* Data :- Data means raw facts and figures for eg:- any value like Shivam, 9428 etc. are called data.
The processed data is called information.

* Data Structure :- Data Structure is the study of how we can store an organise data so that it can be used efficiently. It involve's that how data can be store and accessed from the memory.
The data structure can be classified into two :-

1. Linear data Structure :-
In linear D.S the data elements are stored on continous memory location.
Eg :- Array, Stack, Queue, linked list.

2. Non - linear data Structure :-
Such data Structure which does not store data in linear formate is called non linear D.S.
Eg :- Tree, graph.

* Algorithim :- Step by Step Solution of a problem is called algorithim.
Eg :- Add of 2. no. Step 1. Start / Begin
Step 2 read n_1
Step 3. read n_2
Step 4. Set $c = n_1 + n_2$
Step 5. Write c
Step 6. end / Stop.

Eg:- Multiple two no.

Step 1. Start / Begin
Step 2. read a
Step 3. read b
Step 4. read c.
Step 5. Set mul = a*b*c.
Step 6. Write mul.
Step 7. End / Stop.

ques Write algo. to check a number is +ve, -ve or zero.
Solu
Step 1:- Start
Step 2: read n
Step 3: check if n>0
Write "positive no"
Step 4: check if n<0
Write "negative no"
Step 5: check if n=0
Write "zero"
Step 6: end.

ques Write algo. to check no. is even or odd.
Step 1: Start
Step 2: Read n
Step 3: check if n%2=0
Write "even no"
Otherwise
Write "odd no"
Step 4. end.

ques Write an algo. to print counting from 1 to 10 using for loop.
Sol.
Step 1: Start
Step 2: Set i = 1.
Step 3: for i<=10
i) Write i
ii) i = i+1
Step 4: end.

While loop :-
Step 1: Start
Step 2: Set i=1
Step 3: While i<=10 repeat i, ii.
i, Write i
ii, i = i+1.
Step 4.: end.

Complexity :- means how much time or space is required by the algorithim to solve the problem and generate the require result. We have following two type complexity:
i) Space / Size complexity.
ii) Time complexity.

Space / Size complexity :- In Size complexity we measures the Space required by algorithm to solve a problem. if two algorithm are developed for solving the same problem and we want to select the best algo. then we consid the Size taken by algo.

eg:- Sol of

```
a = a+b
b = a-b
a = a-b
```

Sol of

```
c = a
a = b
b = c
```

first solution is more efficient in term of space complexity.

* Time complexity :- Time complexity means how much time is required by the algo. to solve the problem and generate result. if we have two algo. for solving a problem, then we choose the algo. which takes less time. The time complexity is proportional to the number statements which are executed for solving a problem.

We have following three cases of time complexity of an algo.

. Best case :- When a algo. generates the result in minimum time then it is called as best case of time complexity. of the algo. For example, in linear search if the searched value is found in the begining then it is best case of time complexity.

10 20 30 40 50 60 70 80 90 100

Search : 10.

Best case.

Worst Case : When a algo takes max. time in generating result, then it is called as worst case of time complexity. For example in linear search if the searched value is not present or found at end of the array then it is called, worst case time complexity.

Eg:- 10 20 30 40 50 60 70 80 90 100.

Search : 120.

Worst.

Average case :- Average case means the time which lies in between the time taken in best case and the time taken in worst case.

10 20 30 40 50 60 70 80 90 100

Search 60

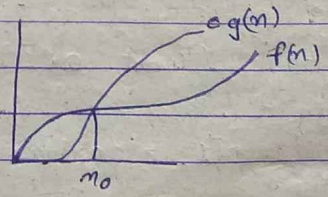
average case.

* Asymptotic notation :- asymptotic notation are used to represent the worst case, Best case and average case of an algo. Mathematically we have following three Asymptotic Notation.

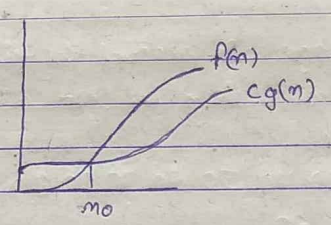
1. Big oh (O(n)) Worst case
2. Omega ($\Omega(n)$) Best case
3. Theta ($\Theta(n)$) Average case.

1. Big oh :- This notation is the formal way to express the upper bound the running of algo. it measures the worst case complexity of an algo.

$O(f(n)) = \{ g(n) : \text{there exist } c>0 \text{ and } n_0 \text{ such that } f(n) <= cg(n) \text{ for all } n > n_0 \}$

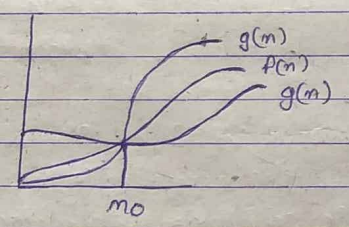


* omega :- This is the formula value to express the lower bound of algo. at running time of algo. it measures the best case time complexity.



$\Omega(f(n)) = \{ g(n) = \text{these exist } c>0 \text{ and } n_0 \text{ such that } cg(n) <= f(n) \text{ for all } n > n_0 \}$

* Theta :- This is the formula value to express the average case of running time of an algo. in other words we can say that it express the both lower and upper bound of an algo.



$\Theta(f(n)) = \{ g(n) \text{ if and only if } g(n) = O(f(n)) \text{ and } g(n) = \Omega(f(n)) \text{ for all } n > n_0 \}$

eg:-
```
read a
read b      } 1 unit. O(1)
c = a+b
```
O(1) = constant time.

eg:-
```
for (i=1; i<=n; i++)
    f(n) = n+1
    f(n) = O(n)
```

eg:-
```
for (i=1; i<=n; i++)
    for (i=1; i<n; i++)
        { for (j=1; j<=n; j++)
        }
```
$f(n) = n + n^2$
$= O(n^2)$

* Array :- Array is a linear data structure which allow us to store multiple value of same data type.
Array allows us to store multiple value under same name.
The value stored in array are called as array element.
All the array element are identified by array name and index number. The index number of n^{th} element is n-1
For example the first element have index no. 0.

To create an array we use syntax:-
datatype arrayname[size];
Note : size of array must non negative constant integer.
int a[5].
. The 1st element is identified as a[0].
" 2nd " " " " a[1] and so on.
The all array element are stored on continous memory location.
We can also intialize array while creating array.
int a[] = {1,2,3,4,5};

* Types of array
i) One dimensional array.
ii) Two dimensional array.
iii) Multidimensional array.

1. One dimensional array:- We can perform following operation on one dimensional arrays:

i) Initialization : providing value to every element of the array is called initialization.

1) At creation time

```
int a[] = {1,2,3,4,5};
   or
int a[3];
a[0]=1; a[1]=2; a[2]=3;
```

2) At run time : We use scanf() to initialize the array elements.

```
int a[3];
scanf("%d",&a[0]); scanf("%d",&a[1]); ....
   or
for(i=0;i<3;i++)
    scanf("%d",&a[i]);
```

Algorithm of initialization

Step 1: Start
Step 2: Set i = 0
Step 3: repeat step 4,5 for i < size.
Step 4: read a[i]
Step 5: Set i = i+1
Step 6: end.

ii) Traverse : To use every element exactly once, is called traversal. Generally printing all element of array is called traversal.

```
for(i=0;i<3;i++)
    printf("%d", a[i]);
```

Algorithm :-

Step 1 : Start
Step 2 : Set i = 0
Step 3 : repeat step 4,5 for i < size.
Step 4 : Write a[i]
Step 5 : Set i = i+1.
Step 6 : end.

iii) Insertion : Insertion means inserting a value in the array. When we insert the value in array then existing values are shifted towards right.

```
for(i=size-1; i>pos; i--)
        arr[i] = arr[i-1]
arr[pos] = data;
```

* Algorithm

Step 1 : Start
Step 2 : read data
Step 3 : read pos.
Step 4 : Set i = size-1
Step 5 repeat steps 6,7 for i > pos.
Step 6 Set arr[i] = arr[i-1]
Step 7 Set i = i-1
Step 8 : Set arr[pos] = data.
Step 9 : end.

```
# include <stdio.h>
  int main()
{     int arr[5] = { 10,20,30,40,50};
      int i, pos, data;
      printf ("enter data");
      scanf ("%d", &data);
     printf ("enter position :");
     scanf ("%d", &pos);
     for (i=4; i>pos; i--)
            arr[i] = arr[i-1];
     arr[pos] = data;
printf (" after inserting %d at %d index the array is ", data
                                                    , pos);

     for(i=0; i<5; i++)
            printf ("%d", arr[i]);
     return 0;
}
```

iv) Deletion:- Deletion means deleting the value from a given index. When we delete values from array then existing values are shifted towards left.

Step 1: Start
Step 2: read pos
Step 3: Set i = pos
Step 4: repeat Steps 4,5 for i<size-1.
Step 5: Set arr[i] = arr[i+1]
Step 6: set i = i+1
Step 7: Set arr[i] = 0
Step 8: end.

```
# include<stdio.h>
  int main()
{
     int arr[5] = { 10,20,30,40,50}
      int i, pos;
     printf ("enter position");
     scanf ("%d", &pos);

     for (i=pos; i<4; i++)
            arr[i] = arr[i+1];

     arr[i] = 0
printf ("after deleting value from %d index array is", pos

     for (i=0; i<5; i++)
          printf ("%d", arr[i]);

     return 0;
}
```

v) Searching :- finding a value in the array is called searching. We can perform following two types of searching :

1. Linear Search :- In linear search we compare the data with every element of the array one by one. it is performed on unsorted array.

Step 1: Start
Step 2: read data
Step 3: Set i = 0
Step 4: repeat steps 5, 6 for i < size
Step 5: check if arr[i] == data
then
a) write "data found"
b) break the loop
Step 6: Set i = i+1
Step 7: check if i == size
then
c) Write "data not found"
Step 8: end.

```
# include <stdio.h>
int main()
{ int arr[5] = { 10, 20, 5, -9, 8 };
  int i, data;
  printf("enter the value to be searched");
  scanf("%d", &data);
  for (i=0; i<5; i++)
      if(arr[i] == data)
  { printf("%d found", data);
    Break;
  }
  if (i==5)
      printf("%d not found in array", data);
  return 0;
}
```

Linear Search have O(n) complexity.

2. Binary Search : it is also a searching method, but it can be performed only on sorted array. In binary search we divide the array in two parts and search acc. it is faster than the linear search.
Binary search have O(log n) complexity.

Step 1: Start
Step 2: read data
Step 3: set low = 0
Step 4: Set high = size-1
Step 5: repeat steps 6, 7 while low <= high.
Step 6: mid = (low + high) / 2
Step 7: check if arr[mid] == data
then
a) write "data found"
b) Break
otherwise check if data > arr[mid]
then
c) Set low = mid + 1
otherwise
d) set high = mid - 1.
Step 8: check if low > high
then e) write "data not found"
Step 9: end

```c
#include <stdio.h>
int main()
{   int arr[5] = { 10,15,17,18,20 };
    int i, data, low = 0, high = 4, mid;
    printf("enter value to searched");
    scanf("%d", &data);
    while (low <= high)
    { mid = (low + high) / 2;
      if (arr[mid] == data
          { printf("%d found", data);
            break.
          }
      else if (arr[mid] > data)
            high = mid - 1;
      else
            low = mid + 1;
    }
     if( low > high)
    printf("%d not found in array", data);
    return 0;
}
```

(i) Sorting: Sorting means arranging the array element in ascending or descending order. We have several sorting mechanism available. The most commonly used, and easy to implement sorting algorithim is selection sort. The worst complexity of selection sort is O(n*n). In selection sort every element is compared to other element one by 1.

Step 1: Start
Step 2: Set i = 0
Step 3: repeat steps 4, 5, 11 for i < size-1.
Step 4: Set j = i+1
Step 5: repeat steps 6, 10 for j < size
Step 6: Check if arr[i] > arr[j] then run step 7, 8, 9
Step 7: Set arr[i] = arr[i] + arr[j].
Step 8: Set arr[j] = arr[i] - arr[j]
Step 9: Set arr[i] = arr[i] - arr[j]
Step 10: Set j = j+1.
Step 11: Set i = i+1.
Step 12: end.

```c
#include <stdio.h>
int main()
{   int arr[5] = { -1,-6,2,-3,0 };
    int i, j;
    for (i=0 , i<4 ; i++)
    { for (j = i+1 ; j < 5; j++)
        { if (arr[i] > arr[j])
            { arr[i] = arr[i] + arr[j],
              arr[j] = arr[i] - arr[j];
              arr[i] = arr[i] - arr[j];
            }
        }
    }
printf("After sorting array is");
for ( i=0; i<5; i++)
      printf("%d", arr[i]);
return 0;
```

vii) Splitting :- Splitting means dividing an array into multiple arrays acc. to some criteria. Split an array in two arrays, the first array stores positive value while second array store negative value.

```
#include <stdio.h>
    int main()
{   int arr[10] = { 10,-5,15,-6,14,-7,-8,-9,14,-74}
    int pos[10], neg[10], i, j, k;
for(i=0, j=0, k=0; i<10; i++)
      { if (arr[i] <0)
        { neg[j] = arr[i];
            j++;
        }
      else
        { pos[k] = arr[i];
            k++;
        }
    }
printf("\n positive no. are");
  for (i=0; i<k; i++)
        printf("%d", pos[i]);
printf(" negative values are :");
  for (i=0; i<j; i++)
        printf("%d", neg[i]);
  return 0;
```

Step 1 : Start
Step 2 : Set i = 0
Step 3 : Set j = 0
Step 4 : Set K = 0
Step 5 : repeat Steps 6, 7 for i < size.
Step 6 : check if arr[i] < 0
then
a) Set neg[j] = arr[i]
b) Set j = j+1
otherwise.
c) Set pos[K] = arr[i]
d) Set K = K+1
Step 7 : Set i = i+1
Step 8 : end.

viii) Merging : Merging means combining two or more than two arrays into a single array.

```
#include <stdio.h>
int main()
{ int pos[] = { 10,20,30,40,50};
  int neg[] = {-1,-2,-3,-4,-5,-6};
  int arr[11];
    int i, j;
for(i=j=0; j<5; j++, i++)
  { arr[i] = pos[j]; }
for (j=0; j<6; j++; i++)
  { arr[i] = neg[j]; }
printf("after merging the array is:");
  for(i=0; i<11; i++)
```

```
printf("%d", arr[i]);
return 0;
```

Step 1: Start
Step 2: Set i=0
Step 3: Set j=0
Step 4: repeat steps 5,6,7 for j< size1
Step 5: Set arr[i] = pos[j]
Step 6: Set i = i+1
Step 7: Set j = j+1
Step 8: Set j=0
Step 9: repeat steps 10,11,12 for j< size2
Step 10: Set arr[i] = neg[j]
Step 11: Set i=i+1
Step 12: Set j=j+1
Step 13: end.

Reversing:- Reversing means storing the element in reverse order. It means the last element will become first element and vice versa and so on.

```
#include<stdio.h>
int main()
{ int arr[5]={10,20,30,40,50}i,j;
for(i=0,j=4; i<j; i++,j--)
{   arr[i] = arr[i]+arr[j];
    arr[j] = arr[i]-arr[j];
    arr[i] = arr[i]-arr[j];
}
printf("after reversing the array");
for(i=0;i<5;i++)
    printf("%d", arr[i]);
return 0;
```

Step 1: Start
Step 2: Set i=0
Step 3: Set j = size-1
Step 4: repeat steps 5,6,7,8,9 for i<j
Step 5: Set arr[i] = arr[i] + arr[j]
Step 6: Set arr[j] = arr[i] - arr[j]
Step 7: Set arr[i] = arr[i] - arr[j]
Step 8: Set i = i+1
Step 9: Set j = j+1
Step 10: end.

* Stack:- Stack is also a linear data structure. A stack works on the concepts of last in first out. In a stack the data elements are inserted and removed from the same end called as top. If stack is empty then top = -1.

A stack has following two standard operations:-

a) Push: Push means inserting a value into the stack. When we push a value in the stack then top is increased by 1. When stack is full (top == size-1) and we try to push value into the stack then it is called as stack overflow condition.

b) Pop :- Pop means deleting value from the stack. When pop an item from stack then the top decreased by 1. if stack is empty (top == -1), and we try to pop value from stack then it is called as stack underflow.

we can implement a stack using array or linkedlist. if we implement stack using linked list then stack can not be overflow because of the dynamic memory allocation.

Implement stack using the array.

Algo. of PUSH.

Step 1: Start
Step 2: read data
Step 3: check if top = size-1.
then
a) write "stack overflow"
otherwise
b) Set top = top+1.
c) Set stack [top] = data.
Step 4: End.

Algo. of Pop.

Step 1: Start
Step 2: check if top = -1.
then
a) write "stack underflow"
otherwise
b) Set stack [top] = 0
c) Set top = top-1
Step 3: end.

Time complexity O(1).

* Types of expression :-

i) Infix expression :- In infix expression we write operator in between operands. Eg :- 10+5

ii) Prefix expression :- In prefix expression we write operator before the operands. it is also called as polish notation
eg :- +ab.

iii) Postfix expression :- In postfix expression we write operator after operands. it is called as reverse polish notation.
eg :- ab+.

In prefix and postfix expression we do not require the ()
to change the precedence. most of the modern compilers used prefix or the postfix notation.

* Infix to prefix. 1. A+B*C
eg :- A+C*BC) D = *BC
⇒ A+D
⇒ +AD
⇒ +A*BC.

① ()
② ^ or **
③ *, /, %.
④ +, -

```
#include <stdio.h>
#define size 5
int main()
{ int stack[size], choice, data, top=-1;
    char con;
  do
    { printf("\n1.push\n2.Pop\nenter yourchoice:")
      scanf("%d", &choice);
     switch(choice)
      { case 1:
          printf("enter data");
          scanf("%d", &data);
        if(top==size-1)
          printf("stack overflow");
       else
       { top++;
         stack[top]=data;
        printf("\n%d pushed at %d postion", data, top);
       }
    break;
 case 2:
        if(top==-1)
          printf("\n stack underflow");
       else
       { data=stack[top];
         stack[top]=0;
         printf("\n%d popped from %d postion",
                                    data, top);
          top--;
```

```
        break;
    }
    printf("\n do not you want to continue");
        con = getche();
    }
        while (con == 'y');
    return 0;
}
```

* Queue :- Queue is also a linear data structure. it works on concept of FIFO. it means that the element added first will be the element which is deleted first. The insertion and deletion operation are performed on two diff ends. The end on which item are added is called rear of Queue. The end from which item is deleted is called front of Queue. When Queue is empty then both front and rear are -1. We can perform following two operation

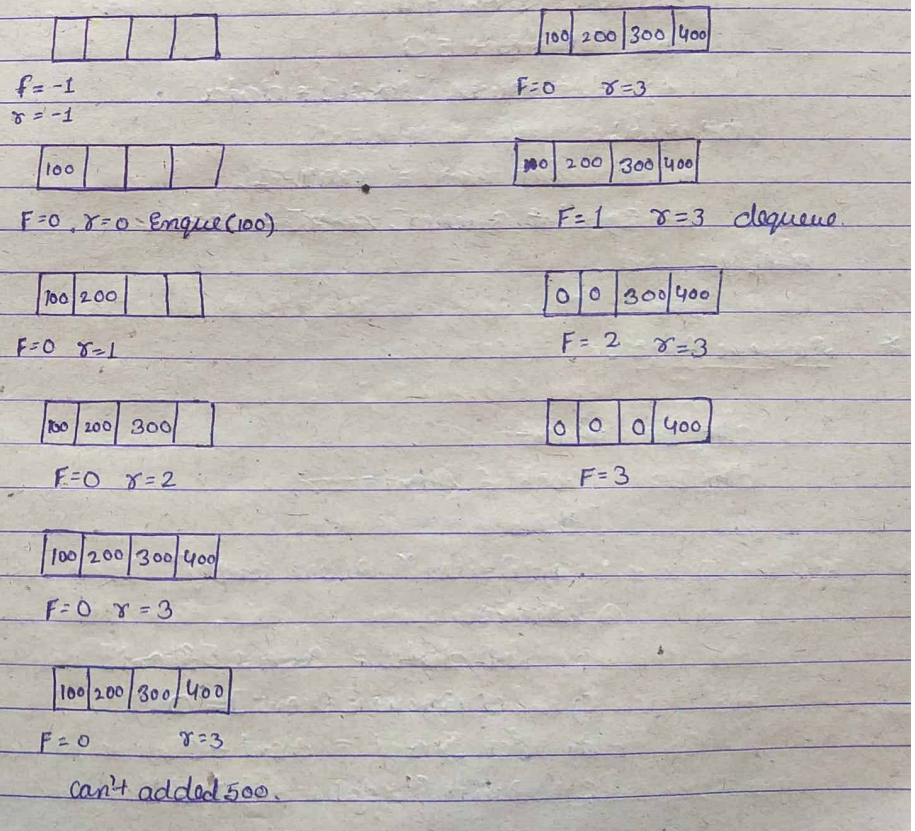
1. enque 2. dequeue

1. Enque :- Enque means inserting value in the Queue When we insert first value in the Queue then both front and rear are set to 0. Then element is added at rear index while adding an other value first we check that rear is = size - 1. then Queue is exhausted. and item can't be added. if the queue is not full then rear is increased by 1 and data is stored on rear.

* Dequeue :- Dequeue means deleting value from the Queue. When we delete an item from Queue then front increased by one.

if front = -1 then it means the Queue is empty and deletion can't be perform on empty Queue.

after deleting an item if front crosses the rear then it means that all the items are removed from Queue and Queue is empty now. So now we reset Both front and rear to -1.



* Enque Algo.

Step 1: Start
Step 2: read data
Step 3: Check if rear = size-1
then
Write "Queue overflow"
Otherwise
check if front = -1
then
a) Set front = 0
b) Set rear = 0
c) Set Queue [rear] = data
Otherwise
d) Set rear = rear+1.
e) Set Queue [rear] = data.
Step 4: end.

* Dequeue Algo.

Step 1: Start
Step 2: Check if front = -1.
then
a) Write " Queue empty"
Otherwise
b) Set data = Queue [front]
c) Set Queue [front] = 0
d) Set front = front + 1
e) Write " data deleted"
f) check if front > rear.
then
g) Set front = rear = -1.
Step 3: end.

```
# include <stdio.h>
# define size 5.
{   int queue [size];
    int front = -1 , rear = -1 , choice , data;
     char con;
do
{ printf ("\n 1. Enqueue \n2. Dequeue \n Enter your choice")
 scanf ("%d", &choice);
  switch (choice)
   { case 1:
       printf (" enter data to be added : ");
       scanf (" %d", & data);
   if ( rear == size-1 )
     printf (" Queue is full %d can't be added", data)
else
  { if(front == -1)
     { front = rear = 0;
      queue [rear] = data;
      printf ("\n %d added at %d position", data
                                    , rear);
     }
      else
    { rear ++;
     queue [rear] = data;
```

```
        printf("\n %d added to %d position", data, rear);
    }
}
    break;
    case 2:
        if (front == -1)
            printf("\n queue is empty");
        else
    { data = queue [front];
      printf("\n %d deleted from %d position", data,
                                              front);
        front++;
      if ( front > rear)
          front = rear = -1;
    }
    break;
}
    printf("\n Do you want to continue");
      con = getche();
    } while (con == 'y');
    return 0;
}.
```

* Circular Queue :- In a normal Queue if we delete some item once a queue is full, then rear = Size - 1. but Queue have empty space, still we can't add items in the Queue.

To Solve this problem circular Queue is implement In a circular Queue if rear = Size - 1 then and there is some space in the Queue then rear is restarted to 0 While adding new items. When a circular Queue is full then front is zero and rear = Size - 1. Or rear + 1 = front.

While deleting an item from circular Queue if front = rear then it means that we have deleted all the values from circular Queue and Queue is empty now. So we reset Both front and rear to -1. if front = Size While deleting the item then front restart from zero.

```
# include <Stdio.h>
# define Size 5.
int queue [Size], front = -1, rear = -1;
 void enqueue (int);
 void dequeue ();
 int main()
    { int ch, data;
  do
  { system("cls");
  printf("\n 1. Enqueue \n 2. Dequeue \n enter choice")
   scanf("%d", & ch);
   switch (ch)
     { case 1:
         printf("enter data to be added");
         scanf("%d", & data);
       enqueue (data);
      break;
      case 2:
          dequeue ();
```

```
        break;
    }
    printf("do you want to continue");
printf("\n press 1 to continue \n press any key to exit");
    scanf("%d", &ch);
  } while (ch==1);
    return 0;
}
void enqueue (int data)
    if((front==0 && rear==size-1) ||
                (rear+1==front))
     printf("Queue fulled");
  else
  { if (front==-1)
        { front = rear=0;
          queue[rear] = data;
        }
else if (front!=0 && rear==size-1)
      { rear = 0;
        queue[rear] = data;
      }
   else
      { rear = rear+1;
        queue[rear] = data;
      }
   printf("\n %d add to queue", data);
  }
}
```

```
void dequeue()
    if(front==-1)
      printf("Queue empty");
    else
  { printf("\n %d delete from equeue", queue[front]);
    queue[front] = 0;
    if(front==rear)
    { front=-1;
      rear=-1;
      return;
    }
     front++;
      if (front==size)
      { front=0;
      }
    }
  }
```

* Algo :- for using function (Enqueue(data))

Step 1: Start

Step 2: read data (x /* this step cancel in function deal*/)

Step 3: Check if f=0 and r=size-1 || rear+1 = front
then
a, Write "Queue full"
Otherwise
Check if f = -1
then
b, Set f = 0
c, Set r = 0
d, Set queue[r] = data.

Otherwise check if f != 0 and r == size - 1
then
e) Set r = 0
f) Set queue[r] = data
Otherwise
g) Set r = r+1.
h) queue[r] = data.
Step 4 : end.

Dequeue
Step 1: Start
Step 2: Check if f = -1
then
a) Write "Queue empty"
Otherwise
b) Set queue[f] = 0
Otherwise check if f == r
then
c) Set f = -1
d) Set r = -1
e) return
f) Set f = f+1
Check if f == size
then.
g) f = 0
Step 3: end.

double ended queue :- In a double ended queue the item can be added or deleted from both ends for of the queue. Generally double ended queue is implemented using a linked list we have following two types of double ended queue.
1. input restricted double ended queue.
2. output restricted double ended queue.

1. input restricted :- In input restricted double ended queue the items can be delete from both end but item can be added from rear only.

2. output restricted :- In output restricted double ended queue the item can be inserted from both end but can be deleted from front only.

* Priority queue :- In this queue the higher priority jobs are done first then the lower priority jobs. it means every element also have its priority and priority decides that which element will be process first. Mostly it is used by Shedulers giving.

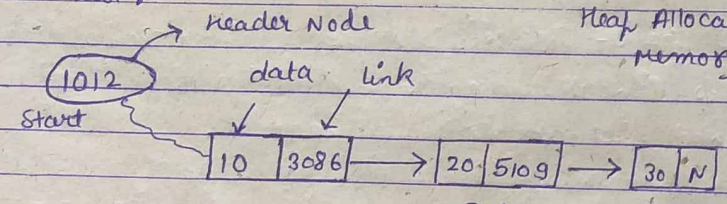
* Linked list :- Linked list is a linear data structure it is linear using the pointers the elements of linked list are called as nodes.
Linked list is mostly created dynamically using the dynamic memory allocation so number of elements in a linked list can be changed it means that the size of linked list can

grow or shrink at programme execution time. we have following four types of linked list.

i) Singly linked list
ii) Circular linked list / Singly circular linked list.
iii) Doubly linked list
iv) Doubly circular linked list

i) Singly linked list :- In a Singly linked list every node have two parts a) data b) link/next.
a) data :- it stores the actual data or information.
b) link :- it is actually a pointer which stores the address of next node.

In a Singly linked list the address of 1st node is stored in a special pointer called as start pointer if a Singly linked list does not have any node then start pointer stores NULL. The data part of node stores data and the next part stores adderess of its successor (next node). the next part of last node stores NULL. which indicates end of linked list. the following diagram shows Singly linked list.



A node of linked list is created using a self referencial structure. The following structure can be used for Singly linked list.

```
Struct linked list
{   int data
    Struct linkedlist * Next;
};
```

We can performe following operation on
i) create ii) display from begining
iii) display from last.
iv) insertion at begning vi) Delation at begning
v) insertion at last vii) delation at last
viii) insertion at middle ix) deletion at middle
x) Searching xi) Sorting.
xii) count.

Note :- A Singly linked can be traversed in a single direction.

i) create :- create means creating a linked list of n nodes.
Step 1 : Start
Step 2 : read n
Step 3 : Set i=1
Step 4 : repeat steps 5,6,7,8,9 for i<=n
Step 5 : create node
Step 6 : read node → data
Step 7 : read node → next = NULL
Step 8 : check if start = NULL
a) Set start = node

Otherwise

b) Set temp = start

c) repeat step d) Write temp → next != NULL
temp = temp → next.

d) temp → next = node.

Step 9: Set i = i+1.

Step 10: end.

ii) traverse from begning :- means displaying data of all the nodes from begning to end

Step 1: start

Step 2: if start = null
a) "List empty" write
otherwise.

b) Set temp = start

c) repeate steps d, e while temp != null

d) Write temp → data

e) Set temp = temp → next.

Step 3: End.

```
# include <stdio.h>
/* # include <stdlib.h> */
struct linked list
    { int data;
      struct linked list * next;
    };
/* void print reverse (struct linked list * ); */
struct linked list * start = NULL;
int main ()
{ struct linked list * node, * temp.
    int choice, i, n, con, pos;
do.
  { system ("cls");
  printf ("\n1. create");
printf ("\n2. Traverse from begning");
printf ("\n3. Traverse from last");
printf ("\n4. Inesertion at begining");
printf ("\n5. Inlsertion at last");
printf ("\n6. Deletion from begining");
printf ("\n7. Deletion from last");
printf ("\n8. Inesertion at middle");
printf ("\n9. Deletion from middle");
printf ("\n10. Searching");
printf ("\n11. Sorting");
printf ("\n12. Cant In Select the operation above:");
scanf ("%d", &choice);
switch (choice)
{ case 1:
  printf ("\n How many nodes you want to create.");
  scanf ("%d", &n);
  for (i=1; i<=n; i++).
{ node = (struct linked list *) malloc (sizeof
                    (struct linked list));
  printf ("enter data :");
  scanf ("%d", & node → data);
    node → next = NULL;
```

```
if (start == NULL);
        start = node;
else
  { temp = start;
    while (temp -> next != NULL)
      temp = temp -> next;
    temp -> next = node;
  }
}
break;
  case 2:
      if (start == NULL)
      printf ("\n linked list empty");
    else
    { printf ("\n linked list is \n");
      temp = start;
    while (temp != NULL)
  { printf ("%d\t", temp -> data);
      temp = temp -> next;
    }
  }
  break;
case 3:    printReverse (start);
        break;
    default:
      printf (" Option 1 se 12 ke bheech input dein");
    }
printf ("press 1 to continue \n any key to exit");
  scanf ("%d", &con);
} while (con == 1);
return 0;
void printreverse (struct linkedlist * temp)
  {  if (temp -> next != NULL)
       printReverse (temp -> next);
       print ("%d", temp -> data);
case 4:
  node = (struct linked list *) malloc (sizeof (struct linked
    printf (" enter data: ");
    scanf ("%d", & node -> start) data);
      node -> next = start;
        start = node;
  break;

case 5:
    node = (struct linkedlist *) malloc (sizeof (struct linkedlist
    printf (" enter data: ");
    scanf ("%d", & node -> data);
    node -> next = NULL;
      if (start == NULL)
            start = Node;
      else
        { temp = start;
          while (temp -> next != NULL)
              temp = temp -> next
          temp -> next = node;
        }
    break;
```

```
case 6:
        if (start == NULL)
          printf ("list empty");
        else
      { if ( start -> next == NULL)
          { printf("\n %d deleted", start -> data);
              free ( start);
                start = NULL;
            }
        else
            { temp = start;
              start = start -> next;
    printf ("\n %d deleted", temp -> data);
            free (temp);
        }
      }
    break;
Case 7:
        if (start == NULL)
          printf ("list empty");
        else
    { if ( start -> next == NULL)
    { printf("\n %d deleted", start -> data);
        free ( start);
          start = NULL;
      }
else
    {   node = start;
      while(node -> next != NULL);
    { temp = node;
      node = node -> next;
    }
      temp -> next = NULL;
  printf ("\n %d deleted", node -> data);
        free(node);
      }
  }
  break;

case 10:
  printf("enter position where you want to insert the node");
   scanf ("%d", &pos);
   for(i=1; i<pos; i++)
   if (start == NULL)
  { printf("empty linked list, searching can't performed");
    scanf("%d", &data);
     temp = start;
    while (temp != NULL)
      { if (temp -> data == data)
          { printf("\n %d found", data);
              break;
          }
        temp = temp -> next;
    }
      if (temp == NULL)
          printf ("\n %d not found", data);
    }
```

```
break;
case 11:
    if (start == NULL)
printf("Linked list empty, Searching can't perform);
  else
{ for (temp = start; temp -> next != NULL; temp = temp -> next)
  { for (q = temp -> next; q != NULL; q = q -> next)
    { if (temp -> data > q -> data)
      { n = temp -> data;
        temp -> data = q -> data;
        q -> data = n;
      }
    }
  }
}
break;

case 12:
    n = 0;
    temp = start;
  while (temp != NULL)
    { n++;
      temp = temp -> next;
    }
printf("\n list have %d nodes", n);
  break;
}
```

Insertion at begining Algo.

Step 1: Start
Step 2: create a new node
Step 3: read node → data
Step 4: Set node → next = start
Step 5: start = node
Step 6: end.

insertion at end algo.

Step 1: Start
Step 2: create a new node
Step 3: read node → data
Step 4: Set node → next = null
Step 5: if start = null
  then a) Set start = node
  Otherwise
    i) Set temp = start
    b) repeat step (while temp → next != null)
    c) Set temp = temp → next
    d) Set temp → next = node
Step 6: end.

Deletion from begining algo.

Step 1: Start
Step 2: if start = null
  then a) Write "list empty".

otherwise
a) if (start → next == null)
b) free (start)
c) set start = NULL;

else otherwise
d) temp = start
e) set start = start → next
f) free(temp)

Step 3: end.

Deletion at last algo:-

Step 1: start
Step 2: check if (start = null)
a) write "list empty"
otherwise
b) if start → next == null
then
c) free (start)
d) set start = null
otherwise
e) node = start
f) while (node → next != null)
g) set temp = node
h) set node = node → next
i) set temp → next = null
j) free (node)
Step 3: end.

algo. of count
Step 1: start
Step 2: set n=0
Step 3: set temp = start
Step 4: while (temp != null)
then a) n++
b) set temp = temp → next;
c) write "list have :-d nodes :" or n.
Step 5: end

algo. of Searching
Step 1: start
Step 2: check if start = null.
Step 3: read data
Step 4: set temp = start
Step 5: while temp != null
then a) check if (temp → data == dat
b) write 'data found.'
Step 6: set temp = temp → next
Step 7: check if (temp == null)
c) then write "data not found"
Step 8: end.

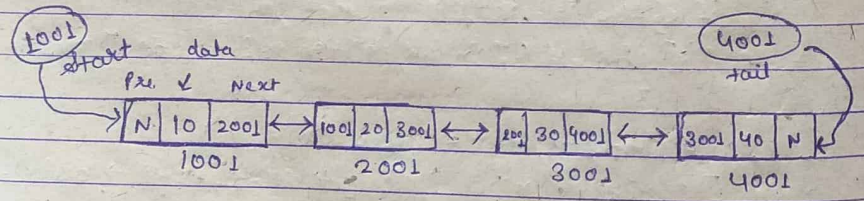
```
case 8:
    printf ("enter position where you want to insert node:");
    scanf ("%d", &n);
node = (struct linkedlist*) malloc (sizeof (struct linkedlist))
    print ("enter data:");
    scanf ("%d", &node->data);
    node -> next = NULL;
      temp = start;
      if (n == 1)
    {   node -> next = temp;
        start = node;
      }
     else
        { for (i=1; i<n-1; i++)
            { if (temp == NULL)
                  break;
              temp = temp -> next;
            }
       if (temp == NULL)
         printf ("\n%d position does not exist", n);
     else
       { if (temp -> next == NULL)
              temp -> next = node;
     else
       { node -> next = temp -> next;
            temp -> next = node;
         }
       }
    }   break;
```

```
case 9:
    printf ("\n enter position from where you want to delete node");
    scanf ("%d", &n);
      if (start == NULL)
          printf ("list empty!!!");
     else
       { temp = start;
         if (n == 1)
        { printf ("\n%d is going to be delete", start->data)
          temp = start;
          start = start -> next;
           free (temp);
        }
         else
           { for (i=1; i<n; i++)
              { q = temp;
                if (temp == NULL)
                   break;
              temp = temp -> next;
           }
       if (temp == NULL)
         { printf ("\n%d position does not exist", n);
         }
     else if (temp -> next == NULL && i == n)
       { printf ("\n%d is going to delete", temp -> data);
          free (temp);
          q -> next = NULL;
       }
```

```
else
{ printf("\n%d is going to delete", temp->data);
    q->next = temp->next;
      free(temp);
    }
  }
} Break;
```

* **Doubly linked list** :- In a doubly linked list every node have three parts.
  i) data/info : used to store the data
  ii) next : It is a pointer which stores address of next node.
  iii) prev : It is a pointer which stores address of previous node.

The address of 1st node of doubly linked list is store in a pointer called start. Similarly address of last node is stored in a special pointer called tail. the following diagram shows doubly linked list having 4 nodes.



the next part of last node and previous part of 1st node stores null. Which indicates end of linkedlist. We can performe following operations on doubly linked list.

```
# include <stdio.h>
struct Doubly linkedlist
{ struct Doubly linkedlist * prev;
  int data;
  struct Doublylinkedlist * next;
};
typedef struct Doublylinkedlist linked list;
  linked list * start = NULL, * tail = NULL;
  int main()
{ Linkedlist * node, * temp, *q;
    int count, n, i, choice, con;
do
  { system("cls");
printf("\n1. create \n2. Diffay from begining \n3.
    Diffay from last" 4. Insert at begining \n5.
  Insert at last \n6. Insert at middle"
  "\n7. Delete from begining \n8. Delete from
  last \n9. Delete from middle"
\n10. Count \n11. Search \n12. Sort \n enter the choice"
scanf("%d", &choice);
  switch(choice)
  { case 1:
      printf("How many nodes you want to create")
scanf("%d", &n);
  for(i=1; i<=n; i++)
{ node = (linkedlist*) malloc(sizeof(linkedlist));
  printf("\n enter data");
scanf("%d", & node->data);
  node->next = node->prev = NULL;
```

```
if (start == NULL)
{ start = tail = node; }
else
  { tail → next = node
    node → prev = tail;
    tail = node;
  }
}
break;
case 2:
    if(start == NULL)
    printf("\n list Khali hai");
else
{ printf("\n list contain (start to End) \n");
    temp = start;
  while (temp != NULL)
{ printf("%d", temp → data);
    temp = temp → next
  }
}
  Break;
case 3:
    if (tail == NULL)
      printf("\n list Khali hai");
  else
{ printf("\n list contain (end to start) \n");
    temp = tail;
  while (temp != NULL)
  { printf("%d", temp → data);
    temp = temp → prev;
  }
}
  break;
} /* at last of case */ or programme */
printf("\n press 1 to continue \n press any key to exit \n");
    scanf("%d", &con);
} while (con == 1);
  return 0;
}
case 4:
    node = (linked list *) malloc (sizeof (linked list));
    printf("enter data :");
    scanf("%d", &node → data);
      node → next = start;
      node → prev = NULL;
  if (start == NULL)
  { tail = start = node;
  }
  else
    { start → prev = node;
      start = node;
    }
  break;
case 5 :
    node = (linked list *) malloc (sizeof (linked list));
  printf("\n enter data :");
  scanf("%d", &node → data);
      node → prev = tail;
```

```
        node -> next = NULL;
        if (tail == NULL)
          { start = tail = node;
          }
        else
          { tail -> next = node;
            tail = node;
          }
        break;
    case 7:
        if (start == NULL)
    { printf("\n linked list is empty"); }
      else if (start == tail)
    { printf("\n %d is going to delete", start -> data);
        free (start);
      start = tail = NULL;
    }
      else
        { temp = start;
          start = start -> next;
          start -> prev = NULL;
    printf("\n %d is going to be deleted", temp -> data);
          free (temp);
        }
      Break;
    case 8:
        if (start == NULL)
    { printf("\n %d is linked list empty");
      }
    else if (start == tail)
      { printf("\n %d is going to be deleted", start -> data);
        free (start);
        start = tail = NULL;
      }
      else
        { temp = tail
          tail = tail -> prev;
          tail -> next = NULL;
    printf("\n %d is going to be deleted", temp -> data);
        free (temp);
      }
    break;
    case 10:
        count = 0;
        temp = start;
        while (temp != NULL)
          { count ++;
            temp = temp -> next;
          }
    printf("\n list contain %d nodes", count);
      break;
    case 11:
          if (start == NULL)
          printf("Nothing to search");
      else
          { printf("Enter Value to be searched:");
            scanf("%d", &n);
            temp = start;
```

```
while (temp1 != NULL)
{   if (temp -> data == n)
    {  printf("\n %d found", n);
       break;
    }
  temp = temp -> next;
}
  if (temp == NULL)
    printf("\n %d not found in list", n);
}
  Break;
```

Case 12:

```
    if (start == NULL)
printf("Linked list empty, Searching/Sorting can't be performed");
else
{ for (temp = start; temp -> next != NULL; temp = temp -> Next)
  { for (q = temp -> next; q != NULL; q = q -> next)
    { if (temp -> data > q -> data)
      {  n = temp -> data;
         temp -> data = q -> data;
         q -> data = n;
      }
    }
  }
}
Breake;
```

case 6:

```
printf("enter position where you want to insert node");
&scanf("%d", &n);
node = (linked list *) malloc(sizeof(linked list));
printf("enter data");
&scanf("%d", &node -> data);
  node -> next = NULL;
  node -> prev = NULL;
   temp -> &start
  if (n == 1)
  { node -> next = temp;
      &start -> prev = node;
       &start = node;
  }
else
   { for (i=1; i<n-1; i++).
      { (if (temp = NULL)
       printf("\n%d Position does not exist", n)
  else
  { if (temp -> next == NULL)
       { if  temp -> next = node);
             node -> prev = tail;
              tail = node;
         }
    else.
      { node -> prev = temp
       node -> next = temp -> next;
        temp -> next -> prev = node;
         temp -> next = node;
  ] }
```

```
        break;
case 3:
        printf("\n enter position from where you want to delete node");
        scanf("%d", &n);
    if (start == NULL)
        printf("list empty !!!");
    else
    { temp = start;
      if(n==1)
    { printf("\n %d is going to be deleted", start->data);
        if(start = tail)
        { free(start);
          start = NULL;
          tail = NULL;
        }
    else
    { temp = start;
      start = start->next;
      start->prev = NULL;
      free(temp);
    }
    else
      for(i=1; i<n; i++)
      { q = temp;
        if(temp == NULL)
            break;
        temp = temp->next;
      }
      if (temp == NULL)
        { printf("\n %d position does not exist", n);
        }
      else if (temp->next == NULL && i == n)
      { printf("\n %d is going to be deleted", temp->data
            free(temp);
            tail = q;
            q->next = NULL;
        }
        else
      { printf("\n %d is going to be delete", temp->data);
            q->next = temp->next;
            temp->next->prev = q;
            free(temp);
        }
      }
    }
    break;

printf("\n Press 1 to continue \n Press any Key to exit\n")
    scanf("%d", &con);
} while(con == 1);
    return 0;
}
```

Algo. case 1:

Step 1: Start
Step 2: create a new node
Step 3: read node → data
Step 4: Set node → next = node → prev = NULL
Step 5: Check if (start == NULL)
then a) start = tail = node;
otherwise
b) tail → next = node
c) node → prev = tail;
d) tail = node.
Step 6: end

case 2: Step 1: Start
Step 2: Check if (start == NULL)
Step 3: Write 'list empty'
otherwise
a) Write 'list contain'
b) Set temp = start
c) while (temp != NULL)
then write temp → data
d) temp = temp → next
Step 4: end.

case 3: Step 1: Start
Step 2: check if (tail == NULL)
a) Write 'list empty'
otherwise
b) Write 'list contain'
c) temp = tail.
d) while (temp != NULL)
then write temp → data
e) temp = temp → prev.
Step 3: end

case 4: Step 1: Start
Step 2: create a node. new.
Step 3: read node → data
Step 4: Set node → next = start,
Step 5: Set node → prev = NULL;
Step 6: check if (start == NULL)
then a) tail = start = node;
otherwise
b) Set start → prev = node
c) Set start = node
Step 7: end.

case 5: Step 1: start
Step 2: create a new node.
Step 3: read node → data
Step 4: Set node → prev = tail
Step 5: Set node → next = NULL
Step 6: check if (tail == NULL)
then a) Set start = tail = node.
otherwise
b) Set tail → next = node
c) Set tail = node;
Step 7: end.

case 7: Step 1 : Start
Step 2 : Check if (Start == NULL)
then write 'List empty'
Otherwise
Step 3: Check if (Start == tail)
then write 'deleted' Start → data
a) then Set free (Start)
b) Set Start = tail = NULL
Otherwise.
c) Set temp = Start
d) Start = Start → next;
e) Start → prev = NULL
then write 'deleted' temp → data
f) Set free (temp)
Step 4: end.

case 8: Step 1 : Start
Step 2: Check if (Start == NULL)
then write 'List empty'
Otherwise
Step 3: Check if (Start == tail)
then write 'deleted' Start → data
a) Set free (Start)
b) Set Start = tail = NULL,
Otherwise.
c) Set temp = tail
d) tail = tail → ~~temp~~ prev;
e) Set tail → next = null
then write 'delete' temp → data
Step 4: free (temp)
Step 5: end.

case 10. Step 1 : Start
Step 2: Set count = 0
Step 3 : Set temp = Start
Step 4: While (temp != NULL)
a) then count ++
b) Set temp = temp → next.
Step 5: Write 'contain nodes' count.
Step 6: end.

case 11: Step 1 : Start
Step 2 : Check if (Start == NULL)
then write 'Nothing to Search'
Otherwise
a) Read n.
b) then Set temp = Start.
Step 3: c) While (temp != NULL)
Step 4. the Check if (temp → data. == n)
d) then Set temp = temp → next.
Step 5: Check if (temp == NULL)
write 'not found in list'
Step 6: end.

Case 12. Step 1: Start
Step 2: Check if (Start == NULL)
then write 'Sorting can't preforme'
Otherwise

Step 3: for repeat (temp = start; temp → next != NULL;
temp = temp → next).

Step 4: repeat for (q = temp → next; q != null; q =
q → next)

Step 5: check if (temp → data > q → data)
then a) Set n = temp → data
b) temp → data = q → data
c) q → data = n,

Step 6: end.

case 6: Step 1: Start.
Step 2: read n
Step 3: create a new node
Step 4: read node → data
Step 5: a) Set node → next = NULL
b) node → prev = NULL
c) temp → start
Step 6: check if (n == 1)
d) node → ~~next~~ = temp
e) start → prev = node
f) start = node
Otherwise
Step 7: repeat for (i = 1; i < n-1; i++
g) then check if temp = NULL
write ('position not exist')
Otherwise
Step 8: check if temp → next == NULL
h) temp → next = node
i) Set node → prev = tail
j) Set tail = node;
Otherwise
k) node → prev = temp
Step 8: l) node → next = temp → next
m) Set temp → next → prev = node.
n) Set temp → next = node.
Step 9: end.

case 9: Step 1: Start.
Step 2: read n
Step 3: check if (start == NULL)
then write ('list empty')
otherwise
a) Set temp = start
b) check if (n == 1)
then write ('deleted') start → data.
c) check if (start = tail)
1. Set free (start)
2. start = NULL
3. tail = NULL.
Otherwise
d) Set temp = start;
e) start = start → next
f) start → prev = NULL;
g) free (temp)
Otherwise
Step 4: repeat for (i = 1; i < n; i++
h) q = temp
(i) check if (temp == NULL)

Set temf = temf → next.

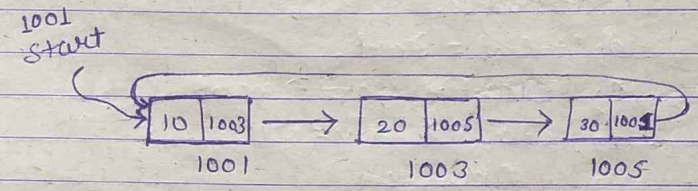
Step 5. Check if (temf == NULL)
write "Position not exist"
Otherwise

Step 6: check if temf → next == NULL && i == n
then write "deleted" temf → data
j) Set free (temf)
k) tail = q
l) q → next = NULL
Otherwise.

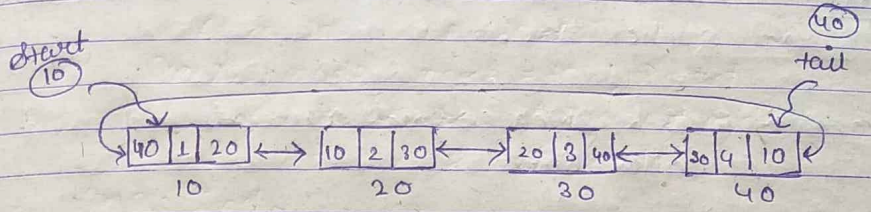
Step 7: write "deleted" temf → data
1. Set q → next = temf → next
2. temf → next → prev = q.
3. free temf.

Step 8: End.

* Single circural linked list :- In singly circular linked list the next part of last node store the address of 1 node instead of null. it means last node is directly connected to first node.



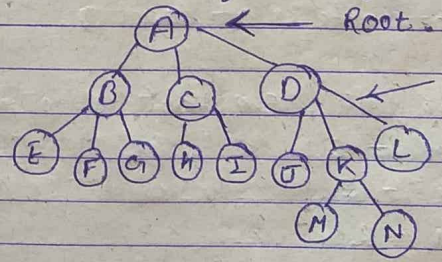
* doubly circular linked list :- it is same as doubly linked list except the next part of last node stores address of 1 node and previous part of 1 node store address last node.



* implementation of Stack using linked list :- When we implement a Stack using linkedlist then stack overflow situation never occurs.
. implement Stack use insertion at last and deletion at last.

* implementation of Queue using linked list :- When we implement a Queue using a linkedlist Queue is never exhausted (never full). To implement Queue we use insertion at last and deletion from begning.

* Tree :- Tree is a non linear data structure it stores data in hierarchical formate a tree looks like the following.



B, C, D are sibling
E, F, G are sibling
H, I are sibling

Tree Terminologies :-

1) **Root :-** A Node from which the tree starts is called root of the tree.

2) **Branch :-** A link which connects a parents and child is called Branch.

3.) **Sibling :-** Children of same parent are called Siblings.

4.) **Ancestor :-** A Node P is an Ancestor of node Q. if there exist a path from a root node to Q and P appears on that path.
A, D, K are ancestor of N. K is parent of N.

5) **Descendant :-** N is Descendant of K

6) **Leaf :-** Node which does not have any child is called leaf node.
eg :- E, F, G, H, I, J, L, M, N.

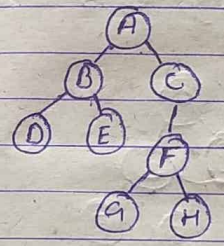
Level :- A set of all nodes at a given depth is called level. Root node is at level 0.

Depth :- Length of the path from root to the node
eg :- Depth of H - 2.
Depth of N - 3
" " M - 2.

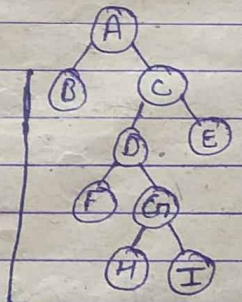
9) **Height :-** Length of the path from Node to the deepest Node. Height of tree is length of path from root node to the deepest node.
eg :- Height of D - 2.
" C - 1
" A - 3

10) **Size :-** Size of a node means the number of Nodes connected to it. + 1.
eg :- Size of D = 6
" A = 14

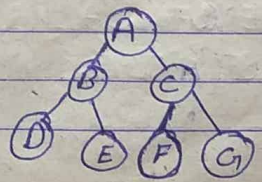
* **Binary tree :-** A tree in which every node can have atmost two children is called Binary tree.



* **Strictly Binary tree :-**
A Binary tree in which any non leaf node can have exactly two children is called Strict Binary tree.

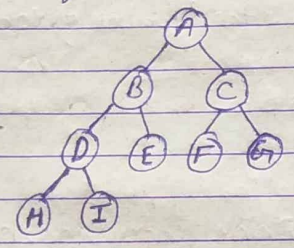


* **full binary tree :-**
A Binary tree in which all non leaf nodes have exactly two children all the leaf nodes are at same level.



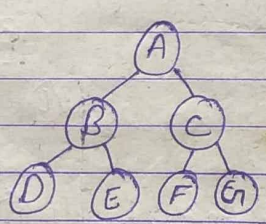
a full binary tree is also called proper binary tree.

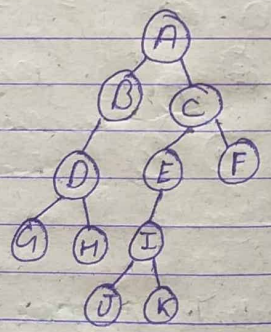
* complete binary tree :- A Binary tree in which every non leaf node have exactly two children and all non leaf node appears on the last level or second last level and level must be filled from left to right.



* Tree traversals :- We can traverse a binary tree in following manner.

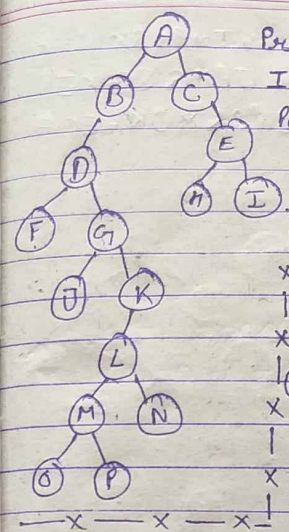
1) Preorder traversal (R-L-R)
2) Postorder traversal (L-R-P)
3) Inorder traversal (L-P-R)



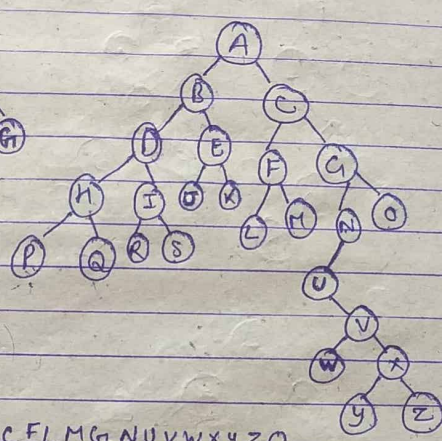


Preorder :- ABDECFG
Postorder :- DEBFGCA
Inorder :- DBEAFCG.

ABDGHCEIJKF
GHDBJKIEFCA
GDHBAJIKECF.

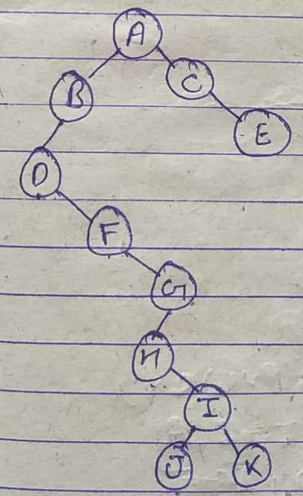


Pre :- ABDFGJKLMOPNCEHI
IN :- FDJGOMPLNKBACHEI
Post :- FJOPMNLKGDBHIECA



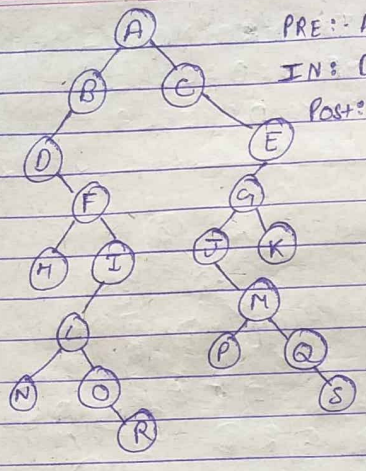
Pre :- ABDHPQIRSEJKCFLMGNUVWXYZO
IN :- PHQDRISBJEKALFMCUWVYXZNGO
Post :- PQHRSIDJKEBLMFWYZXVUNOGCA

(2)



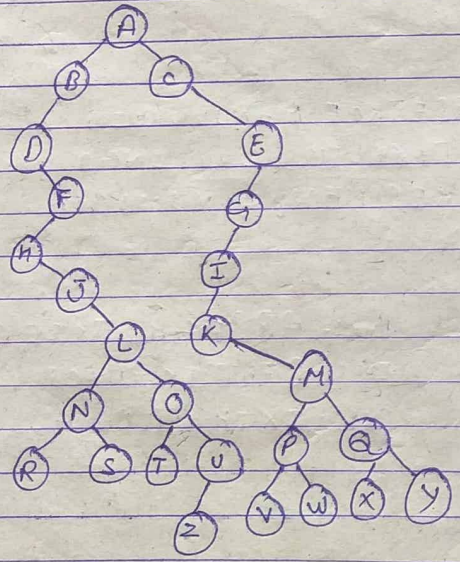
Pre :- ABDFGHIJKCE
IN :- DFHJIKGBACE
Post :- JKIHGFDBECA.

③



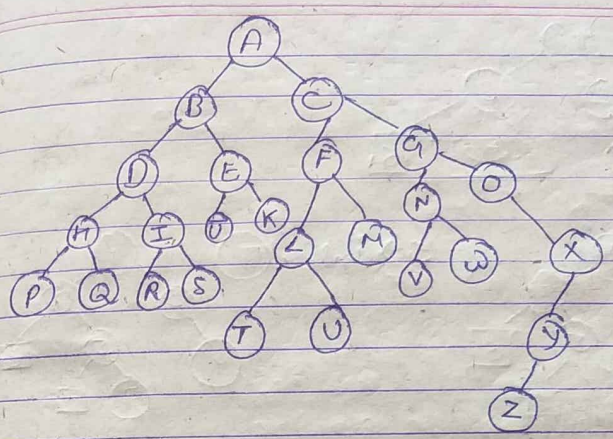
PRE :- A B D F H I L N O R C E G J M P Q S K
IN : D M F N L O R I B A C J P M Q S G K E
Post : H N R O L I F D B P S Q M J K G E C A

④



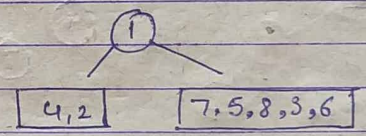
Pre :- A B D F H J L N R S O T U Z C E G I K M P V W Q X Y
Post :- R S N T Z U O L J H F D B V W P X Y Q M K I G E C A
IN :- D H J R N S L T O Z U F B A C K V P W M X Q Y I G E.

Post : P Q H R S I D J K E B F U L M F V W N Z y X O G C A.
IN : P H Q D R I S B J E K A T L U F M C V N W G O Z y x.

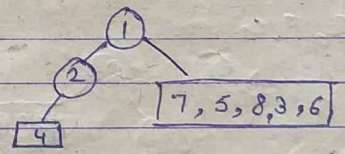


ques Imorder :- 4, 2, 1, 7, 5, 8, 3, 6
Preorder :- 1, 2, 4, 3, 5, 7, 8, 6

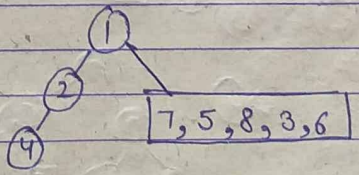
Insert ①



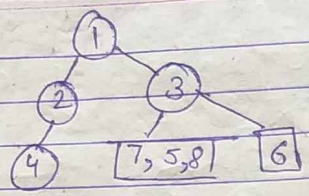
Insert ②



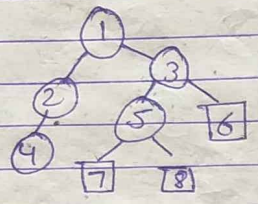
Insert ④



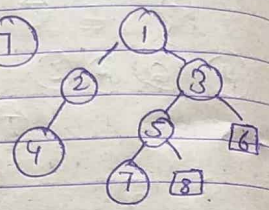
3) Insert (3)



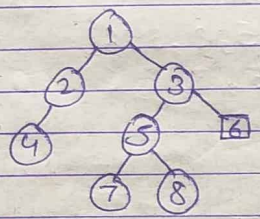
4) insert (5)



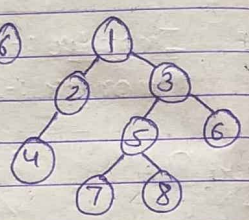
5) insert (7)



6) Insert (8)

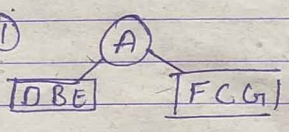


7) Insert (6)

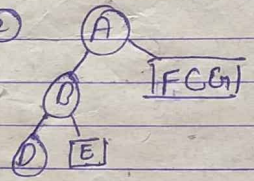


ques In : D, B, E, A, F, C, G.
Pre : A, B, D, E, C, F, G

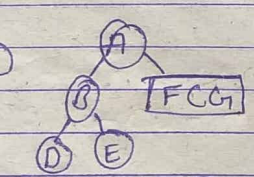
1)



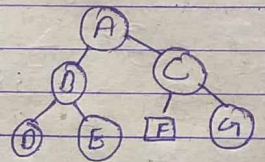
2)



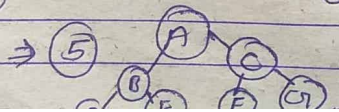
3)



4)

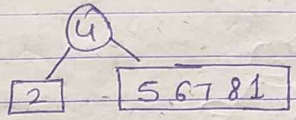


5)

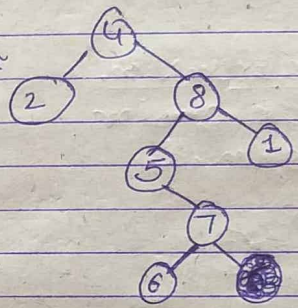


ques In : 2 4 5 6 7 8 1
Pre : 4 2 8 5 7 6 1

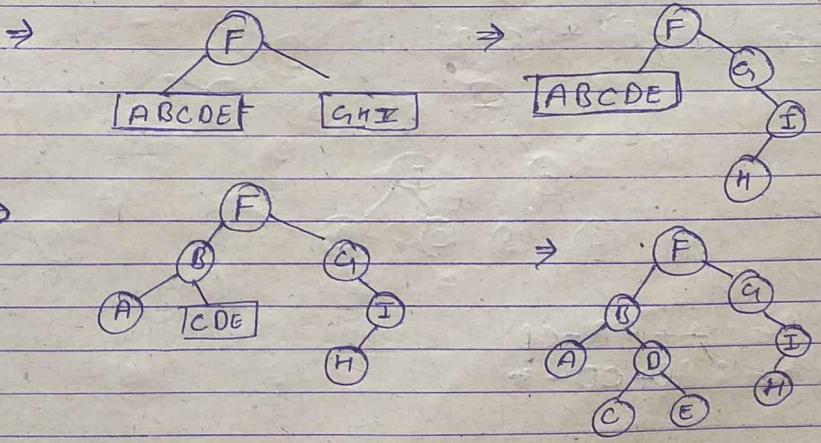
1) insert (4)



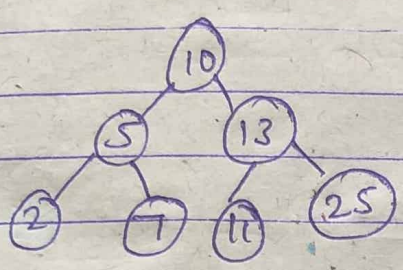
2) insert (2)
⇒ Next step Indicate



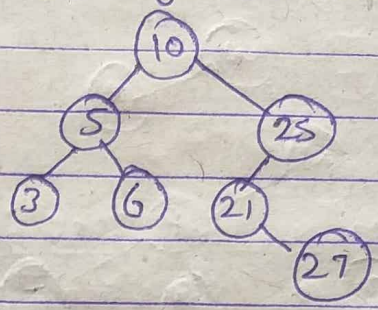
ques Post : A C E D B H I G F
IN : A B C D E F G H I



* BST (Binary Search tree) :- In Binary Search tree left sub tree contains Smaller value then the root and Right sub tree contains larger values then the root this property must be satisfied by every node in the tree. Inorder always in Sorted Assending order.



correct

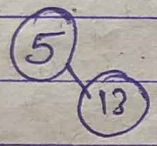


not correct

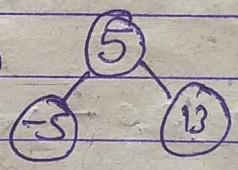
ques 5, 13, -5, 17, 8, 9, 21, 2, 0, -3.

sol.

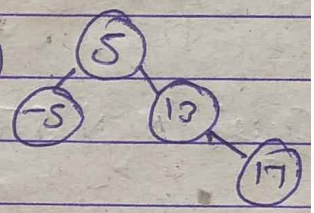
1)



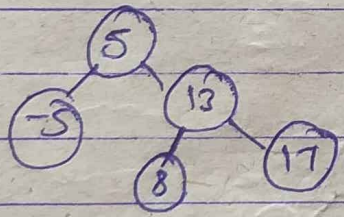
2)



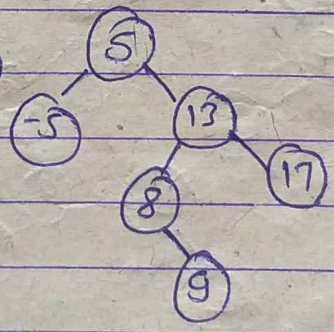
3)



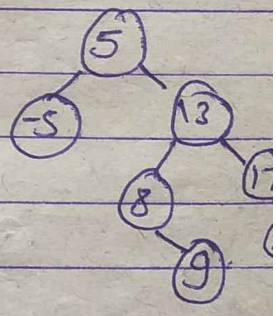
4)



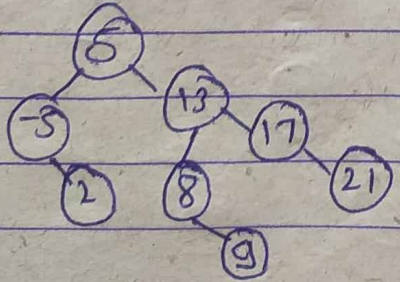
5)



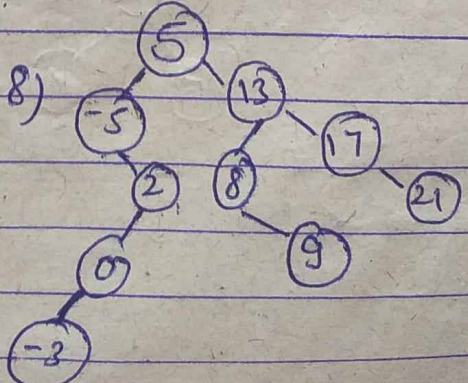
6)



7)

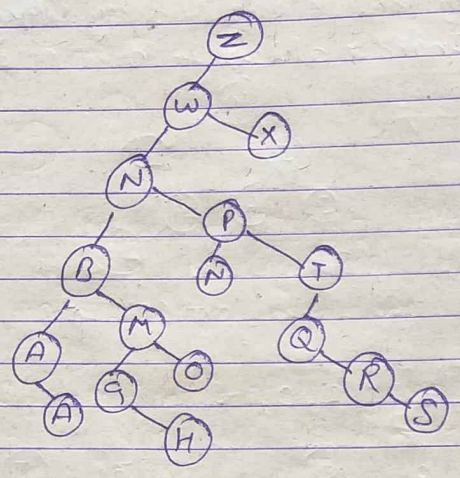


8)

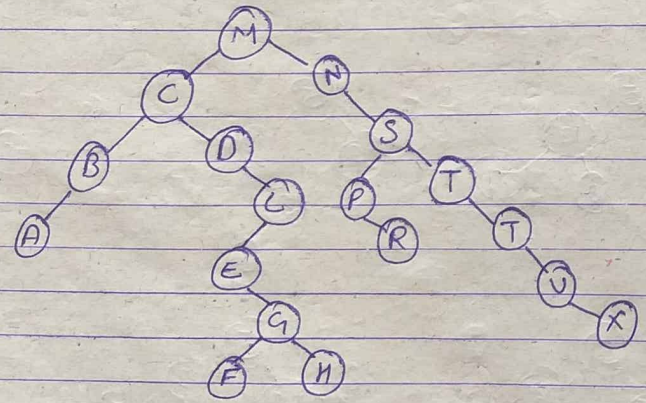


INOrder: -5 -3 0 2 5 8 9 13 17 21.

Ques Z, W, N, B, A, P, N, M, T, A, Q, R, S, O, G, H, X.



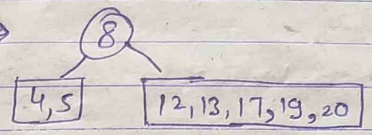
Ques M N S T C B A D L P R T U X E G H F.



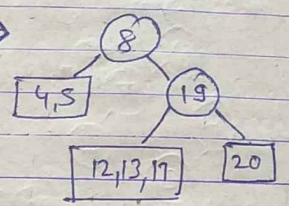
Ques Post order :- 20, 5, 4, 13, 12, 17, 19, 8.

Sol. make IN :- 4, 5, 8, 12, 13, 17, 19, 20

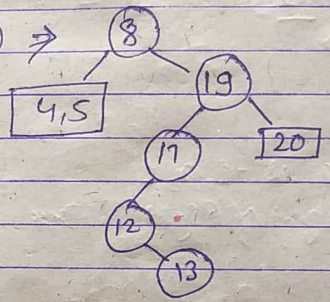
① ⇒



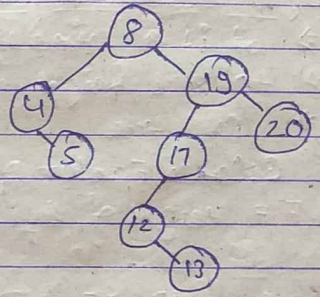
② ⇒



③ ⇒



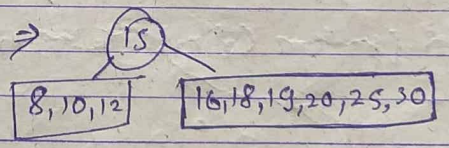
④ ⇒



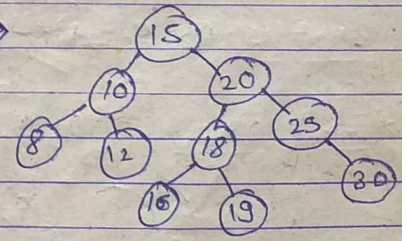
Ques 15, 10, 8, 12, 20, 18, 16, 19, 25, 30 Pre Order.

Sol. IN :- 8, 10, 12, 15, 16, 18, 19, 20, 25, 30

⇒

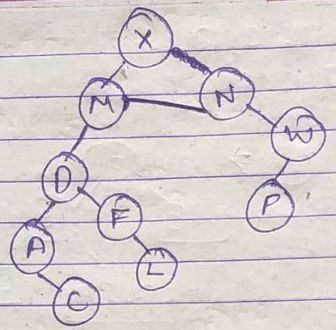


⇒



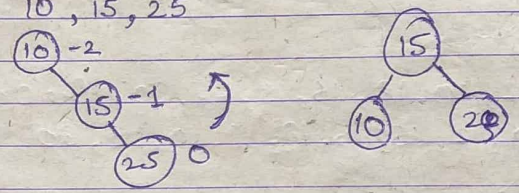
Ques X M N D F A C W L P. Pre order.

IN:- A C D F L M N P W X.



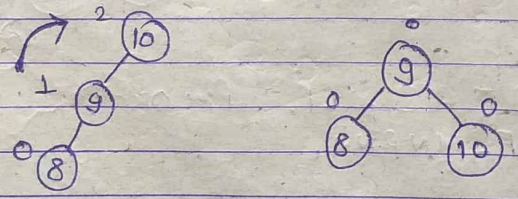
* Height balanced tree :- (AVL tree)
  - ⇒ It is a binary tree
  - ⇒ to keep height minimum
  - ⇒ AVL tree
  - ⇒ Balance Factor of every node is -1, 0, 1
- ⇒ B.F = Height of left subtree - Height of Right subtree.

eg 10, 15, 25

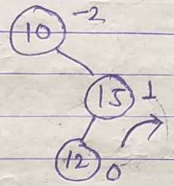


RR Rotation / RR problem (Single AVL rotation)

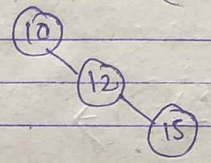
* LL problem 10, 9, 8



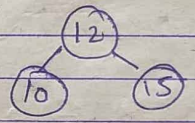
* RL problem



1st rotation (RR) (Double rotation)

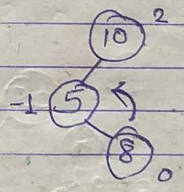


2nd rotation.

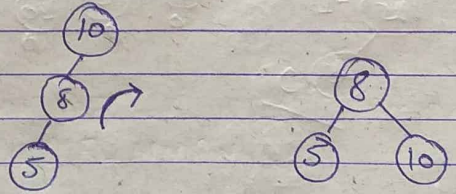


* LR Rotation

10, 5, 8



LL Rotation



* Max Heap tree :-

⇒ Complete Binary tree

⇒ Value of parent node is greater than all the child nodes in it's subtree.

⇒ It means the largest value is stored in the root node.

Create max heap from following values.

19, 10, 13, 5, 14, 12, 21, 6, 7.

1) 9 ⇒ 9

2) 10 ⇒ 9 → 10 / 10 / 9

3) 13 ⇒ 10 → 13 / 9 → 13 / 9 10

4) 5 ⇒ 13 / 9 10 / 5

5) 14 ⇒ 13 / 9 10 / 5 14 → 13 / 14 10 / 5 9 → 14 / 13 10 / 5 9

6) 12 ⇒ 14 / 13 10 / 5 9 12 → 14 / 13 12 / 5 9 10

7) 21 ⇒ 14 / 13 12 / 5 9 10 21 → 21 / 13 14 / 5 9 10 12

6) ⇒ 21 / 13 14 / 5 9 10 12 / 6 → 21 / 13 14 / 6 9 10 12 / 5

7) ⇒ 21 / 13 14 / 6 9 10 12 / 5 7 → 21 / 13 14 / 7 9 10 12 / 5 6

* Min Heap tree :-

It means smallest value is stored in the root node.

10, 15, -2, 9, 4, 1, 6, 18, 7.

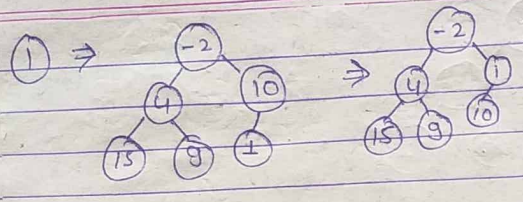
1) 10 ⇒ 10

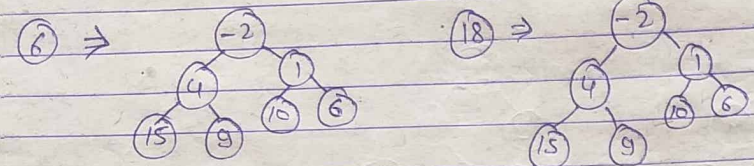
2) 15 ⇒ 10 / 15

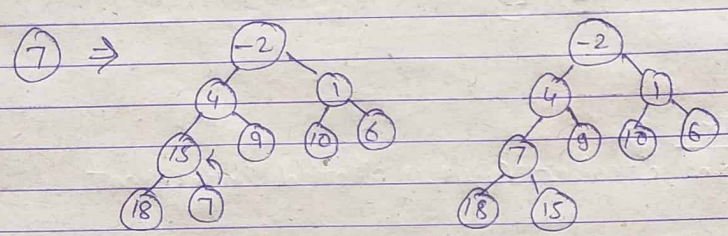
3) 10 / 15 -2 → -2 / 15 10

4) 9 / -2 / 15 10 / 9 → -2 / 9 10 / 15

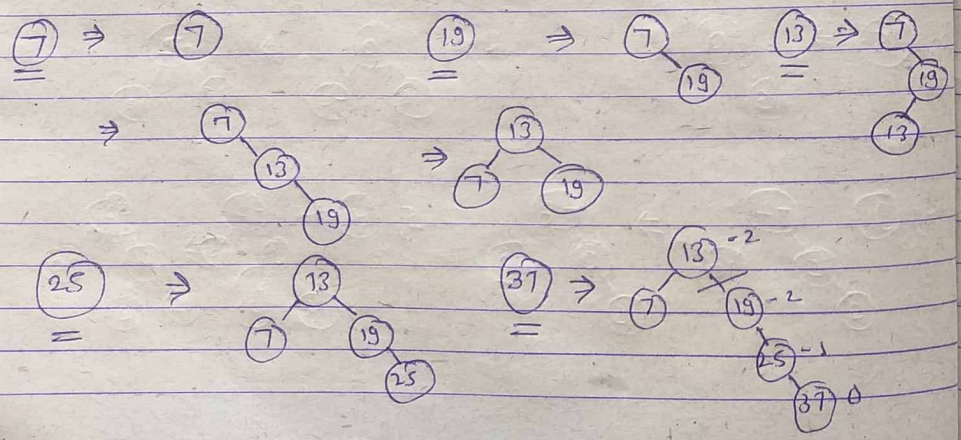
5) 4 ⇒ -2 / 9 10 / 15 4 → -2 / 9 -2 / 15 4 → -2 / 4 -2 / 15 9

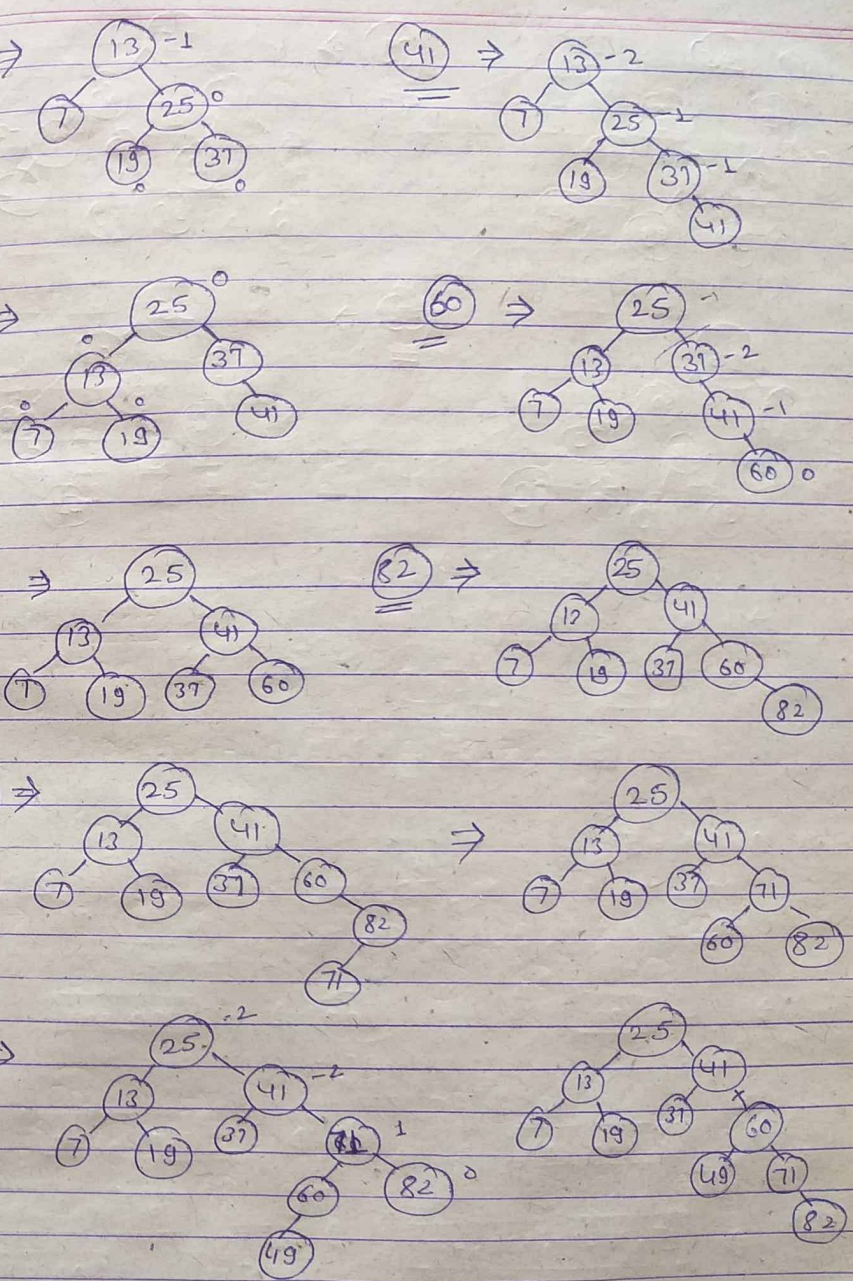


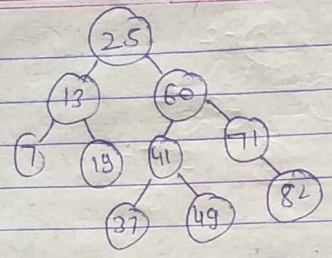




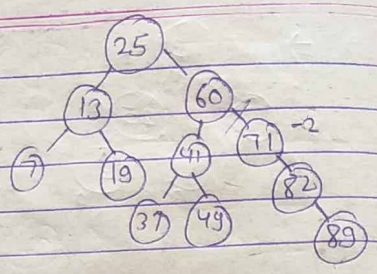
7, 19, 13, 25, 37, 41, 60, 82, 71, 49, 89, 53.

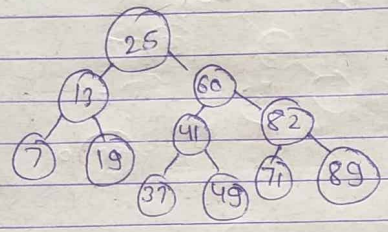




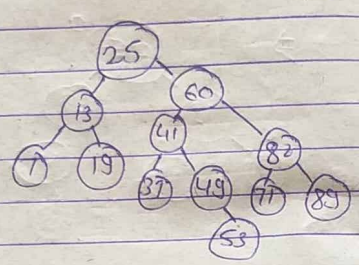


89 ⇒





53 ⇒



* **B-Tree** :- B tree is a m-way tree.
  - ⇒ A node in B-tree can contain m-1 values (Keys)
  - ⇒ And every non leaf node can contain m (have) m children.
  - ⇒ every node in B tree must contain at least m/2 children.
  - ⇒ All leaf node must be at same level.
  - ⇒ A B-tree is also a Binary Search tree.
  - ⇒ It is a m-way tree.

* **Insertion in a B-tree** :- When we insert value in a node and node is already full then the node is splited or (Blasted) and the median value of node goes to the parent node. if Root node is also full then new node Root is also splited and new Root node is created.

Ques Create a Btree of order 4. with following value
1, 5, 6, 2, 8, 11, 13, 18, 20, 79. M = 4

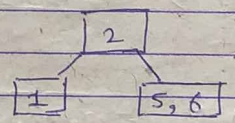
(1) ⇒ [1]

(5) ⇒ [1, 5]

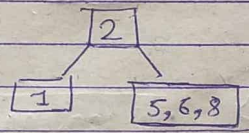
(6) ⇒ [1, 5, 6]

(2) ⇒ [1, 2, 5, 6]

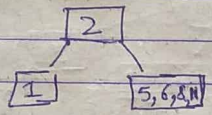
⇒

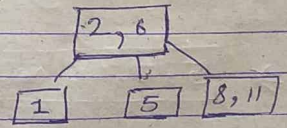


(8) ⇒

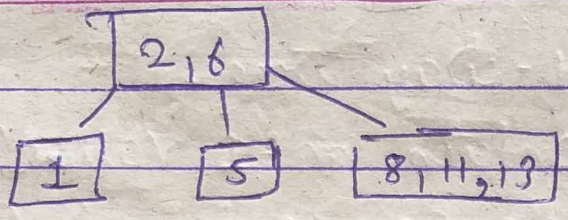


(11) ⇒

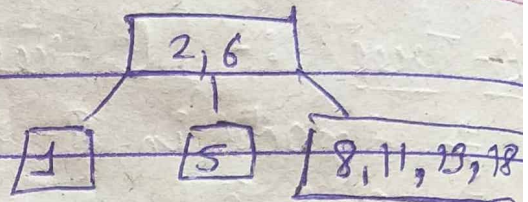




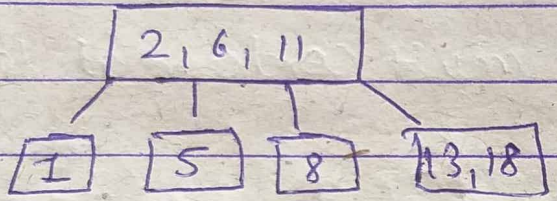
(13) ⇒


2,6
1
5
8,11,13

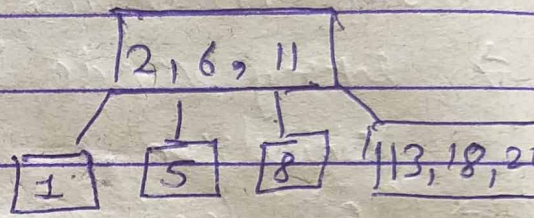
(18) ⇒


2,6
1
5
8,11,13,18

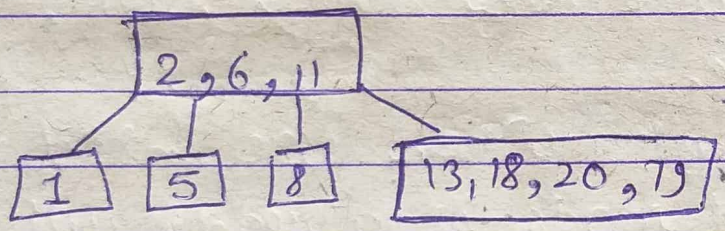
⇒


2,6,11
1
5
8
13,18

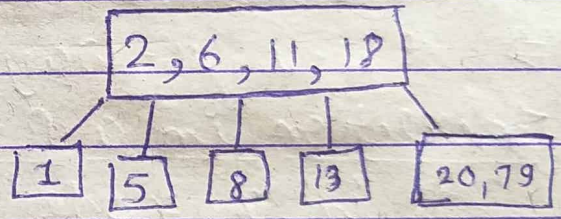
(20) ⇒


2,6,11
1
5
8
13,18,2

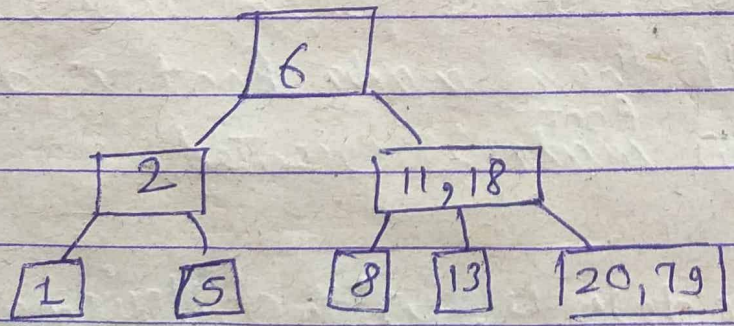
(79) ⇒


2,6,11
1
5
8
13,18,20,79

⇒


2,6,11,18
1
5
8
13
20,79

⇒


6
2
11,18
1
5
8
13
20,79

* graph :-

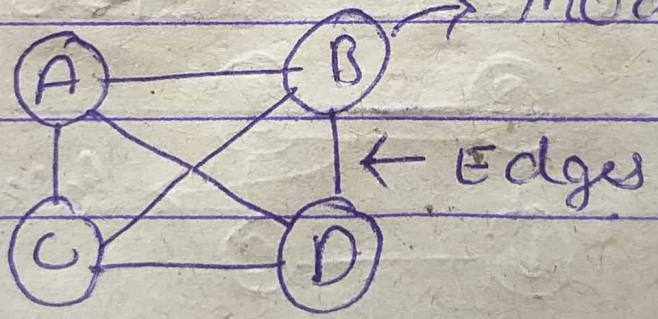
⇒ Non linear data structure.

⇒ Stores data in network format

⇒ many to many relation b/w nodes.

⇒ A graph G = { V, E } where V is set of vertices and E is set of Edges.

undirected graph



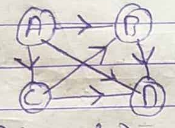
G = { V, E }
V = { A, B, C, D }
E = { AB, AC, AD, BC, BD, CD }

i) **Undirected Edge** :- An edge without direction.

ii) **Undirected graph** :- A graph in which all the edges are undirected.

iii) **Directed Edge** :- An edge which have direction

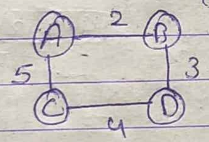
Edges : AB, AC, CB, BD, CD, AD



iv) **Directed graph** :- A graph in which all the edges are directed. (Digraph).

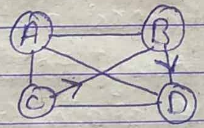
v) **Weighted Edge** :- An edge which have a value associated with it.

vi) **Weighted graph** :- A graph in which all the edges are weighted.

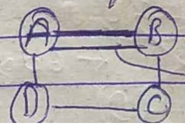


* **Mixed graph** :-

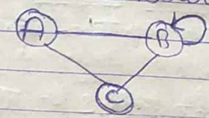
⇒ A graph in which some Edges are directed and some undirected



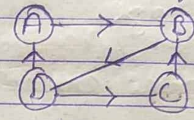
* **Parallel Edge** :- Two edges connecting same set of vertex.



* **Loop / Self loop** :- An edge with same source and destination.



* **Path** :- Set of Edges from a source to destination



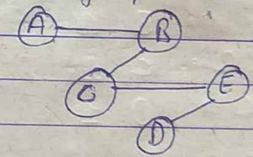
A to C   AB → BD → DC

* **Cycle** : A path which ended on same vertex where it started.

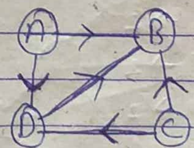
AB → BD → DA.

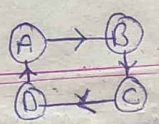
* **Cyclic graph** :- A graph having cycle.

* **Acyclic graph** :- A graph without cycle.



* **Directed Acyclic graph (DAG)** :- A directed graph without cycle.





A ka adjacent B, D

* Adjacent Vertex :-
⇒ Two Vertex connected with same Edge.

* Adjacent Edges :- Two Edges having a common vertex.

CD adjacent edge
DA भी

* Indegree of Vertex :-
No of incoming edges of vertex.

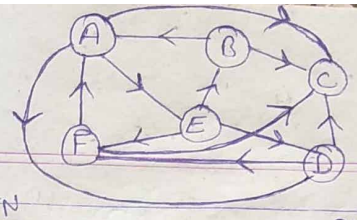
Indegree of A 1
" " B 1
" " C 1
" " D 1

* Outdegree of Vertex :- No. of outgoing edges

Outdegree of A 1
" " B 1
" " C 1
" " D 1

* Degree of node (Indegree + Outdegree)

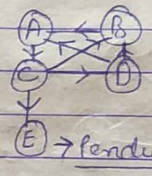
Degree of A → 2
" " B → 2
" " C → 2
" " D → 2



IN
A → 2
B → 1
C → 4
D → 2
E → 1
F → 2

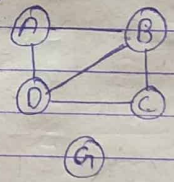
OUT
A → 2
B → 2
C → 0
D → 2
E → 3
F → 2

* Pendulum :- an vertex having indegree 1 and outdegree 0.



E → Pendulum

⇒ Isolated Vertex :- A vertex which is not connected to any other vertex.



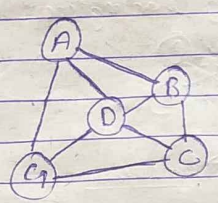
⇒ graph traversal techqiune :- We can use 1 of the following two method for traversal of a graph.
1. BFS → (Breadth for 1st Search)
2. DFS → (Depth first Search)

1. BFS :- In BFS We use
⇒ It is a graph traversal technique
⇒ It uses queue as a supporting data structure.

* Algorithm

1. Select node any as starting node.
2. Add it into Queue
3. Remove node from Queue and insert all it's adjacent vertex in Queue
   (Vertex must not be added twice in Queue)
4. Repeat Step 3 until all nodes are visited.

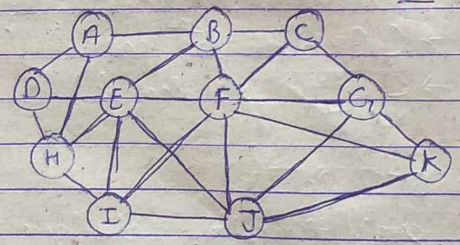
eg :-



Operation	Queue	Traversal order
1. Add A to Queue	A	-
2. Remove A, add B D G in Queue	B, D, G	A
3. Remove B, Add C	D, G, C	B
4. Remove D	G, C	D
5. Remove G	C	G
6. Remove C	-	C

BFS = ABDGC

ques



Operation	Queue	Traversal order
1. Add A	A	-
2. Remove A, Add D H B	D H B	A
3. Remove D, Add E	H B E	D
4. Remove H, Add I	B E I	H
5. Remove B, Add F, C	E I F C	B
6. Remove E, Add J	I F C J	E
7. Remove I,	F C J	I
8. Remove F, Add G K	F C J G K	F
9. Remove C.	J G K	C
10. Remove J	G K	J
11. Remove G	K	G
12. Remove K	-	K

* DFS :- It is a graph traversal technique.
⇒ It uses stack as a supported data structure.

* Algorithm :-

1. Select any node as starting node.
2. Push node to stack.
3. Pop node from stack and push all its adjacent vertex in stack (vertex must not be added twice in stack)
4. repeat all until all nodes are visited.

* Spanning tree :- Spanning tree of a graph is contained all vertex of graph and n-1 edges without any cycle.

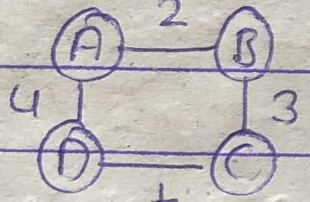
(n = number of vertex / nodes)

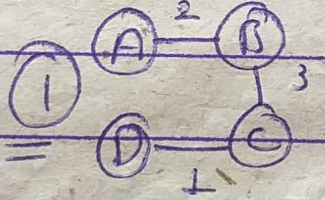
* Minimum cost Spanning tree :-

⇒ We can have multiple tree of a graph.

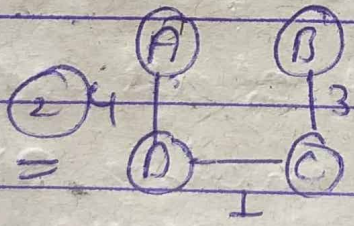
⇒ A tree having minimum weight of a weighted graph is called minimum cost spanning tree.

Eg :-

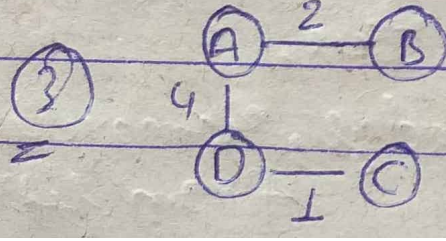




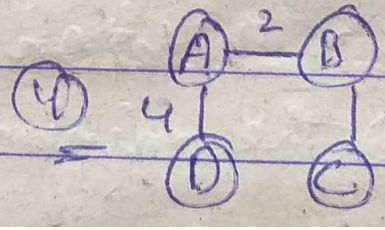
cost = 6



cost = 8



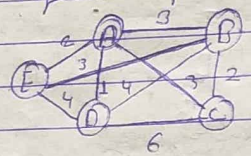
= (1)



⇒ 9

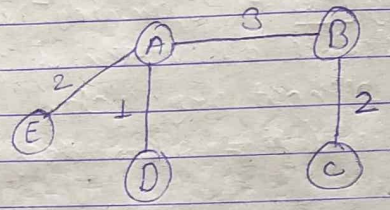
* Prims Algo :-

1. create a list of all edges with their weight.
2. Draw skeleton of spanning tree.
3. Draw edge with minimum weight into skeleton and remove it from list.
4. Draw the adjacent edge of removed edge with minimum weight into skeleton (cycle must not be created).
5. repeat step 4 until n-1 degree are drawn



Edge list

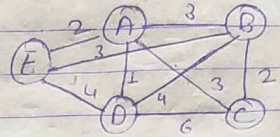
AB	3
AC	3
AD	1
AE	2
BC	2
BE	3
BD	4
CD	6
DE	4



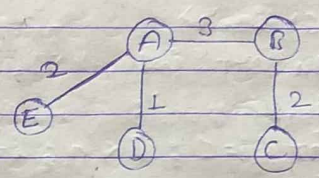
cost = 8.

* Kruskal's Algo :-

1. create the list of all edges. with their weight sorted in ascending order.
2. Draw skeleton of spanning tree.
3. Draw the minimum weight edge, remove it (cycle must be not created)
4. Repeat step 4 until n-1 edge are drawn



AD	1
AE	2
BC	2
AB	3
AC	3
BE	3
~~BE~~	~~4~~
BD	4
DE	4
CD	6



cost = 8

* Insertion Sort :- is also a simple sorting machanism the complexity of insertion sort is $O(n)^2$.

In the insertion sort the value is inserted at its finally position by comparing it to the other elements of the array.

I step		step 2	step 3	
10	5	5	5	
5	10	10	10	1
13	13	13	13	9
9	9	9	9	1
8	8	8	8	8

compare

dc

		Step 4	5	5	5
0	5		9	9	8
1	9		10	10	9
2	10		13	8	10
3	13		8	13	13
4	8				

```
#include <stdio.h>
void insertionSort (int arr[], int n)
  { int i, key, j;
      for (i=1; i<n; i++)
      { key = arr[i];
        j = i-1
/* move element of arr [0..i-1], that are greater
   than key, to one position ahead of their
   current position */

      While (j>=0 && arr[j] > key)
      { arr [j+1] = arr[j];
          j = j-1;
      }
        arr [j+1] = key;
      }
  }
/* A utility function to print an array of size n*/
  void printArray (int arr[], int n)
    {
      int i;
      for ( i=0; i<n; i++)
      printf ("%d", arr[i]);
        printf ("\n");
    }
/* main to execute the insertion sort code*/
  int main ()
  {
      int arr[] = { 12, 11, 13, 5, 6 };
      int n = sizeof (arr) / sizeof [arr[0]);
      insertionSort (arr, n);
      printArray (arr, n);
      return 0;
  }
```

* Bubble Sort :-

Step 1

10	5	5	5	5
5	10	4	4	4
4	4	10	9	9
9	9	9	10	1
1	1	1	1	10

Step 2

5	4	4	4
4	5	5	5
9	9	9	1
1	1	1	9
10	10	10	10

Step 3

4	4	4	1
5	5	1	4
1	1	5	5
9	9	9	9
10	10	10	10

* Bubble Sort is a sorting algorithm that compares two adjacent element and swap them until they are not in the intended order.
Just like the movement of air bubbles in the water that rise up to the surface, each element of the array move to the end in each iteration.
Therefore, it is called a bubble sort.

Working of Bubble Sort
Suppose we are trying to sort the element in ascending order:

1. First Iteration (compare and swap)
2. Starting from the first index, compare the first and the second elements.
3. If the first element is greater than the second element, they are swapped.
4. Now, compare the second and the third elements. Swap them if then they are not in order. The above process goes on until the last element.

The above process goes on until the last element.

```
#include <stdio.h>
void bubbleSort(int array[], int size)
{ int step, i;
    for (step = 0; step < size - 1; ++step)
  { for (i=0; i < size - step - 1; ++i)
    { if (array[i] > array[i+1])
      { int temp = array[i];
        array[i] = array[i+1];
        array[i+1] = temp;
      }
    }
  }
}
void printarray(int array[], int size)
  { int i;
    for (i=0; i < size; ++i)
  {  printf("%d", array[i];
    }
   printf("\n")
}
int main
  { int data[] = {-2, 45, 0, 11, -9, 89, 4};
    int size = sizeof(data) / sizeof(data[0]);
     bubble Sort(data, size);
printf("Sorted Array in Ascending order:\n");
    printArray(data, size);
     return 0;
}
```

Heap Sort:- is a fopular and efficient sorting algo. in computer programming. learning how to write the heap sort algo. requires knowledge of two types of data structure - array and trees.
The initial set of numbers that we want to sort is stored in an array e.g. [10, 3, 76, 34, 23, 32] and after sorting we get sorted array [3, 10, 23, 32, 34, 76]
Heap sort works by visualizing the elements of the array as a special kind of complete binary tree called a heap.

```
# include <stdio.h>
void swap (int *a, int *b)
    int temp = *a;
    *a = *b;
    *b = temp;
}
void heapify (int arr[], int n, int i)
{ int largest = i;
  int left = 2*i+1;
  int right = 2*i+2;
if (left < n && arr[left] > arr[largest])
      largest = left;
if (right < n && arr[right] > arr[largest])
      largest = right.
  if (largest != i)
  { swap (&arr[i], &arr[largest]);
    heapify (arr, n, largest);
  }
}
```

```
void heapSort (int arr[], int n)
  { int i;
    for (i = n/2 - 1; i >= 0; i--)
      heapify (arr, n, i);
    for (i = n-1; i >= 0; i--)
    { swap (&arr[0], &arr[i]);
      heapify (arr, i, 0);
    }
  }
void printarray (int arr[], int n)
{ int i;
    for (i=0; i<n; ++i)
      printf ("%d", arr[i]);
    printf ("\n");
}
int main ()
{ int arr[] = {1, 12, 9, 5, 6, 10, 15, 28};
  int n = sizeof(arr) / sizeof(arr[0]);
  heapSort (arr, n);
  printf ("Sorted array is \n");
  printArray (arr, n);
}
```

* Merge Sort:- is one of the most popular sorting algo. that is based on the principle of divide and conquer algo.
Hence, here a problem is divided into multiple sub problems. each sub problem is solved individually.
Finally, sub-problems are combined to form

the final Solution.

Divide and conquer Strategy.

Using the divide and conquer technique, we divided a problem into subproblems. When the Sol. to each subproblem is ready. We combine the results for from the subproblems to Solve the main problem.

Suppose we had to Sort an array A. A subproblem would be to Sort a sub section of this array starting at index p and ending at index r, denoted as A[p .. r].

* Divide :-

If q is the half way point b/w p and r, then we can split the subarray A[p .. r] into two arrays A[p .. q] and A[q+1, r]

* conquer :-

In the conquer step, we try to Sort both the Subarray A[p .. q] and A[q+1, r]. if we haven't yet reached the base case, we again divided both these Subarray and try to Sort them.

* combine :-

When the conquer step reaches the base step and we get two Sorted Subarray A[p .. q] and A[q+1, r] for array A[p .. r], we combine the result by creating a Sorted array A[p .. r] from two Sorted Subarrays A[p .. q] and A[q+1, r]

```
#include<stdio.h>
void merge(int arr[], int p, int q, int r)
{ int n1 = q-p+1;
  int n2 = r-q;
  int L[n1], M[n2];
  int i,j,k;
  for(i=0; i<n1; i++)
    L[i] = arr[p+i];
  for(j=0; j<n2; j++)
    M[j] = arr[q+1+j];
  i=0;
  j=0;
  k=p;
  while(i<n1 && j<n2){
    if(L[i]<=M[j]){
      arr[k] = L[i];
      i++;
    } else {
      arr[k] = M[j];
      j++;
    }
    k++;
  }
  while(i<n1){
    arr[k] = L[i];
    i++;
    k++;
  }
```

```
While (j < n2)
  arr[k] = M[j];
   j++;
   k++;
  }
}
void mergesort (int arr[], int l, int r) {
   if (l < r) {
    int m = l + (r-l) /2;
mergesort (arr, l, m);
mergesort (arr, m+1, r);
merge (arr, l, m, r);
}
}
Void printarray (int arr[], int size) {
   int i;
    for (i=0; i<size; i++)
    printf ("%d", arr[i]);
    printf ("\n");
}
  int main () {
int arr[] = {6, 5, 12, 10, 9, 1, 78, -9};
int size = sizeof (arr)/sizeof (arr[0]);
mergesort (arr, 0, size-1);
printf ("Sorted array: \n");
printarray (arr, size);
}.
```

* quick sort :-

```
    #include <stdio.h>
void swap (int* a, int* b)
 {  int t = *a;
    *a = *b;
    *b = t;
 }
int partition (int array[], int low, int high)
 {  int pivot = array[high];
    int i = (low - 1);
    int j;=
 for (j = low; j < high, j++)
  { if (array[j] <= pivot)
    { i++
     swap (&array[i], &array[j]);
    }
  }
  swap (&array[i+1], &array[high]);
   return (i+1);
}
 void quicksort (int array[], int low, int high)
{ if (low < high)
  { int pi = partition (array, low, high);
   quickSort (array, low, pi-1);
   quickSort (array, pi+1, high);
  }
}
```

```
void printarray (int array[], int size)
  { int i;
    for (i=0; i< size; ++i) {
      printf ("%d", array[i]);
    }
  printf ("\n");
}
  int main () {
  int data[] = { 8,7,2,1,0,9,6,-7};
  int n = sizeof (data) / sizeof (data[0]);
  printf ("unsorted Array \n");
  printarray (data, n);
  quicksort (data, 0, n-1);
  printf ("sorted array in ascending order: \n");
   printarray (data, n);
  }.
```

* counting Sort:-

counting sort is a storing technique based on keys between a specific range. It works by counting the number of object having distinct key value (kind of hashing). Then do some arithmetic to calculate the position of each object in the output sequence.

counting sort makes assumption about the data for example, it assumes that values are going to be an in the range of 0 to 10 - 99 etc Some other assumption counting sort makes are input data will be all real numbers.

like other algo. this sorting algo. is not a comparison - based algo., it hashes the value in a temporary count array and uses them for sorting.

- It uses a temporary array making it non in place algo.
- Let us understand it with help of an eg.
- For simplicity, consider the data in range 0 to 9

Input data : 1,4,1,2,7,5,2

1) Take a count array to store the count of each unique object

Index	0	1	2	3	4	5	6	7	8	9
count	0	2	2	0	1	1	0	1	0	0

2) Modify the count array such that each element at each index

Index:	0	1	2	3	4	5	6	7	8	9
count :	0	2	4	4	5	6	6	7	7	7

The modify count array indicates the position of each object in the output sequence.

3) Rotate the array clockwise for one time.

Index :	0	1	2	3	4	5	6	7	8	9
count :	0	0	2	4	4	5	6	6	7	7

4) Output each object from the input sequence followed by increasing its count by 1.

PROCESS the input data 1,4,1,2,7,5,2. Position 1 is 0 Put data 1 at index 0 in output. Increase count by 1 to place. next data 1 at an index 1 greater than this index.

* Radix Sort :- The lower bound for comparison based sorting algo. (Merge Sort, Heap Sort, Quick Sort etc) is $\Omega(n \log n)$, i.e, they cannot do better than $n \log n$.
Counting Sort is a linear time sorting algo. that sort in $O(n+k)$ times when element are in the range from 1 to k.
What if the element are in the range from 1 to n^2
We can't use counting sort because counting sort will takes $O(n^2)$ which is worse than comparison based sorting algo. Can we sort such an array in linear time?
Radix sort is the answer. The idea of Radix sort is to do digit by digit sort starting from least significant digit to most significant digit. Radix sort uses counting sort as a subroutine to sort.

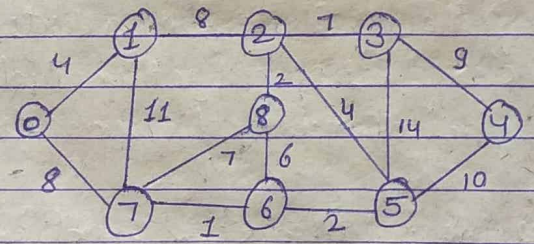
The radix sort Algo.
Do following for each digit i where i value from least significant digit to the most significant digit.
Sort input array using counting sort (or any stable sort) according to the ith digit.
Example:-
original unsorted list:
170, 45, 75, 90, 802, 24, 2, 66
Sorting by least significant digit (1's place) gives:
[* Notice that we keep 802 before 2, because 802 occurred before 2 in the original list, and similarly for pairs 170 & 90 and 45 & 75.]
170, 90, 802, 2, 24, 45, 75, 66.
Sorting by next digit (10's place) give:
[* Notice that 802 again comes before 2 as 802 comes before 2 in the previous list.]
802, 2, 24, 45, 66, 170, 75, 90
Sorting by the most significant digit (100's place) gives:
2, 24, 45, 66, 75, 90, 170, 802.

* Dijkstra's Shortest path algo. :-
Given a graph and a source vertex in the graph find the shortest path from the source to all vertices in the given graph.
Dijkstra's algo. is very similar to prism's algo. for minimum spanning tree. Like Prim's MST we generate a SPT (shortest path tree) with a given source as a root.
We maintain two sets, one set contains vertices included in the shortest-path tree, other set includes vertices not yet included in the shortest path tree.
At every step of the algo. we find a vertex that is in the other set (set of not yet included) and has a minimum distance from the source.
Below are the detailed steps used in Dijkstra's algo to find the shortest path from a single source vertex to all other vertices in the given graph.

* Algorithm :-

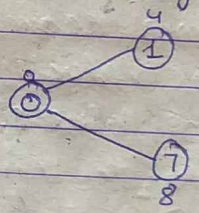
1) Create a set sptSet (shortest path tree set) that keeps track of vertices included in the shortest-path tree, i.e whose minimum distance from the source is calculated and finalized. Initially, this set is empty.
2) Assign a distance value to all vertices in the input graph. Initialize all distance values as Infinite. Assign distance value as 0 for the source vertex so that it is picked first.
3) While sptSet doesn't include all vertices.
   a) Pick a vertex u which is not there in sptSet and has a minimum distance value.
   b) Include u to sptSet.
   c) Update distance value of all adjacent vertices of u. To update the distance value, iterate through all adjacent vertices. For every adjacent vertex v, if the sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

Let us understand with examples:-

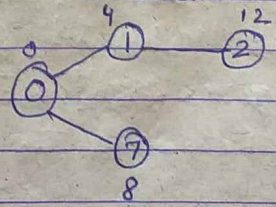


The set sptSet is initially empty and distance assigned to vertices are {0, INF, INF, INF, ", ", ", "} where INF indicates infinite. Now pick the vertex with a minimum distance value. The vertex 0 is picked, include it in sptSet. So sptSet become {0}. After including 0 are 1 and 7. The distance value of 1 and 7 are update as 4 and 8. The following subgraph shows vertices and their distance values, only vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.

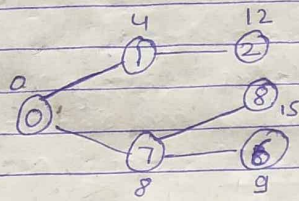
→ to SPT set, update distance value of its adjacent vertices Adjacent vertices of 0



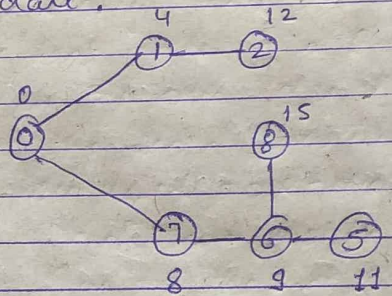
Pick the vertex with minimum distance value and not already included SPT (not in sptSet). The vertex 1 is picked and added to sptSet now becomes {0, 1}. Update the distance value of adjacent vertices of 1. The distance value of vertex 2 become 12.



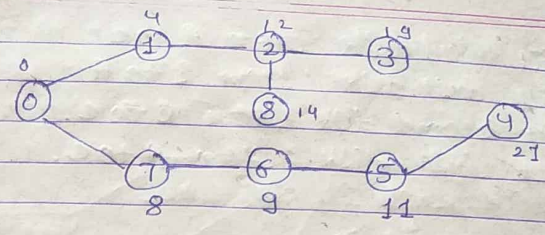
Pick the vertex with minimum distance value and not already included in SPT (not in spt set). Vertex 7 is picked. So spt set now becomes {0, 1, 7}. update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 become finite (15 and 9 respectively)



Pick the vertex with minimum distance value and not already included in spt (not in spt set) vertex 6 is picked. So spt set now become {0, 1, 7, 6} update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are update.



We repeat the above steps until spt set includes all vertices of the given graph. Finally we get following shortest path tree (SPT).



* Tower of hanoi :- Tower of hanoi is a mathematical puzzle where we have three rods and n disk. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:
  - only one disk can be moved at a time.
  - Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
  - No disk may be placed on top of a smaller disk.

Let rod 1 = 'A', rod 2 = 'B', rod 3 = 'C'

Eg with 2 disk :-
- Step 1 : Shift first disk from A to B
- Step 2 : Shift second disk from A to C
- Step 3 : Shift first disk from B to C

Eg with 3 disk :-
- Step 1 : Shift first disk from 'A' to 'C'
- Step 2 : Shift second disk from A to B
- Step 3 : Shift first disk from C to B.

Step 4: Shift third disk from A to C
Step 5: Shift first disk from B to A
Step 6: Shift second disk from B to C.
Step 7: Shift first disk from A to C.

The pattern here is :-
Shift n-1 disk from A to B, using C.
Shift last disk from A to C.
Shift n-1 disks from B to C using A.

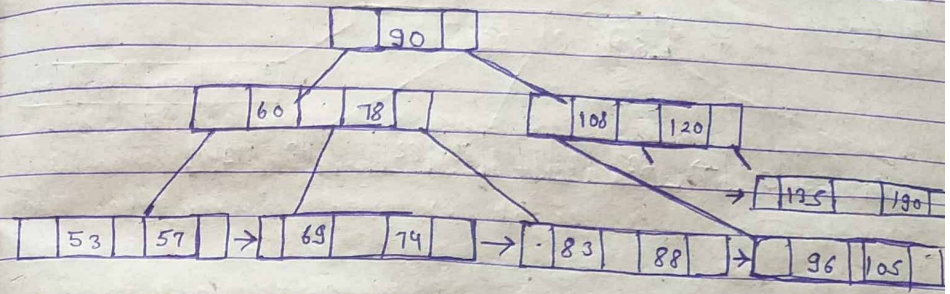
* B+ Tree :- B+ tree is an extension of B tree which allows efficient insertion, deletion and search operation.

In B tree, keys and records both can be stored in the internal as well as leaf node whereas, in B+ tree, records (data) can only be stored on the leaf nodes while internal nodes can only store the key value.

The leaf nodes of a B+ tree are linked together in the form of a singly linked lists to make the search queries more efficient.

B+ tree are used to store the large amount of data which can not be stored in the main memory. Due to the fact that, size of main memory is always limited, the internal nodes (keys to access records) of the B+ tree are stored in the main memory whereas leaf nodes are stored in the secondary memory.

The internal nodes of B+ tree are often called index nodes. A B+ tree of order 3 is shown in following figure.



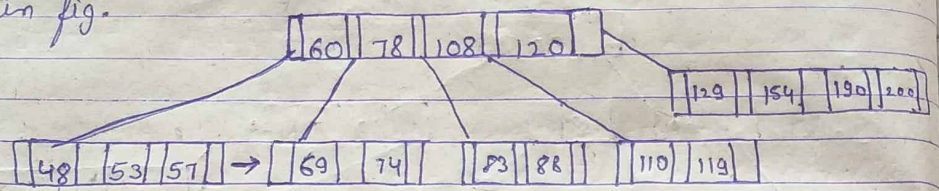
* Advantages of B+ tree :-
1. Records can be fetched in equal number of disk accesses.
2. Height of the tree remains balanced and less as compare to B tree.
3. We can access the data stored in a B+ tree sequentially as well as directly.
4. Keys are used for indexing
5. Faster search queries as the data is stored only on the leaf nodes.

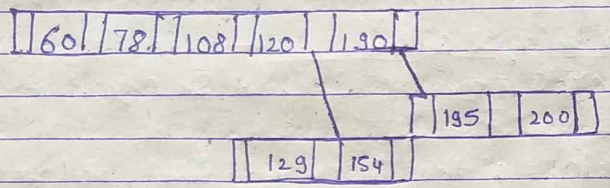
* Insertion in B+ tree :-
- Step 1: Insert the new node as a leaf node.
- Step 2: if the leaf doesn't have required space, split the node and copy the middle node to the index node.
- Step 3: if the index node doesn't have require space, split the node and copy the middle element

element to the next index page.

Example :-

Insert the value 195 into the B+ tree of order 5 shown in fig.





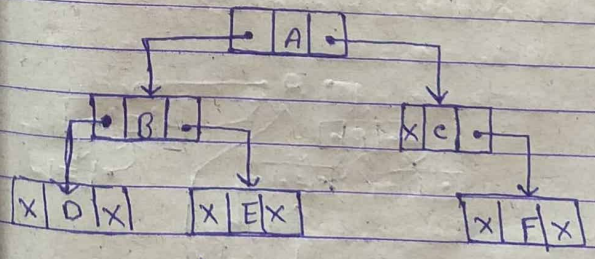
*Threaded Binary tree :-

In the linked representation of binary trees, more than one half of the link fields contain NULL value which result in wastage of storage space. If a binary tree consists of n nodes then n+1 link fields contain NULL value. So in order to effectively manage the space, a method was devised by Perlis and Thornton in which the NULL links are replaced with special links known as threads. Such binary trees with threads are known as threaded binary trees. Each node in a threaded binary tree either contains a link to its child node or thread to other nodes in the tree.

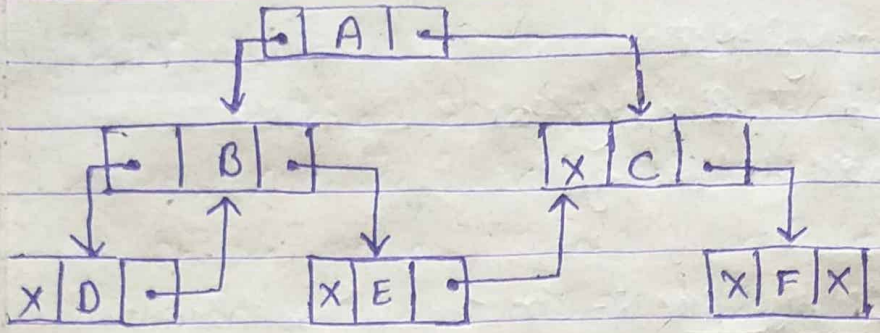
Types of Threaded Binary tree :-

- One-Way threaded Binary tree
- Two-Way threaded Binary tree.

* One Way threaded Binary tree :- In one way threaded binary trees, a thread will appear either in the right or left link field of a node. If it appears in the right link field of a node then it will point to the next node that will appear on performing in order traversal. Such trees are called right threaded binary trees. If thread appear in the left field of a node then it will point to the nodes inorder predecessor. Such trees are called left threaded binary trees, left threaded binary tree are used less often as they don't yield the last advantage of right threaded binary trees. In one-way threaded binary tree the right link field of last node and left link field of first node contains a NULL. In order to distinguish threads from normal links they are represented by dotted lines.



A Binary tree
Inorder traversal
(D, B, E, A, C, F)



A Right - Threaded Binary tree.

The above figure shows the inorder traversal of this binary tree yields D, B, E, A, C, F. When this tree is represented as a right threaded binary tree, the right link field of leaf node D which contain a NULL value is replaced with a threaded that points to node B which is the inorder successor of node D. In the same way other nodes containing values in the right link field will contain NULL value.

Two - Way threaded Binary Trees: In two way threaded Binary trees, the right link field of a node containing NULL values is replaced by a thread that points to nodes inorder successor and left field of a node containing NULL values is replaced by a threaded that points to nodes inorder predecessor.

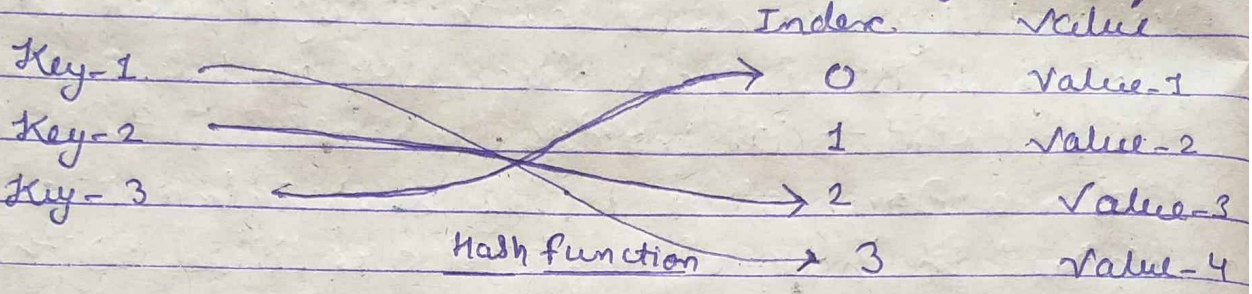
The above figure shows the inorder traversal of this binary tree yields D, B, E, G, A, C, F. If we consider the two way threaded Binary tree, the node E whose left field contain NULL is replaced by a thread pointing to its inorder predecessor i.e node B. Similarly, for node G whose right and left linked field contain NULL values are replaced by threads such that right link field points to its inorder successor and left link field point to its inorder predecessor. In the same way, other nodes containing NULL values in their link fields are filled with threads.

* Hashtable and Hashing :-

- Hash table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.
- Thus, it become a data structure in which insertion and search operation are very fast irrespective of the size of the data. Hash table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

Hashing :-

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key, value) form.



Sr.No	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

Linear Probing :-

As we can see, it may happen that the hashing technique is used to create an already used index of the array. In such a case, we can search the

next empty location in the array by looking into the next cell until we find an empty cell. This technique is called linear probing.

Sr.No	Key	Hash	Array Index	After linear Probing, AI
1	1	1%.20 = 1	1	1
2	2	2%.20 = 2	2	2
3	42	42%.20 = 2	2	3
4	4	4%.20 = 4	4	4
5	12	12%.20 = 12	12	12
6	14	14%.20 = 14	14	14
7	17	17%.20 = 17	17	17
8	13	13%.20 = 13	13	13
9	37	37%.20 = 17	17	18.

Basic Operation

Following are the basic primary operation of a hash table.

- Search - Searches an element in a hash table.
- Insert - insert an element in hash table.
- delete - Deletes an element in hash table.

Data Item.

Define data item having some data and key, based on which the search is to be conducted in a hash table.

```
Struct data item {
    int data;
    int key;
};
```

Hash method :-
Define a hashing method to compute the hash code of the key of the data item.

```
int hash code (int Key) {
    return Key % SIZE; }
```

* Quadratic Probing :- Quadratic probing is an open - addressing scheme where we look for i2'th slot in i'th iteration if the given hash value x collides in the hash table.

- How Quadratic Probing is done?

1. Let hash(x) be the slot index computed using hash func.
2. Let hash(x) % S is full, then we try (hash(x) + 1*1) % S.
3. if (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S.
4. if (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3) % S.

This process is repeated for all the value of i until an empty slot is found.

* Double hashing :-
Double hashing is a collision resolving technique in open addressed Hash tables. Double hashing uses the idea of applying a second hash function to key when a collision occurs.

* Advantage of Double hashing :-

- It is one of effective method for resolving collision.
- The advantage of Double hashing is that it is one of the best form of probing, producing a uniform distribution of records throughout a hash table. This technique does not yield any clusters.
- Double hashing can be done using : (hash1(key) + i * hash2(key)) % TABLE-SIZE
- Here hash1() and hash2() are hash function and table-SIZE is size of hash table. (We repeat by increasing i when collision occurs)