DATA STRUCTURES AND ALGORITHMS

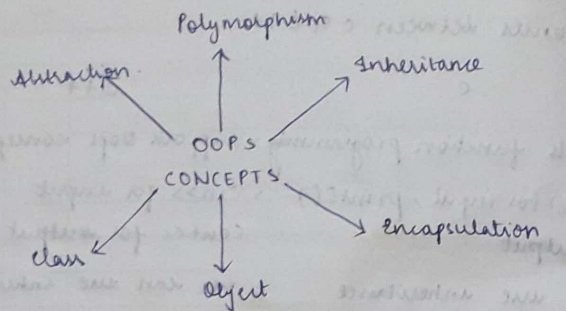
C++

Differences between C & C++

C	C++
> Supports function programming	> supports oops concepts.
> scanf() for input, printf() for output	> cin>> for input cout<< for output.
> can't use inheritance	> We can use inheritance
> file extension is .c	> file extension is .cpp
> C is a structural or procedural programming language.	> C++ is an object oriented programming language
> Emphasis is on procedure or steps to solve any problem.	> Emphasis is on objects rather than procedure.
> Functions are the fundamental building blocks.	> Objects are the fundamental building blocks.
> In C, the data is not secured.	> Data is hidden and can't be accessed by external functions.
> C follows top-down approach	> C++ follows bottom up approach
> Variables must be defined at the begining of the function	> Variables can be defined anywhere in the function.
> In C, name space feature is absent.	> In C++, namespace feature is present.
> C is a middle level language.	> C++ is a high level language.

OBJECT ORIENTED PROGRAMMING (OOPs)

Concepts of object oriented programming :-



The main aim of OOP is to bind together the data and the function that operate on them so that no other part of the code can access this data except the function.

CLASS :-

The building block of C++ that leads to object oriented programming is a class. It is a user defined datatypes which holds its own data member and data functions which can be accessed and used by creating an instance of that class. A class is like a blue print for an object.

A class is a grouping of objects having identical properties, common behaviour and shared relationship.

Eg :- 1) class - Car

Properties :- Company, model, color & capacity

Actions :- speed( ), average( ) and break( ).

OBJECT :-

An object is a identifiable entity with some characteristics and behaviour. An object is an instance of a class. When a class is defined no memory is allocated but when it is instantiated (i.e, an object is created) memory is allocated.

METHOD :-

> An operation required for an object entity when coded in a class is called Method.

> All the objects in a class perform certain common actions or operations.

Note :- Method can also be called as function, module, sub-routine, sub-program.

Example for classes / Syntax for classes :

```
class sample
{  Private : // Access specifiers
       data member 1;
       data member 2;      //members
       data member (3);
   Public : // Access specifiers
       method 1( ) {- - -}
       method 2( ) {- - -}
       method n( ) {- - -}
}
```

Syntax for object creation :-

sample Obj1, Obj2, ---, Objn

ABSTRACTION :- Data abstraction is one of the most essential and important features of object oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

> Classes use the theory of abstraction.

ENCAPSULATION :- It is defined as wrapping up of data and information under a single unit. In object oriented programming, Encapsulation is defined as binding together the data and the functions that manipulate them.

> Encapsulation also leads to data abstraction or hiding. As using Encapsulation also hides the data.

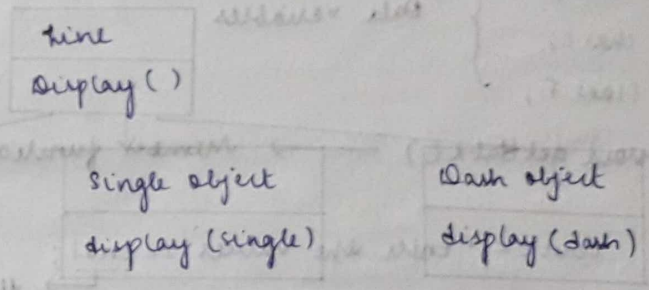
> The data is not accessible by outside functions.

> Data and functions are single unit.

INHERITANCE :- Inheritance is the method by which objects of one class get the properties of objects of another class. It is reusable and programmer can add new properties (files & methods) to the existing class.

```
class X // Base class
   ↓
class Y // Derived class
```

POLYMORPHISM :- Polymorphism allows the same function to act differently in different classes. It takes more than one form. It is possible to code a non-specific (generic) interface.




```
int main()
{
    sum1 = sum(20,30);
    sum2 = sum(20,30,40);
}

int sum(int a, int b)
{
    return(a+b)
}

int sum(int a, int b, int c)
{
    return(a+b+c)
}
```

INHERITANCE :-

Sub class :- The class that inherits properties from another class is called sub class or Derived class.

Super class :- The class whose properties are inherited by sub class is called base class or super class.

Reusability :- Inheritance supports the concept of "reusability", i.e. when we want to create new class and there is already a class that includes some of the code that we want, we can derive our new class from existing class.

ALGORITHM :-

> An algorithm is a finite set of instructions that, if followed, accomplishes a particular task.

> In addition, all algorithms must satisfy the following criteria.

1. Input : Zero or more quantities are externely supplied.
2. Output : At least one quantity is produced.
3. Definiteness : Each instruction is clear and unambiguous.
4. Finiteness : If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. Effectiveness : Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in criterion 3; it also must be feasible.

DEVELOPMENT OF ALGORITHM :-

> Problem statement
> Model formulation
> Algorithm design
> Algorithm correctness
> Implementation
> Algorithm Analysis
> Program Testing
> Documentation.

EFFICIENCY OF ALGORITHMS :-

> The performance of algorithms can be measured on the scales of time and space.

> The problem or that which performs its task in the minimum possible time. In this case the performance measured is termed time complexity.

> An algorithm that consumes or needs limited memory space for its execution. The performance measure in such a case is termed space complexity.

ALGORITHM ANALYSIS.

> Efficiency of an algorithm can be analysed at two different stages, before implementation and after implementation; they are following :-

A Priori Analysis :- This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.

A Posteriori Analysis :- This is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space

required are collected.

A Priori	A Posteriori
> Knowable without experience	> Knowledge requires experience
> Triangles have 3 sides	> Triangle is blue colour
> All bachelors are singles	> All bachelors are unhappy
> A is taller than B, B is taller than C, C is taller than D.	> A is six feet tall.
> Algorithm	> Program
> Language independent	> Language dependent
> Time and space functions	> Watch time & bytes
> Hardware Independent	> Hardware Dependent

ANALYSIS OF ALGORITHM:

It is the process of analysing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of analysis of algorithms is the required time or performance. Generally, we perform the following types of analysis:-

Worst case: The maximum no. of steps taken on any instance of size a.

Best case: The minimum no. of steps taken on any instance of size a.

Average case: An average no. of steps taken on any instance of size a.

ASYMPTOTIC NOTATIONS:-

> We will use asymptotic notations primarily to describe the running time of algorithms.
> However, asymptotic notation actually applies to functions.
> Asymptotic notation is used to describe the limiting behaviour of a function, when its argument tends towards a particular value (often infinity) usually in terms of simpler functions.
> Asymptotic notation can be used to classify algorithms by how they respond to changes in input size.
> While analyzing the run time of an algorithm, we should not only determine how long the algorithm takes in terms of size of its inputs but also should focus on how fast their runtime function grows with the input size.

Frequency count (or) Step count

We can calculate time complexity either by using

1. Frequency count
2. Step count

Frequency count specifies how many times statement is executed.

1. For comments, declarations — step count = 0

2. Return value, assigning — step count = 1

3. Ignore lower order exponents, when higher order exponents are present.

eg: $3n^4 + 4n^3 + 10n^2 + n^0 + 100$.

Higest order exponent = $3n^4$

4. Ignore constant multiplies

Now which is, $3n^4 = O(n^4)$ ⟶ (this is 'O' not zero)

eg:-

```
int sum (int a[], int n)
{
    s=0          ⟶ Assigning (s.c = 1)
    for (i=0; i<n; i++)
    s = s+a[i];
    return s;    ⟶ Returning value (s.c=1)
}
```

For suppose let us assume $n=3$ & $i=0$.

So the execution takes these no. of times

⇒ condition is $i<n$

$0<3$ ⟶ 1 time

$1<3$ ⟶ 2nd time

$2<3$ ⟶ 3rd time

$3<3$ ⟶ even though condition fails it checks. So 4th time.

We've assumed $n=3$ and the execution happed four times that is, $(n+1)$ times.

But, body will be executed n times because the condition $3<3$ is false.

So,

$$\Rightarrow \begin{array}{r} 1 \\ n+1 \\ +\ n \\ 1 \\ \hline 2n+3 \end{array}$$

⟶ Higher order exponential (n).

So the complexity is, $O(n)$.

Asymptotic Notations:-

We can use these notations we can calculate the time complexities of an algorithm.

1. Big Oh Notation
2. Big Omega Notation
3. Theta Notation.

BIG OH NOTATION:-

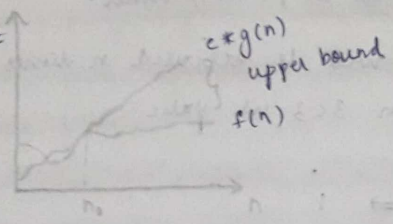
> Mainly represents the upper bound of algorithm's run time.

> To calculate max amount of time taken by the algorithm

> Worst case time complexity.

Def: Let $f(n), g(n)$ be two non-negative function, then $f(n) = O(g(n))$ if there exists two positive constants c, n_0, such that

$f(n) \leq c * g(n)$, $\forall\ n > n_0$



$f(n) = 3n+2$, $g(n) = n$

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

$f(n) \leq c * g(n)$, $\forall\ n > n_0$

$n=1$	$c=4$
$n=2$	$c=4$

$3n+2 \leq c * n$

$5 \leq 4$ ✗

$8 \leq 8$ ✓

(Ω)

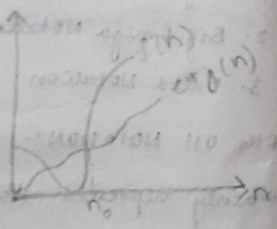
$n_0 > 2$ ✓

Big Omega Notation: To represent lower bound of algorithm.

> To calculate minimum amount of time.

> Best case time complexity.

$f(n) \geq c * g(n)$



$f(n) = 3n+2$, $g(n) = n$

$f(n) \geq c * g(n)$

$3n+2 \geq c * n$

$n=1, c=1$, $5 \geq 1$

$n=2, c=1$, $8 \geq 2$

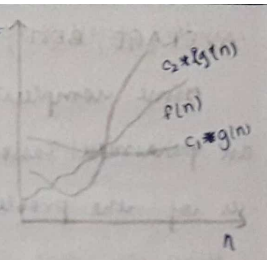
$n_0 \geq 1$

Θ Notation:

It is the average bound of an algorithm.

$f(n) = O(g(n))$ if two positive constants such that

$c_1 * g(n) \leq f(n) \leq c_2 * g(n)$



POLYNOMIAL NOTATION

polynomial algorithms include Quadratic algorithms $O(n^2)$, cubic algorithms $O(n^3)$

represents an algorithm whose performance is directly proportional to the square of the size of the data set.

EXPONENTIAL NOTATION

The exponential notation $O(2^n)$ describes an algorithm whose growth doubles with each addition to the data set

	n^2	2^n
$n=1$	$1^2=1$	$2^1=2$
$n=2$	$2^2=4$	$2^2=4$
$n=3$	$3^2=9$	$2^3=8$
$n=4$	$4^2=16$	$2^4=16$
$n=5$	$5^2=25$	$2^5=32$
$n=6$	$6^2=36$	$2^6=64$
$n=7$	$7^2=49$	$2^7=128$

AVERAGE, BEST, WORST CASE COMPLEXITIES:-

Time complexity of an algorithm is dependent on parameters associated with the input/output instances of the problem

Eg:- I/P$_1$: -1, -23, -11, -2, -4, -6, -9, -8, -9, 2

I/P$_2$: 1, -7, 11, -2, -6, -8, -7, -9

Search for the first occuring even no.'s in the list.

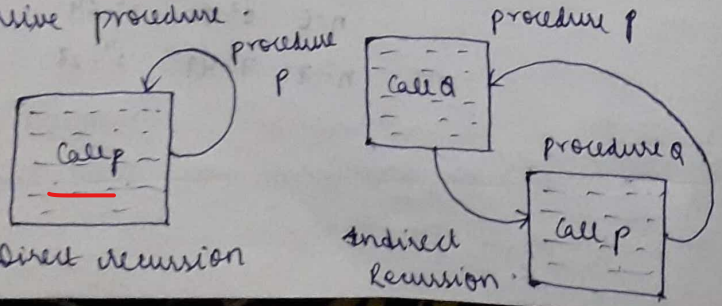
> Incase of input, it takes 10 comparisons to find out

> Incase of input 2, the first element is positive element

[This justifies that the statement that the running time of algorithms are not just dependent on the size of the I/P but also on its nature].

ANALYZING RECURSIVE ALGORITHMS:-

If 'P' is a procedure containing a call statement to itself (or) to another procedure that results in a call to itself, then the procedure P is said to be a recursive procedure.



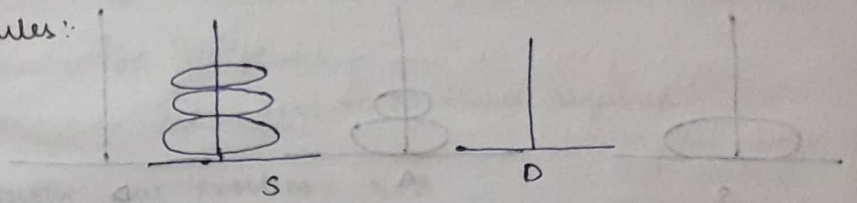
Eg:- Recursion is used to solve towers of Hanoi problems.

TOWERS OF HONAI PROBLEMS:-

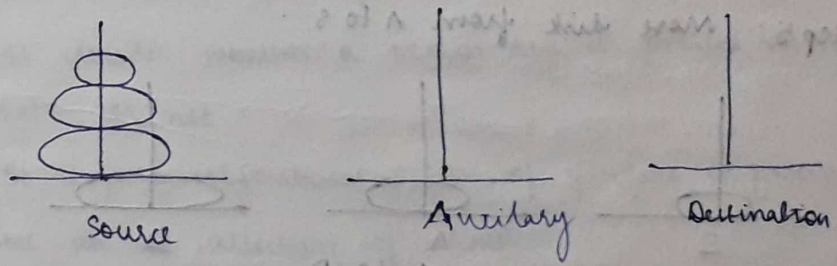
> Tower of Honai is a mathematical puzzle where we have three rods and 'n' disks.

> The objective of the puzzle is to move the entire stack to another rod.

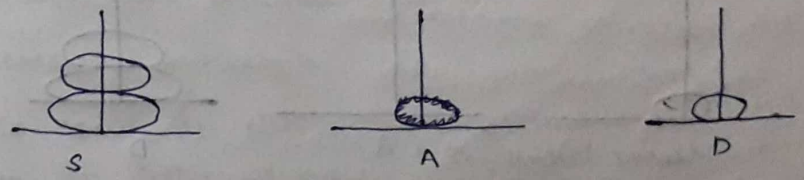
Rules:



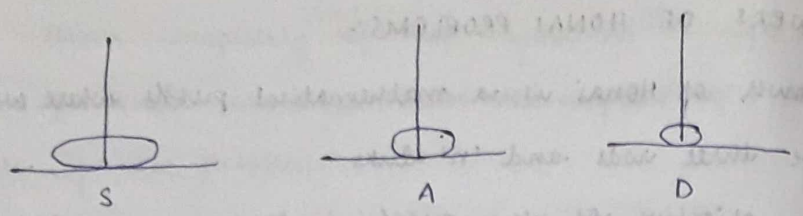
1. Only one disk can be moved at a time
2. Larger disk can't be placed on the top of smaller disk
3. We can use auxilary tower for temporary storage of disks.



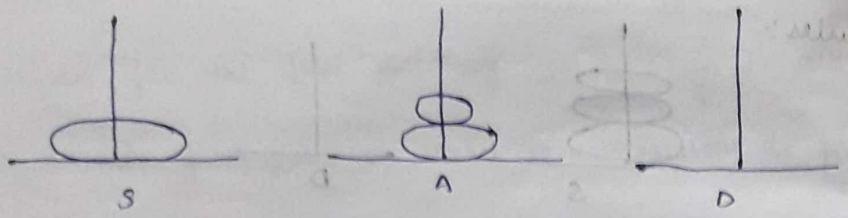
step1:- Move disk from S to D



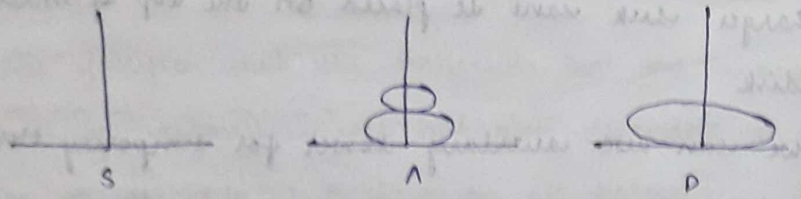
Step 2:- Move disk from S to A



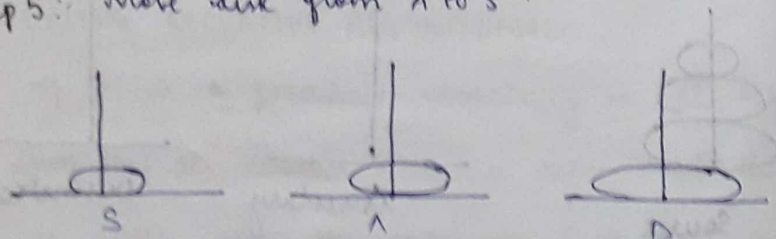
Step 3: Move disk from D to A



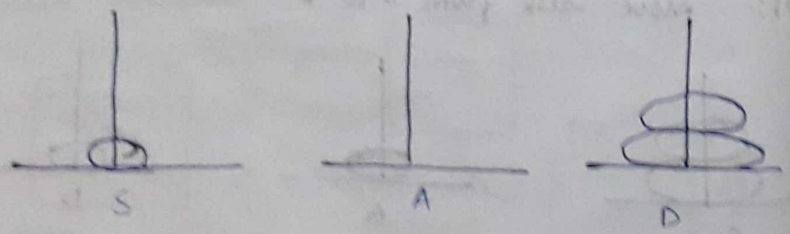
Step 4: Move disk from S to D



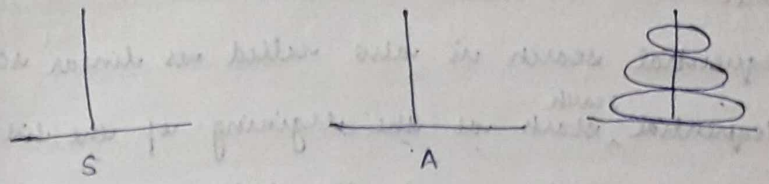
Step 5: Move disk from A to S



Step 6: Move disk from A to D



Step 7: Move disk from S to D



No. of disks = 3

No. of moves required = $2^n - 1$

Eg: $n = 5$

$2^n - 1 = 32 - 1 \rightarrow 31$ Moves required.

From our problem, $n = 3$

$2^3 - 1 = 7$ //

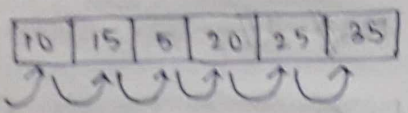
7 moves required. ✓

SEARCHING :-

- Searching is the process of finding a given value position in a list of values.
- It decides whether a search key is present in the data or not.
- It is the algorithmic process of finding a particular item in a collection of items.
- It can be done on internal data structure or an external data structure.
- Searching techniques
- To search an element in a given array, it can be done in following ways:
  1. Sequential search
  2. Binary search

SEQENTIAL SEARCH: / linear

> Sequential search is also called as linear search
> Sequential search starts at the begining of the list and checks every element of the list.
> It is a basic and simple search algorithm.
> Sequential search compares the element with all the other elements given in the list. If the element is matched, it returns the value index, else it returns -1.



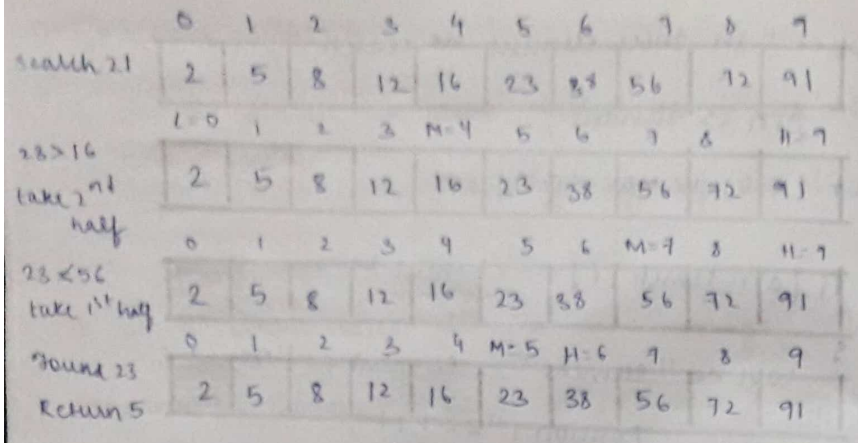
Program:

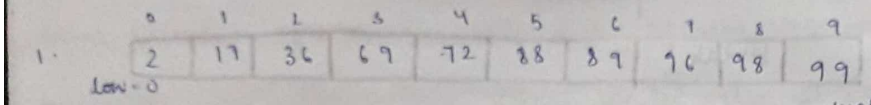
```
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
#define MAX-SIZE 5
int main()
{ int arr_search[max_size], i, element;
  cout << "Linear search example\n";
  cout << "\n Enter" << MAX-SIZE << "Elements for searching";
  for (i=0; i< max-size; i++)        } To read
      cin >> arr-search[i]
      cout << "\n your Data";
  for (i=0; i< MAX-SIZE; i++)        } To print.
  { cout << "\t" << arr-search[i]}
}
```

```
cout << "\n Enter element to search :";
    cin >> element;
for (i=0; i< MAX-SIZE ; i++)
{
  if (arr-search[i] == element)
  { cout << "element" << element << "found :
            position i" << i+1;
    break;
  }
}
  if (i == MAX-SIZE)
    cout << "Element" << element << "Not found";
    getch();
}
```

BINARY SEARCH:

search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.





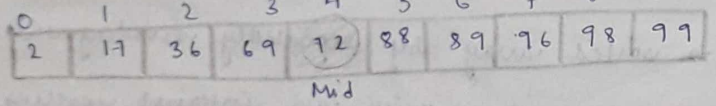
Search element = 89 high = 9

2. Calculate Mid

$$Mid = \frac{low + high}{2}$$

$$= \frac{0+9}{2}$$

$$= \frac{9}{2} = 4.5 \approx 4$$



Mid = 72

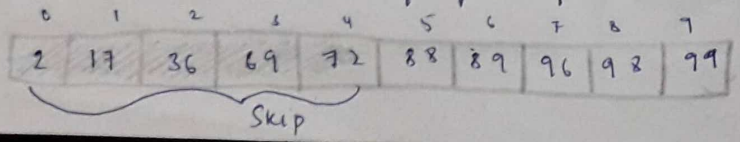
Search element = 89

72 < 89 → key to be searched

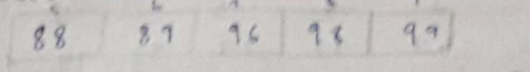
↓

Mid value

{ Note :- search element is greater than the mid value so we can skip left part of the data.



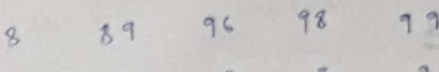
3. Consider new data



4. Calculate Mid

$$Mid = \frac{low + high}{2} \Rightarrow \frac{5+9}{2} = 7$$

5.



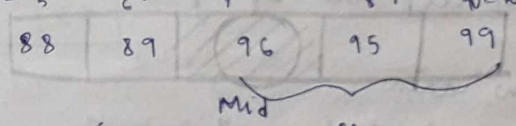
Mid = 96

Search element = 89

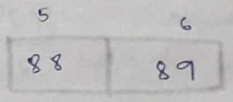
96 > 89 ↓ Mid value

89 < 96 → Mid value ↓ key to be searched

{ Note : Mid value is greather than the search element we can skip right part of the mid.



6.

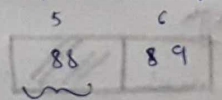


low=5 high=6 $Mid = \frac{5+6}{2}$

7. Mid = 88 $Mid = 5$

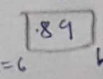
88 < search element (89)

Skip the left part



8.

9.



When low = high we can say that element is found.

8)

low = 0	1	2	3	4	5	6	7	8	high = 9
7	34	56	69	72	87	92	94	111	115

SEARCH ELEMENT = 92

1. Calculate mid

$$Mid = \frac{low + high}{2}$$

$$= \frac{0+9}{2}$$

$$= \frac{9}{2} = 4.5 \approx 4.$$

2. Mid = 72

search element = 92

Mid < search element

72 < 92

So we can skip the left part

3. Consider new data

low 5	6	7	8	9
87	92	94	111	115
				high

4. Calculate mid

$$Mid = \frac{low + high}{2}$$

$$\frac{5+9}{2} = 7//$$

5. Mid = 94

search element = 92

search element < Mid 94 > 92

Mid > search element

So skip the right part.

6.

5	6
87	92
low = 5	Mid = 6

7. ~~Mid = 92~~ ~~87~~ ~~92~~

$$Mid = \frac{5+6}{2}$$

Mid = 5

Mid = 87

search element = 92

87 < 92

Skip the left part

8.

5	6
88	89
skip	

9.

6
89

low = 6 high = 6

When low = high we can say that element is found

Element is found in 6th position.

Program:-

```
#include <iostream>
#include <conio.h>
#include <stdlib.h>
#define MAX_SIZE 5
using namespace std;
int main()
{ int arr_search[max_sise], i, element;
  int low=0, high=max_sise, mid;
cout<< "simple ~~search~~ c++ Binary search example-Array\n";
cout<< "\n Enter" << max_sise << "elements for searching:"
for(i=0; i< max_sise; i++)                    << endl;
 cin >> arr_search[i]
 cout<< "\n your data :";
for (i=0; i<max_sise; i++)
   cout<< "\t" << arr_search[i];
. cout<< "\n Enter element to search : ";
   cin >> element;
   while (low <= high)
{
   mid = (low + high)/2;
   if (arr_search[mid] == element)
{
   cout << "\n search Element :" << element << " : found
   : position : " << mid+1 << ":\n";
   break;
}
   else if (arr_search[mid] < element)
      low = mid + 1;
   else
      high = mid - 1;
}
 if (low > high)
  cout << "\n search element : " << element << " : Not found
                                                \n";
  getch();
}
```

FIBONACCI SEARCH:-

Fibonacci search is a comparison-based technique that uses fibonacci numbers to search an element in a sorted array.

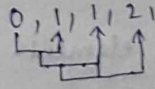
FIBONACCI NUMBERS :-

The first two numbers are 0 & 1 and each subsequent number in the series is equal to the sum of the previous two numbers

$$F(n) = n \quad \text{when } n <= 1$$
$$F(n) = F(n-1) + F(n-2) \quad \text{where } n > 1$$

Fibonacci numbers are

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...



Step1:- Find the smallest number >= n. Let the number be fibm. Let the two fibonacci numbers preceding it be m_1, m_2.

Step 2:- While the array has elements:

1. Compare x with last element of the range covered by m_2.
   - > else if x is less than the element move the fibonacci variables two fibonacci down, indicating elimination of approximately rear two-third of the remaining array
   - > else x is greater than the element, move the three fibonacci variables one fibonacci down. It indicates elimination of the front one-third of the remaining array. Reset offset to index.

Step3:- Since there might be a single element remaining for comparison. Check if m_1 is 1. If yes compare x with that remaining element. If match, return index.

Eg:- Consider no. of elements

10, 20, 30, 40, 50.

Sol:- 0 1 2 3 4
10, 20, 30, 40, 50 } No of elements, n=5.

Search element = 20

Now write fibonacci series upto n.

m_2 m_1 m
0 1 1 2 3

smallest fibonacci number >= n

0 1 1 2 3 5
m_2 m_1 fibm

$m_1 = 3$
$m_2 = 2$
offset = 0
} preceding nos as m_1, m_2

Calculate index $i = \min(offset + m_2, n)$

fibm = 5
$m_1 = 3$
$m_2 = 2$
$i = \min(offset + m_2, n)$
$i = \min(0+2, 5)$
$i = \min(2, 5)$
$i = 2$

Now compare a[2] with search element

x = 20 ; a[2] = 30

'20' : x is less than the index element

x < search element a[i] a[i] > x.

Move two fibonacci variables down, it means

0 1 1 2 3 5 } first case
m_2 m_1 m

two variables down means

0 1 1 2 3 5
m_2 m_1 m

$m_1 = 1$
$m_2 = 1$
fibm = 2

$i = \min(\text{offset} + m_2, n)$

$= \min(0+1, 5)$

$= \min(1, 5)$

$i = 1$, $a[i] = a[1] = 20$

$x = 20$, $a[1] = 20$

nothing but element found at index 1.

Q) 10, 22, 34, 42, 46, 57, 78, 82, 89, 93, 99
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

search element = 99

No. of elements = 11 = n

Now write fibonacci series upto 'n'.

0 1 1 2 3 5 8 (13) 21 34 55
m_2 m_1 fibm

smallest fibonacci number $\geq n$

fibm = 13

$m_1 = 8$

$m_2 = 5$

offset = 0

Calculating index

$i = \min(\text{offset} + m_2, n)$

$i = \min(0 + 5, 11)$

$i = \min(5, 11)$

$i = 5$ $a[i] = a[5] = 57$

$x = 99$

$x > a[i]$

x is greater than index element. One fibonacci down

Now, fibm = 8

$m_1 = 5$

$m_2 = 3$

offset = 5 [it becomes i in the first case when $x > a[i]$]

$i = \min(\text{offset} + m_2, n)$

$i = \min(8, 11)$

$i = 8$

$a[i] = a[8] = 89$

$x = 99$

$x > a[i]$

One fib down again,

fibm = 5

$m_1 = 3$

$m_2 = 2$

offset = 8

$i = \min(\text{offset} + m_2, n)$

$i = \min(8 + 2, 11)$

$i = \min(10, 11)$

$i = 10$

$a[10] = 99 = x$

element found in index 10.

Program:

```
# include <iostream>
using namespace std;
int fibonacci (int a[], int n, int x)
{
    int fibm2=0;
    int fibm1=1
    int fibm = fibm1+fibm2;
    while (fibm<n)
    {
        fibm2 = fibm1
        fibm1 = fibm
        fibm = fibm1+fibm2;
    }
    int offset=0;
    while (fibm>1)
    {
        int i = min (offset+fibm2, n);
        if (a[i]>x)
        {
            fibm = fibm2;
            fibm1 = fibm1- fibm2;
            fibm2 = fibm-fibm1;
        }
        else if (a[i]<x)
        {
            fibm = fibm1;
            fibm1= fibm2
            fibm2 = fibm-fibm1;
        }
        else return i;
}
int main()
{
    int a[50], n, x, j;
    cout<< "Enter the size of the array" << endl;
    cin>>n;
    cout<< "Enter the search element" << endl;
    cin>> x;
    cout<< "enter the array" << endl;
    for (j=0; j<n; j++)
    cin>> a[j];
    cout<< "
    int p= fibonacci (a,n,x);
    cout<< "found at index:" <<p<< endl<<"and position:"
                                                <<p+1;
}
```

BUBBLE SORT

> It is a very simple method that sorts the array elements by repeatedly moving the largest element to the highest index position of the array.

> In bubble sorting, consecutive adjacent pair of elements in the array are compared with each other.

> If the element at the lower index is greater than the element at the higher index.
(The two elements are interchanged so that the element is placed before the bigger one.)

> This process will continue till the list of unsorted elements exhausts.

> This procedure is called Bubble sorting because elements 'bubble' to the top of the list.

TECNNIQUE:-

a) In pass 1, $[A_0]$ and $[A_1]$ are compared, then $[A_1]$ is compared with $[A_2]$; $[A_2]$ is compared with A[3], so on. Finally A[N-2] is compared with A[N-1]. pass 1, involves n-1 comparisons and places the biggest element at the highest index of the array.

b) In pass 2, $[A_0]$ and $[A_1]$ compared with ~~A[N-1]~~. A[1] is compared with A[2], A[2] is compared with A[3], so on. Finally A[N-3] is compared with A[N-2]. pass 2, involves n-2 comparisons and places the second biggest element at the highest index of the array.

c) In pass 3, [A[0]] and [A[1]] compared. Then A[1] compared with A[2], so on. Finally A[N-4] is compared with A[N-3]. pass 3 involves n-3 comparisons and place the third biggest element at the third highest index of the array.

d) In pass n-1, A[0], A[1] are compared, so that A[0] < A[1]. After this step all the elements of the array are arranged in ascending order.

Eg:- A[ ] = {30, 52, 29, 87, 63, 27, 19, 54} N=8.

PASS 1 (N-1) steps

Passes = N-1 = 7 passes

a) Compare 30 and 52, since 30<52 no swapping

b) Compare 52 and 29, since 52>29 swapping required

30, 29, 52, 87, 63, 27, 19, 54.

c) Compare 52 and 87, since 52<87 no swapping

d) Compare 87 and 63, since 87>63 swapping required

30, 29, 52, 63, 87, 27, 19, 54

e) Compare 87 and 27, since 87>27 swapping required

30, 29, 52, 63, 27, 87, 19, 54.

f) Compare 87 and 19, since 87>19 swapping required

30, 29, 52, 63, 27, 19, 87, 54.

g) compare 87 and 54, since 87 > 54, swapping required

30, 29, 52, 63, 27, 19, 54, 87
↳ 1st largest element placed at 1st position.

(largest element placed at the highest index of the array)

Elements are still unsorted.

PASS 2:- (N-2) steps.

30, 29, 52, 63, 27, 19, ~~87, 54~~ 54, 87

a) compare 30 and 29, since 30 > 29, swapping req

29, 30, 52, 63, 27, 19, 54, 87.

b) compare 30 and 52, since 30 < 52, swapping not req

c) compare 52 and 63, since 52 < 63, no swap

d) compare 63 and 27, since 63 > 27, swapping req.

29, 30, 52, 27, 63, 19, 54, 87

e) compare 63 and 19, since 63 > 19, swapping req

29, 30, 52, 27, 19, 63, 54, 87

f) compare 63 and 54, since 63 > 54, swapping req

29, 30, 52, 27, 19, 54, 63, 87
↳ second largest element placed at 2nd highest position

PASS 3:- (N-3) steps

a) 29, 30, 52, 27, 19, 54, 63, 87

a) compare 29 and 30, since 29 < 30, no swap

b) compare 30 and 52, since 30 < 52, no swap

c) compare 52 and 27, since 52 > 27, swap req

29, 30, 27, 52, 19, 54, 63, 87

d) compare 52 and 19, since 52 > 19, swap req

29, 30, 27, 19, 52, 54, 63, 87

e) compare 52 and 54, since 52 < 54, No swap

29, 30, 27, 19, 52, 54, 63, 87
↳ third highest element placed at third position

PASS 4:- (N-4) steps

29, 30, 27, 19, 52, 54, 63, 87

a) compare 29 and 30, since 29 < 30, swap not req

b) compare 30 and 27, since 30 > 27, swap required

29, 27, 30, 19, 52, 54, 63, 87

c) compare 30 and 19, since 30 > 19, swap not required

29, 27, 19, 30, 52, 54, 63, 87

d) compare 30 and 52, since 30 < 52, no swap

29, 27, 19, 30, 52, 54, 63, 87
↳ 4th largest element placed at 4th highest position

PASS 5 :- (N-5) steps.

29, 27, 19, 30, 52, 54, 63, 87

~~a) compare 29 and~~

a) compare 29 and 27, since 29>27 swapping required

27, 29, 19, 30, 52, 54, 63, 87.

b) compare 29 and 19, since 29>19, swapping required

27, 19, 29, 30, 52, 54, 63, 87

c) compare 29 and 30, since 29<30, NO swap

27, 19, 29, 30, 52, 54, 63, 87

→ 5th largest element placed at 5th highest position.

PASS 6 :- (N-6) steps

27, 19, 29, 30, 52, 54, 63, 87

a) compare 27 and 19, since 27>19, swap req.

19, 27, 29, 30, 52, 54, 63, 87

b) compare 27 and 29, since 27<29, no swap

19, 27, 29, 30, 52, 54, 63, 87

→ 6th highest element placed at 6th highest position

PASS 7 :- (N-7) steps

19, 27, 29, 30, 52, 54, 63, 87

a) compare 19 and 27, since 19<27, no swap

19, 27, 29, 30, 52, 54, 63, 87 → 7th highest element placed at 7th highest pos

19, 27, 29, 30, 52, 54, 63, 87

sorted list.

Q) A[] 78, 89, 56, 43, 12, 8, 49, 33, 90, 67, 52, 6.

N = 12

N-1 Passes

PASS 1 (N-1) steps

a) compare 78 and 89, since 78<89, no swap

b) compare 89 and 56, since 89>56, swap req

78, 56, 89, 43, 12, 8, 49, 33, 90, 67, 52, 6.

c) compare 89 and 43, since 89>43, swap required

78, 56, 43, 89, 12, 8, 49, 33, 90, 67, 52, 6.

d) compare 89 and 12, since 89>12, swap required

78, 56, 43, 12, 89, 8, 49, 33, 90, 67, 52, 6.

e) compare 89 and 8, since 89>8, swap required

78, 56, 43, 12, 8, 89, 49, 33, 90, 67, 52, 6.

f) compare 89 and 49, since 89>49, swap required

78, 56, 43, 12, 8, 49, 89, 33, 90, 67, 52, 6.

g) compare 89 and 33, since 89>33, swap required

78, 56, 43, 12, 8, 49, 33, 89, 90, 67, 52, 6.

h) compare 89 and 90, since 89<90, no swap

i) Compare 90 and 67, since 90>67, swap required

78, 56, 43, 12, 8, 49, 33, 89, 67, 90, 52, 6

j) compare 90 and 52, since 90>52, swap required

78, 56, 43, 12, 8, 49, 33, 89, 67, 52, 90, 6

k) compare 90 and 6, since 90>6, swap required

78, 56, 43, 12, 8, 49, 33, 89, 67, 52, 6, 90.

↳ 1st highest element

PASS 2 :- (N-2) steps

a) compare 78 and 56, since 78>56, swap required

56, 78, 43, 12, 8, 49, 33, 89, 67, 52, 6, 90.

b) compare 78 and 43, since 78>43, swap required

56, 43, 78, 12, 8, 49, 33, 89, 67, 52, 6, 90

c) compare 78 and 12, since 78>12, swap required

56, 43, 12, 78, 8, 49, 33, 89, 67, 52, 6, 90.

d) compare 78 and 8, since 78>8, swap required

56, 43, 12, 8, 78, 49, 33, 89, 67, 52, 6, 90.

e) compare 78 and 49, since 78>49, swap required

56, 43, 12, 8, 49, 78, 33, 89, 67, 52, 6, 90.

f) compare 78 and 33, since 78>33, swap required

56, 43, 12, 8, 49, 33, 78, 89, 67, 52, 6, 90.

g) compare 78 and 89, since 78<89, no swap

56, 43, 12, 8, 49, 33, 78, 89, 67, 52, 6, 90.

h) compare 89 and 67, since 89>67, swap required

56, 43, 12, 8, 49, 33, 78, 67, 89, 52, 6, 90.

i) compare 89 and 52, since 89>52, swap required

56, 43, 12, 8, 49, 33, 78, 67, 52, 89, 6, 90

j) compare 89 and 6, since 89>6, swap required

56, 43, 12, 8, 49, 33, 78, 67, 52, 6, 89, 90.

↳ second highest element

PASS 3 (N-3) steps

a) compare 56 and 43, since 56>43, swap required

43, 56, 12, 8, 49, 33, 78, 67, 52, 6, 89, 90.

b) compare 56 and 12, since 56>12, swap required

43, 12, 56, 8, 49, 33, 78, 67, 52, 6, 89, 90.

c) compare 56 and 8, since 56>8, swap required

43, 12, 8, 56, 49, 33, 78, 67, 52, 6, 89, 90.

d) compare 56 and 49, since 56>49, swap required

43, 12, 8, 49, 56, 33, 78, 67, 52, 6, 89, 90.

e) compare 56 and 33, since 56>33, swap required

43, 12, 8, 49, 33, 56, 78, 67, 52, 6, 89, 90.

f) compare 56 and 78, since 56<78, no swap

g) compare 78 and 67, since 78>67, swap required

43, 12, 8, 49, 33, 56, 67, 78, 52, 6, 89, 90.

h) compare 78 and 52, since 78>52, swap required

43, 12, 8, 49, 33, 56, 67, 52, 78, 6, 89, 90.

i) compare 78 and 6, since 78>6, swap required

43, 12, 8, 49, 33, 56, 67, 52, 6, 78, 89, 90

↳ 3rd highest element.

PASS 4: (N-4) steps

a) compare 43 and 12, since 43>12, swap req

12, 43, 8, 49, 33, 56, 67, 52, 6, 78, 89, 90

b) compare 43 and 8, since 43>8, swap req

12, 8, 43, 49, 33, 56, 67, 52, 6, 78, 89, 90.

c) compare 43 and 49, since 43<49, no swap

d) compare 49 and 33, since 49>33, swap req

12, 8, 43, 33, 49, 56, 67, 52, 6, 78, 89, 90.

e) compare 49 and 56, since 49<56, no swap

f) compare 56 and 67, since 56<67, no swap

g) compare 67 and 52, since 67>52, swap required

12, 8, 43, 33, 49, 56, 52, 67, 6, 78, 89, 90

h) compare 67 and 6, since 67>6, swap required

12, 8, 43, 33, 49, 56, 52, 6, 67, 78, 89, 90.

↳ 4th highest element

PASS-5 (N-5) steps

a) compare 12 and 8, since 12>8, swap required

8, 12, 43, 33, 49, 56, 52, 6, 67, 78, 89, 90.

b) compare 12 and 43, since 12<43, no swap

c) compare 43 and 33, since 43>33, swap required

8, 12, 33, 43, 49, 56, 52, 6, 67, 78, 89, 90.

d) compare 43 and 49, since 43<49, no swap

e) compare 49 and 56, since 49<56, no swap.

f) compare 56 and 52, since 56>52, swap required

8, 12, 33, 43, 49, 52, 56, 6, 67, 78, 89, 90.

g) compare 56 and 6, since 56>6, swap required.

8, 12, 33, 43, 49, 52, 6, 56, 67, 78, 89, 90.

↳ 5th highest element

PASS-6 (N-6) steps

a) compare 8 and 12, since 8<12, no swap

b) compare 12 and 33, since 12<33, no swap required

c) compare 33 and 43, since 33<43, no swap

d) compare 43 and 49, since 43<49, no swap

e) compare 49 and 52, since 49<52, no swap

f) compare 52 and 6, since 52>6, swap required

8, 12, 33, 43, 49, 6, 52, 56, 67, 78, 89, 90

↳ 6th highest element.

PASS 7 (N-7) steps

a) compare 8 and 12, since 8<12, no swap

b) compare 12 and 33, since 12<33, no swap

c) compare 33 and 43, since 33<43, no swap

d) compare 43 and 49, since 43<49, no swap

e) compare 49 and 6, since 49>6, swap required

8, 12, 33, 43, 6, 49, 52, 56, 67, 78, 89, 90

↳ 7th highest element.

PASS 8 (N-8) steps.

a) compare 8 and 12, since 8<12, no swap

b) compare 12 and 33, since 12<33, no swap

c) compare 33 and 43, since 12<43, no swap

d) compare 43 and 6 since 43>6, swap req

8, 12, 33, 6, 43, 49, 52, 56, 67, 78, 89, 90

└→ 8th highest element.

PASS 9 (N-9) steps

a) compare 8 and 12, since 8<12, no swap

b) compare 12 and 33, since 12<33, no swap

c) compare 33 and 6, since 33>6, swap req

8, 12, 6, 33, 43, 49, 52, 56, 67, 78, 89, 90

└→ 9th highest element.

PASS 10 (N-10) steps

a) compare 8 and 12, since 8<12, no swap

b) compare 12 and 6, since 12>6, swap required

8, 6, 12, 33, 43, 49, 52, 56, 67, 78, 89, 90

└→ 10th highest element.

~~c) compare~~

Pass 11 (N-11) steps

a) compare 8 and 6, since 8>6, swap required

6, 8, 12, 33, 43, 49, 52, 56, 67, 78, 89, 90.

Sorted list

PROGRAM:-

```
#include<iostream>
#include<stdio.h>
#include<stdlib.h>
using namespace std;
int main()
{
  int arr[50], i, j, n;
  cout<<"simple c++ bubble sort example\n";
  cout<<"enter the size of the array";
  cin>>n;
  cout<<"enter your data"<<endl;
  for(i=0; i<n; i++)
  cin>arr[i];
  cout<<"your data"<<endl;
  for(i=0; i<n; i++
  {
  cout<<arr[i]<<"\t";
  }
  cout<<"\nsorted array"<<endl;
  for(i=0; i<n; i++)
  {
    for(j=0; j<n-i-1; j++)
    {
      if(arr[j+1]<arr[j])
        arr[j] = arr[j]+arr[j+1];
        arr[j+1] = arr[j]-arr[j+1];
        arr[j] = arr[j]-arr[j+1];
    }
  }
  cout<<arr;
```

```
for (i=0; i<n; i++)
    cout << a[i] << " ";
}
```

INSERTION SORT:-

1. The insertion sort works very well when the no. of elements are very less.
2. This technique is similar to the way a librarian keeps the books in the shelf.

Initially all the books are placed on shelf according to their access number.

When a student returns the book to the librarian she compares the access number of that book with all other books of access numbers and inserts it into the correct position, so that all books are arranged in order with respect to their access numbers.

METHOD:-

PASS 1: The 1st element in the array is itself sorted

PASS 2: 2nd element is inserted either before or after 1st element, so that 1st and 2nd elements are sorted

PASS 3: 3rd element is inserted into its proper place, that is before 1st and 2nd (or) between 1st and 2nd elements (or) before 1st and 2nd elements.

PASS 4: Element is inserted into its proper place in 1st, 2nd, 3rd ~~and 4th~~. So at 1st, 2nd, 3rd, 4th elements are sorted.

	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
eg:	39	9	45	63	18	81	108	54	72	36
	0	1	2	3	4	5	6	7	8	9

Pass 1:- 9 39 45 63 18 81 108 54 72 36

(9 sorted; 45 greater (so no prob))

Pass 2: 9 39 45 63 18 81 108 54 72 36

(63 greater (so no prob))

Pass 3: 9 39 45 63 18 81 108 54 72 36

(18 greater than 9; less than 63)

So '18' will be placed between 9 & 39

Pass 4: 9 18 39 45 63 81 108 54 72 36

(81 > 63 so no prob)

Pass 5: 9 18 39 45 63 81 108 54 72 36

(108 greater than 81 so no prob)

Pass 6: 9 18 39 45 63 81 108 54 72 36

(54 greater than 45; less than 108)

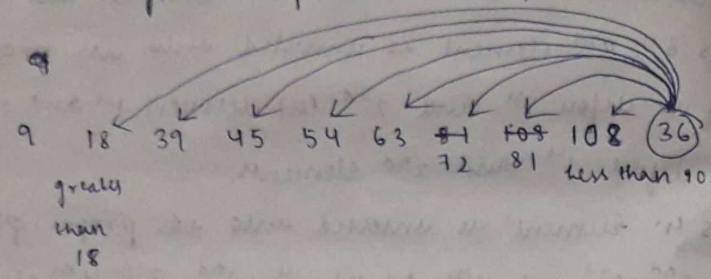
'54' is correct position as in between 45 & 63.

PASS 7:- 9 18 39 45 54 63 81 108 72 36

greater than 63 — less than 108

the correction position for 72 is in b/w 63 & 81

PASS 8:-



correct position for 36 is in b/w 18 and 39.

PASS 9:- 9 18 36 39 45 54 63 72 81 108

(elements in sorted order)

Q) 45, 23, 99, 12, 38, 67, 74, 10, 22, 55, 9

PASS 1:- 23, 45, 99, 12, 38, 67, 74, 10, 22, 55, 9

PASS 2:- 23, 45, 99, 12, 38, 67, 74, 10, 22, 55, 9 (sorted)

PASS 3:- 12, 23, 45, 99, 38, 67, 74, 10, 22, 55, 9

PASS 4:- 12, 23, 38, 45, 99, 67, 74, 10, 22, 55, 9

PASS 5:- 12, 23, 38, 45, 67, 99, 74, 10, 22, 55, 9

PASS 6:- 12, 23, 38, 45, 67, 74, 99, 10, 22, 55, 9

PASS 7:- 10, 12, 23, 38, 45, 67, 74, 99, 22, 55, 9

PASS 8:- 10, 12, 22, 23, 38, 45, 67, 74, 99, 55, 9

PASS 9:- 10, 12, 22, 23, 38, 45, 55, 67, 74, 99, 9

PASS 10:- 9, 10, 12, 22, 23, 38, 45, 55, 67, 74, 99.

(elements in sorted order.

PROGRAM:-

```
#include <iostream>
using namespace std;
int main()
{
  int i, j, n, temp, a[30];
  cout<<"simple c++ insertion sort example\n">>
  cout<<"Enter the no. of elements:";
  cin>>n;
  cout<<"\n Enter the elements \n";
  for(i=0; i<n; i++)
  {
     cin>>a[i];
  }
  for(i=1; i<n; i++)
  {
     temp=a[i];
     for(j=i-1; j>=0 && a[j]>temp; j--)
        a[j+1]=a[j];
     a[j+1]=temp;
  }
  cout<<"\n Sorted list is as follows\n";
  for(i=0; i<n; i++)
  {
     cout<<a[i]<<" ";
  }
}
```

SELECTION SORT:-

Consider an array with N elements. First find the smallest value in the array and place it in the first position. Repeat this procedure until the entire array is sorted.

METHOD:-

In pass 1 find the smallest value position in the array and then swap a[pos] and a[0], thus a[0] is sorted.

In pass 2 find the position of the second smallest value in the array and then swap a[pos] and a[1]. Thus a[1] is sorted.

Thus procedure is repeated until all the elements get sorted. Eg:- 39 (9) 81 45 90 27 72 18
smallest

PASS 1:- 9 39 81 45 90 27 72 (18) — next smallest element; make comparison

PASS 2: 9 18 81 45 90 (27) 72 39 — next smallest element (81>27) swap

PASS 3: 9 18 27 45 90 81 72 (39) — next smallest element (45>90 - swap)

PASS 4:- 9 18 27 39 ~~45~~ 90 81 72 (45) (90>45 swap)

PASS 5: 9 18 27 39 45 81 (72) 90 — 81>72 - swap

PASS 6:- 9 18 27 39 45 72 (81) 90 — next smallest element

PASS 7:- 9 18 27 39 45 72 81 90

Q) 34, 42, 12, 56, 23, (8) smallest element

PASS 1:- ~~8, 34, 42, (12), 56, 23~~

PASS 2:- ~~8, 12, 42, 34, 56, 23~~

PASS 1:- 8, 42, (12), 56, 23, 34

PASS 2:- 8, 12, 42, 56, (23), 34

PASS 3:- 8, 12, 23, 56, 42, (34)

PASS 4:- 8, 12, 23, 34, (42), 56 — next smallest element

PASS 5:- ~~9, 18~~, 8, 12, 23, 34, 42, 56

PROGRAM:

```
#include <iostream.h>
using namespace std;
int main()
{  int i, j, n, temp, a[30], loc, min;
   cout << " simple c++ selection sort example\n";
   cout << " Enter the no. of elements : ";
   cin >> n;
   cout << "\n Enter the elements \n";
   for (i=0; i<n; i++)
   {  cin >> a[i];
   }
   for (i=0; i<n-1; i++)
   {  min = a[i];
      loc = i;
      for (j=i+1; j<n; j++)
      {  if(a[j] < a[i])        // min
         {  min = a[j];
            loc = j;
         }
      }
      temp = a[i];
      a[i] = a[loc];
      a[loc] = temp;
   }
   for
   cout << "\n sorted list is as follows \n";
   for (i=0; i<n; i++)
   cout << a[i] << " ";
}
```

DATA STRUCTURE :-

> Data structure is basically a group of data elements that are put together under one name, and which defines a particular way of storing and organizing data in a computer so that it can be used efficiently

> A data structure is a particular way of organizing data in a computer so that it can be used effectively

Why we need Data structures ?

> Computer science is all about storing and computing from a given data. So studying data structures helps you deal with different ways of arranging, processing and storing data. All codes are made for real time purpose so data structure allows user to provide / use / handle data in different ways.

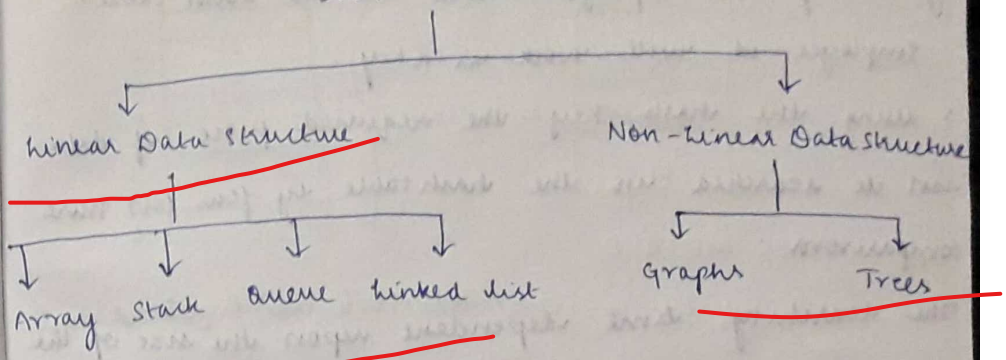
> The data structure is considered to be linear if the data elements construct a sequence of a linear list. The elements are adjacently attached to each other and in a specified order. It consumes linear memory space, the data elements are required to store in a sequential manner in memory.

> The data element are visited sequentially where only a single element can be directly reached

> Non linear data structure does not arrange the data consecutively rather it is arranged in sorted order. In this, data elements can be attached to more than one element exhibiting the hierarchical relationship which involves the relationship between the child, parent and grand parent.

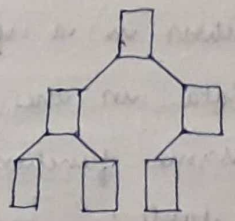
> In the non-linear data structure, the traversal of data elements and insertion or deletion are not done sequentially.

DATA STRUCTURE



LINEAR :-

NON LINEAR :-



HASHING :-

> Hashing is the process of mapping large amount of data item to a smaller table with the help of hashing function.

Adv :- This method is used to handle the vast amount of data.

HASH TABLE :-

> Hash table is a data structure used for storing and retrieving data very quickly.

> Insertion of data in the hash table is based on the key value.

> Every entry in the hash table is associated with some thing.

Eg:- storing an employee record in the hash table.

Employee id will work as a key.

> using the hash key the required piece of data can be searched in the hash table by few (or) more comparisons.

The searching time dependent upon the size of the hash table.

HASH FUNCTION:-

> Hash function is a function which is used to put the data in the hash table. Hence one can use the same function to retrieve the data from the hash table.

Thus hash function is used to implement the the hash table.

* The integer returned by the hash function is called Hash key.

Hash Key (H) = Key % Table size

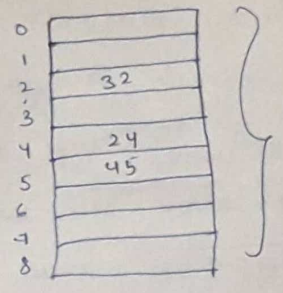
Eg:- 24, 32, 45

Table size = 10

Hash Key (H) = 24 % 10 = 4

Hash Key (H) = 32 % 10 = 2

Hash Key (H) = 45 % 10 = 5



Each index is called as "Bucket"

TYPES OF HASH FUNCTIONS:-

There are various types of hash functions that are used to place the record in the hash table.

1. Division method
2. Mid-Square Method
3. Multiplicative Hash function
4. Digit Folding

DIVISION METHOD:-

The hash function depends on the remainder of division

Eg:- 22, 24, 26, 89, 75, ...

Table size = 10

H(Key) = Key % Table size

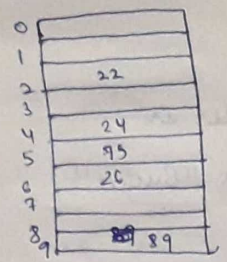
H(22) = 22 % 10 = 2

H(24) = 24 % 10 = 4

H(26) = 26 % 10 = 6

H(89) = 89 % 10 = 9

H(75) = 75 % 10 = 5



MID SQUARE METHOD

In mid square method the key is squared and middle [or] mid part of the result is used as index.

Consider, key - 3111 ; then,

find square of a given number,

$(3111)^2 = 96\underline{783}21$

↳ Middle Num

The Hash index is 783, then place 3111 into 783

MULTIPLICATIVE HASH FUNCTION:-

In multiplication hash function the given record is multiplied with some constant value.

$H(Key) = floor(p * (key * A))$

↳ p

If Key = 107, and p = 50 then

$H(Key) = floor(p * (key * A))$

Author Donald Knuth Suggested A = 0.618033

floor (50 * (107 * 0.618033))

= floor (3306.481...)

H(Key) = 3306

At location 3306, the record 107 will be placed.

DIGIT FOLDING:-

The key is divided into separate parts and using simple function these parts are combined to produce the hash key.

Eg:- num = ~~1237654~~ 1237654

↳ If you want to place the given record into hash table.

First divide the given number into parts.

123, 765, 4...

Now combine = 123 + 765 + 4 = 892

"892" will be the index to place the record i.e, "1237654"

COLLISION:-

> The hash function is a function that return the key value using which the record can be placed in the hash table.

> The function needs to be designed very carefully and it should not return the same hash key address for two different records.

Def:- The main function returns the same hash key for more than one record is called collision and two same hash keys returned for different record is called collision.

Eg:- 12, 21, 32, 44, 76, 67, 86.

Assume Table size = 10.

0	
1	21
2	12
3	
4	44
5	96
6	67
7	
8	
9	

H(12) = 12 % 10 = 2

H(21) = 21 % 10 = 1

H(32) = 32 % 10 = 2 Collision occurred

H(44) = 44 % 10 = 4

H(96) = 96 % 10 = 6

H(67) = 67 % 10 = 7

H(86) = 86 % 10 = 6 collision required

To avoid collision, have some resolution techniques

Collision Resolution Techniques :-

If collision occurs then it should be handled by applying some techniques

1. chaining
2. Open addressing (linear probing)
3. Quadratic Probing
4. Double hashing

1. Chaining :-

. In collision handling method chaining is a concept which introduces an additional field with data i.e, chain. Separate table chain is maintained for colliding data. Data collision occurs when a linked list (chain) is maintained at the home bucket

Eg: 121, 32, 42, 67, 76, 41, 82, 89, 19 - -

Table size = 10

. H(key) = 121 % 10 = 1

H(32) = 32 % 10 = 2

H(42) = 42 % 10 = 2 *

H(67) = 67 % 10 = 7

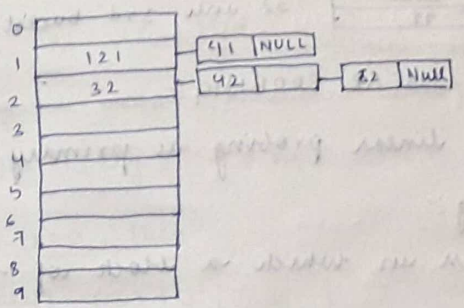
H(76) = 76 % 10 = 6

H(41) = 41 % 10 = 1 *

H(82) = 82 % 10 = 2 *

H(89) = 89 % 10 = 9

H(19) = 19 % 10 = 9 *



2. OPEN - ADDRESSING (LINEAR PROBING) :-

> Easiest method of handling collision

> When collision occurs, i.e; when two records demand for the same bucket in the hash table then collision can be solved by placing "the second record linearly down whenever the empty bucket is found".

Eg :- Records = 31, 42, 36, 99, 45, 32, 76 - -

H (key) = key % table size

H(31) = 31 % 10 = 1

H(42) = 42 % 10 = 2

H(86) = 86 % 10 = 6

H(99) = 99 % 10 = 9

H(45) = 45 % 10 = 5

H(32) = 32 % 10 = 2 *

$H(76) = 76 \% 10 = 6$ *

0	NULL
1	31
2	42
3	32
4	NULL
5	45
6	86
7	96
8	NULL
9	99

← "32" but already filled with some other number.

So go linearly down. Next bucket is free, so insert 32 into 3rd bucket.

PROBLEM WITH LINEAR PROBING :-

> Problem with linear probing is primary clustering

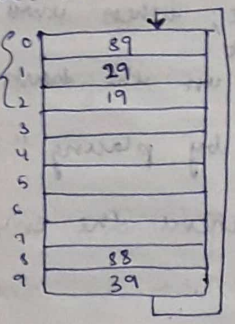
Primary clustering :-

It is a process in which a block of data is formed in the hash table when collision resolved.

Eg: 39, 89, 29, 19, 88 - -

Table size = 10

Cluster is formed



$H(39) = 39 \% 10 = 9$

$H(89) = 89 \% 10 = 9$

$H(29) = 29 \% 10 = 9$

$H(19) = 19 \% 10 = 9$

$H(88) = 88 \% 10 = 8$

3. QUADRATIC PROBING :-

Quadratic probing operates by taking the original hash value and adding successive hash values of an arbitary quadratic polynomial to the starting value.

$$H(Key) = Hash(Key + i^2) \% m$$

m can be table size or prime no.

Eg: 15, 22, 27, 84, 66, 37, - - -

Table size = 10

$H(15) = 15 \% 10 = 5$

$H(22) = 22 \% 10 = 2$

$H(27) = 27 \% 10 = 7$

$H(84) = 84 \% 10 = 4$

$H(56) = 56 \% 10 = 6$

$H(37) = 37 \% 10 = 7$ * (collision)

7th bucket has already an element. Hence apply Quadratic probing.

$H(Key) = H(Key + i^2) \% m$

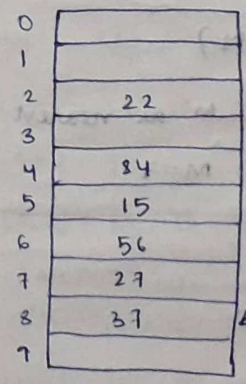
Consider $i = 0$, then

$H(key) = (37 + 0^2) \% 10 = 7$

when $i = 1$

$H(key) = (37 + 1^2) \% 10 = 8$

8th bucket is empty so simply insert 37 into 8th bucket, if 8th bucket is not empty then continue with $i = 2, 3 - - -$



'37' inserted into 8th bucket by using Quadratic Probing.

4. DOUBLE HASHING :-

Double hashing is technique in which a second hash ~~tag~~ function is applied to the key when a collision occurs.

$$H_1(key) = key \,\%\, table\ size$$
$$H_2(key) = M - (key \,\%\, M)$$

M is a prime numbers smaller than the size of the table.

Eg: 37, 49, 25, 62, 33, 17

Table size = 10

H(37) = 37 % 10 = 7

H(49) = 49 % 10 = 9

H(25) = 25 % 10 = 5

H(62) = 62 % 10 = 2

H(33) = 33 % 10 = 3

H(17) = 17 % 10 = 7 collision occured

$H_2(key) = M - (key \,\%\, M)$

Table size 10, take 'M' as nearest prime number to the table size M=7

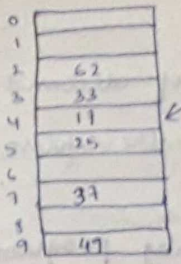
$H_2(17) = 7 - (17 \,\%\, 7)$

$= 7 - 3$

$= 4$

$H_2(17) = 4$

insert 17 into 4th bucket



17 inserted in the 4th bucket by applying Double hashing.

REHASHING :-

A rehashing is a technique in which table is resized i.e; the size of the table is doubled by creating a new table.

- There are three situations in which rehashing is required
  - When table is completely full
  - With quadratic probing when the table is filled half
  - When insertions fail due to overflow.

Eg: Consider some elements

37, 90, 55, 22, 17, 49, 87 - -

H(key) = key % table size

H(37) = 37 % 10 = 7

H(90) = 90 % 10 = 0

H(55) = 55 % 10 = 5

H(22) = 22 % 10 = 2

H(17) = 17 % 10 = 7

H(49) = 49 % 10 = 9

H(87) = 87 % 10 = 7

APPLICATIONS OF HASHING:-

> In compilers to keep track of declared variables
> For online spelling checking the hash functions are used.
> Hashing helps in game playing programs to store the moves made.
> For browser program while crawling the web pages hashing is used

PERFECT HASH FUNCTIONS:-

> Difference between General hash function and perfect Hash function is, Here we are defining the hash functions to generate the Hash Keys.
> In General, hash function was first devised and then the data were processed.
> If the body of the data is fixed and a hash function can be devised after the data are known such a function may really be a perfect hash function. If it hashes all keys on the first attempt
> Perfect Hash functions may be used to implement a look-up table with constant worst case time

QUICK SORT:-

> Quick sort is also known as partition exchange sort
> Quick sort algorithm works by divide and conquer strategy to divide a single unsorted array into two smaller sub-arrays.

ALGORITHM:-

1. Select an element pivot from the array elements
2. Rearrange the elements in the array in such a way that elements ~~the~~ that are less than the pivot appear before the pivot and all the elements greater than the pivot element come after it.
3. After such a partitioning, the pivot is placed in its final position this is called partition operation
4. Recursively sort the two sub-arrays thus obtained (one sub list of values smaller than that of the pivot element and the other having higher value elements)

TECHNIQUE:-

1. Set the index of the first element in the array to loc and left variables. Also set the index of the last elements of the array to the right variable.
   loc = 0, left = 0, right = n-1
2. Start from the element pointed by right and scan the array from right to left, comparing each

element on the way with the element pointed by the variable loc.

i.e; a[loc] should be less than a[right]

> If this is the case, then simply continue comparing until right becomes equal to LOC. Once right = loc, it means the pivot has been placed in its correct position.

> However, if at any point, we have a[loc] > a[right] then interchange the two values and jump to step 3.

> set loc = right

> Start from the element pointed by left and scan the array from left to right, comparing each element on the way with the increment pointed by loc.

i.e; a[loc] should be greater than a[left].

> If that is the case, then simply continue comparing until left becomes equal to loc. Once left = loc it means the pivot has been placed in its correct position.

> If at any point, we have a[loc] < a[left], then interchange the two values and jump to step 2.

> set loc = left.

Eg: 27 10 36 18 25 45 n=6

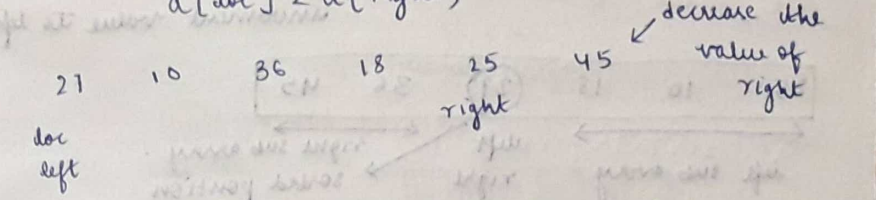
0 1 2 3 4 5

loc, left

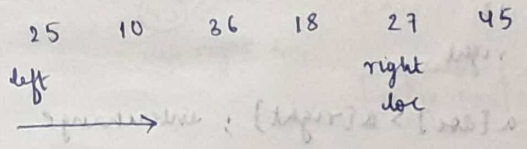
right

> Initially scan from right to left

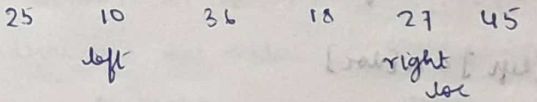
a[loc] < a[right]

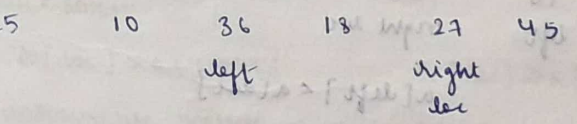


> a[loc] > a[right], interchange two values, loc = right

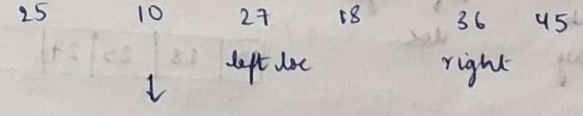


> from left to right (a[loc] > a[left]) → increment the value of left.



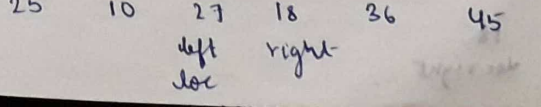
> 

a[loc] < a[left], interchange values set loc = left



Scan from right to left, since a[loc] < a[right]

→ decrement the value of right

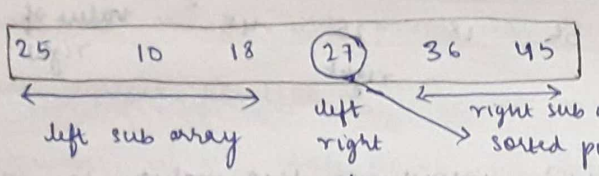


a[loc] > a[right], interchange two values, loc = right

25 10 18 27 36 45

left, right, loc

Start scanning from left to right, a[loc] ≥ a[left]
increment value to left



25 10 18
left loc right

a[loc] > a[right], interchange

18 10 25
left right loc

⟶ a[left] < a[loc]

18 10 25 ⟶ increment
left right loc

a[left] < a[loc]

18 10 25 ← sorted position
left sub array
left right loc

10	18	25	27	

18 10
left loc right

← a[loc] > a[right], interchange

10 18
left loc right

a[left] > a[right loc]

10 > 18, increment to left

36	45

left loc right

← a[loc] < a[right]
36 < 45
decrement right

36 ← sorted position
left loc
right

10	18	25	27	36	

45 ←
left right loc

10	18	25	27	36	45

elements are sorted

QUICK SORT:-

① a[loc] < a[right] — decrement the value of right
a[loc] < a[left] — interchange the values

~~QUICK SORT:-~~

② a[loc] > a[right] — interchange values
a[loc] ≥ a[left] — Increment the value of left

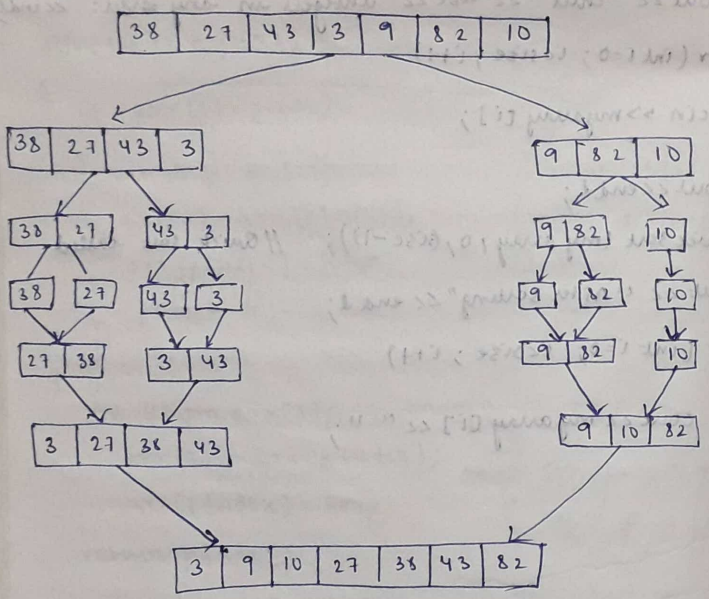
PROGRAM:-

```
#include<iostream>
using namespace std;
int Partition (int arr[], int s, int e)
{
  int pivot = arr[s];
  int pIndex = e;
  for (int i=e ; i>s; i--)
  {
    if (arr[i]> pivot)
    {
      int temp = arr[i];
      arr[i] = arr [pIndex];
      arr[pIndex] = temp;
      pIndex--;
    }
  }
    int temp = arr[s];
    arr[s] = arr[pIndex];
    arr[pIndex] = temp;
  return pIndex;
}
void quicksort (int arr[], int s, int e)
  {
    if (s<e)
    {
      int p= Partition (arr, s, e);
      Quicksort (arr, s, (p-1)); // left partition
      Quicksort (arr, (p+1), e ); // right partition
    }
  }
```

```
int main()
{
  int size = 0;
  cout << "enter size of array: " << endl;
    cin >> size;
    int myarray [size];
  cout << "enter " << size << " integers in any order: " << endl;
  for (int i=0; i<size; i++)
  { cin >> myarray [i];
  }
  cout << endl;
  Quicksort (myarray, 0, (size-1)); // Quick sort called
  cout << "After sorting" << endl;
  for (int i=0; i<size ; i++)
  { cout << myarray [i] << " ";
  }
}
```

MERGE SORT:-

Once the size becomes 1, the merge processes comes into action and starts merging arrays back till arrays back till the complete array is merged



PROGRAM:-

```
# include <iostream>
  using namespace std;
void merge (int arr[], int l, int m, int r, int size)
{
      int i = l;
      int j = m+1;
      int k = l;
    // create temp array
        int temp[size];
    while (i<=m && j<=r)
{     if (arr[i]<=arr[j])
      { temp[k] = arr[i]
        k++, i++;
      }
      else
      { temp[k] = arr[j];
        k++, j++;
      }
}
  // copy the remaining elements of first half, if there any
  while (i<=m)
{   temp[k] = arr[i];
    i++, k++;
}
  while (j<=r)  // copy the remaining elements of second half, if there any
{   temp[k] = arr[j];
    k++, j++;
}
  // copy the temp array to original array
  for (int p=l; p<size r; p++)
    arr[p] = temp[p];
}
// l is for left under and r is for right under of the sub array
void mergesort (int arr[], int l, int r, int size)
    if (l<r)
{     // finding mid point
```

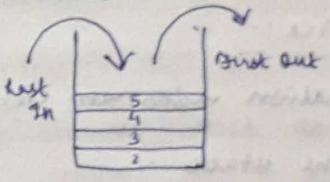
l = first index
r = last index

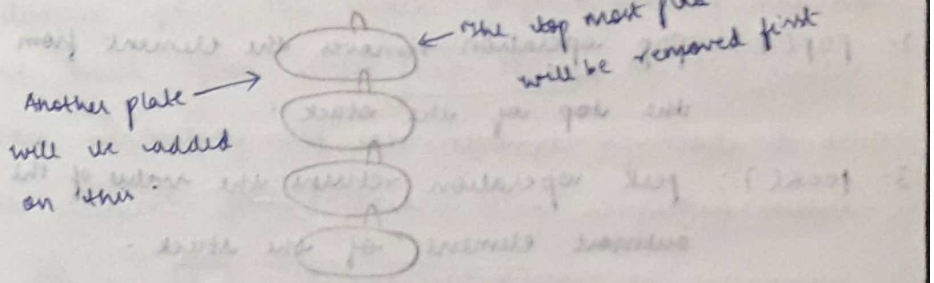
```
    int m=(l+r)/2;
//recursive mergesort for first and second halves.
    mergesort(arr, l, m, size);
    mergesort(arr, m+1, r, size);
 // merge
    merge(arr, l, m, r, size);
    }
}
 int main()
{ cout<<"Enter size of array: "<<endl;
  int size;
  cin>>size;
  int arr[size];
  cout<<"enter"<<size<<"integers in any order: "<<endl
  for(int i=0; i<size; i++)
   cin>>arr[i];
  cout<<"Before sorting"<<endl;
  for(int i=0; i<size; i++)
    cout<<arr[i]<<" ";
    cout<<endl;
    mergesort(arr, 0, (size-1), size);
    cout<<"After sorting"<<endl;
    for(int i=0; i<size; i++)
      cout<<myarray[i]<<" ";
}
```


STACK :-

A stack is a linear data structure which stores elements in an ordered manner like a pile of plates.

- The elements in a stack are added and removed only from one end which is called the "Top".
- Stack is called a "LIFO" (last in first out) data structure, as that was inserted last is the first one to be taken out.





Use of stack:

In order to keep track of the returning point of each active function.

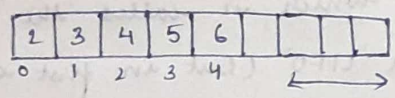
Array Representation of stack:-

- Stack can be represented as a linear array
- Every stack has a variable called 'top' associated

with it which is used to store the address of the top most element of the stack.

> There is another variable 'max' used to store maximum number of elements that the stack can hold.

> If TOP = NULL, it indicates the stack is empty and if TOP = MAX-1, means stack is full



Top = 4 we can insert 4 more element.

OPERATIONS OF A STACK:-

1. push(): push operation adds an element to the top of the stack.
2. pop(): pop operation removes the element from the top of the stack.
3. peek(): peek operation returns the value of the outmost element of the stack.

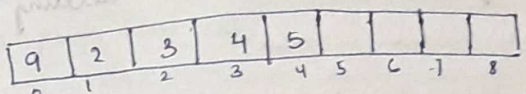
PUSH OPERATION:-

> Push operation is used to insert an element into the stack.

> The new element is added at the top most position of the stack.

Before inserting the value, we must first check if TOP = MAX-1 because in this case, stack is full and no more insertions can be done.

> If the stack is full, in that case if you attempt to insert a value in a stack that is already full overflow will be occured.

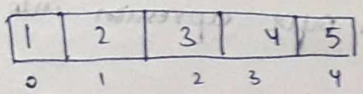


Top = 4

> To insert an element 7, we first check TOP=Max-1 if it is true stack is full and we can't insert if the condition is false, we increment "top" from 4 to 5th index then we simply insert the element into the stack.

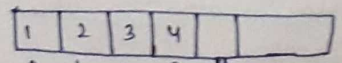
POP OPERATION:-

> This operation is used to delete the topmost element from the stack. Before deleting the value we must first check TOP= NULL, in this case stack is empty. If an attempt is made to delete the element from stack, stack underflow occurs.



Top = 4

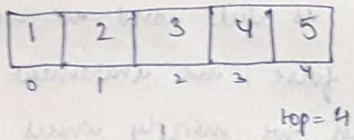
> To delete the top most element, we should first check if TOP = NULL, condition is false. There are some elements in stack, then decrement the top i.e, element get deleted from the stack.



PEEK OPERATION:-

> It is an operation that returns the value of the top most element of the stack without deleting it from the stack.

> In peek operation first check if the stack is empty i.e; TOP=NULL means no elements in stack.



> Peek operation will return '5', it is the value of the top most element of the stack.

APPLICATIONS OF STACK:-

1. Reversing a list
2. Parenthesis checker
3. Conversion of an expression into a postfix expression
4. Evaluation of a post fix expression
5. Conversion of an infix expression into a prefix expression
6. Evaluation of a prefix expression
7. Recursion.

```
#include <iostream>
using namespace std;
int stack[100], n=100, top=-1;
void push(int val)
{
  if (top>=n-1)
  cout<<"underflow occured"<<endl;
  else
  {
    top++;
    stack[top]=val;
  }
}
void pop()
{
  int if (top<=-1)
  cout<<"overflow occured"<<endl;
  else
  {
    cout<<"the popped element is"<<stack[top]<<endl;
    top--;
  }
}
void peek()
{
  if (top>=0)
  cout<<"the stack elements are:";
  for(int i=0; i<n; i++)
    cout<<stack<<" ";
    cout<<endl;
}
```

```
int main()
{ int ch, val;
 cout << "1) Push into stack" << endl;
 cout << "2) pop from stack" << endl;
 cout << "3) Display stack" << endl;
 cout << "4) Exit" << endl;
    do
 {
    cout << "enter choice" << endl;
    cin >> ch;
   switch (ch)
{
    case 1:
   { cout << enter the value to be pushed:";
     cin >> val;
     push (val);
     break;
   }
   case 2:
   { ~~cout << data~~ pop();
             break;
   }
   case 3:
   {    peek();
        break;
   }
   case 4:
   { cout << "exit" << endl;
   } default:
   { cout << "Invalid choice" << endl;
   }
} while (ch!=4);
```

EVALUATION OF ARITHMETIC EXPRESSIONS:-

> Infix postfix, and prefix notations are three different but equal notations of writing algebraic expressions

INFIX NOTATION:-

Operator is placed in between operands

eg: A+B

~~Prefix Notation~~

PREFIX NOTATION:-

operator is placed before operands

eg: +AB

POSTFIX NOTATION:-

· Operator is placed after operands ·

eg:- AB+

Conversion of an infix expression into a postfix expression

1. Step 1:- Add ")" to the end of the infix expression
2. Step 2:- push "(" on to the stack.
3. Step 3:- Repeat until each character in the infix notation is scanned.

> If a "(" is encounted, push it on the stack.

> If an operand (whether a digit (or) a character) is encountered add it to the postfix expression.

> If a ")" is encountered then,

a. Repeatedly pop from stack and add it to the postfix expression until a "(" is encountered.

b. Discard the "(" i.e; remove the "(" from stack and do not add it to the postfix expression.

> If an operator O is encountered then

a. Repeatedly pop from stack and add each operator to the postfix expression which has the same precedence a higher precedence than O.

b. Push the operator 'O' from the stack.

Step 4:- Repeatedly pop from the stack and add it to the postfix expression until the stack is empty

Step 5:- EXIT.

The order of evaluation of a postfix expression is always from left to right.

⟹ The stack is used to hold operators rather than numbers purpose of the stack is to reverse the order of the operators in the expression

> Since no operator can be printed until both of its operands have appeared.

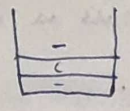
RULES:-

1. Print operands as they arrive
2. If the stack is empty (or) contains a left paran-thesis on top, push the incoming operator onto the stack.
3. If the incoming symbol is a left parenthesis, push it on to the stack.
4. If the incoming symbol is a right parenthesis, ~~push it on the stack~~ pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right

Eg:- a − (b * c − d) / e

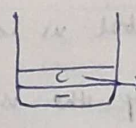
INFIX	STACK	POST FIX
a		a
−		a
(		a
b		ab
*		ab
c		abc
−		abc *

d



abc*d

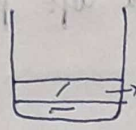
) → when it is encountered remove all operator until it reaches to '('



abc*d-

→ discarded

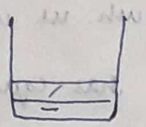
/



→ highest precedence than '-', so we can push into stack

abc*d-

e



abc*de-/

no element → remove all operator from the stack

abc*de/-

Eg :- A* B+C

Current symbol	Operator Stack	Postfix string
A		A
*	*	A
B	*	AB
+	+	AB* { + has lower precedence so * is poped
C	+	AB*C
No element	Remove all operator	AB*C+

Eg :- A+ B*C

Current symbol	Operator Stack	Postfix string
A		A
+	+	A
B	+	AB
*	+*	AB
C	+*	ABC
No element	Remove all operator	ABC*+ (because * is on the top of stack it comes out 1st).

Eg :- A* (B+C)

Current symbol	Operator Stack	Postfix string
A		A
*	*	A
(	*(	A
B	*(	AB
+	*(+	AB
C	*(+	ABC
)	*	ABC+
		ABC+*

Eg:- A-B+C

Current Symbol	Operator stack	Postfix string
A		A
-	-	A
B	-	AB
+	+	AB-
C	+	AB-C
		AB-C+

Eg:- A*B^C+D

Current symbol	Operator stack	Postfix string
A		A
*	*	A
B	*	AB
^	*^	AB
C	*^	ABC
+	+	ABC^*
D	+	ABC^*D
		ABC^*D+

Eg:- A*(B+C*D)+E

Current symbol	Operator stack	Postfix string
A		A
*	*	A
(	*(	A
B	*(	AB
+	*(+	AB
C	*(+	ABC
*	*(+*	ABC
D	*(+*	ABCD
)	*	ABCD*+
+	+	ABCD*+*
E	+	ABCD*+*E
		ABCD*+*E+

Eg:- A+B*C/(E-F)

Current symbol	Operator stack	Postfix string
A		A
+	+	A
B	+	AB
*	+*	AB
C	+*	ABC
/	+/	ABC*
(	+/(	ABC*
E	+/(	ABC*E
-	+/(-	ABC*E
F	+/(-	ABC*EF
)	+/	ABC*EF-
		ABC*EF-/+

eg:- (A/(B-C+D)) * (E-A) * C

Current symbol	Operator stack	Postfix string
(	(	
A	(	A
/	(/	A
(	(/(	A
B	(/(	AB
-	(/(-	AB
C	(/(-	ABC
+	(/(+	ABC-
D	(/(+	ABC-D
)	(/	ABC-D+
)		ABC-D+/
*	*	ABC-D+/
(	*(	ABC-D+/
E	*(	ABC-D+/E
-	*(-	ABC-D+/E
A	*(-	ABC-D+/EA
)	*	ABC-D+/EA-
*	*	ABC-D+/EA-
C	*	ABC-D+/EA-C
		ABC-D+/EA-C*

eg:- A-B+C

Current symbol	Operator stack	Postfix string
A		A
-	-	A
B	-	AB
+	+	AB-
C	+	AB-C
		AB-C+

eg:- 9-((3*4)+8)/4

Current symbol	Operator stack	Postfix string
9		9
-	-	9
(	-(	9
(	-((	9
3	-((	93
*	-((*	93
4	-((*	934
)	-(	934*
+	-(+	934*
8	-(+	934*8
)	-	934*8+
/	-/	934*8+
4	-/	934*8+4
		934*8+4/-

Evaluation of postfix expression:-

> In postfix evaluation, every character of the postfix expression is scanned from left to right.

> If the character encountered is an operand, it is pushed to the stack.

> If an operator is encountered, then the top two values are popped from the stack and operator is applied on these values. The result is then pushed onto the stack.

Eg:- $9-((3*4)+8)/4 \iff 934*8+4/-$

character scanned	stack	
9	9	
3	9,3	
4	9,3,4	 4*3=12
*	9,12	
8	9,12,8	 12+8=20
+	9,20	
4	9,20,4	 20/4=5
/	9,5	
-	4	

Eg:- 456+*

character scanned	stack
4	4
5	4,5
6	4,5,6
+	4,11
*	44

Q) 10+2-8+3

current symbol	Operator stack	Postfix string
10		10
+	+	10
2	+	10 2
-	-	102+
8	-	102+8
+	+*	102+8-
3	-+	102+83
		~~102+83-+~~ 102+8-3+

Now 102+8-3+

character scanned ~~Current symbol~~	~~Operator~~ stack	~~Postfix string~~
10	10	
2	10,2	
+	12	
8	12,8	
-	4	
3	4,3	
+	7	

Q) 628*+3-

character scanned	stack
6	6
2	6,2
8	6,2,8
*	6,16
+	22
3	22,3
-	19

Q) 231*+9-

character scanned	stack
2	2
3	2,3
1	2,3,1
*	2,3
+	5
9	5,9
-	-4

Q) 53+62/*35*+

character scanned	stack
5	5
3	5,3
+	8
6	8,6
2	8,6,2
/	8,3
*	24
3	24,3
5	24,3,5
*	24,15
+	39

Q) (6-4/2)*(4/2-1)

current symbol	operator stack	postfixing
(	(	
6	(	6
-	(-	6
4	(-	64
/	(-/	64
2	(-/	642
)		642/-
*	*	642/-
(	*(	642/-
4	*(	642/-4
/	*(/	642/-4
2	*(/	642/-42
-	*(-	642/-42/
1	*(-	642/-42/1
)	*	642/-42/1-
		642/-42/1-*

642/-42/1-*

character scanned	stack
6	6
4	6,4
2	6,4,2
/	6,2
-	4
4	4,4
2	4,4,2
/	4,2
1	4,2,1
-	4,1
*	4

PROGRAM:-

```
#include <iostream>
using namespace std;
int main()
{ int x1, x2, x3, a;
  char str[200];
  stack <int> s1;
  cout<<" enter a postfix expression "<<endl;
  cin>>str;
  for (int i=0; str[i]!='\0'; i++)
  {
    a= str[i];
    if (isdigit(a))
    {
      a = str[i]-48;
      s1.push(a);
      cout<<"push value"<<a<<endl;
    }
    else
    { x1= s1.top()
      cout<<" value on top"<<x1<<"going to pop"<<endl;
      s1.pop();
      x2=s1.top();
      cout<<" value on top"<<x2<<"going to pop"<<endl;
      s1.pop();
      switch (a)
      { case '+':
        { x3 = x1+x2;
          cout<<"push value"<<x3<<endl;
          s1.push(x3);
          break;
        }
        case '-':
        { x3 = x2-x1;
          s1.push(x3)
          cout<<"push value "<<x3<<endl;
          break;
        }
        case '*':
        { x3 = x3* x2;
          s1.push(x3);
          cout<<"push value"<< x3<<endl;
        }
        break;
        case '/':
        {
          x3= x3/x2
          s1.push(x3);
          cout<<"push value"<<x3<<endl;
          break;
        }
      }
    }
  cout<<"ans is"<< s1.top()<<endl;
}
```

ASCII [48 to 57]

QUEUE:-

> An Queue is a FIFO (First-In-First-Out) data structure in which that element that is inserted first is first one to be taken out.

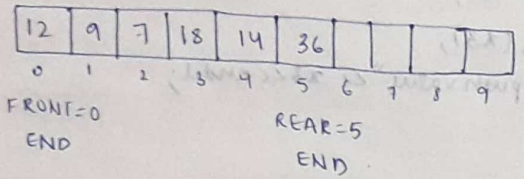
> The elements in a Queue are added at one end called the "REAR" and removed from the other end called the "FRONT".

> Queue can be implemented by using arrays or linked list.

Array Representation of Queue:-

> Queues can be easily represented using linear arrays.

> Every Queue has FRONT and REAR that point to the position from where deletion and insertion can be done.



> In Queue insertion can be done at Rear End and deletion can be done at FRONT END.

> In the above example FRONT=0, REAR=5, suppose we want to add one element, we must increment "REAR" position and then value would be stored at the position pointed by REAR.

> Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.

> Queues are used to transfer data asynchronously (data not necessarily received at same rate as sent) between two processes.

Eg: pipes, file IO, sockets.

> Queues are used as buffers on MP3 players and portable CD players ipad playlist.

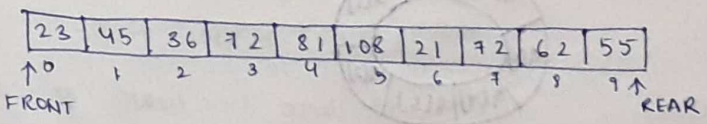
> Queues are used in playlist for jukebox to add songs to the end play from front of the list.

> Queues are used in operating system for handling interrupts.

TYPES OF QUEUS

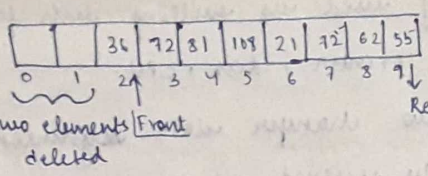
1. Circular Queue
2. Dequeue
3. Priority Queue
4. Multiple Queue

> In a linear Queue, insertions can be done only at one end called the REAR and deletions are always done from the other end called the FRONT.



FRONT=0, REAR=9.

> For suppose if you want to delete two elements from Queue.



> If you want to insert an element into the Queue, not be possible because queue is already full (Rear = Max-1)

> There is no empty space after the Rear

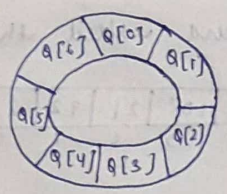
> eventhough there is a space available before the front end the "overflow" condition still exist because the condition REAR = MAX-1 (Queue full). (this is major drawback of linear queues)

> To solve the problem we have to solutions. First, shift the elements to the left so that vacant space can be occupied and utilised officially (but time consuming)

CIRCULAR QUEUES:-

Second option is circular Queue.

> In circular Queue the first index comes right after the last index.

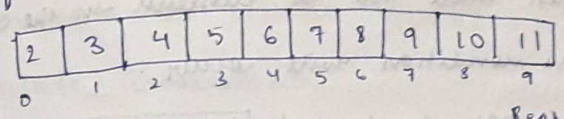


The circular Queue will be full only when FRONT=0 and REAR = MAX-1.

> Circular Queue is implemented in the same manner as a linear Queue is implemented.

> For insertion, we now have to check the following three conditions.

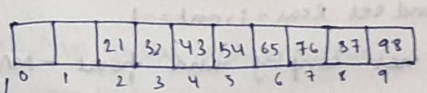
i) If Front = 0 and Rear = Max-1, then the circular Queue is full.



ii) If Rear != Max-1 then Rear will be incremented and the value will be inserted.

(increment rear so that it points to location '9' and insert the value here).

iii) If FRONT != 0 and Rear = Max-1 then it means that the Queue is not full. So, set Rear = 0 and insert the new element there



(Queue with vacant location)

Set Rear=0 and insert the value here.

Algorithm to insert an element into a circular Queue

Step 1: If Front = 0, and Rear = Max-1
write "overflow"
Goto step 4 [END OF IF]

Step 2: If Front = -1 and Rear = -1
Set front = Rear = 0
Else if Rear = MAX-1 and front != 0
Set Rear = 0
else
Set Rear = Rear + 1 [end of if]

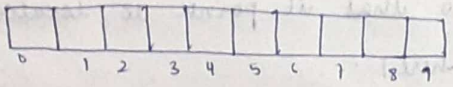
Step 3: set Queue [Rear] = val

Step 4: Exit.

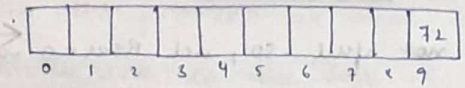
→ To delete an element, again we check for these cases

i) if front = -1 then there are no elements in the Queue so, an underflow condition will occur.

ii) if the Queue is not empty and front = Rear then after deleting the element the at the front the Queue became empty and so front and Rear are set to be.



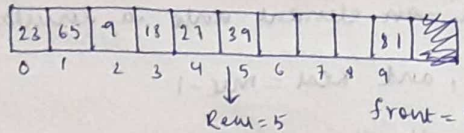
front = Rear = -1



front = Rear = 9

Delete this element and set Rear = front = -1

iii) if the Queue is not empty and front = MAX-1 then after deleting the element at the front, front is set to be 0.



Delete this element and set Front = 0.

Algorithm:

Step 1: if front = -1
write "underflow"
Go to step 4.
(end of if)

Step 2: Set value = Queue [Front]

Step 3: if front = Rear
set front = Rear = -1
else
if front = MAX-1

PROGRAM ON LINEAR QUEUE:

```
#include <iostream.h>
#include <stdlib.h>
#define max_size 100
using namespace std;
int main()
{ int item, choice, i, rear=0, front=0, exit=1;
  int queue [max_size];
  cout << "\nSimple Queue Example - Array";
  do
  {
    cout << "\n Queue main menu";
    cout << "\n 1. Insert \n 2. Remove \n 3. Display \n Others to exit:";
    cin >> choice;
    switch (choice)
  { case 1:
    {
      if (rear == max_size)
      cout << "\n Queue reached max size";
      else
      {
        cout << "\n Enter the value do be inserted:";
        cin >> item;
        cout << "\n Position: " << rear+1 << ", insert value:
             << item;
        queue [rear] = item;
        rear++;
```

```
    break;
}
 case 2:
{ if (front == rear)
  cout << "\n Queue is empty";
  else
  { cout << "\n Position: " << front << ", Remove value: "
                          << queue[front];
    front++;
  }
   break;
}
  case 3:
{
    cout << "\n Queue size: " << (rear - front);
    for (i= front; i< rear ; i++)
      cout << "\n Position: " << i << ", Value: " << queue[i];
    break;
}
  default:
{ cout << "invalid choice";
   break;
}
}
} while (exit);
}
```

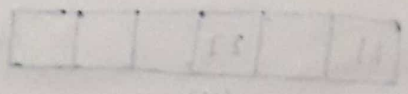
PRIORITY QUEUE :-

> An priority queue is a data structure in which each element is assigned a priority.

> The priority of the element will be used to determine the order in which the element will be processed.

There are two types of priority queues.

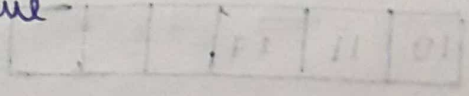
1. Ascending priority queue.



1 - - - - - - - - 10

highest lowest.

2. For descending priority queue -

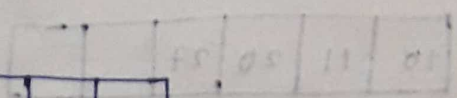


1 - - - - - - - - 10

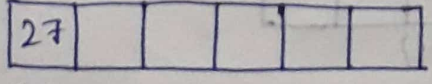
lowest highest

Eg:-

Element	Priority
27	4
11	2
10	1
20	3
15	6
12	5

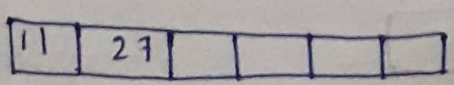


Insertion



(4)

> Insert '11' → before that check the priorities of the new element with the existed element.



(2) (4)

> Insert 10 (every insertion can be done from right to left).

↓

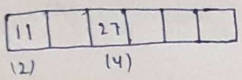
priority = 1

i. 27 → priority = 4 > 10 priority ⇒ 1

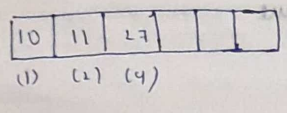
which is less than the existed element

Move 27 to next position

ii. 11 ⇒ (2) ⇒ 10 ⇒ 1

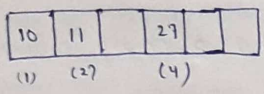


iii. 11(2) > ⇒ 10(1) → move

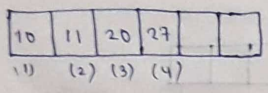


> Insert 20(3)

i. 27(4) > 20(3) ↵ move 1 position right

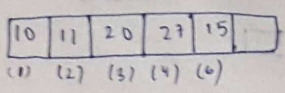


ii. 11(2) < 20(3) ← we can insert 20 without shifting



> Insert 15(6)

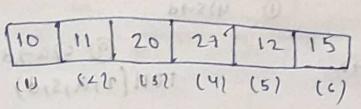
i. 27(4) ⇒ 15(6) ↑ priority is More



> Insert 12(5)

i. 15(6) ⇒ 12(5) ↵ priority less than the previous element

So move one position right



Priority Queue are widely used in operating systems to execute the highest priority process first.

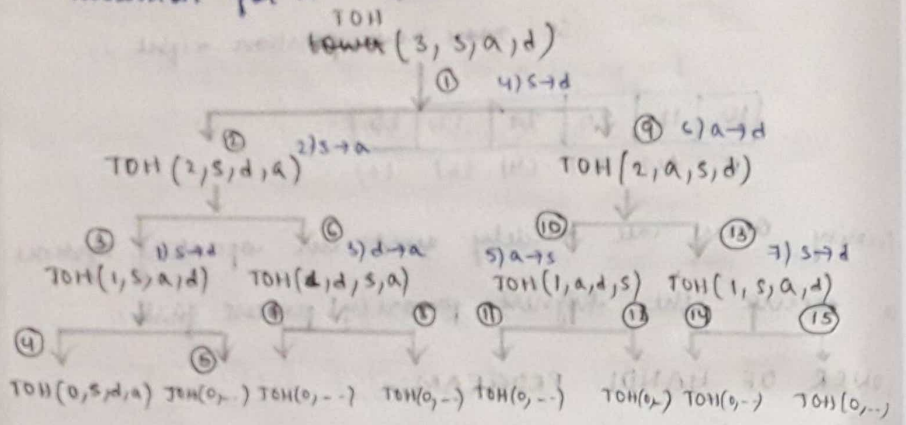
TOWER OF HANOI PROGRAM:-

```
#include<iostream.h>
using namespace std;
// tower of HANOI function implementation
void TOH(int n, char source, char Aux, char Dest)
{
  if(n==1)
  {
    cout<<"Move Disk"<<n<<"from"<<source<<"to"<<Dest<<endl;
    return;
  }
  TOH(n-1, source, Dest, Aux);
  cout<<"Move Disk "<<n<<"from"<<source<<"to"<<Dest<<endl;
  TOH(n-1, Aux, source, Dest);
}
// main program
int main()
{
  int n;
  cout<<"Enter no. of disc";
  cin>>n;
  //calling TOH
```

TOH (n, s, a, d);

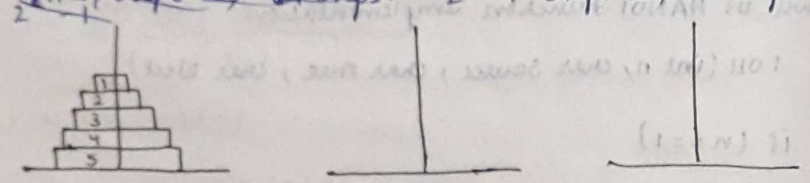
}

Recursion for n = 3 :-



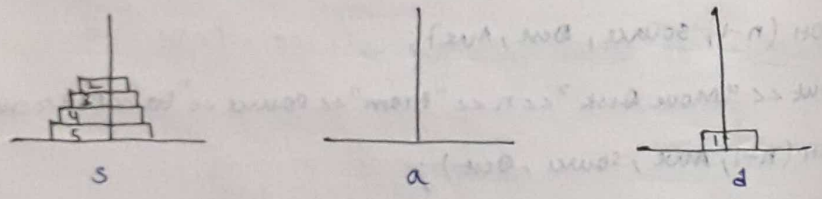
TOWER OF HANOI ALGORITHM FOR N = 5

~~In ... steps ⟹ ... steps.~~ $2^n - 1$ steps ⟹ 31 steps

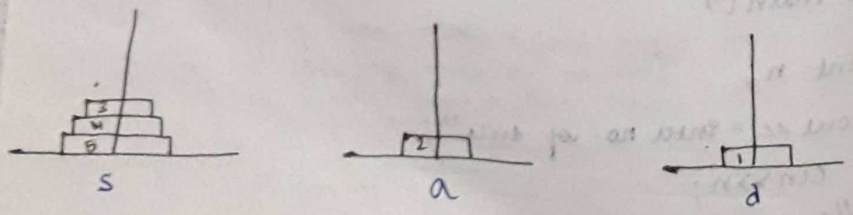


Source (s) Auxiliary (a) Destination (d)

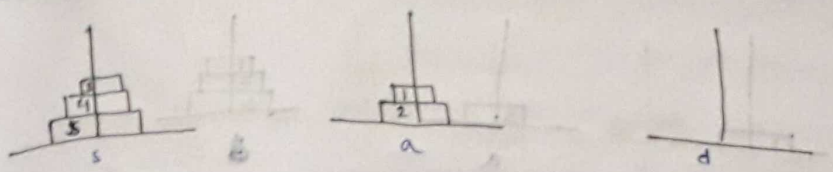
1. Move disc 1 from s to d



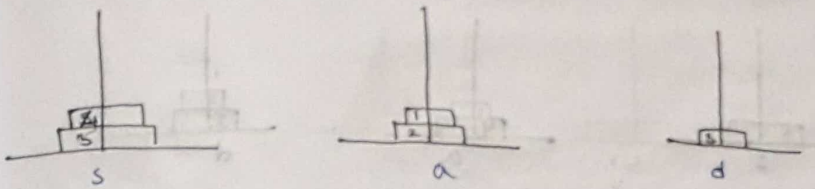
2. Move disc 2 from s to a



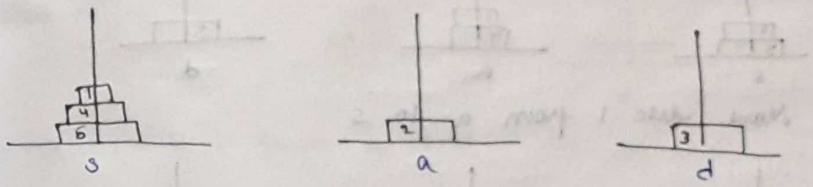
3. Move disc 1 from d to a



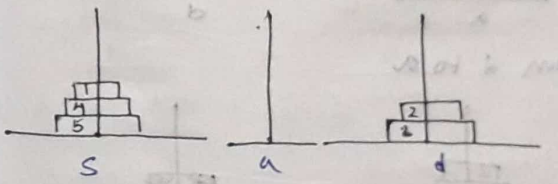
4. Move disc 3 from s to d



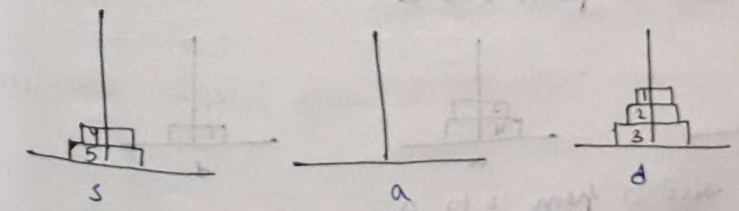
5. Move disc 1 from a to s.



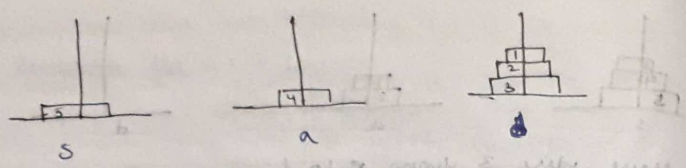
6. Move disc 2 from a to d



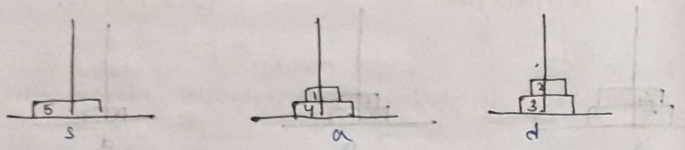
7. Move disc 1 from s to d



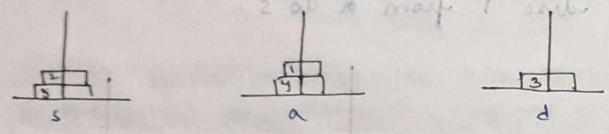
8. Move disc 4 from s to a



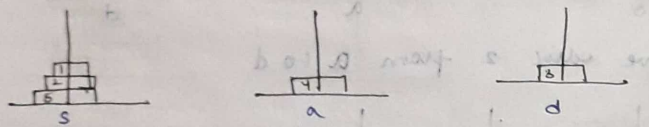
9) Move disc 1 from d to a



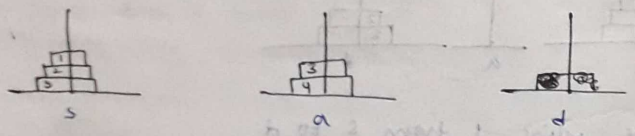
10) Move disc 2 from d to s



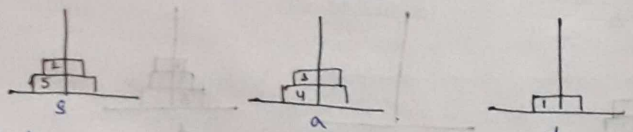
11) Move disc 1 from a to s



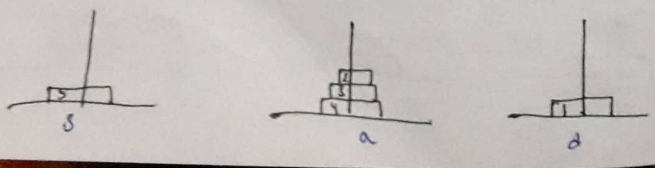
12) Move disc 3 from d to a



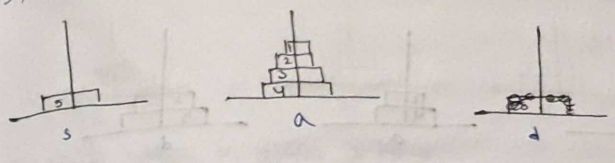
13) Move disc 1 from s to d



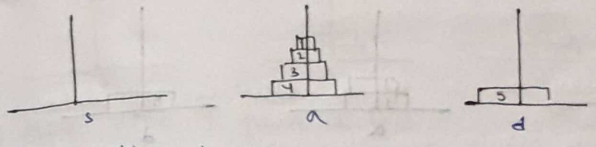
14) Move disc 2 from s to a



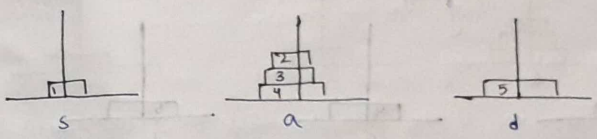
15) Move disc 1 from d to a



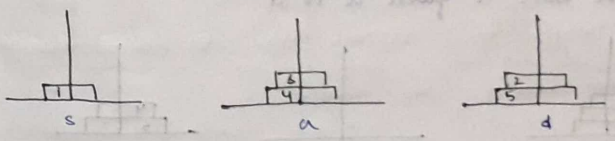
16) Move disc 5 from s to d



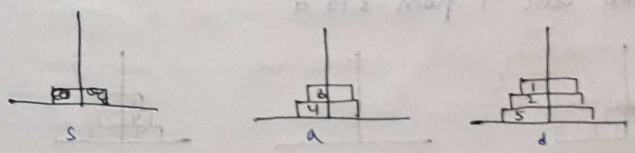
17) Move disc 1 from a to s



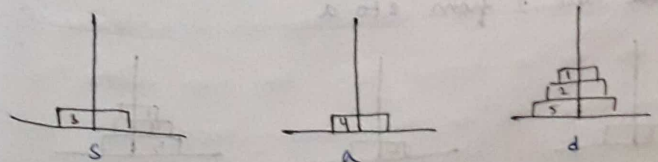
18) Move disc 2 from a to d



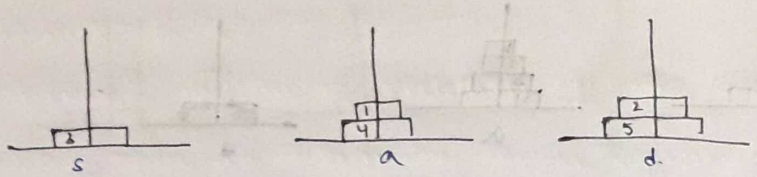
19) Move disc 1 from s to d



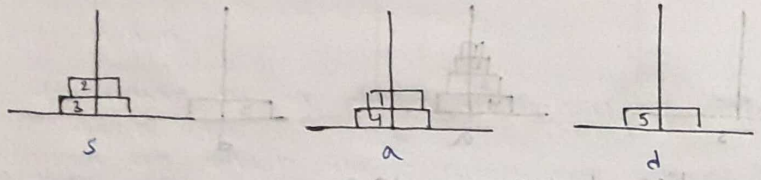
20) Move disc 3 from a to s



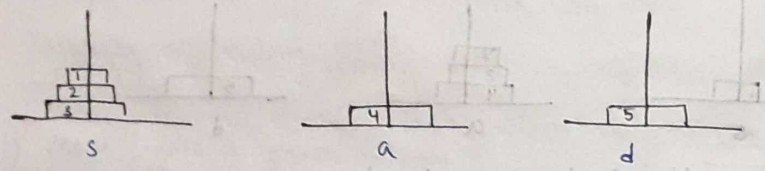
21) Move disc 1 from d to a



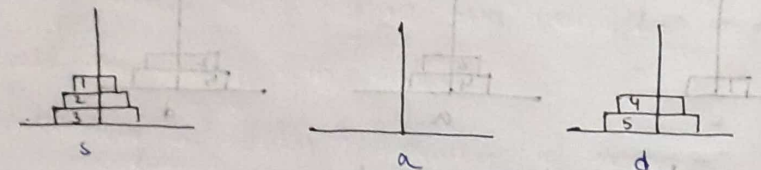
22) Move disc 2 from d to s



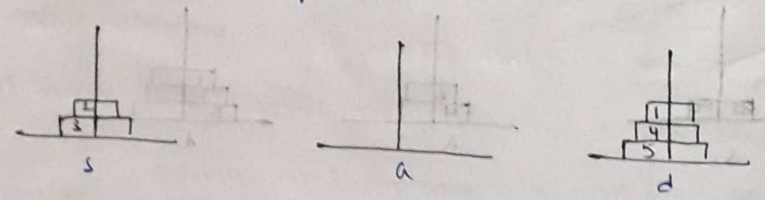
23) Move disc 1 from a to s



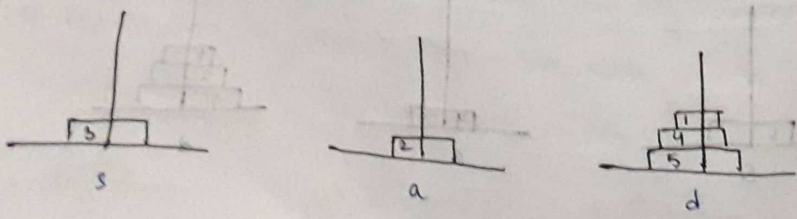
24) Move disc 4 from a to d



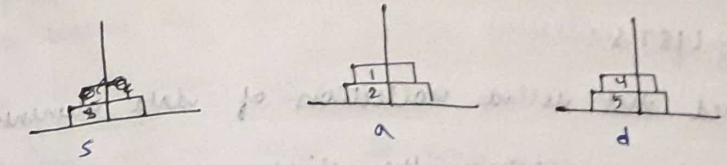
25) Move disc 1 from s to d



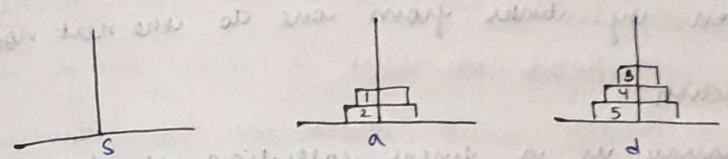
26) Move disc 2 from s to a



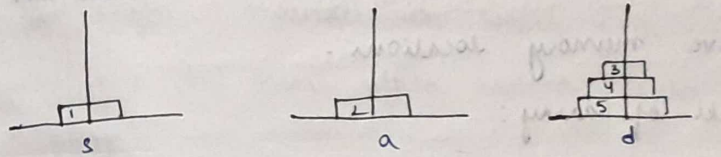
27) Move disc 1 from d to a



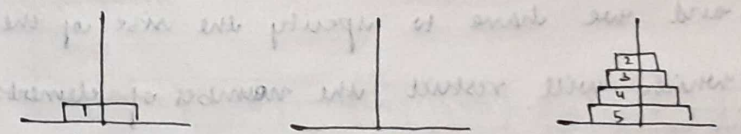
28) Move disc 3 from s to d



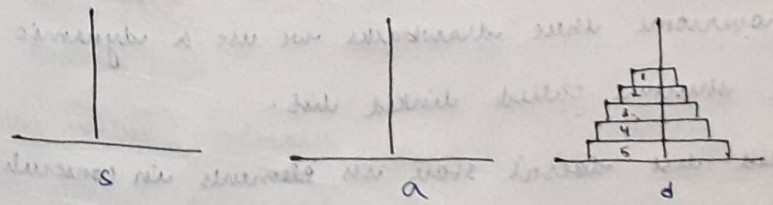
29) Move disc 1 from a to s



30) Move disc 2 from a to d



31) Move disc 1 from s to d



Source Auxillary Destination

$2^n - 1$ steps $\Rightarrow$ 31 steps //.

UNIT-3

LINKED LISTS:

> An linked list is a collection of data elements called nodes in which the linear representation is given by links from one to the next node.

Introduction:

An array is a linear collection of data elements in which the elements are stored in consecutive memory locations.

Drawbacks of array:

1> Array is a static data structure which defines arrays, and we have to specify the size of the array, which will restrict the number of elements that the array can store.

2> Memory is wasted (∴ Wastage of memory)

> To overcome these drawbacks we use a dynamic data structure called linked list.

> Linked list doesn't store its elements in consecutive memory location and the user can add number of elements to it.

> In array, we can access the elements randomly, where as in a linked list. we can access the elements only in a sequential order.

Definition:

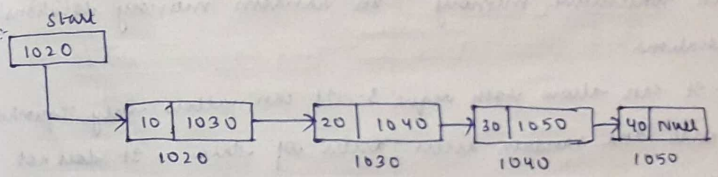
> An linked list is a linear collection of data elements. where data elements are called nodes.

> Linked list is used to implement the data structures such as stacks, Queues and graphs.

> An linked list is used to implement the data structures such as stacks, Queues and graphs.

> An linked list can be perceived as a train (or) a sequence under in which each node contains one (or) more data and a pointer to the next node.

Eg:



Every node contains two parts

1. Data part
2. Address part.

> NULL value indicates the end of the list

> In linked every node contains a pointer to another node which is of the same type so it is called self refferential data type.

> Linked list contains a pointer variable START that stores the address of the first node in the list.

> We can transverse the entire using START which contains the address of the first node, next part of

the first node in the contains the address of succeeding node.

> if START = NULL then the linked list is empty

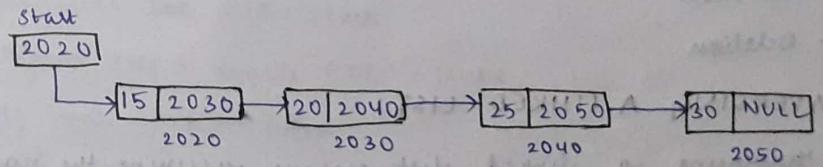
Syntax:
```
struct node
{
    int data;
    struct node *next;
};
```

LINKED LIST	ARRAYS
1. It is a linear collection of data elements	1. It is also a linear collection of data elements.
2. It stores the data elements in consecutive memory locations	2. It stores the data elements in random memory locations
3. It can allow both sequential and random access of data elements	3. It can allow only sequential access of data. It does not allow random access
4. It is a static data structure (i.e; initially we fix the size of the array)	4. It is a dynamic data structure during runtime it can shrink the size of the linked list.

ALGORITHM TO PRINT THE NUMBER OF NODES IN A LINKED LIST:

1. Initialize set count = 0
2. Initialize set PTR = START
3. Repeat steps 4 & 5 while PTR != NULL
4. set count = count + 1;
5. set PTR = PTR → NEXT
   (End of loop)
6. Print count value.
7. Exit.

Eg:



Initially count = 0 / PTR = START = 2020

while (PTR != NULL) (2020 != NULL)

C = C+1 = 0+1 = 1

PTR = PTR → NEXT = 2030 PTR → NEXT = NULL

while (2030 != NULL) (NULL != NULL)

C = 1+1 = 2

PTR = PTR → NEXT = 2040

(2040 != NULL)

C = 2 + 1 = 3

(2050 != NULL)

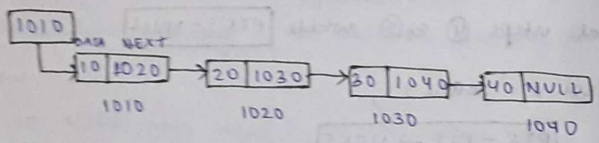
C = 3 + 1 = 4.

SINGLE LINKED LIST:

> A single linked list is the simplest type of linked list in which every node contains an data and a pointer to next node of the same data type.

> It allows traversal of data only in one way

Eg:-



Operations on single linked list

1. Traversing
2. Searching
3. Insertion
4. Deletion.

TRAVERSING A LINKED LIST:

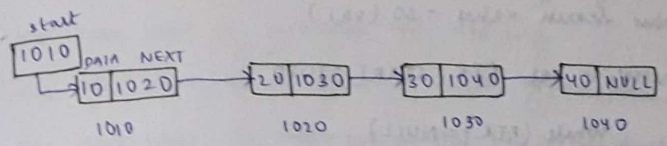
Traversing a linked list means accessing the nodes of the list inorder to perform some processing on them. A visit to every node of a data structure is called a traversal.

ALGORITHM FOR TRAVERSING A LINKED LIST

1. Initialise set PTR = START
2. Repeat steps ③ & ④ while PTR != NULL
3. Apply process to PTR → DATA
4. Set PTR = PTR → NEXT

   End of loop
5. Exit.

Eg:-



set PTR = START = 1010

(while PTR != NULL) (1010 != NULL)

{ process PTR → DATA = 10
  PTR = PTR → NEXT = 1020 }

SEARCHING FOR A VALUE IN LINKED LIST

> Searching a linked list means to find a particular element in the linked list.

> Searching means finding whether a given value is present in the data part of the node or not.

> If it is present, the algorithm returns the address of the node that contains the value.

ALGORITHM:

1. Initialize set PTR = start
2. Repeat step 3 while PTR != NULL
3. If val = PTR → DATA

   Set POS = PTR

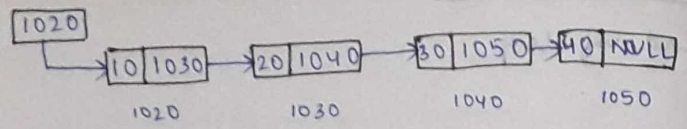
   Go to step 5.

   else

   set PTR = PTR → NEXT

   (end of if)
4. Set POS = NULL
5. Exit.

Eg:



Now search value = 30 (val)

Initialise set PTR = START = 1020

while (PTR != NULL)

(1020 != NULL)

if (val = PTR ⇒ DATA)

(30 = 10) x

else

Set PTR = PTR ⟶ NEXT = 1030

while (PTR != NULL) (1030 != NULL)

if (val = PTR → DATA)

30 = 20 x

else

Set PTR = PTR → NEXT = 1040

while (PTR != NULL) (1040 != NULL)

if (val = PTR → DATA)

30 = 30

set par = PTR

par = 1040

Goto step ⑤ exit

element found in the list.

INSERTING A NEW NODE IN A LINKED LIST:

Here we consider four cases

1. The new node is inserted at the begining
2. The new node is inserted at the end
3. The new node is inserted after a given node
4. The new node is inserted before a given node.

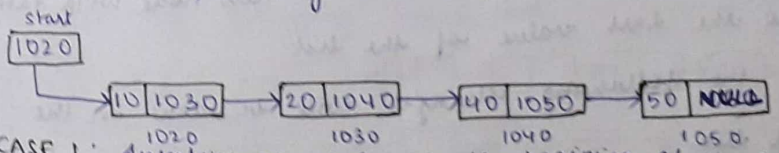
Inserting a new node at the begining of a linked list:

> Before discussing insertion operations in all these four cases, let us first discuss an important term called overflow.

> Overflow is a condition where it occurs when AVAIL = NULL (or) no free memory cell is present in the system.

> When this condition occurs, the program must give an appropriate message

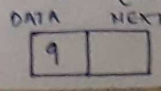
Consider the following linked list:



CASE 1: Inserting new node at the begining of the list.

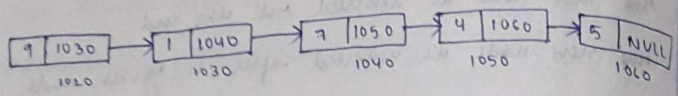
Now suppose we want to add a new node until a data and add it at the first node of the list. Then the following changes will be done in the linked list.

1) Allocate memory for the new node and initialise its DATA part to 9. (DATA = 9)

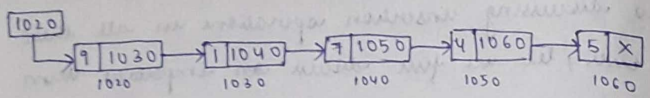


(New node)

2) Add the new node as the first node of the list by making the NEXT part of the new node contains the address of START.



3) Now make START to point to the first node of the list.

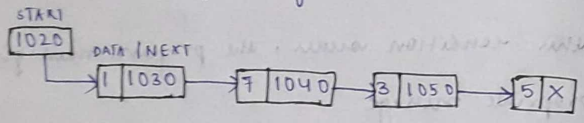


CASES:

CASE 2:

(2) (Inserting a node at the end of LL) :-

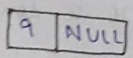
Consider the following linked list.



Suppose now we need to add a new node with data '9' as the last value of the list.

Then the following changes will be done in the linked list.

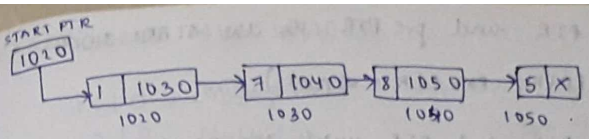
1) Allocate memory for new node and initialize its DATA part to be 9 and next part NULL.



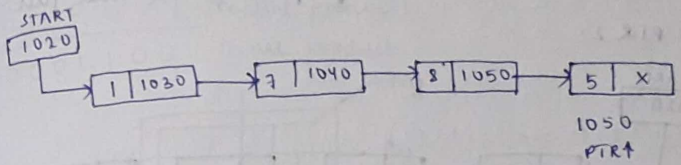
new node.

2) Make a PTR (pointer variable) which points to start

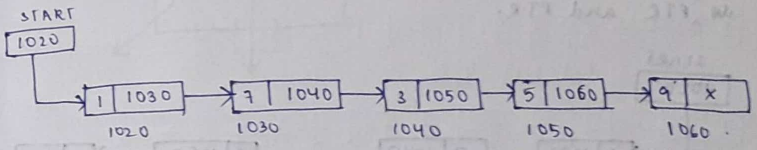
PTR = START



3) Move PTR so that it points to the last node of the list



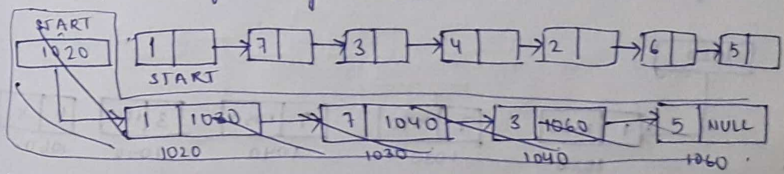
4) Add the new node after the node pointed by PTR. This is done by storing the address of the new node in the PTR → NEXT



CASE 3:-

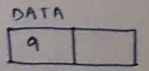
Inserting a Node after given node in a linked list.

Consider the following linked list



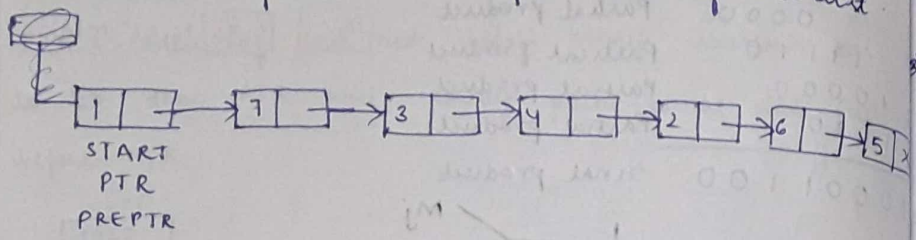
Suppose we want to add a new node with 9 after the node containing data 3.

1) Allocate memory for the new node and initialise its DATA part to 9

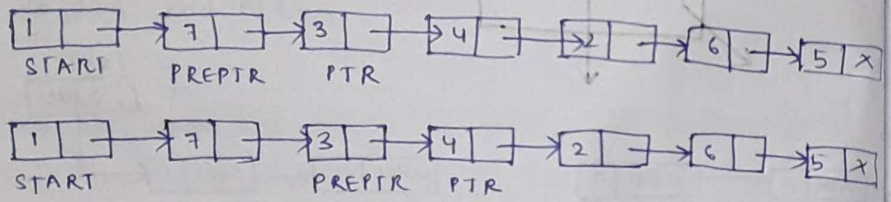


(new node)

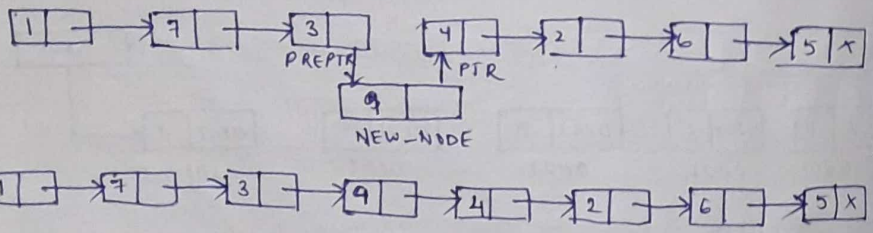
2. Take two pointer variables PTR and PREPTR and initialise them with START so that START, PTR and PREPTR point to the first node of the list.



3. Move PTR and PREPTR until the DATA part of PREPTR = value of the node after which insertion has to be done. PREPTR will always point to the node just before PTR.

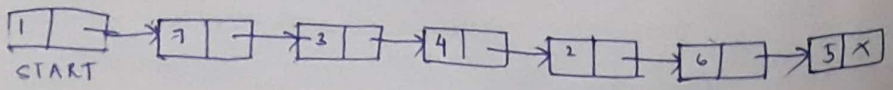


4) Add the new node in between the nodes pointed by PREPTR and PTR.

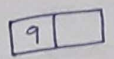


CASE 4:

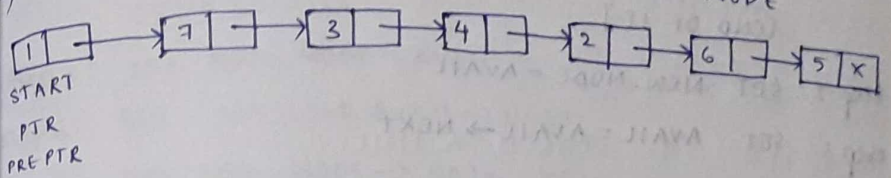
1) Inserting a node before a given node in a linked list.



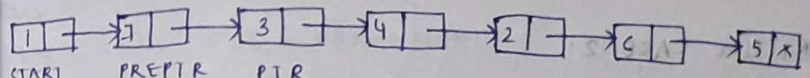
2) Allocate memory for the new node and initialise its data part to 9.



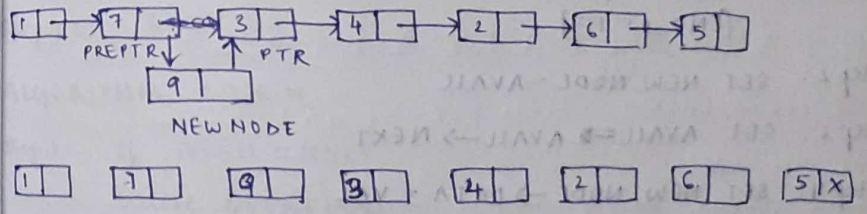
3) Initialise PREPTR and PTR to the START NODE



4) Move PTR and PREPTR until the DATA PART of PTR = value of the node before insertion has to be done. PREPTR will always point to the node just before PTR.



5) Insert the new node in between the nodes pointed by PREPTR and PTR



ALGORITHM CASE 1:

Step 1: [INITIALIZE] SET PTR = START

Step 2: Repeat ste

ALGORITHM CASE 1:

Step 1: If AVAIL = NULL
write OVERFLOW
Go to step 7
[END OF IF]

Step 2: SET NEW_NODE = AVAIL

Step 3: SET AVAIL = AVAIL → NEXT

Step 4: SET NEW_NODE → DATA = VAL

Step 5: SET NEW_NODE → NEXT = START

Step 6: SET START = NEW_NODE

Step 7: EXIT.

ALGORITHM CASE 2:

Step 1: If AVAIL = NULL
write Overflow
Go to step 10
[END OF IF]

Step 2: SET NEW_NODE = AVAIL

Step 3: SET AVAIL = AVAIL → NEXT

Step 4: SET NEW_NODE → DATA = VAL

Step 5: SET NEW_NODE → NEXT = NULL

Step 6: SET PTR = START

Step 7: Repeat step 8 until PTR → NEXT != NULL

Step 8: SET PTR = PTR → NEXT
[END OF LOOP]

Step 9: SET PTR → NEXT = NEW_NODE

Step 10: EXIT.

ALGORITHM CASE 3:

Step 1: IF AVAIL = NULL
~~Go to step~~
write OVERFLOW
Go to step 12
[END OF IF]

Step 2: SET NEW_NODE = AVAIL

Step 3: SET AVAIL = AVAIL → NEXT

Step 4: SET NEW_NODE → DATA = VAL

Step 5: SET ~~NEW_NODE → NEXT =~~ PTR = START

Step 6: SET PREPTR = PTR

Step 7: Repeat steps 8 and 9 while PREPTR → DATA != NUM

Step 8: SET PREPTR = PTR

Step 9: SET PTR = PTR → NEXT
[END OF LOOP]

Step 10: SET PREPTR → NEXT = NEW_NODE

Step 11: SET NEW_NODE → NEXT = PTR.

Step 12: EXIT.

ALGORITHM CASE 4:

Step 1: If AVAIL = NULL
write OVERFLOW
Go to step
[END OF IF]

Step 2: SET NEW_NODE = AVAIL

Step 3: SET AVAIL = AVAIL → NEXT

Step 4: SET NEW_NODE → DATA = VAL

Step 5: SET PTR = START

Step 6: SET PREPTR = PTR

Step 7: Repeat steps 8 and 9 while PTR → DATA != NUM

Step 8: SET PREPTR = PTR

Step 9: SET PTR = PTR -> NEXT

[END OF LOOP]

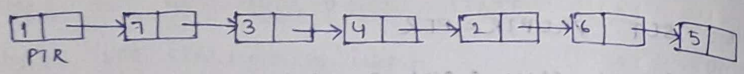
Step 10: PREPTR -> NEXT = NEW_NODE

Step 11: SET NEW_NODE -> NEXT = PTR

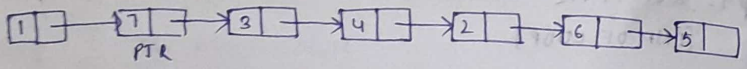
Step 12: EXIT.

ALGORITHM TO SEARCH A LINKED LIST:

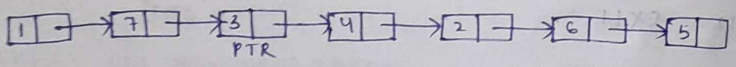
We search VAL = 4, initialize PTR = START



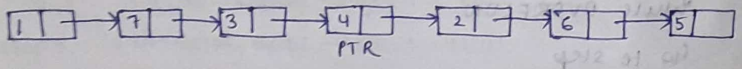
Here PTR -> DATA = 1, since PTR -> DATA != 4, we move to the next node.



Here PTR -> DATA = 7, since PTR -> DATA != 4, we move to the next node.



Here PTR -> DATA = 3, since PTR -> DATA != 4, we move to the next node.



Here PTR -> DATA = 4, since PTR -> DATA = 4, POS = PTR, POS now stores the address of the node that contains VAL

Step 1: [INITIALIZE] SET PTR = START

Step 2: Repeat step 3 while PTR != NULL

Step 3: If VAL = PTR -> DATA

SET POS = PTR

Go to step 5

ELSE

SET PTR = PTR -> NEXT

[END OF IF]

[END OF LOOP]

Step 4: SET POS = NULL

Step 5: EXIT.

DELETING A NODE FROM LINKED LIST

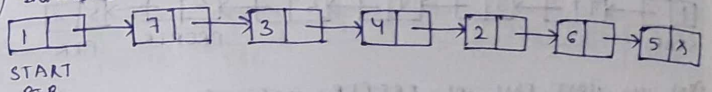
Case 1: First node is deleted

Case 2: Last node is deleted

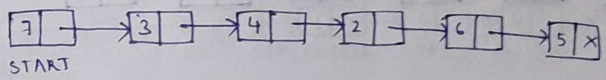
Case 3: The node after a given node is deleted.

CASE 1: Deleting the first node from the linked list.

1) Initialize PTR = START.



2) Make START to point to the next node in sequence



Step 1: If START = NULL

Write underflow

Go to step 5

[end of IF]

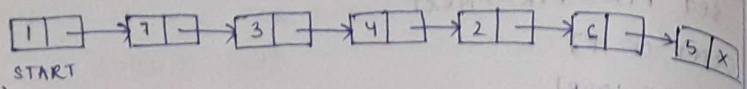
Step 2: SET PTR = START

Step 3: SET START = START -> NEXT

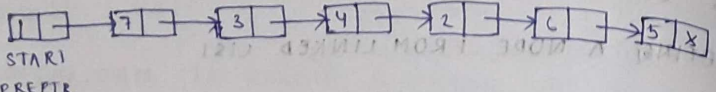
Step 4: FREE PTR

Step 5: EXIT.

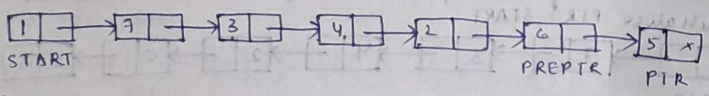
CASE 2: Deleting last node from a linked list.



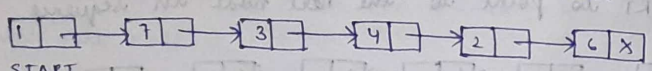
1) Take pointer variables PTR and PREPTR which initially point to START.



2) Move PTR and PREPTR such that NEXT part of PTR = NULL. PREPTR always points to the node just before the node pointed by PTR.

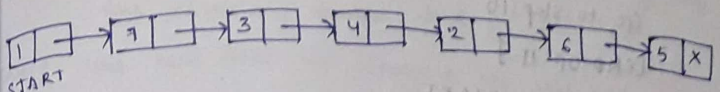


3) Set the NEXT part of PREPTR node to NULL

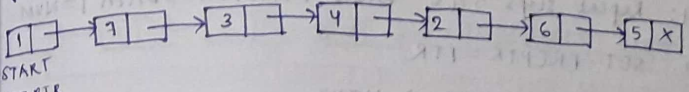


Step 1: If START = NULL
Write UNDERFLOW
Go to step 8.
[END OF IF]
Step 2: SET PTR = START
Step 3: SET PREPTR. Repeat steps 4 and 5 until while PTR -> NEXT != NULL
Step 4: PREPTR = PTR
Step 5: PTR = PTR -> NEXT.
[END OF LOOP]
Step 6: SET PREPTR -> NEXT = NULL
Step 7: FREE PTR
Step 8: EXIT

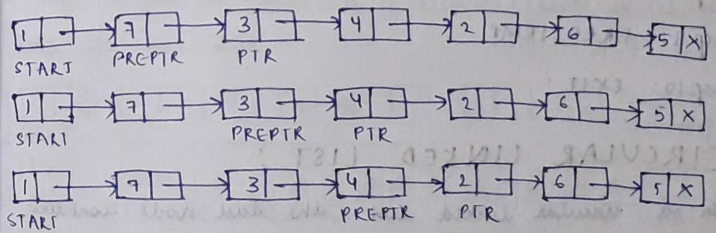
CASE 3:
DELETING THE NODE AFTER A GIVEN NODE IN A LINKED LIST



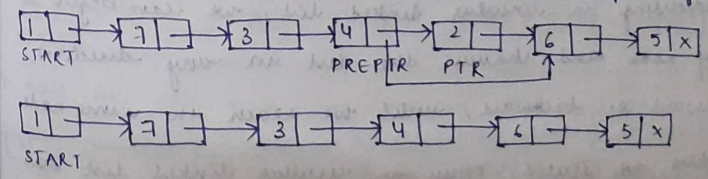
1. Take pointer variables PTR and PREPTR which initially point to START.



2. Move PTR and PREPTR such that PREPTR points to the node containing VAL and PTR points to the succeeding node



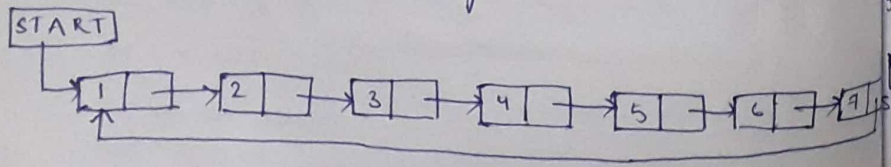
3. Set the NEXT part of PREPTR to the next part of PTR



Step 1: If START = NULL
Write UNDERFLOW
Go to step 10
[END OF IF]
Step 2: SET PTR = START
Step 3: SET PREPTR = PTR
Step 4: Repeat steps 5 & 6 while PREPTR → DATA != NULL
Step 5: SET PREPTR = PTR
Step 6: SET PTR = PTR → NEXT.
[END OF LOOP]
Step 7: SET TEMP = PTR
Step 8: SET PREPTR → NEXT = PTR → NEXT
Step 9: FREE TEMP
Step 10: EXIT

CIRCULAR LINKED LIST :

In a circular linked list, the last node contains a pointer to the first node of the list. While traversing a circular linked list, we can begin at any node and traverse the list in any direction, forward or backward, until we reach the same node where we started. Thus, a circular linked list has no beginning and no ending.

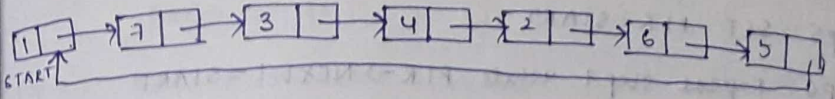


INSERTING A NEW NODE IN A CIRCULAR LINKED LIST

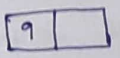
Case 1: The new node is inserted at the beginning of the circular linked list

Case 2: The new node is inserted at the end of the circular linked list.

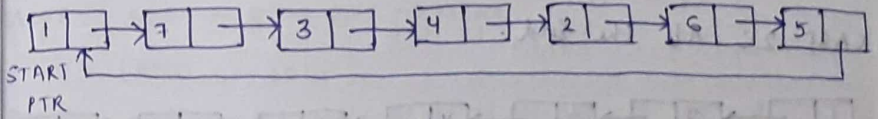
CASE 1: At the Begining



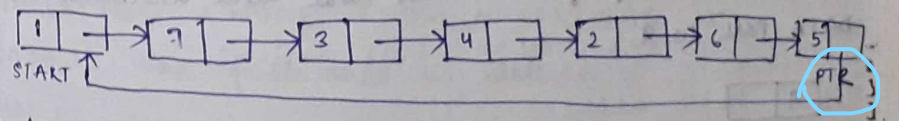
1. Allocate memory for the new node and initialise its data part to 9.



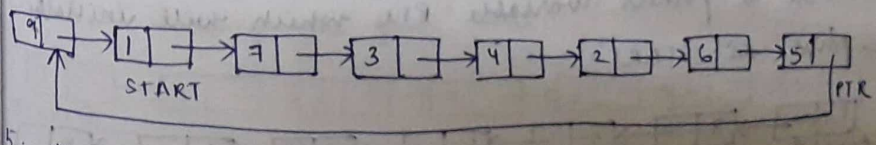
2. Make a pointer variable PTR that points to the start node of the list.



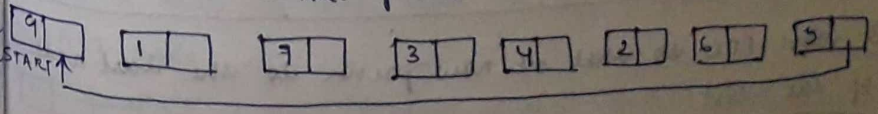
3. Move PTR so that it now points to the last node of the list



4. Add the new node in between PTR and START

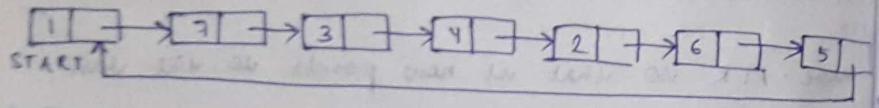


5. Make ~~start~~ START point to the newnode.

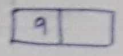


Step 1: If AVAIL = NULL
write OVERFLOW
Go to step 11
[END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat step 7 while PTR -> NEXT != START
Step 7: PTR = PTR -> NEXT
[END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = START
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET START = NEW_NODE
Step 11: EXIT.

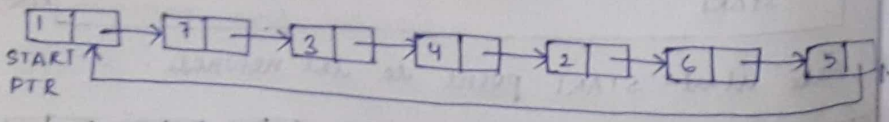
CASE 2: At the end.



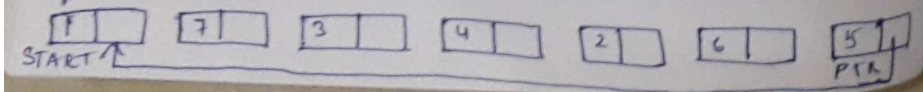
1. Allocate memory for the new node and initialise its DATA part to 9:



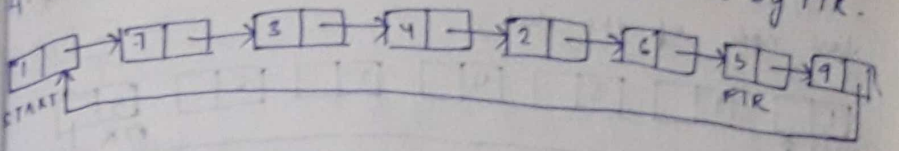
2. Make a pointer variable PTR which will initially point to START.



3. Move PTR so that it now points to the last node of the list



4. Add the new node after the node pointed by PTR.

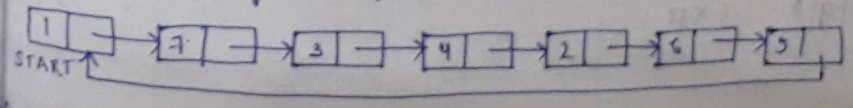


Step 1: If AVAIL = NULL
Write OverFlow
Go to step 10
[END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET ~~PTR = START~~ NEW_NODE -> NEXT = START
Step 6: SET PTR = START
Step 7: Repeat step 8 while PTR -> NEXT != START
Step 8: SET PTR = PTR -> NEXT
[END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: EXIT.

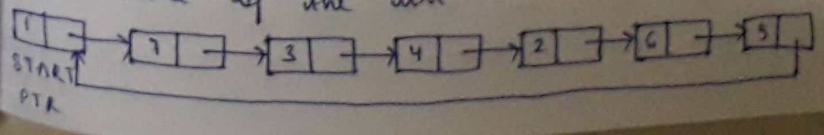
DELETING A NODE FROM A CIRCULAR LIST:

CASE 1: The first node is deleted.
CASE 2: The last node is deleted.

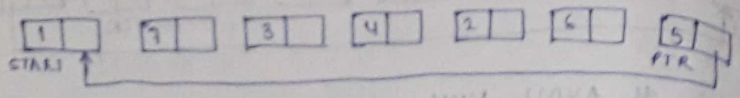
CASE 1: Deleting the first node.



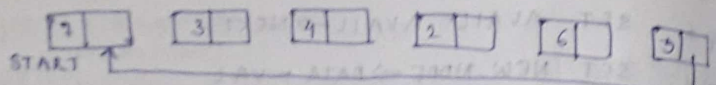
1. Make a variable PTR and make it point to the START node of the list.



2. Move PTR further so that it now points to the last node of the list

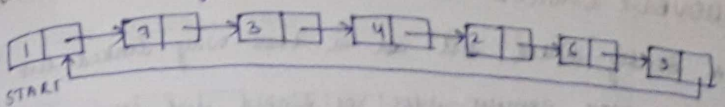


The NEXT part of PTR is made to point to the second node of the list and the memory of the first node is freed. The second node becomes the first node of the list.

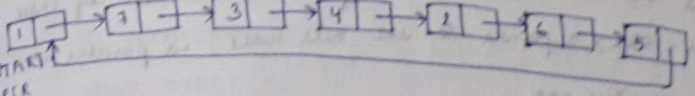


Step 1: If START = NULL
Write UNDERFLOW
Go to Step 8.
[END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> NEXT != START
Step 4: SET PTR = PTR -> NEXT
[END OF LOOP]
Step 5: SET PTR -> NEXT = START -> NEXT
Step 6: FREE START
Step 7: SET START = PTR -> NEXT
Step 8: EXIT.

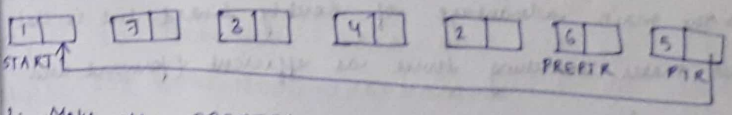
CASE 2: Deleting the last node.



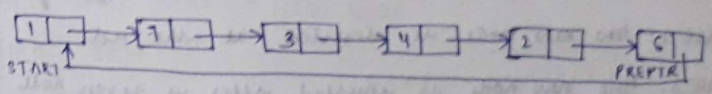
1. Make two pointers PREPTR and PTR which will initially point to START.



2. Move PTR so that it points to the last node of the list. PREPTR will always point to the node preceding PTR.



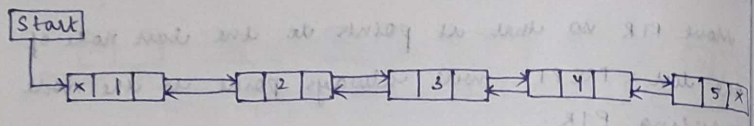
3. Make the PREPTR's next part store START node's address and free the space allocated for PTR. Now PREPTR is the last node of the list.



Step 1: If START = NULL
Write UNDERFLOW
Go to Step 8.
[END OF IF]
Step 2: SET PTR = START
Step 3: Repeat steps 4 & 5 while PTR -> NEXT != START
Step 4: PREPTR = PTR
Step 5: PTR = PTR -> NEXT
[END OF LOOP]
Step 6: SET PREPTR -> NEXT = START
Step 7: FREE PTR
Step 8: EXIT.

DOUBLY LINKED LIST:

A doubly linked list or a two way linked list is a more complex type of linked list which contain a pointer to the next as well as the previous node in the sequence. Therefore it consists of three parts - data, a pointer to the next node, a pointer to the previous node.



> The main advantage of doubly linked list is that it makes searching twice as efficient (forward and backward).

INSERTING A NODE IN A DOUBLY LINKED LIST:

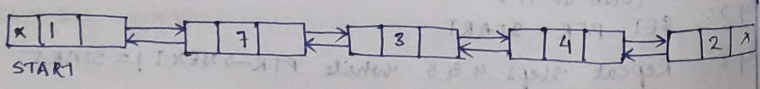
CASE 1: The new node is inserted at the begining

CASE 2: The new node is inserted at the end

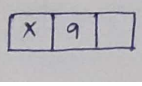
CASE 3: The new node is inserted after a given node

CASE 4: The new node is inserted before a given node.

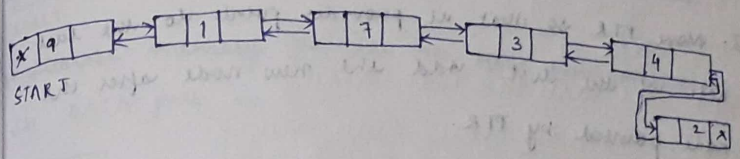
CASE 1: Inserting at the begining.



1. Allocate memory for the new node and initialise its DATA part to 9 and PREV field to NULL.



2. Add the new node before the START node. Now the new node becomes the first node of the list.



Step 1: If AVAIL = NULL
write OVERFLOW
{Go to step 9}
[END OF IF]

Step 2: SET NEW_NODE = AVAIL

Step 3: SET AVAIL = AVAIL -> NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET NEW-NODE -> PREV = NULL

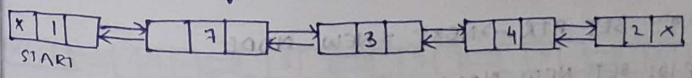
Step 6: SET NEW_NODE -> NEXT = START

Step 7: SET START -> PREV = NEW_NODE

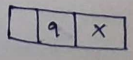
Step 8: SET START = NEW-NODE

Step 9: EXIT.

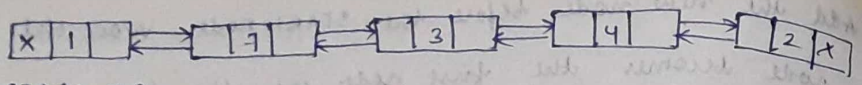
CASE 2: Inserting at the end.



1. Allocate memory for the new node and initialise its DATA part to 9 and its NEXT field to NULL

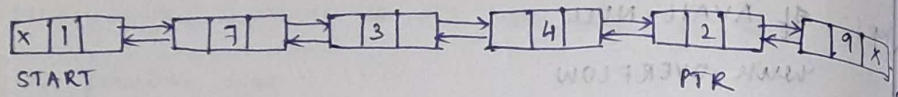


2. Make a pointer variable PTR and make it to point to the first node of the list.



START, PTR

3. Move PTR so that it ~~provides~~ points to the last node of the list. Add the new node after the node pointed by PTR.



START PTR

Step 1: If AVAIL = NULL
write OVERFLOW
Go to step 11
[END OF IF]

Step 2: SET NEW_NODE = AVAIL

Step 3: SET AVAIL = AVAIL -> NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET NEW_NODE -> NEXT = NULL

Step 6: SET PTR = START

Step 7: Repeat Step 8 while PTR -> NEXT != NULL

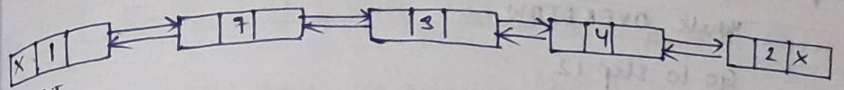
Step 8: SET PTR = PTR -> NEXT
[END OF LOOP]

Step 9: SET PTR -> NEXT = NEW_NODE

Step 10: SET NEW_NODE -> PREV = PTR.

Step 11: EXIT.

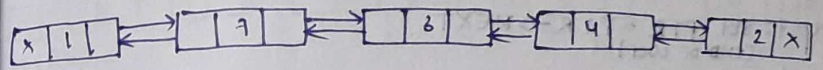
CASE 3: Inserting after a given node.



START

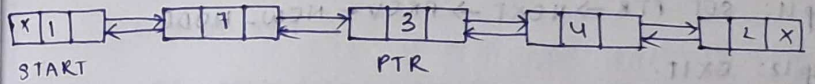
1. Allocate memory for the new node and initialize its DATA part to 9.

2. Take a pointer variable PTR and make it point to the first node of the list.



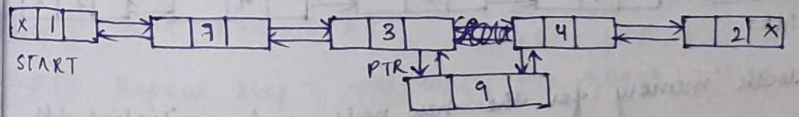
START, PTR.

3. Move PTR further until the data part of PTR = value after which the node has to be inserted.

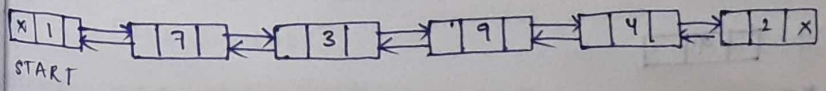


START PTR

4. Insert the new node between the PTR and the node succeeding it.



START



START

Step 1: If AVAIL = NULL
write OVERFLOW
Go to step 12
[END OF IF]

Step 2: SET NEW_NODE = AVAIL

Step 3: SET AVAIL = AVAIL -> NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET PTR = START

Step 6: Repeat step 7 while PTR -> DATA != NUM

Step 7: SET PTR = PTR -> NEXT
[END OF LOOP]

Step 8: SET NEW_NODE -> NEXT = PTR -> NEXT

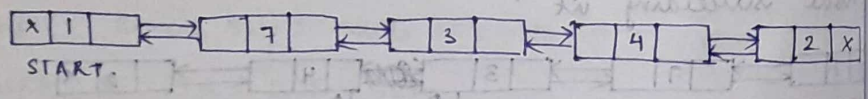
Step 9: SET NEW_NODE -> PREV = PTR

Step 10: SET PTR -> NEXT = NEW_NODE

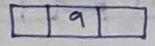
Step 11: SET PTR -> NEXT -> PREV = NEW_NODE

Step 12: EXIT.

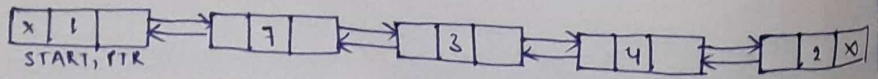
CACE 4: Inserting before a given node:



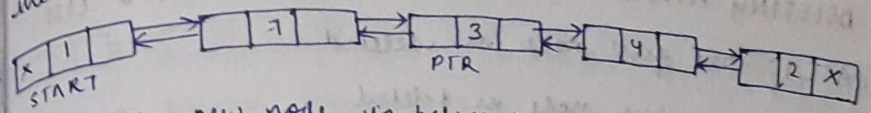
1. Allocate memory for the new node and initialise its datapart to 9.



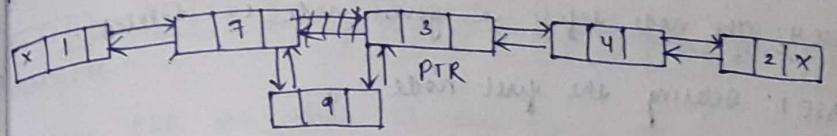
2. Make a pointer variable PTR and make it point to the first node of the list.

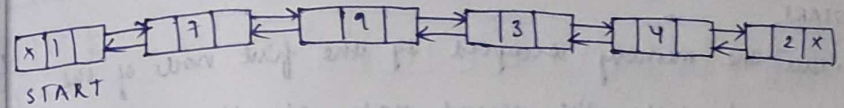


3. Move PTR further so that it now points to the node whose data is equal to the value before which the node has to be inserted.



4. Add the new node in between the node pointed by PTR and the node preceding it.





Step 1: If AVAIL = NULL
write OVERFLOW
Go to step 12
[END OF IF]

Step 2: SET NEW_NODE = AVAIL

Step 3: SET AVAIL = AVAIL -> NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET PTR = START

Step 6: Repeat step 7 ~~until~~ while PTR -> DATA != NUM

Step 7: PTR = PTR -> NEXT
[END OF LOOP]

Step 8: SET NEW_NODE -> NEXT = PTR

Step 9: SET NEW_NODE -> PREV = PTR -> PREV

Step 10: SET PTR -> PREV = NEW_NODE

Step 11: SET PTR -> PREV -> NEXT = NEW_NODE

Step 12: EXIT.

DELETING ANDDE FROM A DOUBLY LINKED LIST

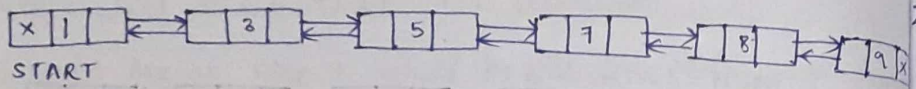
CASE 1: The first node is deleted

CASE 2: The last node is deleted

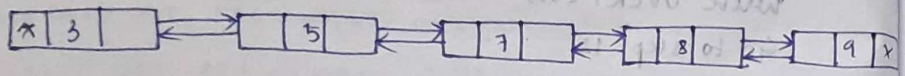
CASE 3: The node after a given node is deleted

CASE 4: The node before a given node is deleted.

CASE 1: Deleting the first node.



1. Free the memory occupied by the first node of the list and make the second node of the list as the START node.



Step 1: If START = NULL
write Underflow
Go to step 6
[END OF IF]

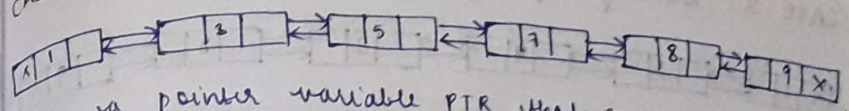
Step 2: SET PTR = START

Step 3: SET START = START → NEXT

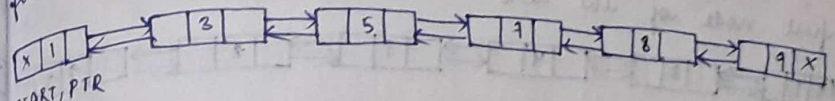
Step 4: FREE PTR

Step 5: EXIT

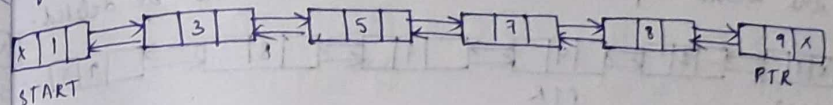
CASE 2: Deleting the last node.



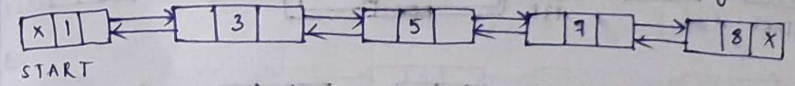
1. Make a pointer variable PTR that points to the first node of the list.



2. Move PTR so that it now points to the last node of the list.



3. Free the space occupied by the node pointed by PTR and store NULL in NEXT field of its preceding node



Step 1: If START = NULL
write UNDERFLOW
Go to step 7
[END OF IF]

Step 2: SET PTR = START

Step 3: Repeat step 4 until PTR → NEXT != NULL

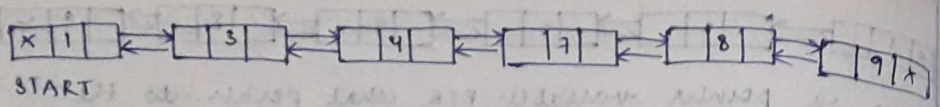
Step 4: SET PTR = PTR → NEXT
[END OF IF]

Step 5: SET PTR → PREV → NEXT = NULL

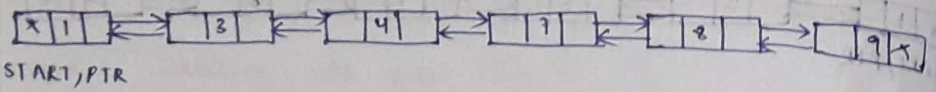
Step 6: FREE PTR

Step 7: EXIT.

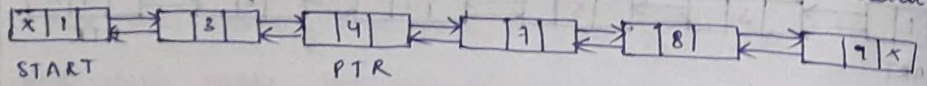
CASE 3: Deleting after a given node.



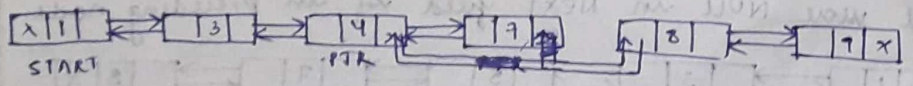
1. Make a pointer variable PTR and make it point to the first node of the list



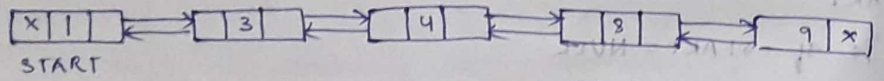
2. Move PTR further so that its data part is equal to the value after which the node has to be deleted.



3. Deleting the node succeeding PTR



4.



Step 1: If START = NULL
Write UNDERFLOW
Go to step 9
[END OF IF]

Step 2: SET PTR = START

Step 3: Repeat Step 3 while PTR → DATA ! = NUM

Step 4: SET PTR = PTR → NEXT

Step 5: SET TEMP = PTR → NEXT

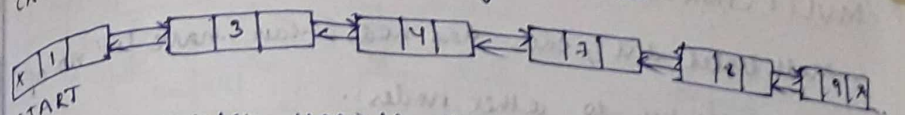
Step 6: SET PTR → NEXT = TEMP → NEXT

Step 7: SET TEMP → NEXT → PREV = PTR

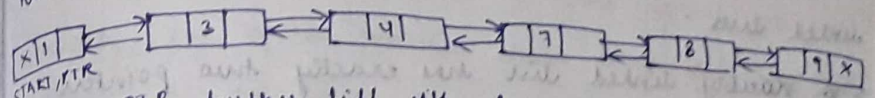
Step 8: FREE TEMP

Step 9: EXIT.

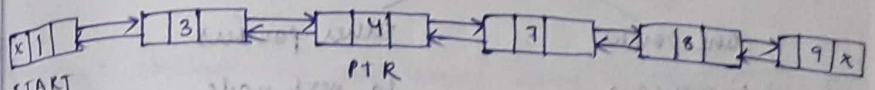
CASE 4: Deleting before a given node.



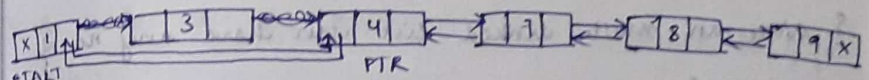
1. Make a pointer variable PTR that points to the first node of the list.



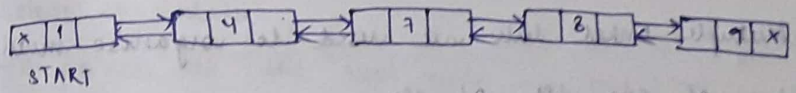
2. Move PTR further till its data part is equal to the value before which the node has to be deleted.



3. Delete the node preceding PTR.



4.



Step 1: If START = NULL
Write UNDERFLOW
Go to step 9
[END OF IF]

Step 2: set PTR = START

Step 3: Repeat step 4 while PTR → DATA ! = NUM

Step 4: SET PTR = PTR → NEXT
[END OF LOOP]

Step 5: SET TEMP = PTR → PREV

Step 6: SET TEMP → PREV → NEXT = PTR.

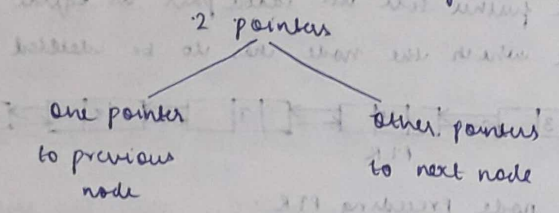
Step 7: SET PTR → PREV = TEMP → PREV

MULTI LINKED LIST:

> Multi linked list, each ~~too~~ node can have 'n' no of ~~position~~ pointers to other nodes.

> ~~A~~ doubly linked list is a special case of multi linked lists.

> ~~A~~ doubly linked list has exactly two pointers.



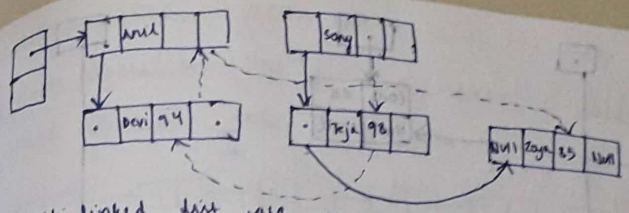
{ Node in multi linked list can have any no of pointers }.

> Multiple linked lists are used to organise multiple orders of one set of elements.

Eg: If we have linked list that stores name and marks obtained by students in a class then we can organise the nodes of the list in two ways.

> Organise the nodes alphabetically.

> Organise the nodes according to decreasing order of marks so that the information of student who got highest marks comes before other students.



> Multi linked list are used to store sparse matrices. Generally, the sparse matrices have very few non-zero values, if we use a normal array to store such matrices, we will end up wasting a lot of space.

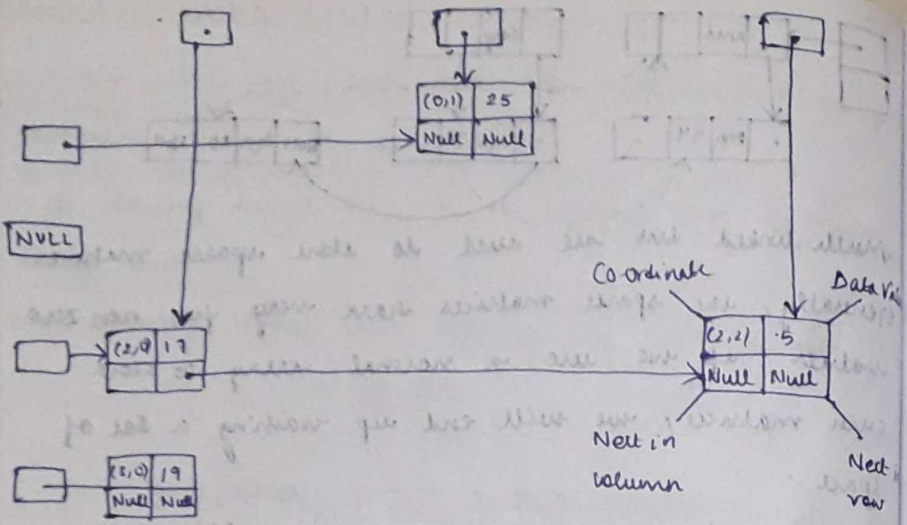
> Sparse matrix is represented using a linked list for every row and column.

> A node in the multi-linked list will have four parts.

1. Stores the data
2. A pointer to the next node in the row.
3. Stores a pointer to the next node in the column
4. Stores the coordinates (or) the row and column number

Eg:

y \ x	0	1	2
0	0	25	0
1	0	0	0
2	17	0	5
3	19	0	0

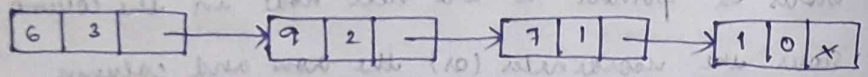


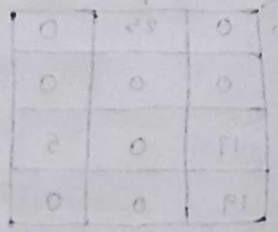
Polynomial Representation:

Using linked list:

Eg: $6x^3 + 9x^2 + 7x + 1$

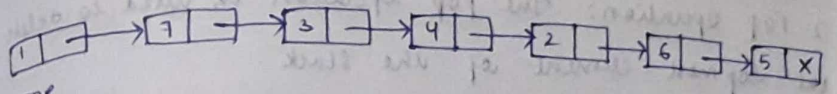
Every individual term in a polynomial consists of two parts, a coefficient and power (exponent)





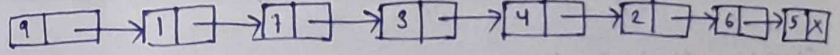
LINKED LISTS IN STACKS:

1. Push operation: The push operation is used to insert element into the stack.



→ To insert an element with value 9, we check the condition TOP = NULL. If this is the case then we allocated memory for a new node, store the value in its DATA part and NULL in its NEXT part. The new node then will be called TOP.

However if TOP != NULL, then we insert the new node at the begining of the linked stack and name new node as TOP.



Step 1: Allocate memory for the new node and name it as NEW_NODE.

Step 2: SET NEW_NODE -> DATA = VAL

Step 3: If TOP = NULL

SET NEW_NODE -> NEXT = NULL

SET TOP = NEW_NODE

Else

SET NEW_NODE -> NEXT = TOP

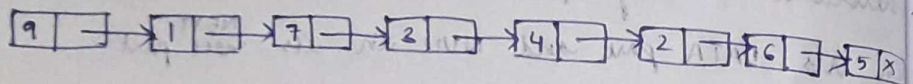
SET TOP = NEW_NODE

[END OF IF]

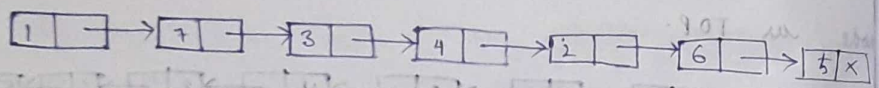
Step 4: END.

2- Pop Operation: The pop operation is used to delete the top most element of the stack.

> Before deleting the value, we must first check if TOP = NULL, if it is true stack is empty and underflow message will be printed.



> In case TOP != NULL then we delete the node pointed by TOP, and make TOP point to the second element of the linked list.



Step 1: If TOP = NULL

PRINT "UNDERFLOW"

GO to step 5

[END OF IF]

Step 2: SET PTR = TOP

Step 3: SET TOP = TOP -> NEXT

Step 4: FREE PTR

Step 5: END.

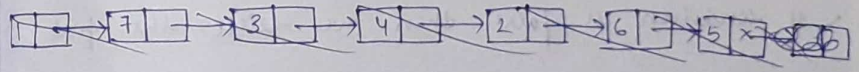
LINKED REPRESENTATION OF QUEUS:

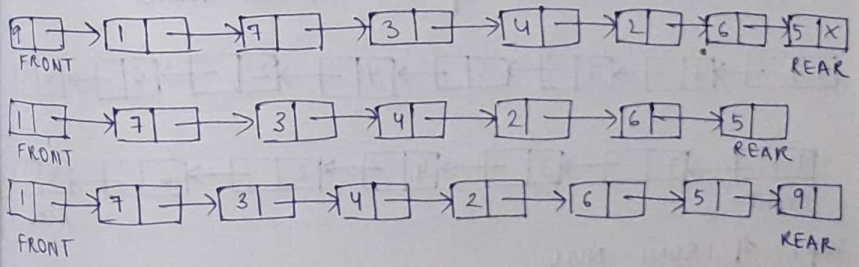
1. Insert operation: Insert operation is used to insert an element into a queue. The new element is added as the last element of the queue.

> To insert an element we first check if FRONT = NULL, if it is true then the queue is empty.

> So we allocate memory for a new node, store the value in its ~~data~~ DATA part and NULL to its NEXT part. The new node then will be called FRONT & REAR.

> If FRONT != NULL, then we will insert the new node at the rear end of the linked queue and name this new node as REAR.





Step 1: Allocate memory for the new node and name it as PTR.

Step 2: SET PTR -> DATA = VAL

Step 3: If FRONT = NULL

SET FRONT = REAR = PTR

SET FRONT -> NEXT = REAR -> NEXT = NULL

ELSE

SET REAR -> NEXT = PTR

SET REAR = PTR

SET REAR -> NEXT = NULL

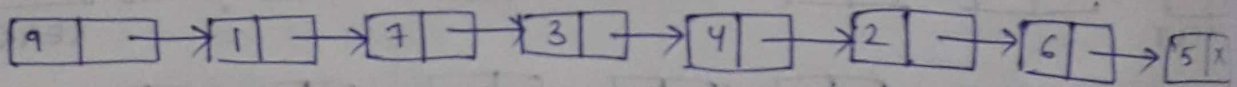
[END OF IF]

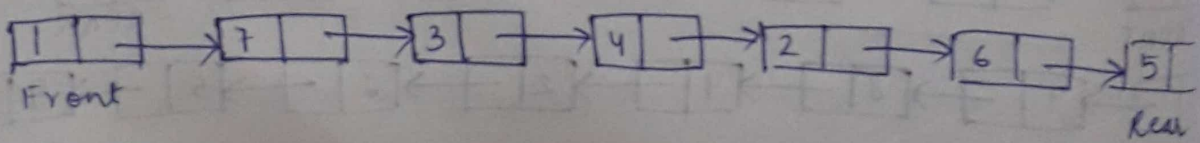
Step 4: END.

2. Delete Operation: The delete operation is used to delete the first inserted in a queue i.e, the element whose address is stored in FRONT.

> Before deleting the value we must check if FRONT = NULL because if this is the case, then the queue is empty and no more deletions can be done. An underflow message will be printed.

>. If condition is false then we delete the first node pointed by FRONT. The front will now point to the second element of the linked queue.





Step 1: If FRONT = NULL
Write UNDERFLOW
Go to step 5
[END OF IF]

Step 2: SET PTR = FRONT

Step 3: SET ~~PTR -> NEXT = FRONT~~ FRONT = FRONT -> NEXT

Step 4: FREE PTR

Step 5: END.

UNIT - 4

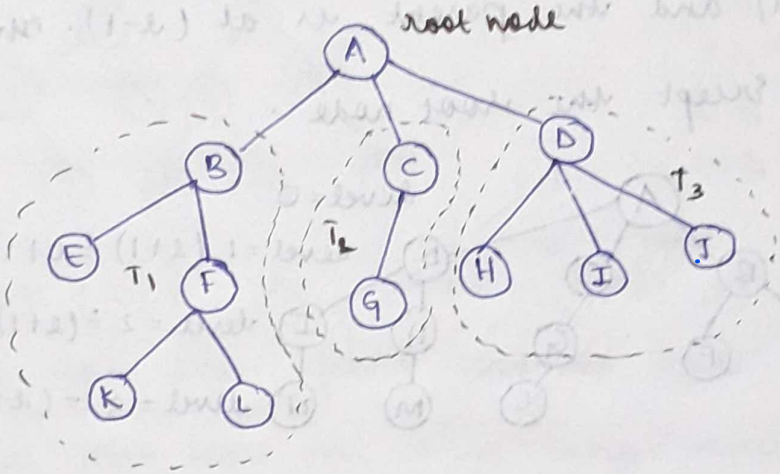
TREES

A tree is a finite set of one (or) more nodes such that

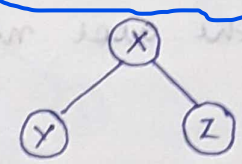
i, There ~~are nodes~~ and is a specially designed node called root node.

ii, the remaining nodes are partitioned into m n ($n > 0$) disjoint sets $T_1, T_2, \ldots T_n$ where each T_i ($i = 1, 2, \ldots n$) is a tree $T_1, T_2 \ldots T_n$ are called subtrees of the root

eg

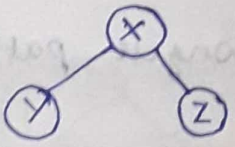


Parent: The parent of a node is the immediate predecessor of a node.



'X' is the parent of 'Y' and 'Z'.

Child: If the immediate predecessor of a node in the parent of the node then all the immediate successor of a node are known as child.



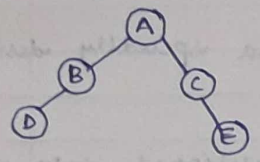
'Y' and 'Z' are children of 'X'.

The child which is on the left side is called left child and on the right side is called right child.

LINK: This is a pointer to a node in a bus.

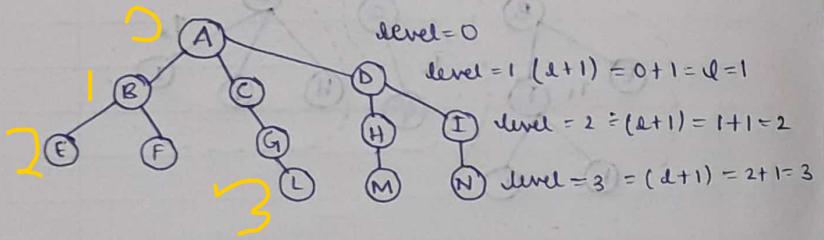
ROOT: This is specially designed node with no parent

LEAF: The node which is at the end and does not have any child is called leaf node.

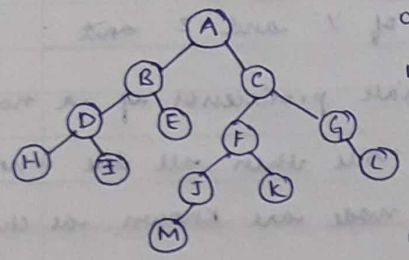


Here D, E are leaf nodes

LEVEL: Level is the rank in the hierarchy. The root node has level 0. If a node is at level l, then its child level $(l+1)$ and the parent is at $(l-1)$. This is true for all nodes. Except the root node.



HEIGHT: The maximum number of nodes that is possible in a pointer starting from the root node to leaf node is called height of a tree.

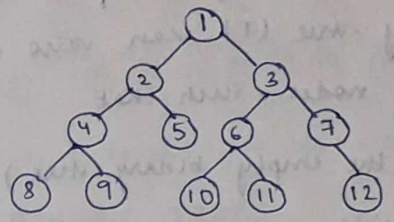


0
1
2
3
4

height = dmax + 1
= 4 + 1
= 5.

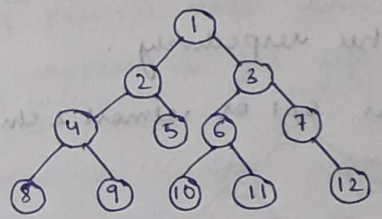
In the above tree the longest path is A-C-F-J-M and height of the tree is 5.

DEGREE: Degree of a node is equal to the number of children that a node has (degree of a leaf node is zero).



Degree of node '4' is 2
Degree of node '5' is 0
Degree of node '7' is 1

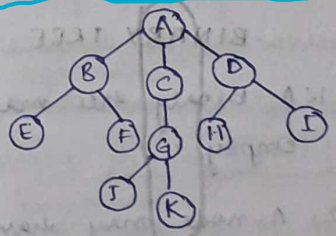
SIBLING: The nodes which have the same parent are called siblings.



Nodes: 2 & 3
4 & 5
8 & 9
6 & 7
10 & 11
} Siblings.

ANCESTOR NODE: An ancestor node of a node is any predecessor node on the path from root to that node.

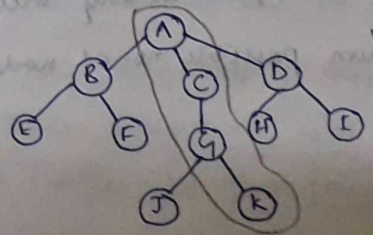
Root node does not have any ancestors



Eg: A, C & G are the ancestors of node K.

DESCENDANT NODE: A descendant node is any successor node on any path from the node to leaf node.

> leaf node do not have any descendants.



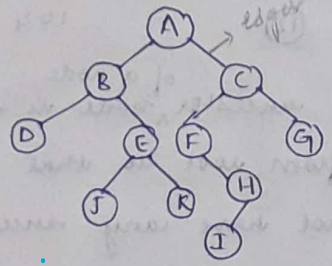
Eg: C, G, K are the Descendants of node A.

BINARY TREES:

A binary tree is a special form of a tree like of general tree, a binary tree (T) can also be defined as the finite set of nodes, such that

1. T is empty (called the empty binary tree) or
2. T contains a specially designed node called the root of T and the remaining nodes of T form two disjoint binary trees T_1 and T_2 which are called left subtree and right subtree respectively.

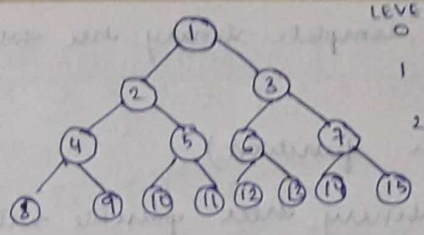
In binary tree each node has 0, 1 or atmost 2 children



GENERAL TREE	BINARY TREE
1. A tree can never be empty	1. A binary tree may be empty.
2. A node may have any number of children	2. A node may have atmost two children

FULL BINARY TREES:

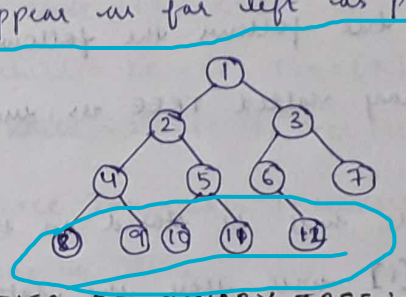
A binary tree is said to be full binary tree if it contains the maximum possible no. of nodes at all levels.



Height = $l_{max} + 1$ ⇒ $3 + 1 = 4$.

COMPLETE BINARY TREE:

A binary tree said to be complete binary tree if its levels, except possibly the last level have the maximum no. of positive nodes and all the nodes in the last level appear as far left as possible.



PROPERTIES OF BINARY TREE:-

1. In any binary tree the maximum no. of nodes on level 'l' is 2^l where $\boxed{l \geq 0}$
2. The ~~maximum~~ maximum no. of nodes possible in a binary tree of height 'h' is $\boxed{2^h - 1}$
3. The ~~maximum~~ minimum no. of nodes possible in a binary tree of height h is $\underline{h}$.
4. For any non-empty binary tree, if 'n' is the no. of nodes, and 'e' is the no. of edges, then $\boxed{n = e + 1}$
5. For any non-empty binary tree T, if n_0 is the no of leaf nodes (degree = 0) and n_2 is the no. of internal nodes (degree = 2) then $\boxed{n_0 = n_2 + 1}$

6. The height of a complete binary tree with 'n' no. of nodes is

$$\lceil \log_2 (n+1) \rceil \quad (\text{ceil function})$$

7. The total no. of binary trees possible with 'n' nodes is

$$\frac{1}{n+1} {}^{2n}C_n$$

ARRAY REPRESENTATION (SEQUENTIAL) OF BINARY TREES:

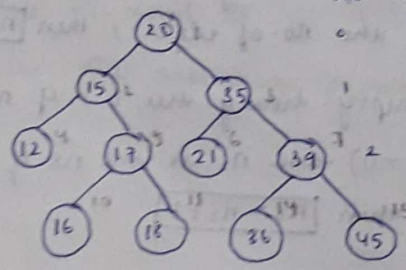
> Sequential representation of trees is done using single (or) one-dimensional array.

A sequential binary tree follows the following rules

1. A one-dimensional array called TREE is used to store elements of tree
2. The root of the tree will be stored in the first location that is TREE[1] will store the data of the root element.
3. The children of a node stored in location K will be stored in location 2K (left child) and 2K+1 (right child)

0 → 2K+1
2K → 2

4. The maximum size of the array TREE is given by $2^h - 1$ where 'h' is the height of the tree.
5. An empty tree (or) sub tree is specified using NULL. If TREE[1] = NULL then the tree is empty.



left = 2K
right = 2K+1

Height 4

Array size = $2^h - 1$

$2^4 - 1$

= 15 //

20	15	35	12	17	21	39			16	18			36	45
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Node = 20, K = 1

Tree[K] = Tree[1] = 20

left child Tree[2K] = Tree[2] = 15

Right child = Tree[2K+1] = Tree[3] = 35.

Now, IInd Node = 39, K = 7

left child = 2K = 14, Tree[14] = 36

Right child = 2K+1 = 15, Tree[15] = 45

IIIrd Node = 17, K = 5

left child = 2K = 10, Tree[K] = 16

Right child = 2K+1 = 11, Tree[11] = 18.

Binary Tree Traversals (Recursion):

> Traversing a binary tree is the process of visiting each node in the tree exactly once in a systematic way.

> In linear data structure such as stacks, queues the elements are traversed sequentially.

> Tree is a non-linear data structure in which the element can be traversed in many different ways.

> There are different algorithm for tree traversals these algorithms differ in the order in which the nodes are visited.

> Here we discuss three traversal algorithm

1. pre-order traversal
2. In-order traversal
3. post-order traversal
4. level-order traversal.

Order traversals:

Pre order traversal: (Root, left, right)

To traverse a non-empty binary tree in pre-order the following operation are performed recursively at each node.

1. Visiting the root node.
2. Traverse the left subtree in pre order.
3. Traverse the right subtree in pre order.

The pre-order traversal is also called as depth-first traversal.

Algorithm:

1. Repeat steps 2 & to 4 while TREE != NULL
2. Write WRITE → DATA
3. PREORDER (TREE → LEFT)
4. PREORDER (TREE → RIGHT)
   [END OF LOOP]
5. END.

> Pre-Order is also known as NLR (Node-left-Right) traversal algorithm.

> NLE is used to extract a prefix notation from an expression tree.

In-Order Traversal: (left, root, right)

> Traversing the left subtree in inorder visiting the root node.

> Traversing the right subtree in inorder.

> This is also called as symmetric traversal (or) LNR (left-Node-Right) traversal algorithm.

Algorithm:

1. Repeat steps 2 to 4 while TREE != NULL
2. INORDER (TREE → LEFT)
3. Write (TREE → DATA)
4. INORDER (TREE → RIGHT)
   [END OF LOOP]
5. END.

Post Order Traversal: (left, right, root)

1. Traversing the left subtree in post order
2. Traversing the right subtree in post order.
3. Visiting the root node.

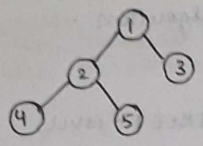
This is also called as LRN (left-Right-Node) traversal algorithm.

> Post order traversals are used to extract postfix notation from an expression tree.

Algorithm:

1. Repeat steps 2 to 4 while TREE != NULL
2. POST ORDER (TREE → LEFT)
3. POST ORDER (TREE → RIGHT)
4. Write TREE → DATA
   [END OF LOOP]
   END.

Sg : i.

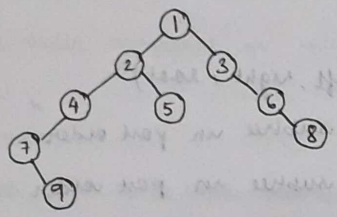


a) Inorder (left, root, right): 4, 2, 5, 1, 3

b) Pre order (Root, left, right): 1, 2, 4, 5, 3

c) Post order (left, right, root): 4, 5, 2, 3, 1

ii)

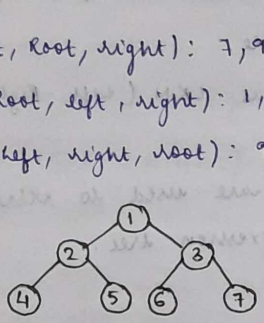


a) Inorder (left, Root, right): 7, 9, 4, 2, 5, 1, 8, 6, 8 3

b) Pre order (Root, left, right): 1, 2, 4, 5 7, 9, 8, 3, 6, 8.

c) Post order (left, right, root): 9, 7, 4, 5, 2, 8, 6, 3, 1.

iii)

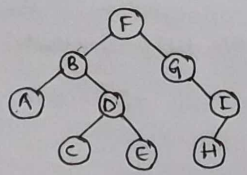


a) Inorder (left, root, right): 4, 2, 5, 1, ~~3, 6, 7~~ 6, 3, 7

b) Pre order (Root, left, right): 1, 2, 4, 5, 3, 6, 7

c) Post order (left, right, root): 4, 5, 2, 6, 7, 3, 1

iv)

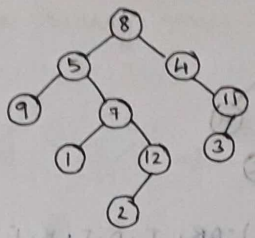


a) Inorder (left, root, right): A, B, C, D, E, F, H, I, G

b) Pre order (Root, left, right): F, B, A, D, C, E, G, I, H

b) Post order (~~Root~~ left, right, root): A, C, E, D, B, H, I, G, F

v)

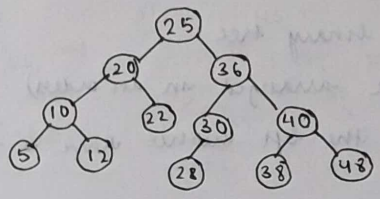


a) Inorder (left, root, right): ~~9, 5, 1, 7, 2, 12~~ 9, 5, 1, 7, 8, 2, 12, 8, 3, 11, 4.

b) Pre order (Root, left, right): 8, 5, 9, 7, 1, 2, 12, 2, 4, 11, 3

c) Post order (left, right, root): 9, 2, 12, 1, 7, 5, 3, 11, 4, 8.

vi)

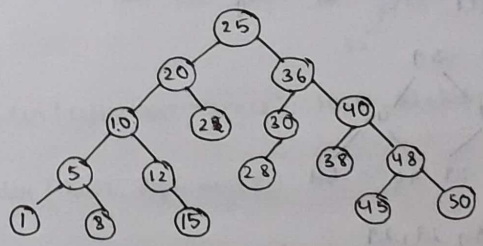


a) Inorder (left, root, right): 5, 10, 12, 20, 22, 25, ~~28~~, 30, 36, 38, 40, 48.

b) Pre order (Root, left, right): 25, 20, 10, 5, 12, 22, 36, 30, 28, 40, 38, 48

c) Post order (left, right, root): 5, 12, 10, 22, 20, 28, 30, 38, 48, 40, 36, 25.

vii)

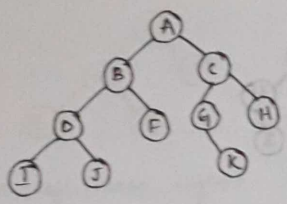


a) Inorder (left, root, right): 1, 5, 8, 10, 15, 12, 20, 22, 25, 28, 30, 36, 38, 40, 45, 48, 50.

b) Pre order (Root, left, right): 25, 20, 10, ~~22~~ 5, 1, 8, 12, 15, 22, 36, 30, 28, 40, 38, 45, 50, 48.

c) Post order (left, right, root): 1, 8, 5, 15, 12, 10, 22, 20, 28, 30, 38, 45, 50, 48, 40, 36, 25

viii)



i) Inorder (left, root, right): ~~D,B~~, I, D, J, B, F, A, ~~C, G, K,~~ H

ii) Preorder (Root, left, right): A, B, D, I, J, F, C, G, K, H

iii) Postorder (~~left~~, right, root): I, J, D, F, B, K, G, H, C, A

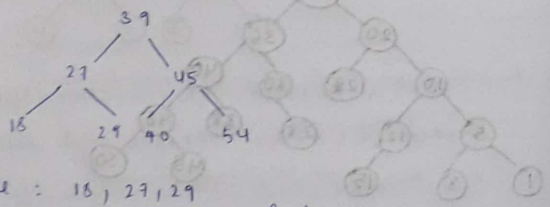
BINARY SEARCH TREE:

It is order based binary tree

(All the nodes are arranged in an order)

- All the nodes in the left subtree have a value less than of root.
- All the nodes in the right-subtree have a value greater than of the root.
- These rules applicable to every sub-tree

Eg:

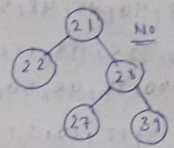


Left Subtree : 18, 27, 29

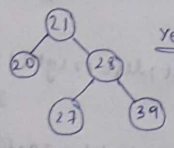
Root : 39

Right subtree : 40, 45, 54

Eg:-



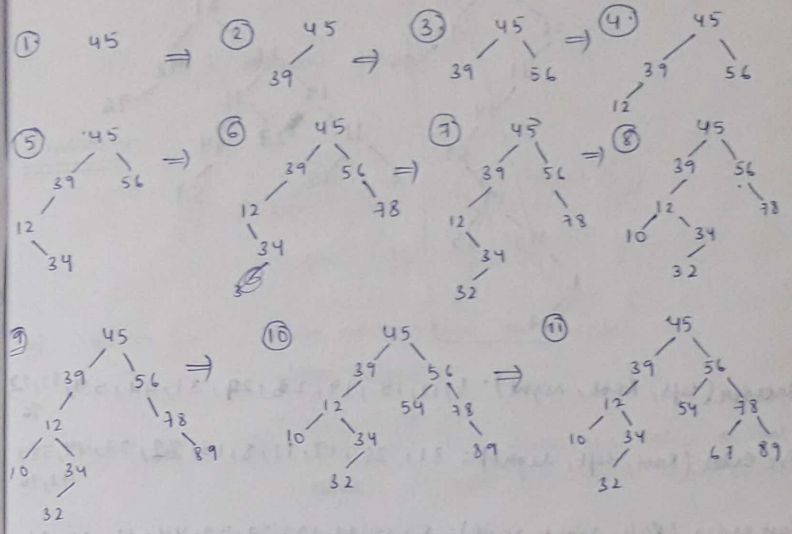
NO



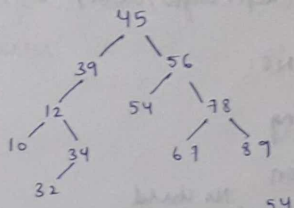
Yes

Construct a Binary search tree

i, 45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67.



Final Binary search tree:

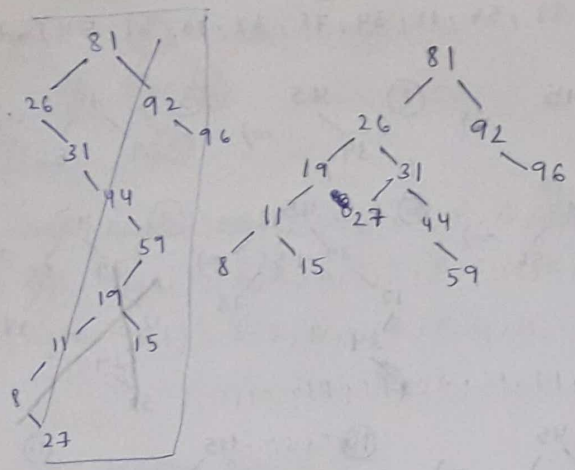


Inorder (left, root, right): 10, 12, 32, 34, 39, 45, ~~54, 56, 67, 78~~ 54, 56, 67, ~~78~~ 89

Preorder (Root, left, right): 45, 39, 12, 10, 34, 32, 56, 54, 78, 67, 89

Postorder (left, right, root): 10, 32, 34, 12, 39, ~~54~~ 54, 67, 89, 78, 56, 45.

ii 81, 92, 26, 31, 44, 59, 19, 96, 11, 15, 8, 27



Inorder (left, Root, right): 8, 11, 15, 19, 26, 27, 31, 44, 59, 81, 92, 96

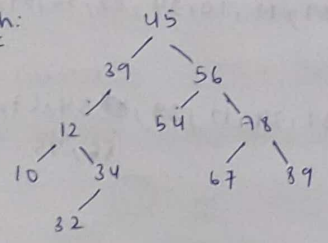
Pre Order (Root, left, right): 81, 26, 19, 11, 8, 15, 31, 27, 44, 59, 92, 96

Post order (left, right, root): 8, 15, 11, 19, 27, 59, 44, 31, 26, 96, 92, 81

OPERATIONS:

1. Searching
2. Insertion
3. Deletion ← No child / One child / Two child

Search:



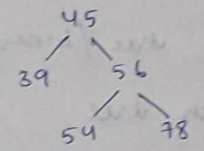
find 78.

1. 45 (78) compare / search starts from root
   ↳ search element
2. Greater than the root, so move right

2. 56 78 compare
   ↳ search value is greater so move right

3. 78 = 78 ← element found

Insertion:



⇒ Insert (55)

⇒ 45 ⟺ 55 New node value is greater
Move right

⇒ 56 ⟺ 55 New node is less than the value at that node
Move left

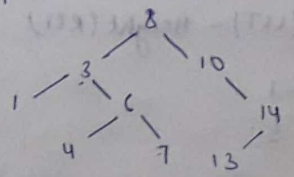
⇒ 54 ⟺ 55 55 is greater so insert at the right position

Deletion:

Deleting a node having i) 0 children ii) one child iii) two child

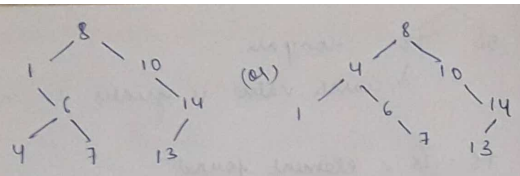
Replace the parent with child from either

left (Maximum element from left) or Right (Minimum element from right)



Delete 3

Left possibility: 1
Right possibility: 4

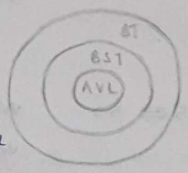


AVL :- (Adelson, Velski & Landis)

> It is a type of binary tree / binary search tree

> self balancing binary search tree

Balancing of left & right subtrees:

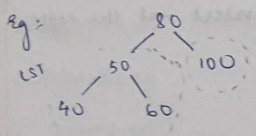


Balancing factor = Height of left subtree
- Height of right subtree

every node should contain 0, 1, -1.

Height Calculation : longest path from leaf node to that node.

Eg:

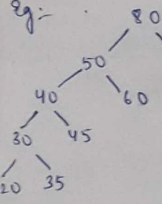


'80' => Height (LST) - Height (RST)

=> 2 - 1

= 1 ✓

Eg:



Not an AVL

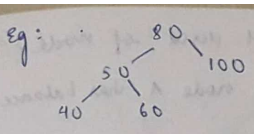
because

Balancing factor = Height (LST) - Height (RST)

4 - 1

= 3. ✗

Eg:

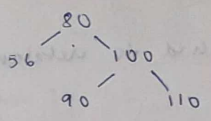


Balancing factor = 2 - 1

= 1. ✓

left Heavy Tree

Eg:

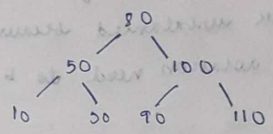


Balancing factor = 1 - 2

= -1 ✓

Right Heavy tree

Eg:



Balancing factor = 2 - 2 = 0 ✓

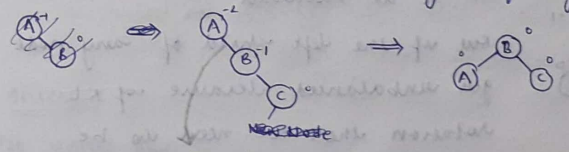
Balanced AVL Tree

* Balancing factor
- 1 (left Heavy tree)
- 0 (Balanced tree)
- -1 (Right Heavy tree)

RELATIONS:

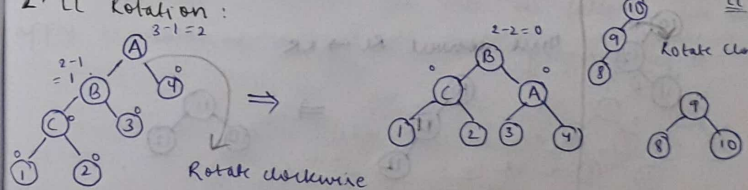
1. RR Rotation:

Node is inserted in the right of right sub tree



Rotate anticlockwise

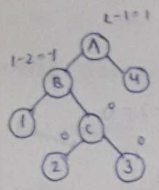
2. LL Rotation:



when node 1' gets inserted as a left child of node 2, then AVL property gets destroyed i.e., node A has balance factor 1 2.

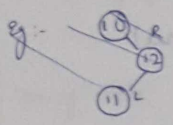
The LL rotation has to be applied to rebalance the node.

3. LR Rotation:

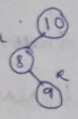


It is balanced but the right child of any node get unbalanced because of LR, then LR rotation need to be applied.

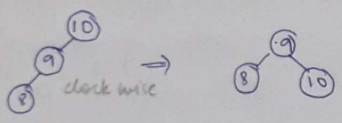
Eg:-



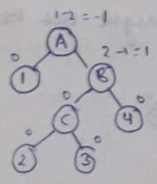
Eg:



First convert LR → LL



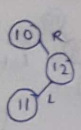
4. RL Rotation:



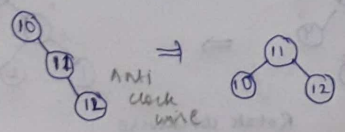
⇒ It is balanced

But if the left child of any node get unbalanced because of RL rotation then RL need to be applied.

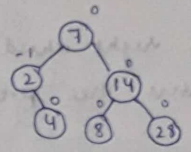
Eg:-



First convert RL → RR

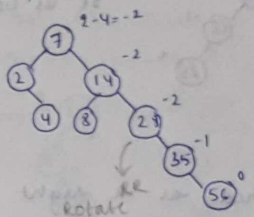


Eg: 7, 2, 14, 4, 8, 28.

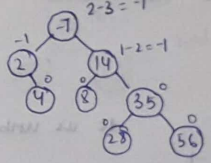


⇒ It is balanced

→ Insert 35, 56

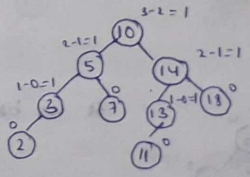


⇒ It is unbalanced



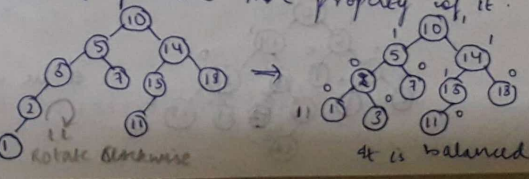
⇒ It is balanced.

Eg: Insert 1, 25, 28, 12 in the given AVL tree



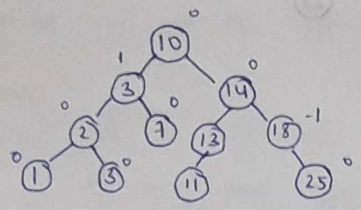
Insert 1:

To insert node 1 we're to attach it as a left child of 2. This will unbalance the tree as follows. we will apply LL rotation to preserve AVL property of it.



Insert 25:

We will insert 25 as a right child of 18. No balancing is required as entire tree preserves the AVL property.

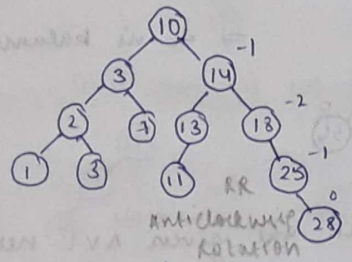


It is balanced.

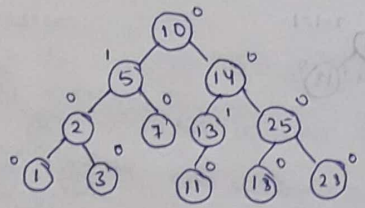
Insert 28:

The node 28 is attached as a right child of 25.

RR rotation is required to rebalance.



It is unbalanced

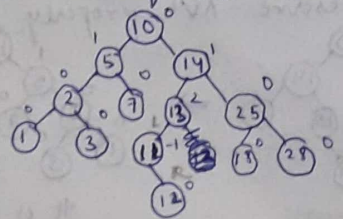


It is balanced.

Insert 12:

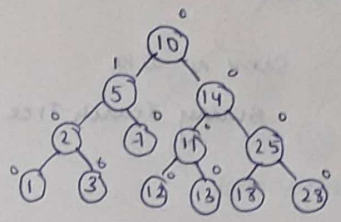
The node 12 is attached as a right child of 11.

LR rotation is required to rebalance



It is unbalanced

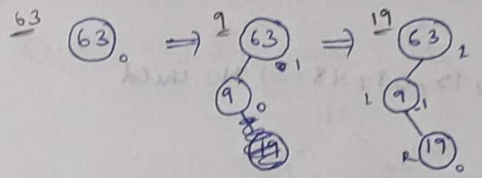
LR → LL (clockwise)



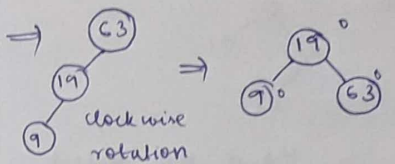
It is balanced.

Q) construct an AVL Tree

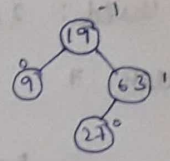
63, 9, 19, 27, 18, 108, 99, 81.

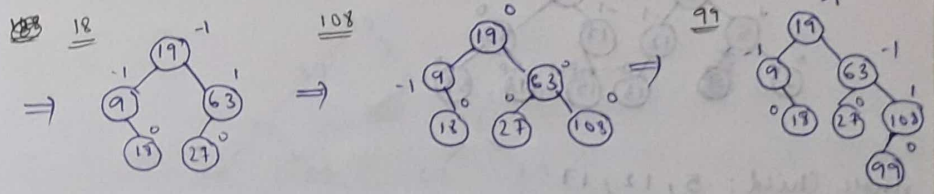


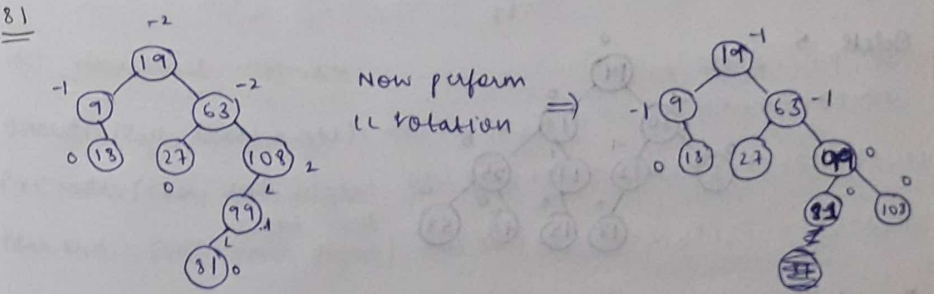
Now perform LR Rotation
First convert to LL



Balanced tree







Now perform
LL rotation

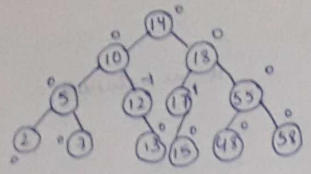
Balanced //

In order: 9, 18, 19, 27, 63, 81, 99, 108.

Pre order: 19, 9, 18, 63, 27, 99, 81, 108.

Post order: 18, 9, 27, 81, 108, 99, 63, 19.

DELETION IN AVL TREE



Same as like Binary Search Tree

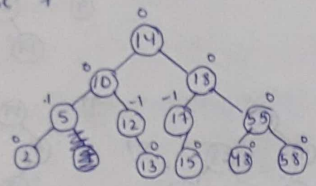
1. Single child
2. No child
3. Two children

~~Single child~~: ~~2, 7, 13, 15, 58, 48~~ => No child

Single child:

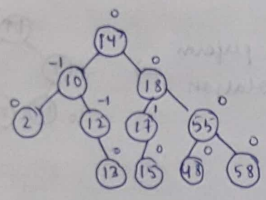
No child: 2, 7, 13, 15, 58, 48.

Delete 7



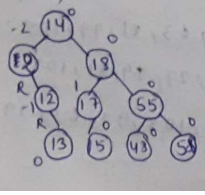
Single Child: 5, 12, 17

Delete 5

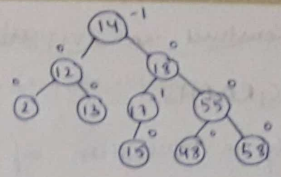


Two children: 10, 14, 18, 55.

Delete 10:



Perform RR rotation, rotate anticlockwise

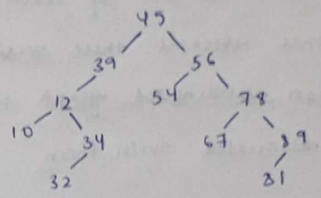


Balanced //.

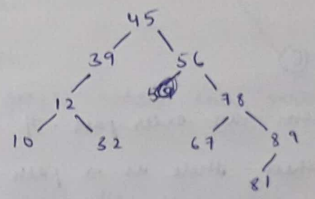
Q) Create Binary search tree for the following elements

45, 39, 56, 12, 54, 78, 32, 10, 89, 54, 67, 81.

Q) Delete 54, 34 from the tree



Delete 54 & 34



~~Q) construct AVL~~

67, 78, 81, 89, 56

Inorder (Left, root, right): 10, 12, 32, 39, 45, ~~81, 89, 67, 78, 56~~

Pre order (Root, left, right): 45, 39, 12, 10, 32, 56, 78, 67, 89, 81

Post order (left, ~~root~~ right root, ~~right~~): 10, 32, 12, 39, 81, 89, 67, 78, 56, 45

Q) ~~Write pre-order to~~ Construct an AVL tree for the

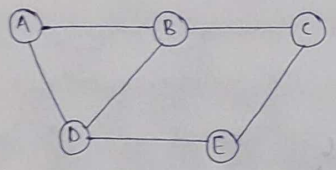
UNIT - 5 GRAPHS

Graph: It is basically a collection of vertices (also called nodes) and edges that connect these vertices

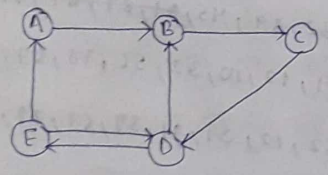
> Graph is viewed as a generalisation of the tree structure.

Definition: A graph G is defined as an ordered set (V, E), where V(G) represents the set of vertices and E(G) represents the edges that connect these vertices

Undirected Graph: In an undirected graph edges do not have any direction associated with them



Directed Graph: Edges form an order pair. If there is an edge from A to B, then there is a path from A to B but not from B to A.

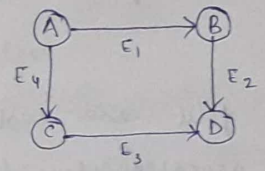


Ordered pairs
(A, B)
(B, C)
(E, A)
(F, D)
(D, E)
(D, B)
(C, D)

Graph Terminology

Adjacent nodes (or) neighbours:

For every edge e = (u, v) that connects nodes, the nodes u & v and the end points are said to be adjacent nodes (or) neighbours



$E_1 = (A, B)$
$E_2 = (B, C)$
$E_3 = (C, D)$
$E_4 = (A, C)$

} Adjacent nodes.

Degree of Node:

Degree of a node is the total no. of edges containing the node

If degree of a node is "0" then that node is called isolated node

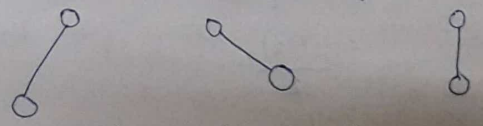
Regular Graph:

It is a graph where each vertex has the same no. of neighbours.

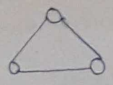
i.e., Every vertex has the same degree.

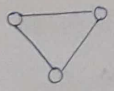
> A regular graph with vertices of degree K is called K-regular graph of degree K.

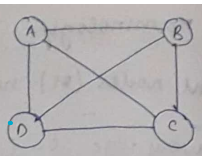
0 = regular graph.



1- regular Graph







2- regular graphs

Path: A path p written as $p = \{v_0, v_1, v_2, v_3 \ldots v_n\}$ of length 'n' from a node u to v is defined as a sequence of (n+1) nodes.

Cycle: A path in which the first and last vertices are same. A simple cycle has no repeated edges / vertices except the first and last vertices.

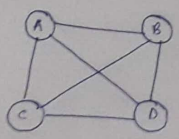
Connected Graph: A graph is said to be connected if for any two vertices (u,v) there is a path from u to v.

> A connected graph that does not have any cycle is called a tree. Therefore a tree is treated as special graph.

Complete Graph: A graph G is said to be complete if all nodes are fully completed.

i.e., there is a path from one node to every other node in the graph

> A complete graph has $n(n-1)/2$ edges where 'n' is no. of nodes in G.



No. of nodes (n) = 4

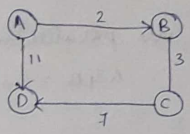
No. of edges $= \frac{n(n-1)}{2}$

$= \frac{4(3)}{2}$

$= 6$.

Labelled Graph / Weighted Graph:

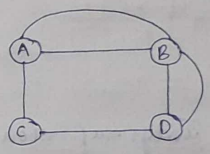
A graph is said to be labelled graph if every edge in the graph is assigned some data (weight).



Multiple Edges

Distinct edges which connect the same end points are called multiple edges.

i.e., $e = (u,v)$ and $e' = (u,v)$ are known as multiple edges of G.



Loop: An edge that has identical end points is called a loop i.e., $e = (u,v)$

Strongly connected directed graphs:

A digraph is said to be strongly connected if and only if there exists a path between every pair of nodes in graph G.

i.e., if there is a path from node u to v, then there must be path from node v to u.

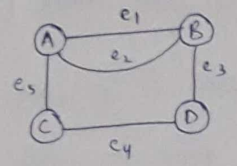
Weakly connected Graph:

A digraph is said to be weakly connected if it is connected by ignoring the direction of edges.

Parallel / Multiple Edges:

Distinct edges which connect the same end-points are called Multiple edges.

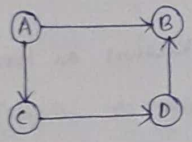
Eg:



e_1, e_2 are parallel edges between A & B.

Simple Directed Graph:

An digraph is said to be simple directed graph if and only if it has no parallel edges.



Representation of Graphs:

There are three common ways of storing graphs in the computer memory.

1. Sequential representation by using adjacency matrix.
2. Linked Representation by using an adjacency list that store the neighbours of a node using a linked list.
3. Adjacency multilist which is an extension of linked representation.

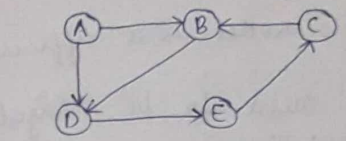
1. Adjacency matrix representation:

> An adjacency matrix is used to represent which nodes are adjacent to one another.

> In an adjacency matrix, the rows and columns are labelled by graph vertices.

> For any graph "G" having n' nodes the adjacency matrix will have the definition of n×n.

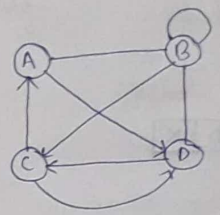
> Adjacency matrix contain only 0's and 1's. It is also called as bit matrix (or) Boolean matrix.



(Directed Graph)

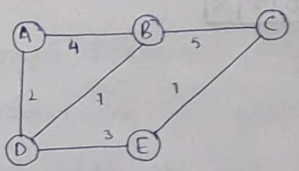
Matrix

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



(Directed graph with loop)

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



(Weighted Graph)

	A	B	C	D	E
A	0	4	0	2	0
B	1	0	0	7	0
C	0	5	0	0	0
D	1	0	0	0	3
E	0	0	1	0	0

2. Adjacency list representation

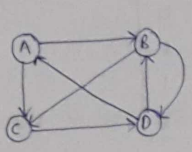
An adjacency list is a another way in which graphs can be represented in the computer's memory. This structure consists of a list of all nodes in Graph G.

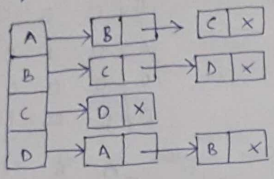
Advantages of adjacency list,

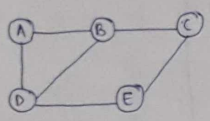
> It is easy to follow and clearly shows the adjacent node of a particular node.

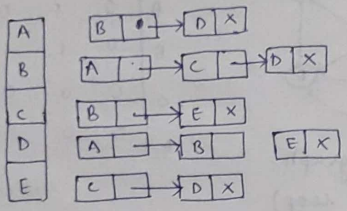
> It is used for storing graphs that have a small to moderate no. of edges.

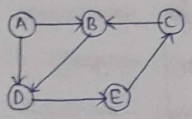
> Adding new nodes in G is easy and straight forward when G is represented using an adjacent list, whereas in adjacency matrix adding new node is a difficult task, as the size of the matrix need to be changed.

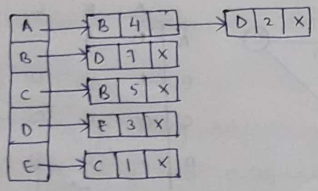












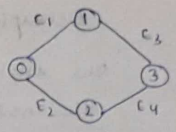
Adjacency Multilist Representation:

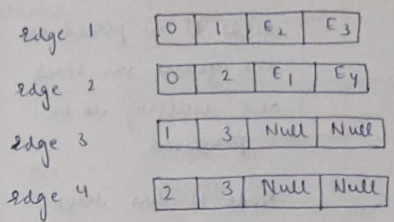
> Graphs can also be represented using multi-lists which can be said to be modified version of adjacency lists.

> Adjacency multi-list is an edge-based rather than a vertex based representation of graphs.

Multilist representation basically consists of two parts

> A directory of nodes information and a set of linked lists containing information about edges.





Vertex	List of edges
0	Edge 1, Edge 2
1	Edge 1, Edge 3
2	Edge 2, Edge 4
3	Edge 3, Edge 4

Graph Traversal Algorithms

> Traversing a graph means method of examining the nodes and edges of the graph.

> There are two standard methods of graph traversal

1. BFS
2. DFS

BFS: (Breadth first search)

> Breadth first uses a queue as an auxillary data to store nodes for further processing.

> Where as DFS uses stack, but both these algorithms are using a variable called STATUS

> During the execution of the algorithm every node in the graph will have the variable STATUS set to 1 or 2 depending on its current state.

Status	State of the node	Description
1	Ready	The initial state of the node N
2	waiting	Node N is placed on the queue or stack and waiting to be processed
3	Processed	Node N has been completely processed.

Algorithm

Step 1: set STATUS = 1 (ready state) for each node G

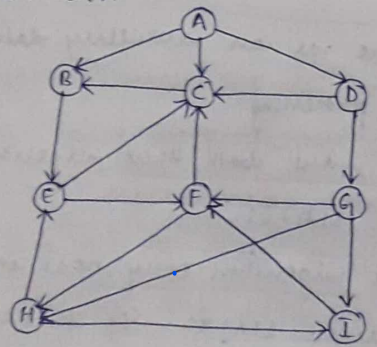
Step 2: enqueue the starting node A and set its STATUS = 2 (waiting state)

Step 3: Repeat steps 4 and 5 until Queue is empty

Step 4: Dequeue a node N. process it and set its STATUS = 3 (processed state)

Step 5: enqueue all the neighbours of N that are in the ready state (whose STATE = 1) and set their STATUS = 2 (waiting state)
[END OF LOOP]

Step 6: EXIT



Adjacency list

A : B, C, D
B : E
C : B, G
D : C, G
E : C, F
F : C, H
G : F, H, I
H : E, I
I : F

Here we need queues : Main Queue, Origin Queue, Status Queue

Initially status of each node in a Queue is

Status -

A	B	C	D	E	F	G	H	I
1	1	1	1	1	1	1	1	1

O -

NULL								

Q -

A	B	C	D	E	F	G	H	I
A								

Front = Rear = 0

> Dequeue 'A' from Queue and add adjacent nodes of A into main Queue, 'A' is inserted into origin (O) Queue by incrementing status to '3'.

Q =

	B	C	D					

O =

A								

S =

A	B	C	D	E	F	G	H	I
3	2	2	2	1	1	1	1	1

> Dequeue 'B' from Queue and add adjacent nodes of B into main Queue by incrementing status to 2 & B is inserted into origin Queue by incrementing status to '3'.

Q =

		C	D	E				

O =

A	B							

S =

A	B	C	D	E	F	G	H	I
3	3	2	2	2	1	1	1	1

> Dequeue 'C' from main Queue and add adjacent nodes of C into main queue by incrementing status to '2' and C inserted into origin queue by incrementing status to '3'.

Q = | | | E | D | E | G | | | | | |

O = | A | B | C | | | | | | |

S = | 3 | 3 | 3 | 2 | 2 | 1 | 2 | 1 | 1 |
A B C D E F G H E

> Dequeue 'D' from main Queue and add adjacent nodes of 'D' into main queue by incrementing status to '2' and insert D into origin Queue by incrementing status to 3.

Q = | | | | | E | G | | | |

O = | A | B | C | D | E | | | | |

S = | 3 | 3 | 3 | 3 | 3 | 2 | 2 | 1 | 1 |
A B C D E F G H I

> Dequeue 'E' from main Queue and add adjacent nodes of 'E' into main Queue

Q = | | | | | G | F | | | |

O = | A | B | C | D | E | | | | |

S = | 3 | 3 | 3 | 3 | 3 | 2 | 2 | 1 | 1 |

> Degree 'G' from main Queue and insert into adjacent nodes of 'G' into main Queue by incrementing status '2' and insert 'G' into origin Queue by incrementing status to 3.

Q = | | | | | | | F | H | I |

O = | A | B | C | D | E | G | | | |

S = | 3 | 3 | 3 | 3 | 3 | 2 | 3 | 2 | 2 |
A B C D E F G H I

> Degree 'F' from main Queue and insert into origin simply by incrementing status to 3.

Q = | | | | | | | | H | I |

O = | A | B | C | D | E | G | F | | |

S = | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 2 | 2 |
A B C D E F G H I

> Degree 'H' from main Queue and insert into origin Queue, simply by incrementing status to '3'.

Q = | | | | | | | | | I |

O = | A | B | C | D | E | G | I | H | |

S = | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 2 |
A B C D E F G H I

> Degree I from main Queue and insert into origin Queue, simply by incrementing status to 3

Q = | | | | | | | | | | } Empty

O = | A | B | C | D | E | F | G | H | I |

S = | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |

Output : A B C D E G H I } By Applying

DEPTH FIRST SEARCH ALGORITHM

> Depth first search algorithm progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found.

Algorithm:

Step 1: Set STATUS = 1 (ready state) for each node

Step 2: Push the starting node 'A' on the stack and set its STATUS = 2 (waiting state)

Step 3: Repeat steps 4 and 5 until stack is empty

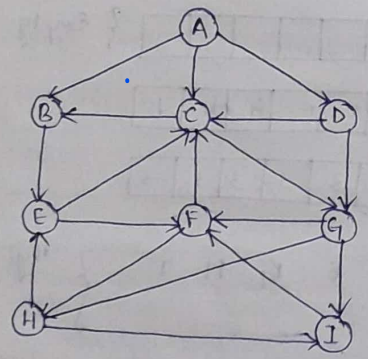
Step 4: Pop the top node N process it and set its status = (processed state).

Step 5: Push on the stack all the neighbours of N that are in the ready state (whose STATUS = 1) and their STATUS = 2 (waiting state).

[END OF LOOP].

Step 6: EXIT

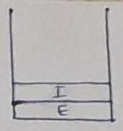
Eg:-



a) Push H onto the stack

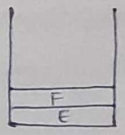
b) pop and print the top element of the stack i.e., 'H'. Push all neighbours of H onto the stack that are in the ready state.

PRINT : H



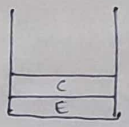
c) Pop and print the top element of the STACK i.e., 'I' push all neighbours of I onto the stack that are in the ready state

PRINT : I



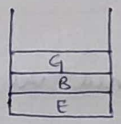
d) Pop and print the top element of the stack i.e., 'F' and all neighbours of 'F' onto the stack that are in the ready state

PRINT : F



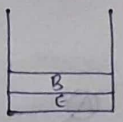
e) Pop and print the top element from the stack i.e., 'C' push all neighbours of 'C' onto the stack that are in the ready state

PRINT : C



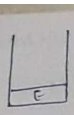
f) Pop and print the top element from the stack i.e, 'G' there 'G' neighbours are already in ready state i.e, no push operation

PRINT : G



g) Pop and print the top element from the stack i.e., 'B' there 'B' neighbours are already in ready state. No push operation

PRINT : B



h) Pop and print the top element from the stack i.e, E
Here C' neighbours are already in ready state. No push operation

PRINT : E

SHORTEST PATH ALGORITHMS

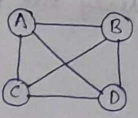
There are three different algorithms to calculate the shortest path between the vertices of a graph G.

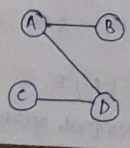
- Minimum Spanning tree
- Dijkstra's Algorithm
- Warshall's Algorithm

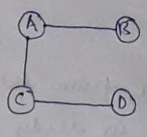
Minimum Spanning Trees:

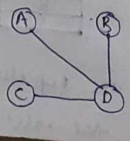
A spanning tree of connected, undirected graph G is a subgraph of G which is a tree connected all vertices together.

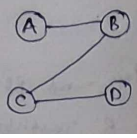
A minimum tree is defined as a spanning tree with weight less than or equal to the weight of every other spanning tree.

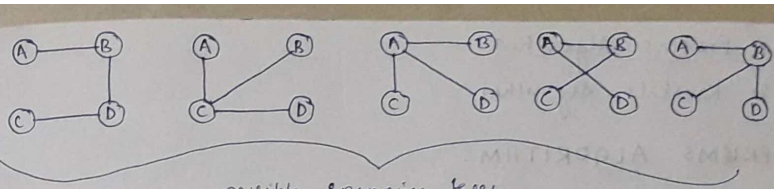




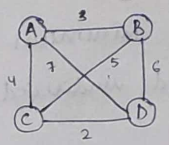


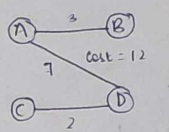


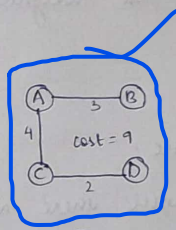


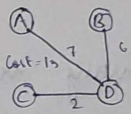


possible spanning trees.

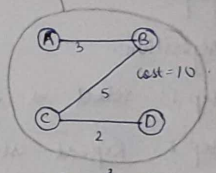


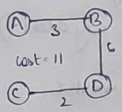


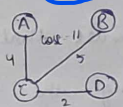


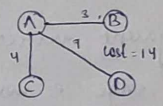


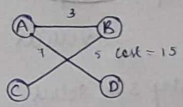
Minimum Spanning tree

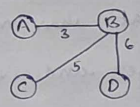












cost = 14

Applications:

- Minimum spanning trees are mainly used for designing networks.
- Minimum spanning trees are used to find airline routes.
- Minimum spanning trees are used to find cheapest way to connect terminals.
- Minimum spanning trees are applied in routing algorithms for finding the most efficient path.

Minimum spanning trees to find the shortest path between nodes, we have two algorithms.

1. Prioms Algorithm
2. Kruskals Algorithm

PRIMS ALGORITHM:

Prism's algorithm is greedy algorithm

Prism's algorithm is used to find a minimum spanning tree for a connected weighted undirected graph.

Algorithm:-

Step 1: Select a starting vertex.

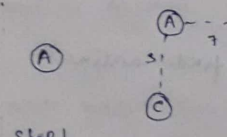
Step 2: Repeat steps 3 & 4 until there are fringe vertices (Adjacent vertices (nodes that are adjacent

Step 3: Select an edge e connecting the tree vertex an fringe vertex that has minimum weight.

Step 4: Add the selected edge and the vertex to the minimum spanning tree T.

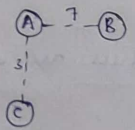
[END OF LOOP]

Step 5: EXIT

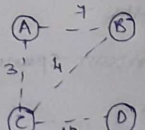


Step 1

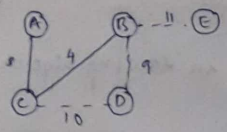
Step 2



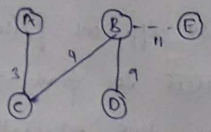
Step 3



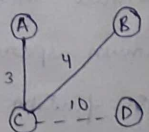
Step 4



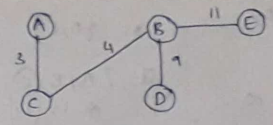
Step 6



Step 7

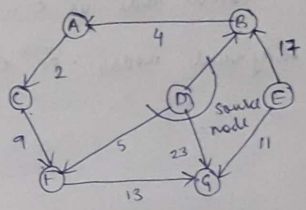


Step 5

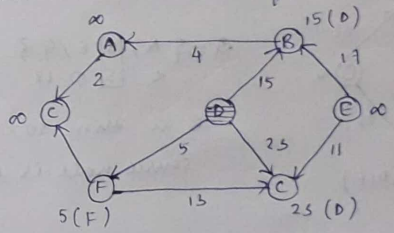


Step 8

weight / cost = 27



⇒ Find the distance from source node to adjacent



⇒ Write source and Queue sets

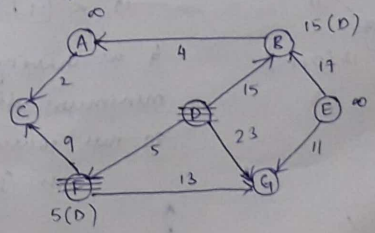
source S = {D}

Q = {A, B, C, D, E, F, G}

∞ 15 ∞ ∞ ∞ 5 23

→ min distance

⇒ Now find the adjacent nodes of 'F' and the distances from source to node to F adjacent nodes

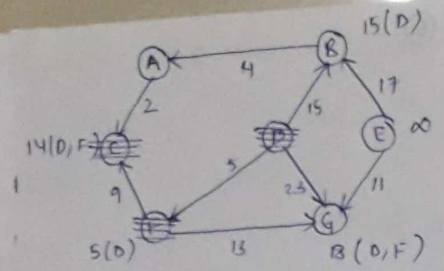


F' adjacent nodes

C, G

Find distances for both C & G

(Next page)



S = {D, F}

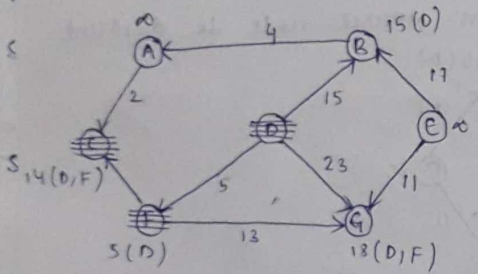
Q = {A, B, C, E, G}
∞ 15 14 ∞ 18

C is minimum so next source node is 'C'.

⇒ C does not have any adjacent nodes. so include in source set

S = {D, F, C}

Q = {A, B, E, G}

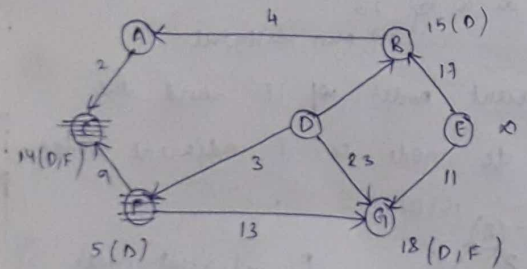


Q = {A, B, E, G}
∞ 15 ∞ 18

B is min so, source node is 'B'

⇒ Now find adjacent nodes of 'B' and find distance from source node to adjacent nodes.

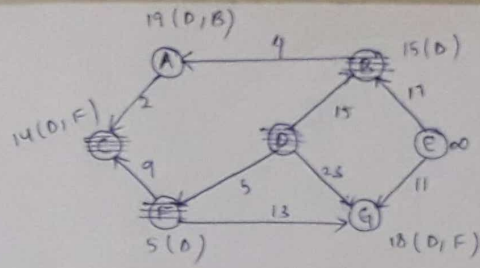
B adjacent node is A.



S = {D, F, C, B}

Q = {A, E, G}
19 ∞ 18

G is having minimum distance so next source node is G.

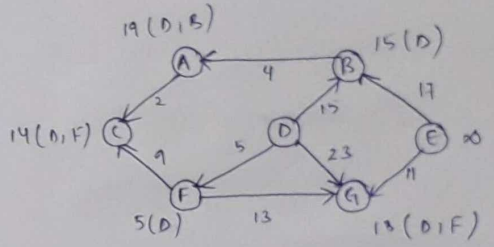


G does not have any adjacent nodes

S = {D, F, C, B, G}

Q = {A, E}
19 ∞

A is next source node



S = {A, B, C, D, F, G}

Q = {E}

↓

Nothing but E is not reachable from source node 'D'.

A, B, C, D, F, G are reachable nodes from source node D.