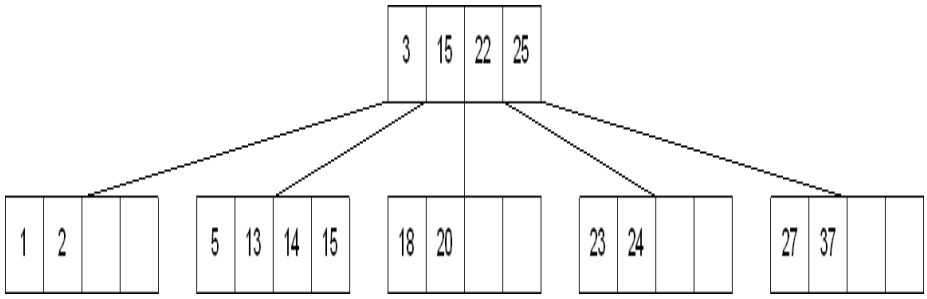
This case can lead to problems with tree reorganization but it will be solved in a manner similar to deletion from a binary search tree.

The key to be deleted will be replaced by its immediate predecessor (or successor) and then the predecessor (or successor) will be deleted since it can only be found in a leaf node.

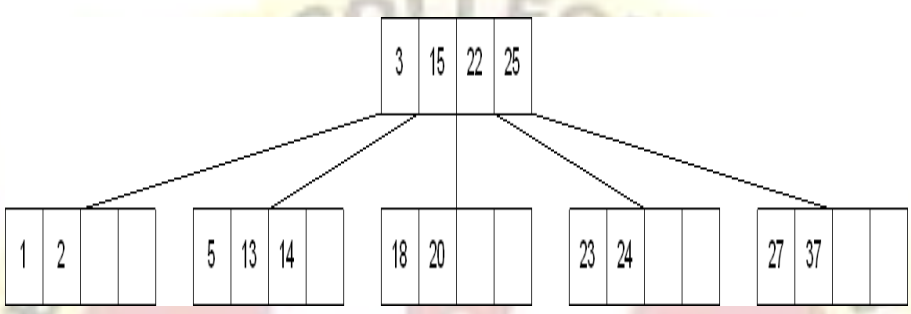
Deleting 16 from the tree above results in:



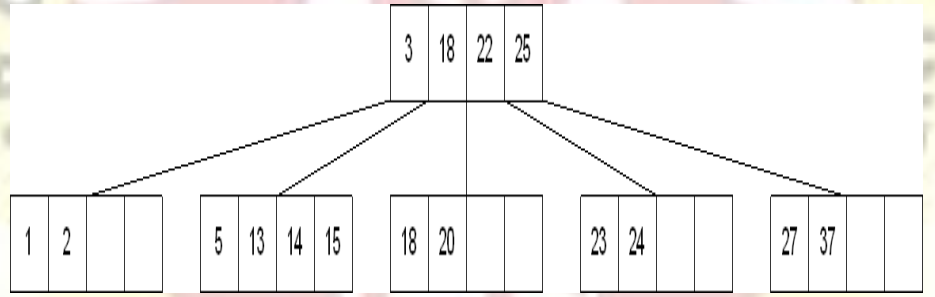
The "gap" is filled in with the immediate predecessor:



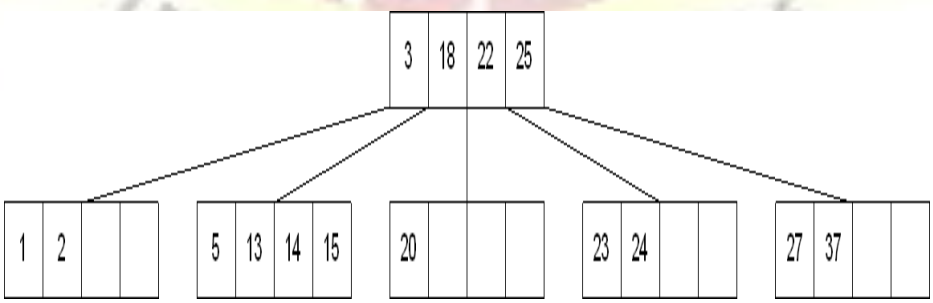
and then the immediate predecessor is deleted:



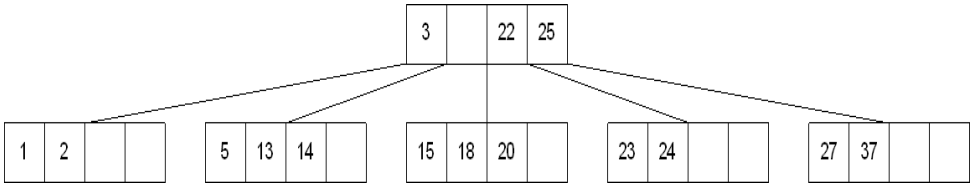
If the immediate successor had been chosen as the replacement:



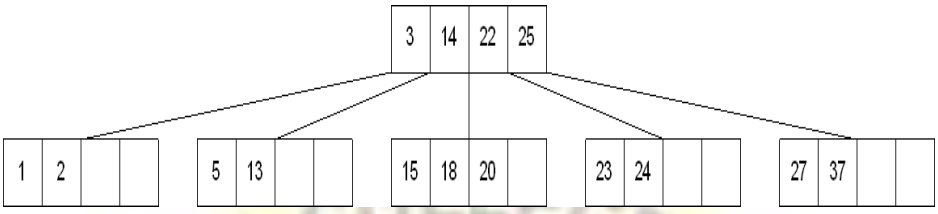
Deleting the successor results in:



The vales in the left sibling are combined with the separator key (18) and the remaining values. They are divided between the 2 nodes:



and then the middle value is moved to the parent:

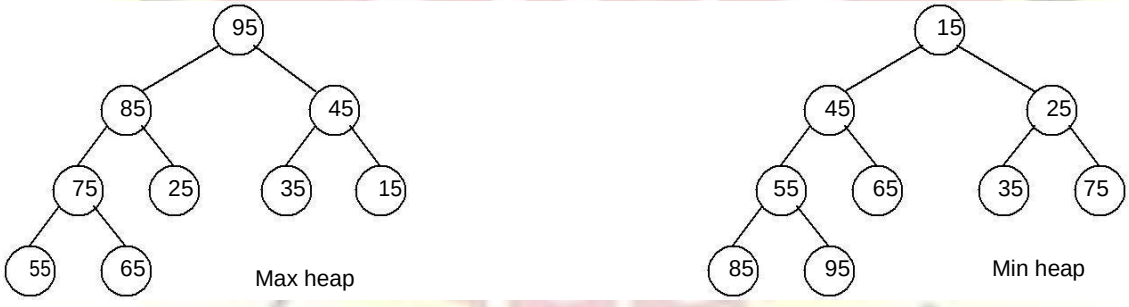


Heap and Heap Sort:

Heap is a data structure, which permits one to insert elements into a set and also to find the largest element efficiently. A data structure, which provides these two operations, is called a priority queue.

Max and Min Heap data structures:

A max heap is an almost complete binary tree such that the value of each node is greater than or equal to those in its children.



A min heap is an almost complete binary tree such that the value of each node is less than or equal to those in its children.

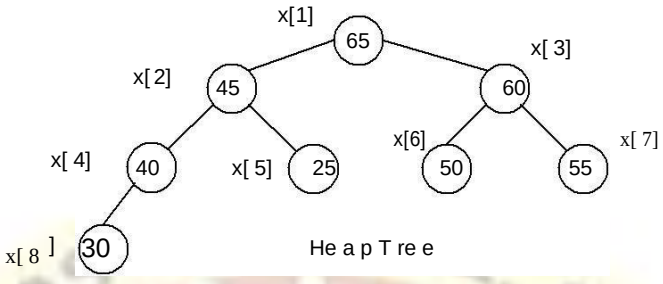
Representation of Heap Tree:

Since heap is a complete binary tree, a heap tree can be efficiently represented using one dimensional array. This provides a very convenient way of figuring out where children belong to.

- The root of the tree is in location 1.
- The left child of an element stored at location *i* can be found in location 2**i*.
- The right child of an element stored at location *i* can be found in location 2**i*+1.
- The parent of an element stored at location *i* can be found at location floor(*i*/2).

The elements of the array can be thought of as lying in a tree structure. A heap tree represented using a single array looks as follows:

X[1]	X[2]	X[3]	X[4]	X[5]	X[6]	X[7]	X[8]
65	45	60	40	25	50	55	30



Operations on heap tree:

The major operations required to be performed on a heap tree:

1. Insertion,
2. Deletion and
3. Merging.

Insertion into a heap tree:

This operation is used to insert a node into an existing heap tree satisfying the properties of heap tree. Using repeated insertions of data, starting from an empty heap tree, one can build up a heap tree.

Let us consider the heap (max) tree. The principle of insertion is that, first we have to adjoin the data in the complete binary tree. Next, we have to compare it with the data in its parent; if the value is greater than that at parent then interchange the values. This will continue between two nodes on path from the newly inserted node to the root node till we get a parent whose value is greater than its child or we reached the root.

For illustration, 35 is added as the right child of 80. Its value is compared with its parent‘s value, and to be a max heap, parent‘s value greater than child‘s value is satisfied, hence interchange as well as further comparisons are no more required.

As another illustration, let us consider the case of insertion 90 into the resultant heap tree. First, 90 will be added as left child of 40, when 90 is compared with 40 it requires interchange. Next, 90 is compared with 80, another interchange takes place. Now, our process stops here, as 90 is now in root node. The path on which these comparisons and interchanges have taken places are shown by *dashed line*.

The algorithm Max_heap_insert to insert a data into a max heap tree is as follows:

```
Max_heap_insert (a, n)
{
        //inserts the value in a[n] into the heap which is stored at a[1] to a[n-1] int i,
```

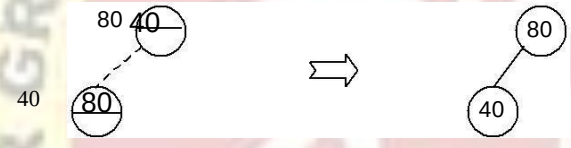
```
        n;
        i = n;
        item = a[n];
        while ( (i > 1) and (a[ i/2 ] < item ) do
        {
                a[i] = a[  i/2  ] ;                    // move the parent down i
                =  i/2      ;
        }
        a[i] = item ;
        return true ;
}
```

Example:

Form a heap using the above algorithm for the data: 40, 80, 35, 90, 45, 50, 70.

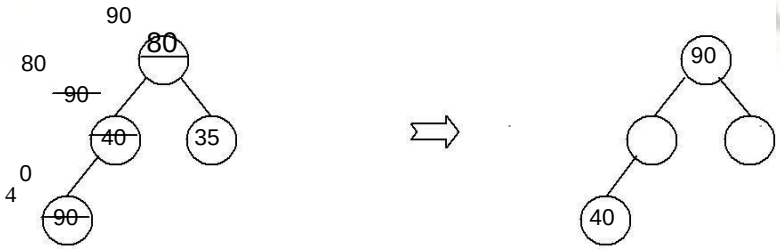
1. Insert 40:

2. Insert 80:

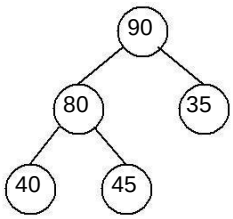


3. Insert 35:

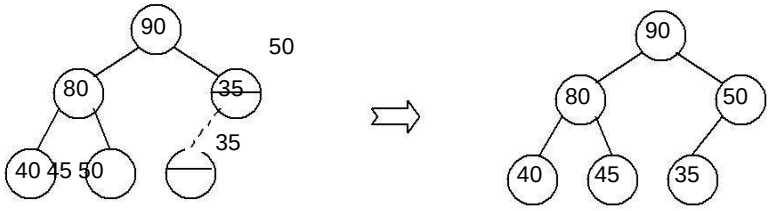
4. Insert 90:



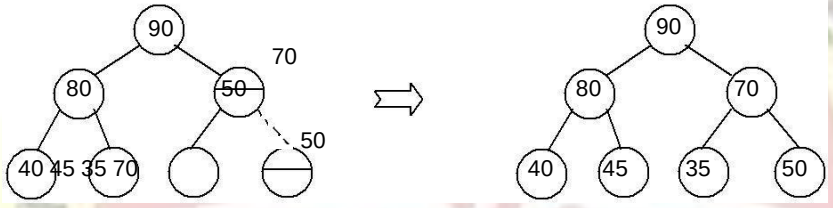
5. Insert 45:



6. Insert 50:



7. Insert 70:



Deletion of a node from heap tree:

Any node can be deleted from a heap tree. But from the application point of view, deleting the root node has some special importance. The principle of deletion is as follows:

- Read the root node into a temporary storage say, ITEM.
- Replace the root node by the last node in the heap tree. Then re-heap the tree as stated below:
  - Let newly modified root node be the current node. Compare its value with the value of its two child. Let X be the child whose value is the largest. Interchange the value of X with the value of the current node.
  - Make X as the current node.
  - Continue re-heap, if the current node is not an empty node.

The algorithm for the above is as follows:

delmax (a, n, x)

```
// delete the maximum from the heap a[n] and store it in x
{
        if (n = 0) then
        {
                write (―heap is empty‖);
                return false;
        }
        x = a[1]; a[1] = a[n];
        adjust (a, 1, n-1);
```

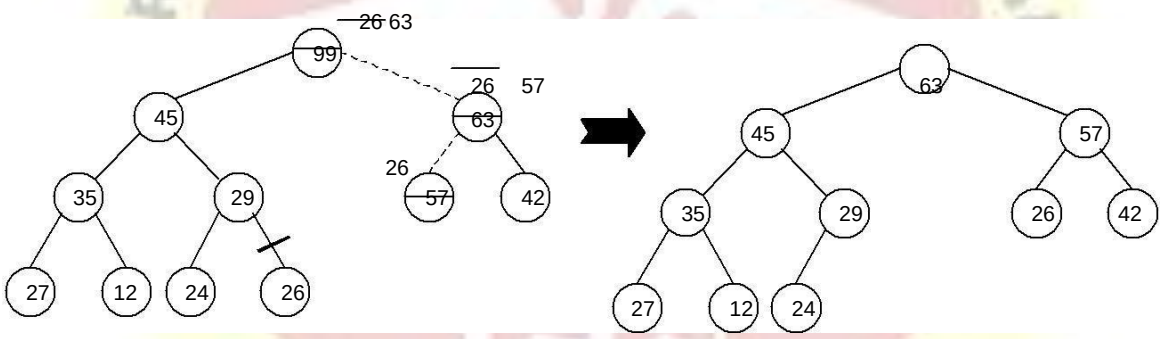
```
        return true;
}
```

adjust (a, i, n)

```
// The complete binary trees with roots a(2*i) and a(2*i + 1) are combined with a(i) to form
a single heap, 1 ≤ i ≤ n. No node has an address greater than n or less than 1. //
{
        j = 2 *i ;
        item = a[i] ;
        while (j ≤ n) do
        {
                if ((j < n) and (a (j) < a (j + 1)) then j        j + 1;
                        // compare left and right child and let j be the larger
                child if (item ≥ a (j)) then break;
                                                // a position for item is found else
                a[ j / 2 ] = a[j] // move the larger child up a level j = 2 * j;

        }
        a [ j / 2 ] = item;
}
```

Here the root node is 99. The last node is 26, it is in the level 3. So, 99 is replaced by 26 and this node with data 26 is removed from the tree. Next 26 at root node is compared with its two child 45 and 63. As 63 is greater, they are interchanged. Now, 26 is compared



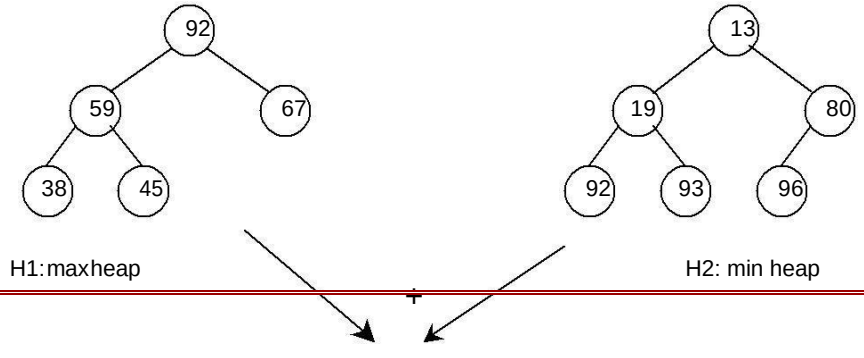
with its children, namely, 57 and 42, as 57 is greater, so they are interchanged. Now, 26 appears as the leave node, hence re-heap is completed.

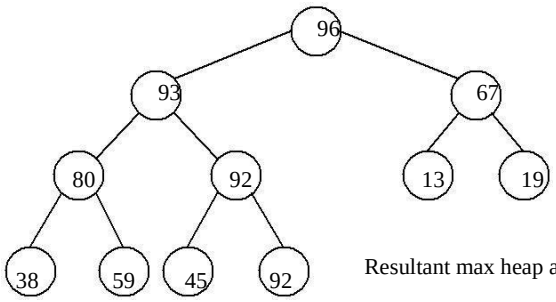
De le t ing t he no de w it h dat a 99 — Aft er De le t io n of no de w it h dat a 99

Merging two heap trees:

Consider, two heap trees H1 and H2. Merging the tree H2 with H1 means to include all the node from H2 to H1. H2 may be min heap or max heap and the resultant tree will be min heap if H1 is min heap else it will be max heap. Merging operation consists of two steps: Continue steps 1 and 2 while H2 is not empty:

1. Delete the root node, say x, from H2. Re-heap H2.
2. Insert the node x into H1 satisfying the property of H1.





Resultant max heap after merging H1 and H2

Application of heap tree:

They are two main applications of heap trees known are:

1. Sorting (Heap sort) and
2. Priority queue implementation.

HEAP SORT:

A heap sort algorithm works by first organizing the data to be sorted into a special type of binary tree called a heap. Any kind of data can be sorted either in ascending order or in descending order using heap tree. It does this with the following steps:

1. Build a heap tree with the given set of data.
2. a. Remove the top most item (the largest) and replace it with the last element in the heap.
   b. Re-heapify the complete binary tree.
   c. Place the deleted node in the output.
3. Continue step 2 until the heap tree is empty.

Algorithm:

This algorithm sorts the elements a[n]. Heap sort rearranges them in-place in non-decreasing order. First transform the elements into a heap.

heapsort(a, n)

```
{
        heapify(a, n);
        for i = n to 2 by – 1 do
        {
                temp = a[i]; a[i]
                = a[1]; a[1] =
                temp;
                adjust (a, 1, i – 1);
        }
}
```

heapify (a, n)

```
//Readjust the elements in a[n] to form a heap.
{
        for i       n/2 to 1 by – 1 do adjust (a, i, n);
}
```

adjust (a, i, n)

```
// The complete binary trees with roots a(2*i) and a(2*i + 1) are combined with a(i) to form
a single heap, 1 ≤ i ≤ n. No node has an address greater than n or less than 1. //
{
        j = 2 *i ;
        item = a[i] ;
        while (j ≤ n) do
        {
                if ((j < n) and (a (j) < a (j + 1)) then j       j + 1;
                        // compare left and right child and let j be the larger
                child if (item ≥ a (j)) then break;
                                                // a position for item is found else
                a[ j / 2 ] = a[j] // move the larger child up a level j = 2 * j;

        }
        a [ j / 2 ] = item;
}
```

Time Complexity:

Each ‗n' insertion operations takes O(log k), where ‗k' is the number of elements in the heap at the time. Likewise, each of the ‗n' remove operations also runs in time O(log k), where ‗k' is the number of elements in the heap at the time.

Since we always have k ≤ n, each such operation runs in O(log n) time in the worst case.

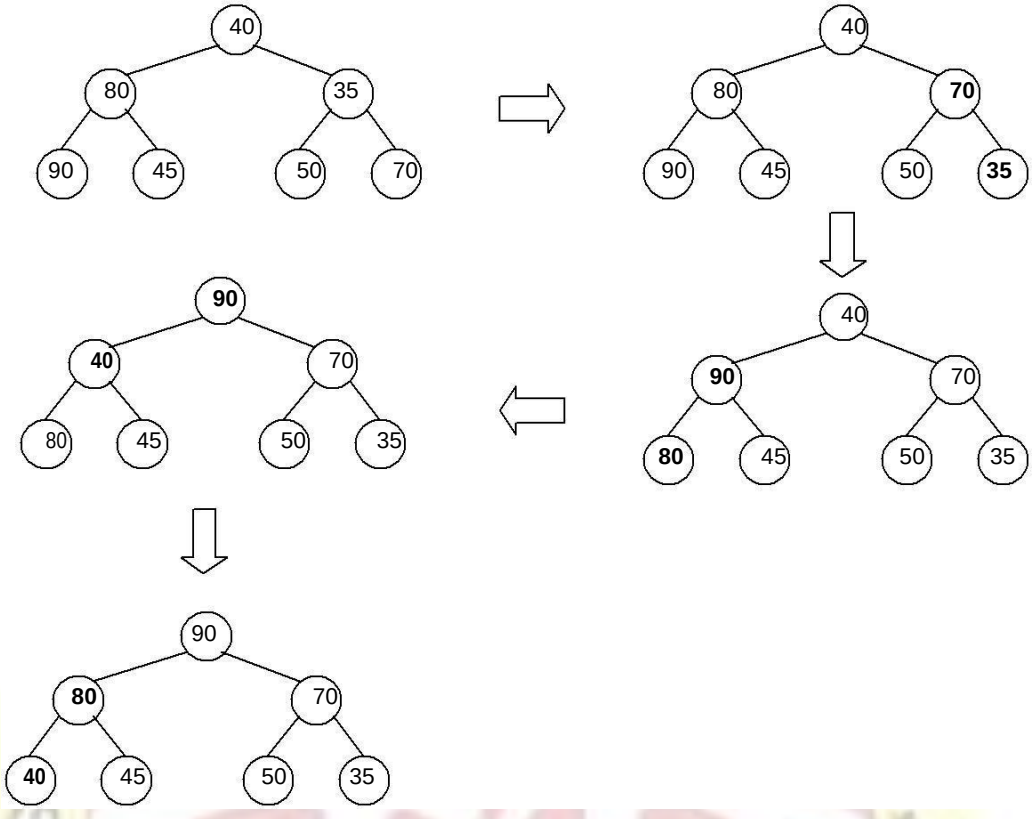
Thus, for ‗n' elements it takes O(n log n) time, so the priority queue sorting algorithm runs in O(n log n) time when we use a heap to implement the priority queue.

Example 1:

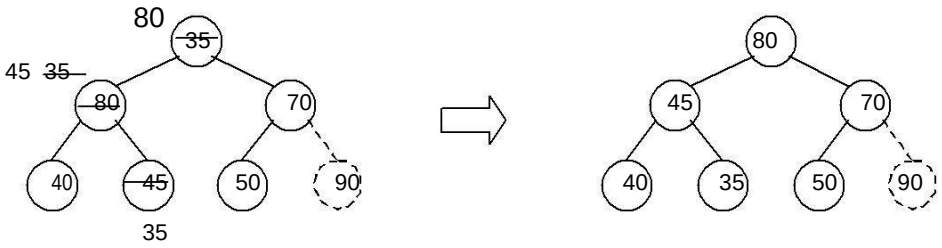
Form a heap from the set of elements (40, 80, 35, 90, 45, 50, 70) and sort the data using heap sort.

Solution:

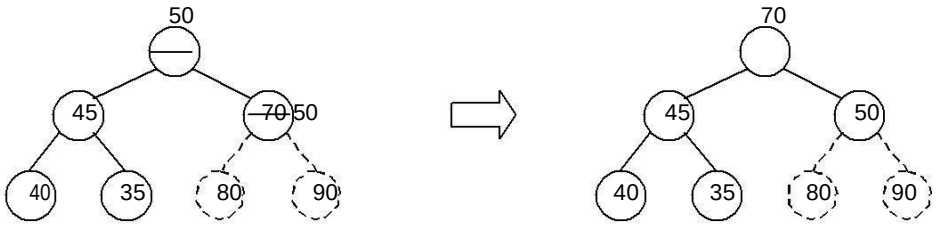
First form a heap tree from the given set of data and then sort by repeated deletion operation:



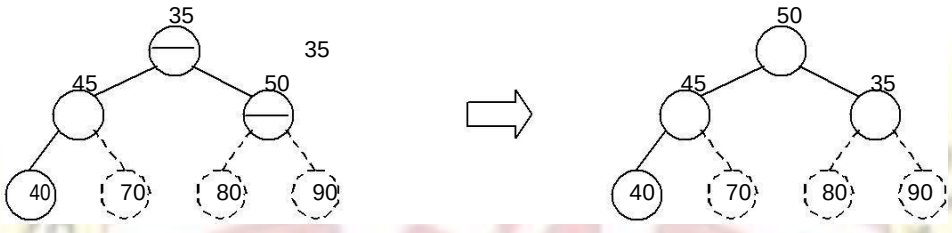
1. Exchange root 90 with the last element 35 of the array and re-heapify



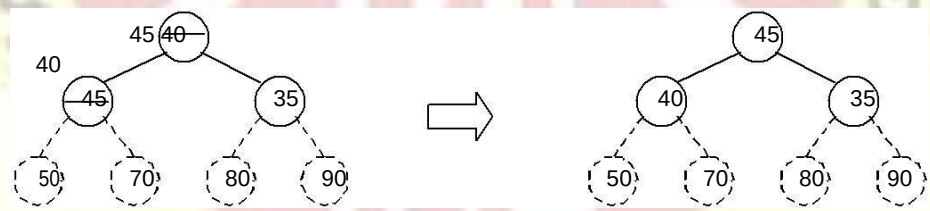
2. Exchange root 80 with the last element 50 of the array and re-heapify 70



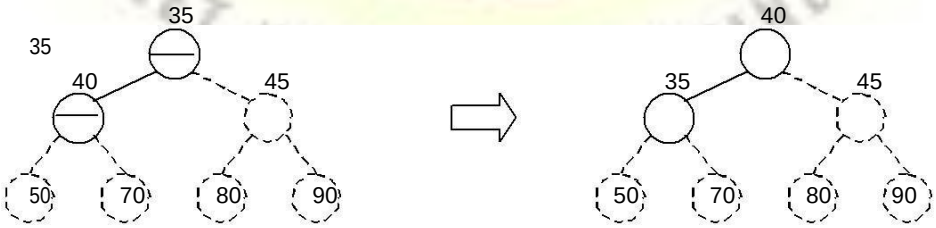
3. Exchange root 70 with the last element 35 of the array and re-heapify 50



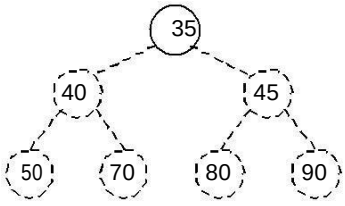
4. Exchange root 50 with the last element 40 of the array and re-heapify



5. Exchange root 45 with the last element 35 of the array and re-heapify 40



6. Exchange root 40 with the last element 35 of the array and re-heapify



The sorted tree

Priority queue implementation using heap tree:

Priority queue can be implemented using circular array, linked list etc. Another simplified implementation is possible using heap tree; the heap, however, can be represented using an array. This implementation is therefore free from the complexities of circular array and linked list but getting the advantages of simplicities of array.

As heap trees allow the duplicity of data in it. Elements associated with their priority values are to be stored in from of heap tree, which can be formed based on their priority values. The top priority element that has to be processed first is at the root; so it can be deleted and heap can be rebuilt to get the next element to be processed, and so on. As an illustration, consider the following processes with their priorities:

Process	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}
Priority	5	4	3	4	5	5	3	2	1	5

These processes enter the system in the order as listed above at time 0, say. Assume that a process having higher priority value will be serviced first. The heap tree can be formed considering the process priority values. The order of servicing the process is successive deletion of roots from the heap.

UNIT IV

GRAPH

Basic Graph Concepts:

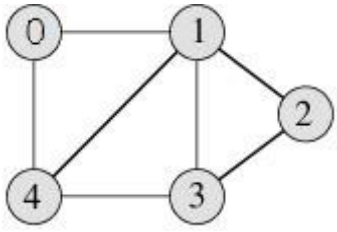
Graph is a data structure that consists of following two components:
1. A finite set of vertices also called as nodes.
2. A finite set of ordered pair of the form (u, v) called as edge.

The pair is ordered because (u, v) is not same as (v, u) in case of directed graph (di-graph). The pair of form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

Graph and its representations:

Graphs are used to represent many real life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, facebook. For example, in facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender and locale.

Following is an example undirected graph with 5 vertices.



Following two are the most commonly used representations of graph.
1. Adjacency Matrix
2. Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of the graph representation is situation specific. It totally depends on the type of operations to be performed and ease of use.

Adjacency Matrix:

Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

The adjacency matrix for the above example graph is:

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

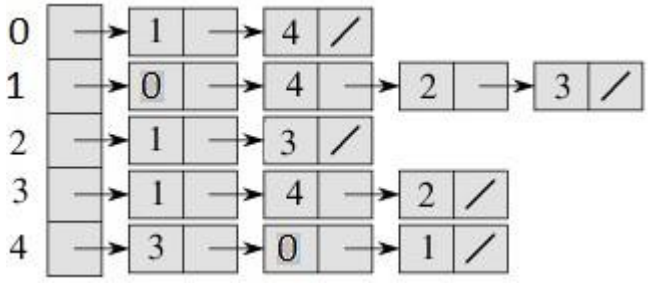
Adjacency Matrix Representation of the above graph

Pros: Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex ‘u’ to vertex ‘v’ are efficient and can be done O(1).

Cons: Consumes more space O(V^2). Even if the graph is sparse (contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time.

Adjacency List:

An array of linked lists is used. Size of the array is equal to number of vertices. Let the array be array[]. An entry array[i] represents the linked list of vertices adjacent to the ***i***th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists. Following is adjacency list representation of the above graph.

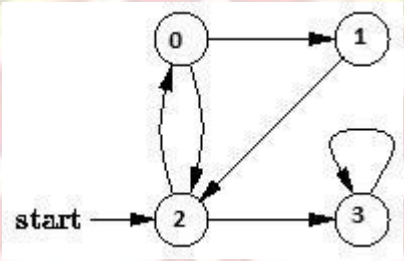


Adjacency List Representation of the above Graph

Breadth First Traversal for a Graph

Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. Breadth First Traversal of the following graph is 2, 0, 3, 1.



Algorithm: Breadth-First Search Traversal

BFS(V, E, s)

```
for each u in V − {s}
    do color[u] ←
        WHITE d[u]
        ← infinity
        π[u] ← NIL
        color[s] ←
        GRAY
       d[s] ←
      0 π[s]
      ← NIL
      Q ← {}
   ENQUEUE(Q, s)
  while Q is non-empty
do u ← DEQUEUE(Q)
        for each v adjacent to u
            do if color[v] ← WHITE
                then color[v] ←
                      GRAY d[v] ←
                      d[u] + 1
                        π[v] ← u
                       ENQUEUE
```

(Q, *v*) DEQUEUE(Q)
color[*u*] ← BLACK

Applications of Breadth First Traversal

1) Shortest Path and Minimum Spanning Tree for unweighted graph In unweighted graph, the shortest path is the path with least number of edges. With Breadth First, we always reach a vertex from given source using minimum number of edges. Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.

2) Peer to Peer Networks. In Peer to Peer Networks like BitTorrent, Breadth First Search is used to find all neighbor nodes.

3) Crawlers in Search Engines: Crawlers build index using Bread First. The idea is to start from source page and follow all links from source and keep doing same. Depth First Traversal can also be used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of built tree can be limited.

4) Social Networking Websites: In social networks, we can find people within a given distance 'k' from a person using Breadth First Search till 'k' levels.

5) GPS Navigation systems: Breadth First Search is used to find all neighboring locations.

6) Broadcasting in Network: In networks, a broadcasted packet follows Breadth First Search to reach all nodes.

7) In Garbage Collection: Breadth First Search is used in copying garbage collection using Cheney's algorithm.

8) Cycle detection in undirected graph: In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. In directed graph, only depth first search can be used.

9) Ford–Fulkerson algorithm In Ford-Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow. Breadth First Traversal is preferred as it reduces worst case time complexity to $O(VE^2)$.

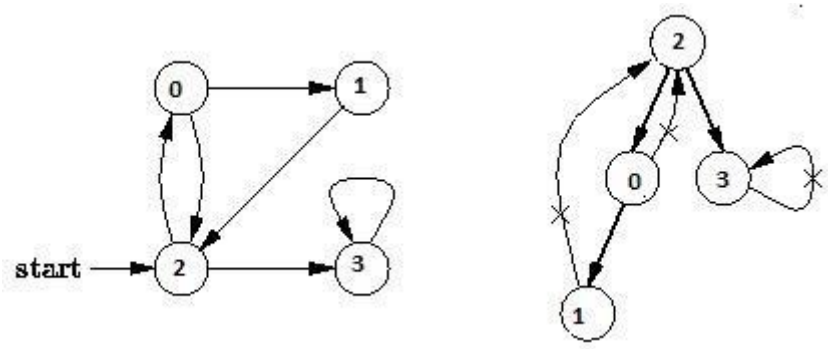
10) To test if a graph is Bipartite We can either use Breadth First or Depth First Traversal.

11) Path Finding We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.

12) Finding all nodes within one connected component: We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node.

Depth First Traversal for a Graph

Depth First Traversal (or Search) for a graph is similar to Depth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. Depth First Traversal of the following graph is 2, 0, 1, 3



Algorithm Depth-First Search

The DFS forms a depth-first forest comprised of more than one depth-first trees. Each tree is made of edges (*u*, *v*) such that *u* is gray and *v* is white when edge (*u*, *v*) is explored. The following pseudocode for DFS uses a global timestamp time.

DFS (V, E)

```
for each vertex u in V[G]
    do color[u] ←
          WHITE
          π[u] ← NIL
  time ← 0
for each vertex u in V[G]
   do if color[u] ← WHITE
      then DFS-Visit(u)
```

DFS-Visit(u)

```
color[u] ←
GRAY time
← time + 1
d[u] ← time
for each vertex v adjacent to u
do if color[v] ← WHITE
      then π[v] ← u
            DFS-
Visit(v) color[u]
← BLACK time
← time + 1
f[u] ← time
```

Applications of Depth First Search

Depth-first search (DFS) is an algorithm (or technique) for

traversing a graph. Following are the problems that use DFS as a

building block.

1) For an unweighted graph, DFS traversal of the graph produces the minimum spanning tree and all pair shortest path tree.

2) Detecting cycle in a graph

A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges. (See this for details)

3) Path Finding

We can specialize the DFS algorithm to find a path between two given vertices u and z.
i) Call DFS(G, u) with u as the start vertex.
ii) Use a stack S to keep track of the path between the start vertex and the current vertex.
iii) As soon as destination vertex z is encountered, return the path
as the contents of the stack

4) Topological Sorting

5) To test if a graph is bipartite

We can augment either BFS or DFS when we first discover a new vertex, color it opposite its parents, and for each other edge, check it doesn't link two vertices of the same color. The first vertex in any connected component can be red or black! See this for details.

6) Finding Strongly Connected Components of a graph A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex. (See this for DFS based also for finding Strongly Connected Components)

C program to implement the Graph Traversal
(a) Breadth first traversal
(b) Depth first traversal

DFS search starts from root node then traversal into left child node and continues, if item found it stops otherwise it continues. The advantage of DFS is it requires less memory compare to Breadth First Search (BFS).

UNIT V

SEARCHING

Searching is used to find the location where an element is available. There are two types of search techniques. They are:

- Linear or sequential search
- Binary search

Sorting allows an efficient arrangement of elements within a given data structure. It is a way in which the elements are organized systematically for some purpose. For example, a dictionary in which words is arranged in alphabetical order and telephone director in which the subscriber names are listed in alphabetical order. There are many sorting techniques out of which we study the following.

- Bubble sort
- Quick sort
- Selection sort and
- Heap sort

There are two types of sorting techniques:

Internal sorting

External sorting

If all the elements to be sorted are present in the main memory then such sorting is called **internal sorting** on the other hand, if some of the elements to be sorted are kept on the secondary storage, it is called **external sorting**. Here we study only internal sorting techniques.

- ***Linear Search:***

This is the simplest of all searching techniques. In this technique, an ordered or unordered list will be searched one by one from the beginning until the desired element is found. If the desired element is found in the list then the search is successful otherwise unsuccessful.

Suppose there are ‗n' elements organized sequentially on a List. The number of comparisons required to retrieve an element from the list, purely depends on where the element is stored in the list. If it is the first element, one comparison will do; if it is second element two comparisons are necessary and so on. On an average you need [(n+1)/2] comparison's to search an element. If search is not successful, you would need 'n' comparisons.

The time complexity of linear search is ***O(n)***.

Algorithm:

Let array a[n] stores n elements. Determine whether element ‗x' is present or not.

```
linsrch(a[n], x)
{
        index = 0;
        flag = 0;
        while (index < n) do
        {
                if (x == a[index])
                {
                        flag = 1;
                        break;
                }
                index ++;
        }
        if(flag == 1)
                printf(—Data found at %d position—, index);
        else
                printf(—data not found‖);

}
```

Example 1:

Suppose we have the following unsorted list: 45, 39, 8, 54, 77, 38, 24, 16, 4, 7, 9, 20 If we are searching for:

45, we'll look at 1 element before success
39, we'll look at 2 elements before success
8, we'll look at 3 elements before success 54,
we'll look at 4 elements before success 77,
we'll look at 5 elements before success 38
we'll look at 6 elements before success 24,
we'll look at 7 elements before success 16,
we'll look at 8 elements before success 4,
we'll look at 9 elements before success 7,
we'll look at 10 elements before success 9,
we'll look at 11 elements before success 20,
we'll look at 12 elements before success

For any element not in the list, we'll look at 12 elements before failure.

Example 2:

Let us illustrate linear search on the following 9 elements:

Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Elements	-15	-6	0	7	9	23	54	82	101

Searching different elements is as follows:

Searching for x = 7 Search successful, data found at 3rd position.

Searching for x = 82 Search successful, data found at 7th position.

Searching for x = 42 Search un-successful, data not found.

A non-recursive program for Linear Search:

```
\{include <stdio.h>
\{include <conio.h>

main()
{
        int number[25], n, data, i, flag = 0;
        clrscr();
        printf("\n Enter the number of elements: ");
        scanf("%d", &n);
        printf("\n Enter the elements:
        "); for(i = 0; i < n; i++)
                scanf("%d", &number[i]);
        printf("\n Enter the element to be Searched: ");
        scanf("%d", &data);
        for( i = 0; i < n; i++)
        {
                if(number[i] == data)
                {
```

```
                flag = 1;
                break;
            }
        }
        if(flag == 1)
            printf("\n Data found at location: %d", i+1);
        else
            printf("\n Data not found ");
}
```

A Recursive program for linear search:

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

void linear_search(int a[], int data, int position, int n)
{
        if(position < n)
        {
                if(a[position] == data)
                        printf("\n Data Found at %d ", position);
                else
                        linear_search(a, data, position + 1, n);
        }
        else
                printf("\n Data not found");
}

void main()
{
```

```
        int a[25], i, n, data;
        clrscr();
        printf("\n Enter the number of elements: ");
        scanf("%d", &n);
        printf("\n Enter the elements:
        "); for(i = 0; i < n; i++)
        {
                scanf("%d", &a[i]);
        }
        printf("\n Enter the element to be seached: ");
        scanf("%d", &data);
        linear_search(a, data, 0, n);
        getch();
}
```

1. BINARY SEARCH

If we have ‗n' records which have been ordered by keys so that $x_1 < x_2 < \ldots < x_n$. When we are given a element ‗x', binary search is used to find the corresponding element from the list. In case ‗x' is present, we have to determine a value ‗j' such that a[j] = x (successful search). If ‗x' is not in the list then j is to set to zero (un successful search).

In Binary search we jump into the middle of the file, where we find key a[mid], and compare ‗x' with a[mid]. If x = a[mid] then the desired record has been found. If x < a[mid] then ‗x' must be in that portion of the file that precedes a[mid]. Similarly, if a[mid] > x, then further search is only necessary in that part of the file which follows a[mid].

If we use recursive procedure of finding the middle key a[mid] of the un-searched portion of a file, then every un-successful comparison of ‗x' with a[mid] will eliminate roughly half the un-searched portion from consideration.

Since the array size is roughly halved after each comparison between ‗x' and a[mid], and since an array of length ‗n' can be halved only about $\log_2 n$ times before reaching a trivial length, the worst case complexity of Binary search is about $\log_2 n$.

Algorithm:

Let array a[n] of elements in increasing order, $n \geq 0$, determine whether ‗x' is present, and if so, set j such that x = a[j] else return 0.

```
binsrch(a[], n, x)
{
        low = 1; high = n;
        while (low ≤ high) do
        {
                mid = (low + high)/2 if
                (x < a[mid])
                        high = mid – 1;
                else if (x > a[mid])
                        low = mid +
                1; else return mid;
        }
        return 0;
 }
```

low and *high* are integer variables such that each time through the loop either ‗x' is found or *low* is increased by at least one or *high* is decreased by at least one. Thus we have two sequences of integers approaching each other and eventually *low* will become greater than *high* causing termination in a finite number of steps if ‗x' is not present.

Example 1:

Let us illustrate binary search on the following 12 elements:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
Elements	4	7	8	9	16	20	24	38	39	45	54	77

If we are searching for x = 4: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 1, high = 2, mid = 3/2 = 1, check 4, ***found***

If we are searching for x = 7: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 1, high = 2, mid = 3/2 = 1, check 4
low = 2, high = 2, mid = 4/2 = 2, check 7, ***found***

If we are searching for x = 8: (This needs 2 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8, ***found***

If we are searching for x = 9: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 4, high = 5, mid = 9/2 = 4, check 9, ***found***

If we are searching for x = 16: (This needs 4 comparisons) low
= 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 4, high = 5, mid = 9/2 = 4, check 9
low = 5, high = 5, mid = 10/2 = 5, check 16, ***found***

If we are searching for x = 20: (This needs 1 comparison) low =
1, high = 12, mid = 13/2 = 6, check 20, ***found***

If we are searching for x = 24: (This needs 3 comparisons) low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 7, high = 8, mid = 15/2 = 7, check 24, ***found***

If we are searching for x = 38: (This needs 4 comparisons) low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 7, high = 8, mid = 15/2 = 7, check 24
low = 8, high = 8, mid = 16/2 = 8, check 38, ***found***

If we are searching for x = 39: (This needs 2 comparisons) low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39, ***found***

If we are searching for x = 45: (This needs 4 comparisons) low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 10, high = 12, mid = 22/2 = 11, check 54
low = 10, high = 10, mid = 20/2 = 10, check 45, ***found***

If we are searching for x = 54: (This needs 3 comparisons) low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 10, high = 12, mid = 22/2 = 11, check 54, ***found***

If we are searching for x = 77: (This needs 4 comparisons) low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 10, high = 12, mid = 22/2 = 11, check 54
low = 12, high = 12, mid = 24/2 = 12, check 77, ***found***

The number of comparisons necessary by search element: 20

– requires 1 comparison;
8 and 39 – requires 2 comparisons;
4, 9, 24, 54 – requires 3 comparisons and
7, 16, 38, 45, 77 – requires 4 comparisons

Summing the comparisons, needed to find all twelve items and dividing by 12, yielding 37/12 or approximately 3.08 comparisons per successful search on the average.

Example 2:

Let us illustrate binary search on the following 9 elements:

Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Elements	-15	-6	0	7	9	23	54	82	101

Solution:

The number of comparisons required for searching different elements is as follows:

1. If we are searching for x = 101: (Number of comparisons = 4)

low	high	mid
1	9	5
6	9	7
8	9	8
9	9	9

found

2. Searching for x = 82: (Number of comparisons = 3)

low	high	mid
1	9	5
6	9	7
8	9	8

found

3. Searching for x = 42: (Number of comparisons = 4)

low	high	mid
1	9	5
6	9	7
6	6	6

6 not found

4. Searching for x = -14: (Number of comparisons = 3)

low	high	mid
1	9	5
1	4	2
1	1	1
2	1	not found

Continuing in this manner the number of element comparisons needed to find each of nine elements is:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Elements	-15	-6	0	7	9	23	54	82	101
Comparisons	*3*	*2*	*3*	*4*	*1*	*3*	*2*	*3*	*4*

No element requires more than 4 comparisons to be found. Summing the comparisons needed to find all nine items and dividing by 9, yielding 25/9 or approximately 2.77 comparisons per successful search on the average.

There are ten possible ways that an un-successful search may terminate depending upon the value of x.

If x < a(1), a(1) < x < a(2), a(2) < x < a(3), a(5) < x < a(6), a(6) < x < a(7) or a(7) < x < a(8) the algorithm requires 3 element comparisons to determine that ‗x' is not present. For all of the remaining possibilities BINSRCH requires 4 element comparisons.

Thus the average number of element comparisons for an unsuccessful search is: (3 +

3 + 3 + 4 + 4 + 3 + 3 + 3 + 4 + 4) / 10 = 34/10 = 3.4

Time Complexity:

The time complexity of binary search in a successful search is O(log n) and for an unsuccessful search is O(log n).

A non-recursive program for binary search:

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

main()
{
        int number[25], n, data, i, flag = 0, low, high, mid;
        clrscr();
        printf("\n Enter the number of elements: ");
        scanf("%d", &n);
        printf("\n Enter the elements in ascending order: ");
        for(i = 0; i < n; i++)
                scanf("%d", &number[i]);
        printf("\n Enter the element to be searched: ");
        scanf("%d", &data);
        low = 0; high = n-1;
        while(low <= high)
        {
                mid = (low + high)/2;
                if(number[mid] == data)
                {
                        flag = 1;
                        break;
                }
                else
                {
                        if(data < number[mid])
                                high = mid - 1;
                         else
                                low = mid + 1;
                }
        }
        if(flag == 1)
                printf("\n Data found at location: %d", mid + 1);
         else
                printf("\n Data Not Found ");
}
```

Bubble Sort:

The bubble sort is easy to understand and program. The basic idea of bubble sort is to pass through the file sequentially several times. In each pass, we compare each element in the file with its successor i.e., X[i] with X[i+1] and interchange two element when they are not in proper order. We will illustrate this sorting technique by taking a specific example. Bubble sort is also called as exchange sort.

Example:

Consider the array x[n] which is stored in memory as shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	X[5]
33	44	22	11	66	55

Suppose we want our array to be stored in ascending order. Then we pass through the array 5 times as described below:

Pass 1: (first element is compared with all other elements).

We compare X[i] and X[i+1] for i = 0, 1, 2, 3, and 4, and interchange X[i] and X[i+1] if X[i] > X[i+1]. The process is shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	X[5]	Remarks
33	44	22	11	66	55	
	22	44				
		11	44			
			44	66		
				55	66	
33	22	11	44	55	66	

The biggest number 66 is moved to (bubbled up) the right most position in the array.

Pass 2: (second element is compared).

We repeat the same process, but this time we don‘t include X[5] into our comparisons. i.e., we compare X[i] with X[i+1] for i=0, 1, 2, and 3 and interchange X[i] and X[i+1] if X[i] > X[i+1]. The process is shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	Remarks

33	22	11	44	55	
22	33				
	11	33			
		33	44		
			44	55	
22	11	33	44	55	

The second biggest number 55 is moved now to X[4].

Pass 3: (third element is compared).

We repeat the same process, but this time we leave both X[4] and X[5]. By doing this, we move the third biggest number 44 to X[3].

X[0]	X[1]	X[2]	X[3]	Remarks
22	11	33	44	
11	22			
	22	33		
		33	44	
11	22	33	44	

Pass 4: (fourth element is compared).

We repeat the process leaving X[3], X[4], and X[5]. By doing this, we move the fourth biggest number 33 to X[2].

X[0]	X[1]	X[2]	Remarks
11	22	33	
11	22		
	22 33		

Pass 5: (fifth element is compared).

We repeat the process leaving X[2], X[3], X[4], and X[5]. By doing this, we move the fifth biggest number 22 to X[1]. At this time, we will have the smallest number 11 in X[0]. Thus, we see that we can sort the array of size 6 in 5 passes.

For an array of size n, we required (n-1) passes.

Program for Bubble Sort:

```
#include <stdio.h>
#include <conio.h>
void bubblesort(int x[], int n)
{
        int i, j, temp;
        for (i = 0; i < n; i++)
```

```
        {
                for (j = 0; j < n–i-1 ; j++)
                {
                        if (x[j] > x[j+1])
                        {
                                temp = x[j]; x[j]
                                = x[j+1]; x[j+1]
                                = temp;
                        }
                }
        }
}

main()
{
        int i, n, x[25];
        clrscr();
        printf("\n Enter the number of elements: ");
        scanf("%d", &n);
        printf("\n Enter Data:");
        for(i = 0; i < n ; i++)
                scanf("%d", &x[i]);
        bubblesort(x, n);
        printf ("\n Array Elements after sorting: ");
        for (i = 0; i < n; i++)
}               printf ("%5d", x[i]);
```

Time Complexity:

The bubble sort method of sorting an array of size n requires (n-1) passes and (n-1) comparisons on each pass. Thus the total number of comparisons is (n-1) * (n-1) = n^2 − 2n + 1, which is O(n^2). Therefore bubble sort is very inefficient when there are more elements to sorting.

Selection Sort:

Selection sort will not require no more than n-1 interchanges. Suppose x is an array of size n stored in memory. The selection sort algorithm first selects the smallest element in the array x and place it at array position 0; then it selects the next smallest element in the array x and place it at array position 1. It simply continues this procedure until it places the biggest element in the last position of the array.

The array is passed through (n-1) times and the smallest element is placed in its respective position in the array as detailed below:

Pass 1: Find the location j of the smallest element in the array x [0], x[1],..... x[n-1], and then interchange x[j] with x[0]. Then x[0] is sorted.

Pass 2: Leave the first element and find the location j of the smallest element in the sub-array x[1], x[2], ... x[n-1], and then interchange x[1] with x[j]. Then x[0], x[1] are sorted.

Pass 3: Leave the first two elements and find the location j of the smallest element in the sub-array x[2], x[3], ... x[n-1], and then interchange x[2] with x[j]. Then x[0], x[1], x[2] are sorted.

Pass (n-1): Find the location j of the smaller of the elements x[n-2] and x[n-1], and then interchange x[j] and x[n-2]. Then x[0], x[1], ... x[n-2] are sorted. Of course, during this pass x[n-1] will be the biggest element and so the entire array is sorted.

Time Complexity:

In general we prefer selection sort in case where the insertion sort or the bubble sort requires exclusive swapping. In spite of superiority of the selection sort over bubble sort and the insertion sort (there is significant decrease in run time), its efficiency is also ***O(n^2)*** for n data items.

Example:

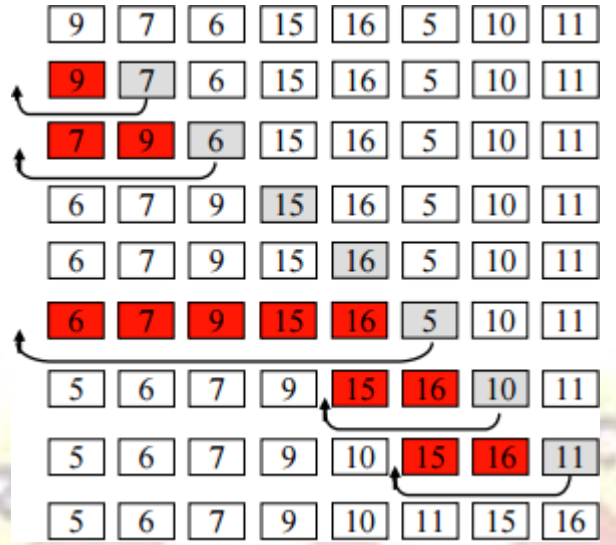
Let us consider the following example with 9 elements to analyze selection Sort:

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*Remarks*
65	70	75	80	50	60	55	85	45	find the first smallest element
i								j	swap a[i] & a[j]
45	70	75	80	50	60	55	85	65	find the second smallest element
	i			j					swap a[i] and a[j]
45	**50**	75	80	70	60	55	85	65	Find the third smallest element
		i				j			swap a[i] and a[j]
45	**50**	**55**	80	70	60	75	85	65	Find the fourth smallest element
			i		j				swap a[i] and a[j]
45	**50**	**55**	**60**	70	80	75	85	65	Find the fifth smallest element
				i				j	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	80	75	85	70	Find the sixth smallest element
					i			j	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	75	85	80	Find the seventh smallest element
						i j			swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	**75**	85	80	Find the eighth smallest element
							i	J	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	**75**	**80**	**85**	The outer loop ends.

INSERTION SORT

An algorithm consider the elements one at a time, inserting each in its suitable place among those already considered (keeping them sorted). Insertion sort is an example of an **incremental** algorithm. It builds the sorted sequence one number at a time. This is a suitable sorting technique in playing card games. Insertion sort provides several advantages:

- Simple implementation
- Efficient for (quite) small data sets
- Adaptive (i.e., efficient) for data sets that are already substantially sorted: the time complexity is O(n + d), where d is the number of inversions
- More efficient in practice than most other simple quadratic (i.e., O(n^2)) algorithms such as selection sort or bubble sort; the best case (nearly sorted input) is O(n)
- Stable; i.e., does not change the relative order of elements with equal keys
- In-place; i.e., only requires a constant amount O(1) of additional memory space
- Online; i.e., can sort a list as it receives it



Step-by-step example:

Step 1	12 **3** 1 5 8	Checking second element of array with element before it and inserting it in proper position. In this case, 3 is inserted in position of 12.
Step 2	3 12 **1** 5 8	Checking third element of array with elements before it and inserting it in proper position. In this case, 1 is inserted in position of 3.
Step 3	1 3 12 **5** 8	Checking fourth element of array with elements before it and inserting it in proper position. In this case, 5 is inserted in position of 12.
Step 4	1 3 5 12 **8**	Checking fifth element of array with elements before it and inserting it in proper position. In this case, 8 is inserted in position of 12.
	0 1 3 8 12	Sorted Array in Ascending Order

Figure: Sorting Array in Ascending Order Using Insertion Sort Algorithm

Suppose, you want to sort elements in ascending as in above figure. Then,

1. The second element of an array is compared with the elements that appear before it (only first element in this case). If the second element is smaller than first element, second element is inserted in the position of first element. After first step, first two elements of an array will be sorted.

2. The third element of an array is compared with the elements that appears before it (first and second element). If third element is smaller than first element, it is inserted in the position of first element. If third element is larger than first element but, smaller than second element, it is inserted in the position of second element. If third element is larger than both the elements, it is kept in the position as it is. After second step, first three elements of an array will be sorted.

3. Similarly, the fourth element of an array is compared with the elements that appear before it (first, second and third element) and the same procedure is applied and that element is inserted in the proper position. After third step, first four elements of an array will be sorted.

If there are *n* elements to be sorted. Then, this procedure is repeated *n-1* times to get sorted list of array.

Time Complexity:

Worst Case Performance	**$O(n^2)$**
Best Case Performance(nearly)	**O(n)**
Average Case Performance	**$O(n^2)$**

Output:

Enter no of elements:5

Enter elements:1 65 0 32 66

Elements after sorting: 0 1 32 65 66

Hashing and Collision:

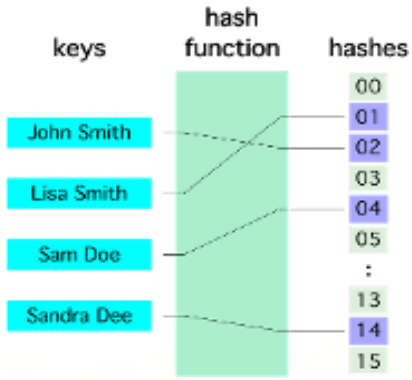
Hashing is the technique used for performing almost constant time search in case of insertion, deletion and find operation. Taking a very simple example of it, an array with its index as key is the example of hash table. So each index (key) can be used for accessing the value in a constant search time. This mapping key must be simple to compute and must helping in identifying the associated value. Function which helps us in generating such kind of key-value mapping is known as Hash Function.

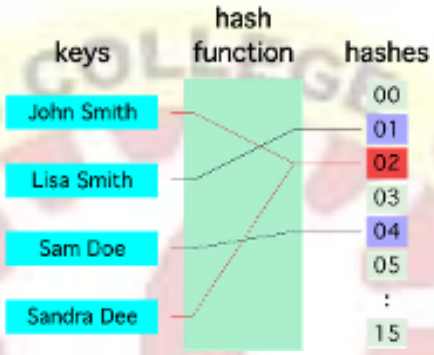
In a hashing system the keys are stored in an array which is called the Hash Table. A perfectly implemented hash table would always promise an average insert/delete/retrieval time of O(1).

Hashing Function:

A function which employs some algorithm to computes the key *K* for all the data elements in the set *U*, such that the key *K* which is of a fixed size. The same key *K* can be used to map data to a hash table and all the operations like insertion, deletion and searching should be possible. The values returned by a **hash function** are also referred to as **hash** values,

hash codes, **hash** sums, or **hashes**.





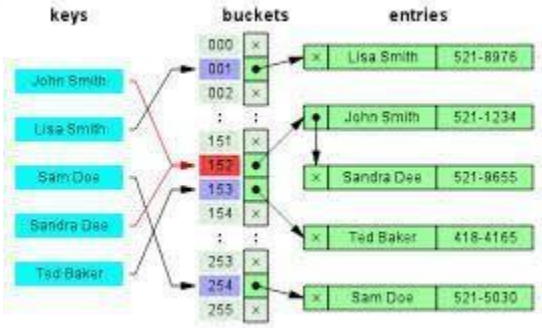
Hash Collision:

A situation when the resultant hashes for two or more data elements in the data set *U*, maps to the same location in the has table, is called a hash collision. In such a situation two or more data elements would qualify to be stored / mapped to the same location in the hash table.

Hash collision resolution techniques:

Open Hashing (Separate chaining):

Open Hashing, is a technique in which the data is not directly stored at the hash key index (k) of the Hash table. Rather the data at the key index (k) in the hash table is a pointer to the head of the data structure where the data is actually stored. In the most simple and common implementations the data structure adopted for storing the element is a linked-list.



n this technique when a data needs to be searched, it might become necessary (worst case) to traverse all the nodes in the linked list to retrieve the data.

Note that the order in which the data is stored in each of these linked lists (or other data structures) is completely based on implementation requirements. Some of the popular criteria are insertion order, frequency of access etc.

Closed hashing (open Addressing)

In this technique a hash table with pre-identified size is considered. All items are stored in the hash table itself. In addition to the data, each hash bucket also maintains the three states: EMPTY, OCCUPIED, DELETED. While inserting, if a collision occurs, alternative cells are tried until an empty bucket is found. For which one of the following technique is adopted.

Liner Probing

1. Quadratic probing
2. Double hashing (in short in case of collision another hashing function is used with the key value as an input to identify where in the open addressing scheme the data should actually be stored.)

A comparative analysis of Closed Hashing vs Open Hashing

Open Addressing	Closed Addressing
All elements would be stored in the Hash table itself. No additional data structure is needed.	Additional Data structure needs to be used to accommodate collision data.
In cases of collisions, a unique hash key must be obtained.	Simple and effective approach to collision resolution. Key may or may not be unique.
Determining size of the hash table, adequate enough for storing all the data is difficult.	Performance deterioration of closed addressing much slower as compared to Open addressing.
State needs be maintained for the data (additional work)	No state data needs to be maintained (easier to maintain)
Uses space efficiently	Expensive on space

Applications of Hashing:

A **hash function** maps a variable length input string to fixed length output string -- its **hash value**, or **hash** for short. If the input is longer than the output, then some inputs must map to the same output -- a **hash collision**. Comparing the hash values for two inputs can give us one of two answers: the inputs are definitely not the same, or there is a possibility that they are the same. Hashing as we know it is used for performance improvement, error checking, and authentication. One example of a performance improvement is the common hash table, which uses a hash function to index into the correct bucket in the hash table, followed by comparing each element in the bucket to find a match. In error checking, hashes (checksums, message digests, etc.) are used to detect errors caused by either hardware or software. Examples are TCP checksums, ECC memory, and MD5 checksums on downloaded files. In this case, the hash provides additional assurance that the data we received is correct. Finally, hashes are used to authenticate messages. In this case, we are

trying to protect the original input from tampering, and we select a hash that is strong enough to make malicious attack infeasible or unprofitable.

- Construct a *message authentication code* (MAC)
- Digital signature
- Make commitments, but reveal message later
- Timestamping
- Key updating: key is hashed at specific intervals resulting in new key

*****