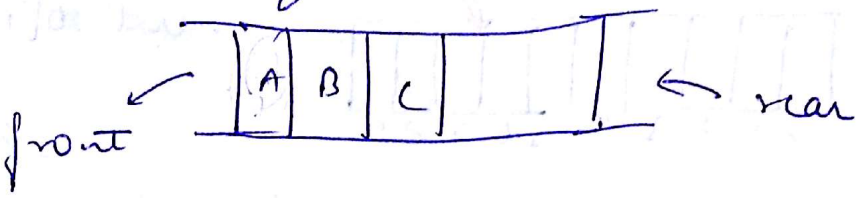
Queue :

It is a collection of items from which an item can be deleted from one end (ie called front end of queue) and in which item can be inserted from the other end (ie. called as rear end of queue).



The item which you insert first will be deleted first.

Real applicaⁿ

Theatre - ticket - person's line (queue)

→ first in first out data structure (FIFO)
the item which is first is out first

∮ Stack is LIFO

Operations

1) Enqueue (Push)
Inserts an item x at the rear end.
O(1)

2) Dequeue (Pop)
Deletes an item from front end.
O(1)

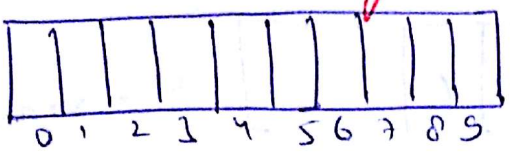
3) Isempty.
Checks whether the queue is empty or not.
Empty → 0
Not " → 1
O(1)
Time complexity

will not depend on the size of queue

Implementations

1) Using Array → linear Queue
   → Circular Queue
2) Using linked list → linear L.L
   → Circular L.L

1) linear queue (array)

eg: Q → 

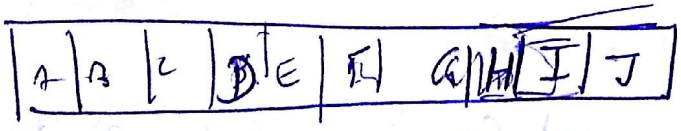
int Q[10];

[-1] front contains the index of the first element inserted

[-1] rear contain the index of the last element inserted.

→
```
int IsEmpty()
{
    if (front == -1 && rear == -1)
        return 1;
    else
        return 0;
}
```

eg:



int Q[10];
init front = -1
init rear = -1

overflow if tried to enter "K"

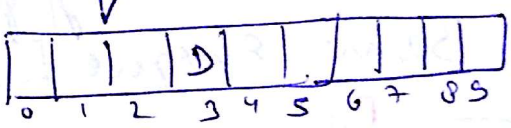
```
void Enqueue (int x)
{
  if (rear == m-1)
  {
    printf ("overflow");
  }
  else if (Isempty())
  {
    front = rear = 0
    a[rear] = x;
  }
  else
  {
    rear++;
    a[rear] = x;
  }
}
```

m = size of array

can also write if IsFull()

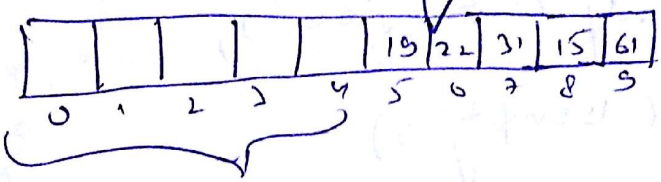
If nothing (no element) is present in an array & we r trying to delete then this condn is known as underflow

→ if front = rear, then only one element is present in array.



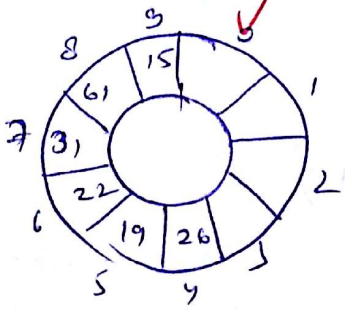

```
int Dequeue ( )
{
  if (Isempty())
    printf ("Underflow");
  else if (front == rear)
    front = rear = -1;
  else
    front = front + 1;
}
```

Disadv:
Using linear queue we can't insert an extra element if there is an empty space out in array.



four spaces empty but can't insert it here ∵ inserted at rear end.

2) Circular Queue (array)



m = max. size of array

(Next) To find neighbour $\Longrightarrow$ $(i+1)\%m$

Previous neighbour $\Longrightarrow$ $(i+m-1)\%m$

→ Same IsEmpty func^n as of linear queue

→ Same Enqueue func^n

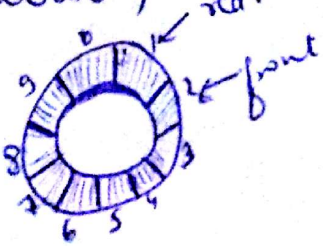
[change rear++ ⟶ (rear+1)%m]

→ same dequeue func^n

[change front++ ⟶ (front+1)%m]

if (rear+1)%m == front

return ;

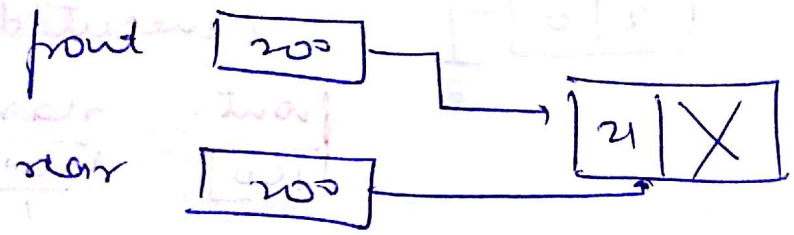


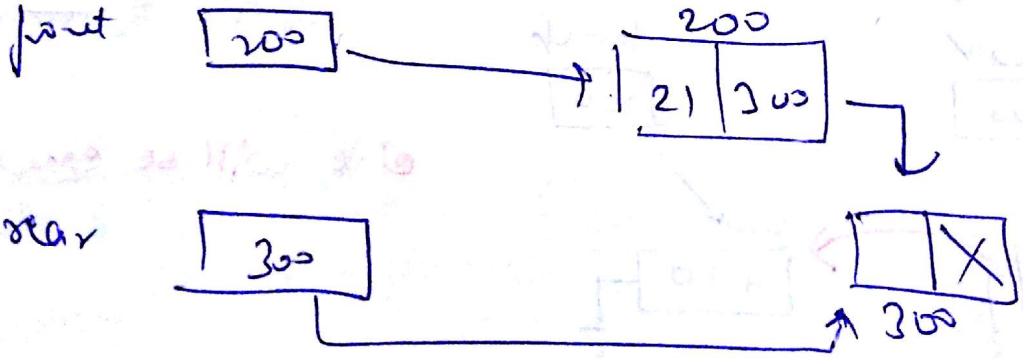
3) Using linked list (singly)

```
struct node
{
  int x;
  struct node *next;
};
```

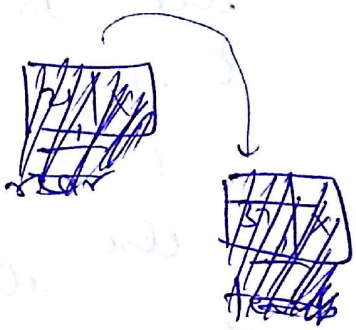
front will contain the address of first node in the linked list.
rear will contain the address of last node in the linked list.

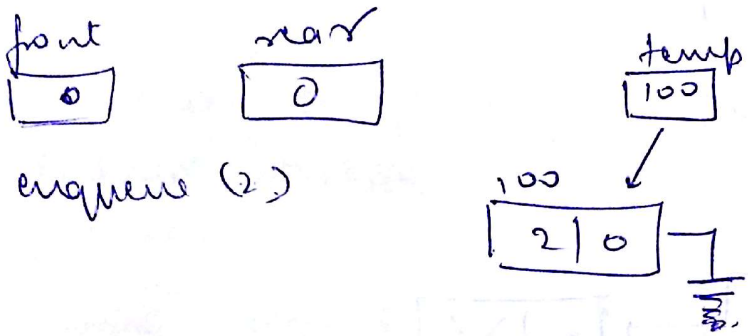
```
int Is_empty()
{
  if (front == NULL && rear == NULL)
    return 1;
  else
    return 0;
}
```





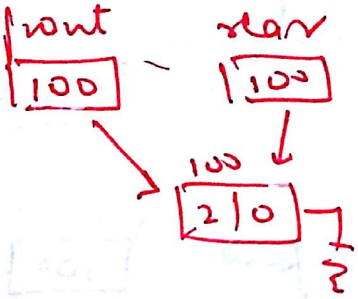
```
Void enqueue (int x)
{
    struct node *temp = (struct node *) malloc
                        sizeof(struct Node));
    temp -> data = x
    temp -> next = NULL;
    if (isempty())
    {
        front = rear = temp;
    }
    else
    {
        rear -> next = temp;
        rear = temp;
    }
```



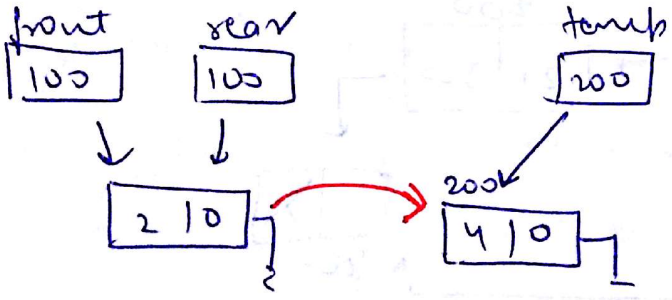


① enqueue (2)

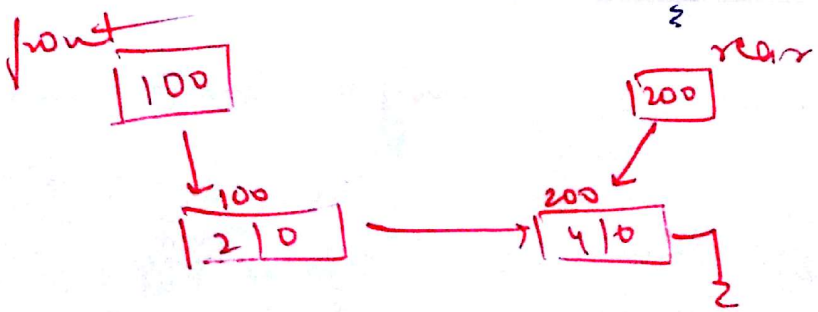
if will be executed



② enqueue (4)



else will be executed



Also analyze the time complexity of (push & pop)-stack (dequeue & enqueue) → queue

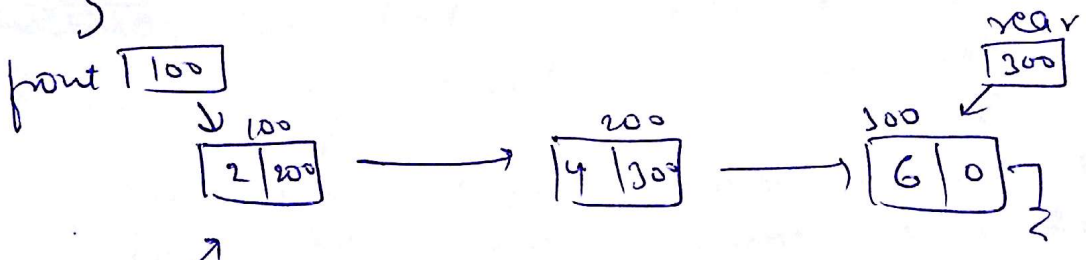
Q. Implement a queue using two stacks.

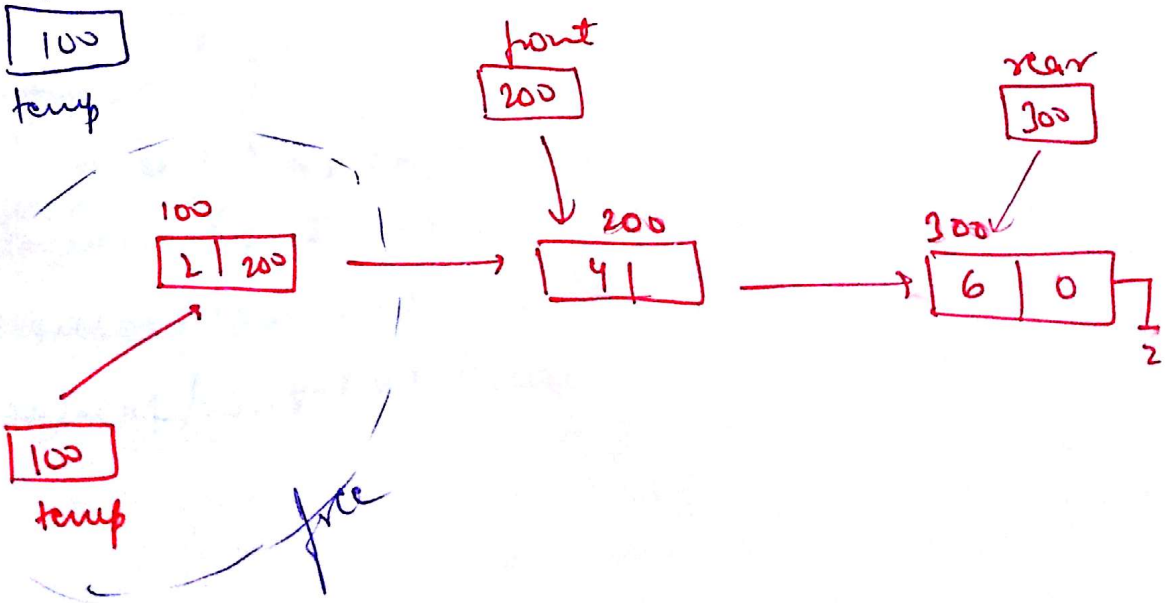
Q. Implement a stacks using two queue.

```
void Dequeue ()
{
  struct Node * temp = front ;

  if (front == NULL)
  return;
  if (front == rear)
  {
     front = rear = NULL;
  }
  else
  {
     front = front -> next;
  }
  free (temp) ;
}
```

storing address of current front in temp ptr.





Circular linked list

If the "next" field in the last node contains a pointer to the first node rather than a null pointer then such a list is called CLL.

Double Ended Queue (DEQUE) / DECK

Deque & Dequeue are different

DEQUE is a collection of items where items are enqueued (inserted) & dequeued (deleted) from either of the end front & rear.

- double ended :: insertion & deletion → at front or rear end doesn't matter
- while in queue
  - insertion → rear
  - deletion → front

Types of Deque

1) Input Restricted
2) Output Restricted

INPUT

Insertion can take place at only one end but deletion can be performed from both ends

OUTPUT

Deletion can take place at only one end but insertion can be performed from both ends.

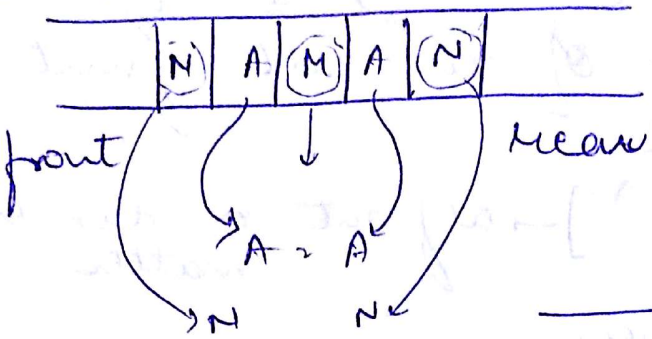
Operations

1) IsEmpty ( )
2) EnqueueRear (x)
3) EnqueueFront (x)
4) DequeueRear (x)
5) DequeueFront (x)

4 & 5 → Stack
2 & 3 → Queue

6. To check whether given string is palindrome or not using Deck. Write a program in C.

Sol MADAM NAMAN



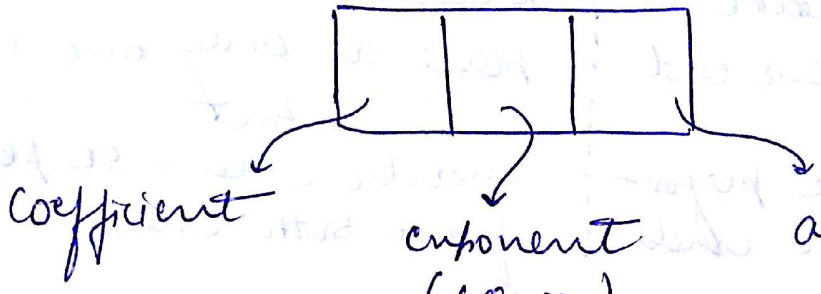
→ delete (Dequeue)

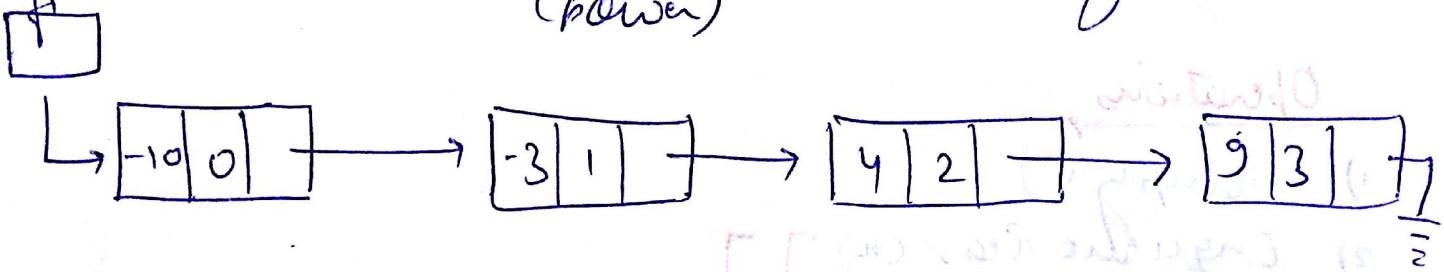
Applications of LL

→ POLYNOMIAL OPERATIONS (Single valued)

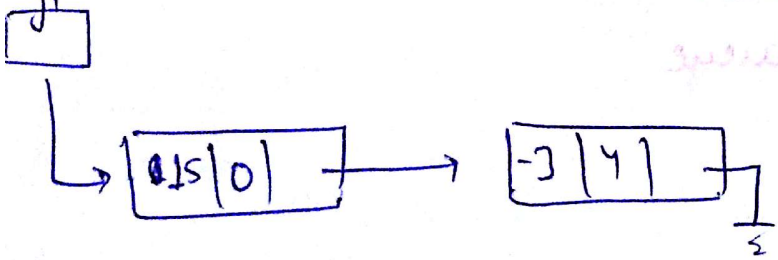
$f(x) = c_0 + c_1x + c_2x^2 + \dots + c_nx^n$

$f_1(x) = 9x^3 + 4x^2 - 3x - 10$





$g_1(x) = -3x^4 + 15$



1) Addⁿ
2) Subtracⁿ
3) Multiplicaⁿ

→ SPARSE MATRIX

Q. Write a funcⁿ to insert a node at the beginning of a Linked list?

```
struct node
{   int data;
    struct node * next;
};
struct node *head;
main()
{  head = NULL;
   AddBeg()
   Display();
}
void AddBeg(int n)
{  struct node *temp;
   temp = (struct node *) malloc (size of (struct node));
   temp -> data = n;
   temp -> next = head;
   head = temp;
```

Time Complexity

→ Motivation

eg: find whether the given no. is prime or not

	A	B
Pseudo code	for (i=2 to (n-1))	for (i = 2 to $\sqrt{n}$)
2	if (i divides n)	if (i divides n)
3	print (no. is non prime)	print (no. is non prime)
5		
9		
11		

Assuming that the division operation is performed in 1 ms.

Worst case → max. no. of times the loop will run

eg Worst case

	A	B
	(n-2) times	($\sqrt{n}$ - 1) times
n = 3	1 ms	0.71 ms
n = 9	7 ms	2 ms
n = 1000	≈ 10^3 ms	24 ms
n = 10^{10}	115 days	1.66 minutes
	$T_A(n) \propto n$	$T_B(n) \propto \sqrt{n}$

Time complexity is a simple measurement of how fast the time taken by the program grows if the input size ↑es.

We should write a program that behaves well for larger input.

The running time of algorithm mainly depends on following four factors:

1. Single v/s Multiprocessor
2. Read / Write speed
3. Operating system
4. Input size

(1), (2), (3) → Machine dependent

(4) → Machine independent

Our motive →
to find rate of growth of time taken by the algorithm wrt input size

Eg. Write a program to find the summation of two no.s. Assuming that addn operation takes constant amt of time whatever be the size of input.

```
int a, b, c;      → c1 amt. of time
c = a + b ;
    |   |
    c3  c2
```

Total time taken = $c_1 + c_2 + c_3$

$T \propto C$ [CONSTANT TYM ALGORITHM]

Eg. Write a program to find sum of n elements in array.

```
int S = 0                          c1
for (i = 0; i <= n; i++)           c2   } execn time
  S = S + A[i]                     c3
```

Analyze of algorithm

Cost	freq
c_1	1
c_2	$n+1$
c_3	n

$$\text{Time} = c_1 + c_2 n + c_2 + c_3 n$$
$$= (c_1 + c_2) + (c_2 + c_3) n$$
$$= a + bn$$

LINEAR TIME ALGORITHM

e.g. ~~find~~ write a program to find the sum of all given elements in a matrix.

```
c1   S=0;
c2   for(i=0; i<=n-1; i++)
     {
c3       for(j=0; j<=n-1; j++)
         {
c4           S = S + A[i][j];
         }
     }
```

Cost	freq
c_1	1
c_2	$n+1$
c_3	$n \times (n+1)$
c_4	n^2

$$\text{Time} = a + bn + cn^2$$

QUADRATIC TIME ALGORITHM

Linear Search (naive)

Eg to search 9. 10, 5, 3, 9, 20
first compare with 10 → 5 → 3
then 9 comes
↳ so return 9.

→ Worst Case ($T_L \propto n$)
The maximum run time of algorithm

→ Avg. Case ($T_L \propto n/2$)
The avg. / expected run time.

→ Best Case ($T_L \propto 1$)
The minimum run time

to search 9

```
for (i = 0 ; i <= n ; i++)
if (a[i] == 9)
    return a[i] ;
printf ("found = %d=", i)
```

→ can't return in loop

Order of Growth

Eg $f_1(n) = 15n^2 + 16n + 3$, $n \to \infty$ $\approx n^2$

$f_2(n) = 21n^2 + 9$, $n \to \infty$ $\approx n^2$

asymptotically equivalent fns →

$\lim_{n \to \infty} f_1(n) \approx \lim_{n \to \infty} f_2(n)$ [in terms of power of n]

In this case we consider only the leading terms, since the lower order terms are relatively insignificant for larger values of n ($n \to \infty$). We also ignore the leading term constant coefficient a, since the constant factors are less significant than the rate of growth in determining the computational efficiency of large inputs.

Asymptotic notations

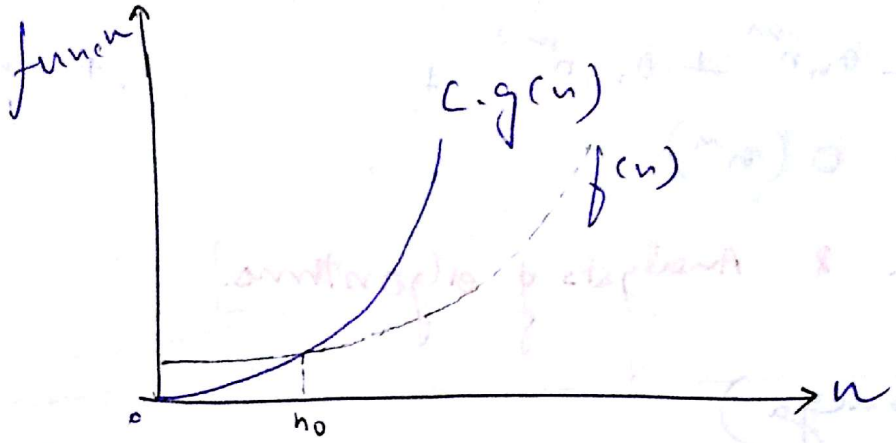
We have a formal mathematical way of classifying the func^n (running time) in the form of asymptotic notations.

There are 3 types of asymptotic notations which are used to determine the time complexity →

1) O (Big-oh) (Worst case)
2) Ω (Big-omega) (best case)
3) Θ (Big theta) (avg case)

) O notation

$O(g(n)) = \{ f(n) : \exists$ constants c & $n_0 :$
$f(n) \leq c \cdot g(n),$
$\forall\ n \geq n_0 \}$



$f(n)$ never grows at a rate faster than $c.g(n)$ after n_0.

O gives us an upper bound on the rate of growth of a func^n $\quad n \geq n_0$

Eg: Suppose $f(n) = 5n^2 + 2n + 1$ for $n \to \infty$ Suppose $g(n) = n^2$
Prove that $f(n) \in O(g(n))$.

Sol: To find c & n_0

$c = 5 + 2 + 1 = 8$

$c.g(n) = 8n^2$

Now for n_0 $\quad 5n^2 + 2n + 1 = 8n^2$

$3n^2 - 2n - 1 = 0$

$(3n+1)(n-1) = 0$

$\therefore \boxed{n \geq 1}$

$n_0 = 1$

1) Sum of 2 elements $\quad O(1)$
2) Sum of elements in a array $\quad O(n)$
3) " " " " matrix $\quad O(n^2)$

If $f(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots + a_1 n + a_0$

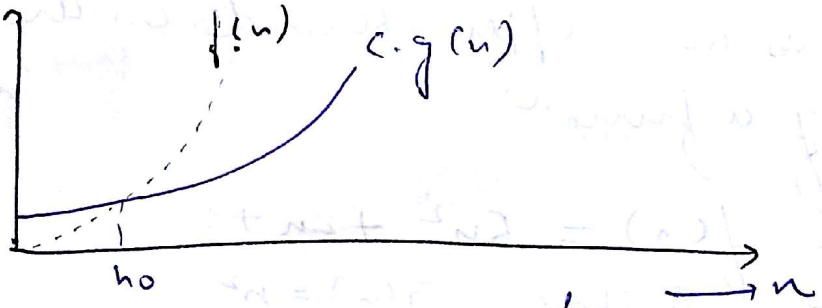
$\equiv O(n^m)$

[Design & Analysis of algorithms.]

2) Ω (omega)

$\Omega(g(n)) = \{ f(n) : \exists$ constant c & n_0

$f(n) \geq c\, g(n), \quad \forall\, n \geq n_0$



$cg(n)$ is a lower bound for $f(n)$

Q. $f(n) = 5n^2 + 2n + 1$ $\quad c = 5$

$g(n) = n^2$

$f(n) = \Omega(g(n))$

$5n^2 + 2n + 1 \geq c \times n^2 \quad \forall\, n \geq n_0$

$2n + 1 \geq 0$

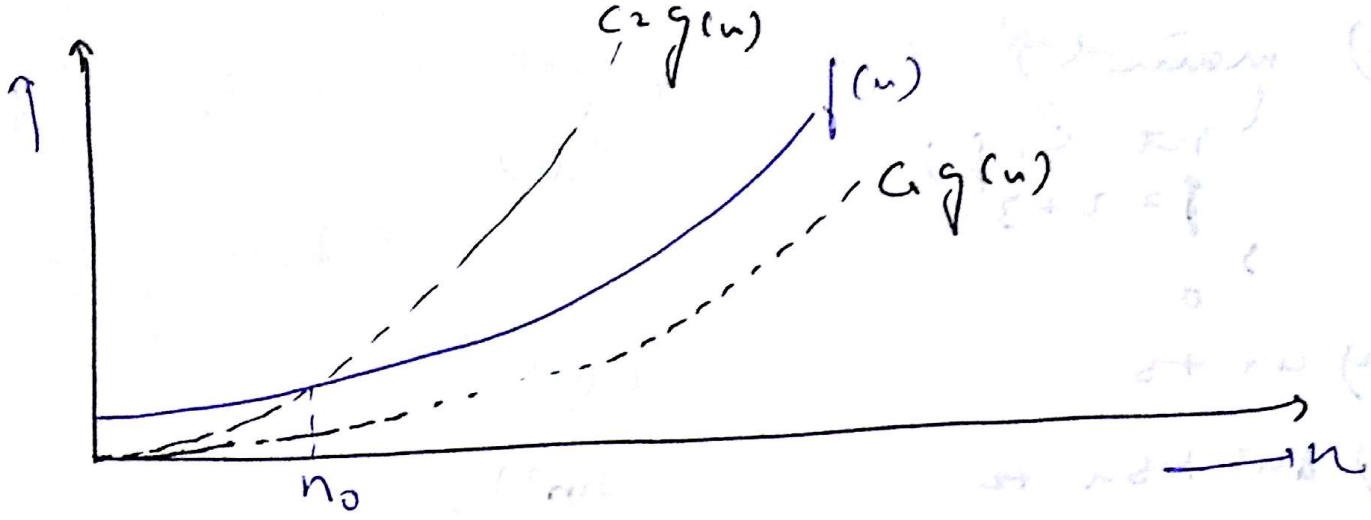
$n \geq -1/2$

$\Rightarrow n_0 = 0$

) ⊛ Theta

⊛ $g(n) = \{ f(n) : \exists$ constants c_1, c_2 & n_0 :

$c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall\, n \geq n_0 \}$



Q. $5n^2 + 2n + 1$

$\Theta(n^2)$

$\forall\ n \geq n_0$

$c_1 n^2 \leq f(n) \leq c_2 n^2$

$5n^2 \leq 5n^2 + 2n + 1 \leq 8n^2$

$\boxed{\forall\ n \geq 1}$

$n \geq -1/2$ $\qquad$ $n \in \left(-\infty, -\frac{1}{3}\right) \cup (1, \infty)$

Q. 1) Prove that $3n+2 = O(n)$
$\Omega(n)$
$\Theta(n)$

2) $6 \cdot 2^n + n^2 \rightarrow O(2^n)$
$\Omega(n^2)$ or $\Omega(2^n)$?
$\Theta(n^2)$ or $\Theta(2^n)$?

3) $10n^2 + 4n + 2$
$O(n^2)$ or $\Omega(n^2)$ or $\Omega(n)$ or $\Theta(1)$?

1) main ()
{ int c, f;
f = 2 + 3
}
0

O(1)

2) an + b

O(n)

3) $an^2 + bn + c$

$O(n^2)$

4) linear search

O(n)

5) main ()
{
int n ; ⟶ c_1
printf ( ) ; ⟶ c_2
scanf ( ) ; ⟶ c_3
while (n > 1) ⟶ $(\log_2 n + 1)$
{ n = n/2 ⟶ $c_5 \times \log_2 n$
}
}

↳ 1 will be executed till we end

Iterations → n, n/2, n/4, n/8, ..., n/2, n/2², n/2^k

kth iteration

at last $\frac{n}{2^k} = 1$

$n = 2^k$

$\log_2(n) = k \log(2)$

$k = \log_2 n$.

Time Complexity ⟹ $O(\log_e n)$

$2^n > n^3 > n^2 > n\log(n) > n > \log(n) > 1$

Q. A)
```
main()
{ int n, i;
  while (n >= 2)
  {
    n = √n
  }
}
```

$\therefore n \longrightarrow n^{1/2} \longrightarrow n^{1/4} \longrightarrow n^{1/8} \longrightarrow n^{1/16}$

$n^{1/2^k} = 2$

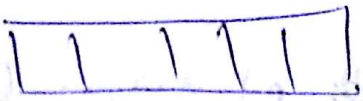
$\log(n) = 2^k \log(2) \Longrightarrow k = \log 2^{n^2}$

B)
```
main()
{ int i, j, n, y, z, n;
  for (i=1; i<=n; i++)
  {
    for (j=1; j<=n; j=2*j)
    {
      n = y + z;
    }
  }
}
```

List

1) Array (Total allocation at the compile time) — space complexity

int a[10] ; O(m)



Insertion

at end O(1)

at start O(n) — different

Deletion O(n) not to leave any empty space

Access O(1)

Traversal O(n) visiting the data items

Search

2) linked list O(m) — no. of nodes at the compile time

Insertion →

at the beginning O(1)

at the end O(n)

Deletion

at the beginning O(1)

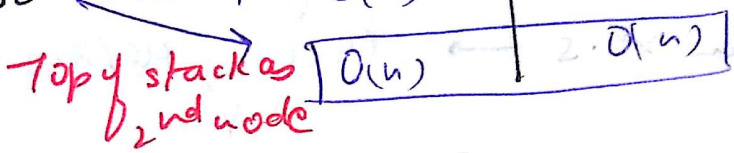
at " end O(n)

Access O(n)

Search O(n)

Stack

	Push	Pop
1) Array	O(1)	O(n)
2) Linked list	O(1)	O(1)



Queue

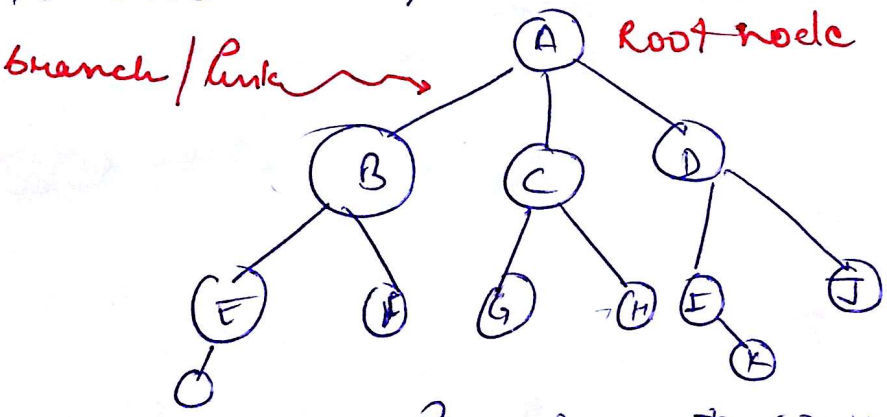
	Enqueue	Dequeue
1.) Linear (Array)	O(1)	O(1)
2) Circular "	O(1)	O(1)
3) Linear (LL)	O(1)	O(1)
4) Circular (LL)	O(1)	O(1)

TREE

Linear D.S ⟶ array, LL, stack, queue, dequ

Non-linear D.S ⟶ tree, graphs.

Tree is a non-linear data structure.
It is a collⁿ of entities called nodes.
which are linked together to simulate a hierarchy.
Each node is used to store the data item.
It can be represented as



- The node which is not connected to any other node is called leaf Node
- All other nodes except leaf node & root node are called internal / intermediate node
- When we go from any node A to node B then A is the ancestor of B & B is the descendent of A.

Ancestors of F = B & A
descendents of D = I, J, K

→ Node A is an ancestor of node B.
Node B " a descendent of " A
if A is either the parent of B / parent of some ancestor of B

→ ~~A node B is left descendent of a node A if B is other the left child~~

Trees:

1) Node
2) Parent & child
3) leaf node
4) Internal / intermediate
5) Ancester & Descendent
6) Sibling.
7) Level of a node
8) Height of a node
9) Depth of a node

- If two nodes have same parent then they are said to be sibling.
- The level of a node is defined - the root of a tree has a level zero. and the level of any other node is one more than the level of its parent.
- length of path from root node to a particular node → Depth

length → no. of link from A to G (2) in above g.

Depth of a root node = 0

- No. of edges in longest path from a node to leaf node → Height

Ht of node D = 0
Ht. of node A = 3

Ht. of leaf node = 0

Ht. of a tree = depth of a tree = Ht. of a root node
The maximum level of any leaf in a tree

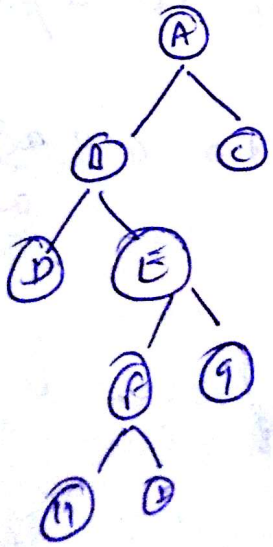
⇒ PROPERTIES OF A TREE →

1) unique path (between any two nodes)
2) NO cycle (exist in case of graph)
3) If tree has n no. of nodes
then no. of edges (links) = n − 1

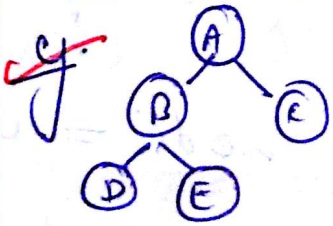
⇒ BINARY TREE (BT)

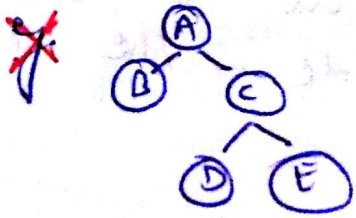
Each node can have almost 2 children

Strict (BT)/2-Tree
A node can have either 0 child / 2 children.
It can't have 1 child



Almost comp. BT
All levels except possibly last level are comp. filled & all nodes must be filled from left - right at each level.





Comp./ full/ Perfect
All the levels are completely filled.
A binary tree with ht : h is called a comp. BT if all levels are completely filled

Q. A strict binary tree containing n no. of leaves has exactly (2n-1) nodes. (Prove)

Sol

⇒ A - almost complete B.T
C - complete B.T.

then C ⊂ A

Q find the ht. of comp. binary tree for level l = m no. of nodes

l+1 = 2m

$l \Rightarrow 2^l$ max. no. of nodes in l level

Proof a complete B.T

Let ht be h, & nodes = n

$n = 2^0 + 2^1 + 2^2 + \dots + 2^h$

$= 2^{h+1} - 1$

$n+1 = 2^{h+1}$

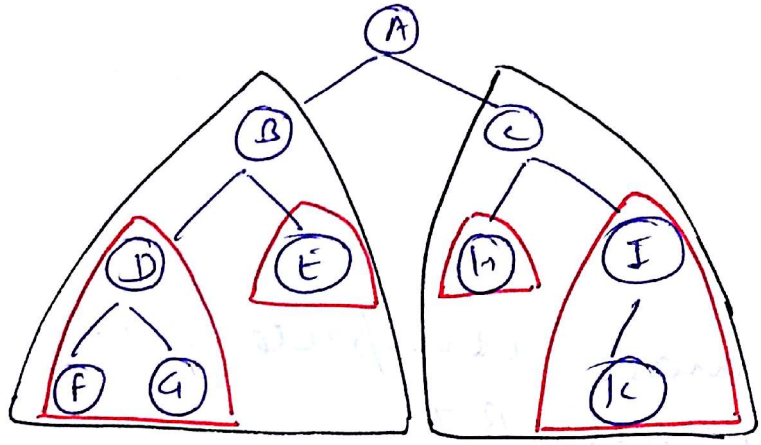
$\log(n+1) = h+1$

$h = \log_2(n+1) - 1$

Binary tree is a recursive data structure

A binary tree is specified by a ~~tree~~ three set.

Root node | left subtree | Right subtree

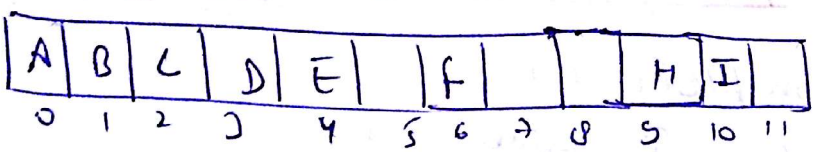


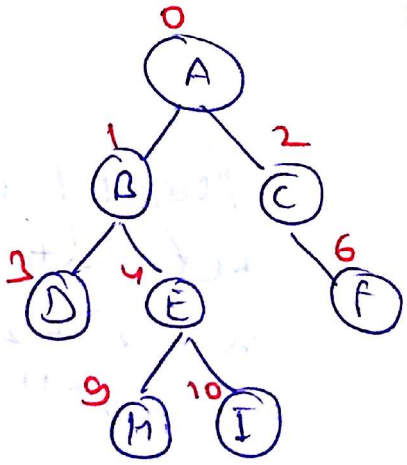
⟹ ~~BINARY SEARCH TREE (BST)~~

Representation of Binary Trees.

Using Array | Using linked list

1) USING ARRAY





→ Static memory allocation
(So can't increase the size of array)

→ Loss of memory space (some array blocks are empty)

* Parent node (i) ← 2i+1 (Left child)
2i+2 (Right child)

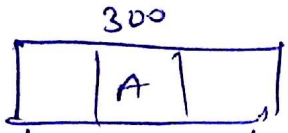
↓ Parent

(i-1)/2

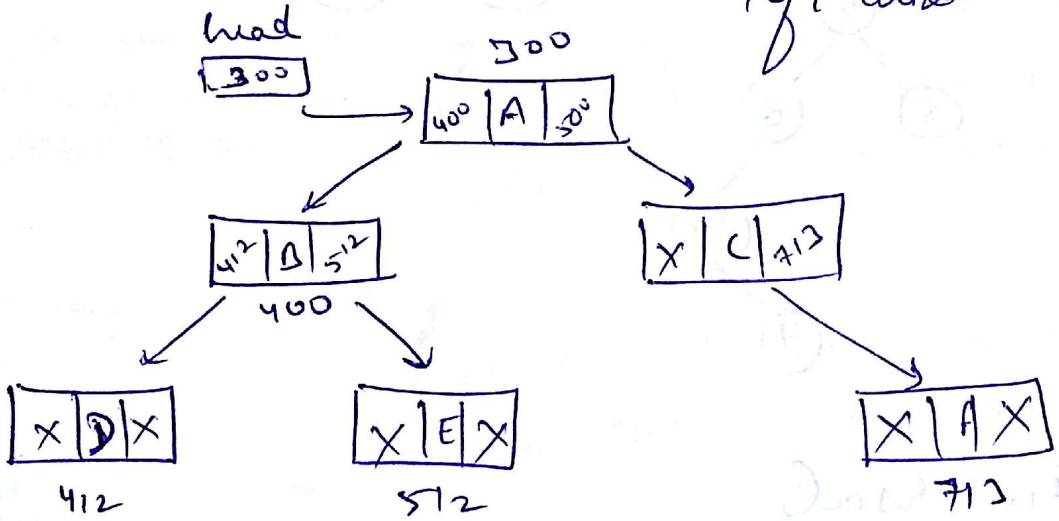
[For complete B-T this array implementation is the best choice.

2) Using linked list

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

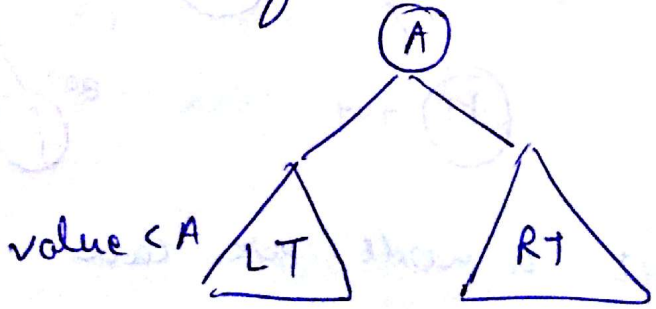


address of left child address of right child



Binary search Tree (BST)

BT which has a property that all the elements in the left subtree of any node are less than the value of m & all the elements on the right sub tree of m are ~~are~~ >= to the value of node m.

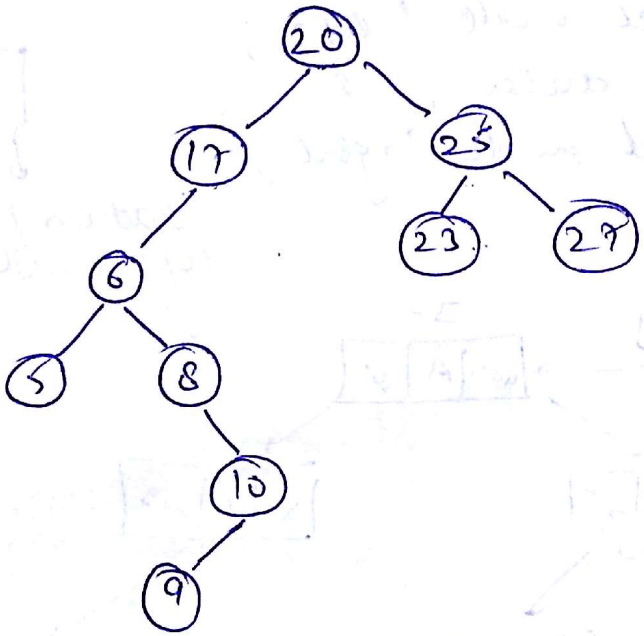


→ Operations

1) Creation / insertion

Create a BST for the following data.

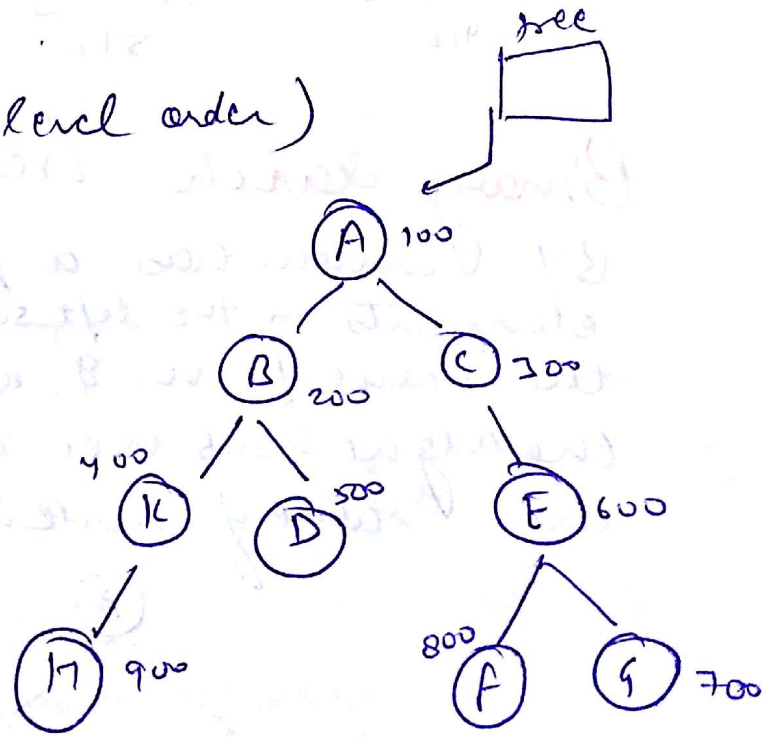
20, 17, 6, 8, 25, 10, 23, 5, 27, 9.



→ Traversal

1) Breadth First (level order)

2) Depth first



* whenever we visit a node we can keep the addresses of all the children in a queue so that we can visit them later.

* A node in a queue is called discovered node whose address is known to us but we have not visited yet.
* The node which is discovered first will be visited / dequeued first.

In above eg. A = Visited node
B, C = Discovered node
K, D, E, G, F, H: Undiscovered node

⇒ In Depth First; we will visit the whole sub-tree of a child before going to the other child. (going in vertical direction) ↓

b) Root - LT - RT (Pre-order)
2) Root - RT - LT
3) LT - Root - RT (In-order)
4) RT - Root - LT
5) LT - RT - Root (Post order)
6) RT - LT - Root

Q. Given Inorder & Pre order sequence of binary tree can be uniquely constructed binary tree.

→ Inorder (stackless SOL^n)

```
Void inorder (struct Bnode *tree)
{
    struct Bnode * p = tree;
    do
    {
        while (p != NULL)
        {
            push (S, p);
            p = p -> left;
        }
        if (! isempty (S))
        {
            p = pop (S);
            printf ("%d", p -> data);
            p = p -> right;
        }
    }
    while (! isempty (S) || p != NULL);
}
main ()
{
    struct Bnode * tree = NULL;
    struct Bnode * s[100];
    inorder (tree);
```

```
void inorderR (struct Bnode * tree)
{
    if (tree != NULL)
    {
        inorderR (tree -> left);
        printf ("%d", tree -> data);
        inorderR (tree -> right);
    }
}
```

In our computer memory there is some special kind of space (implicit stack) i.e. used to complete the function call

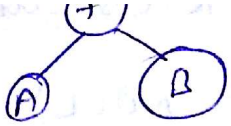
- → Tree
- → Terminology
- → Binary trees
- → Types of binary tree (BT)
- → Represntaⁿ of BT
- → Traversal methods and their recursion and iterative implementaⁿ.
- → Exprsⁿ tree.

Expression Tree

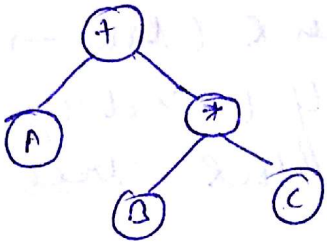
An expression of operands & operators can be represented by a binary B.T. The root of that binary B.T contains an operator i.e to be applied to the results of other expressions represented by lft & right subtree.

The operators are represented by internal nodes
The operands " " " leaf "

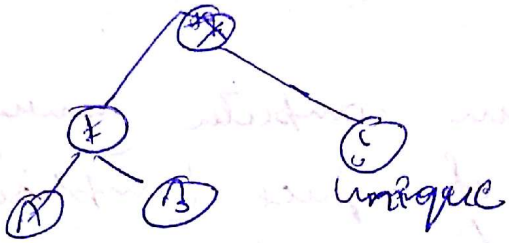
1) A + B



2) A + B * C



3) (A + B) * C



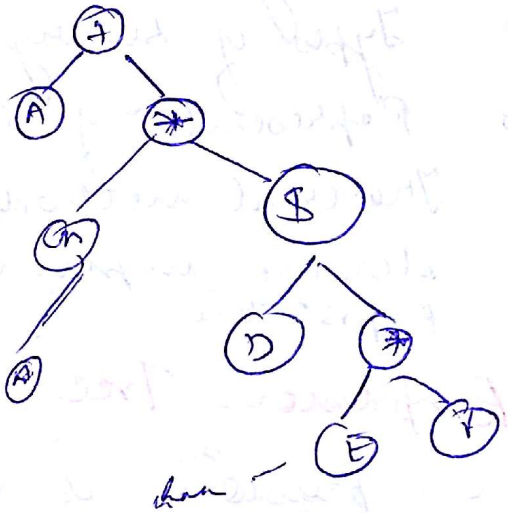
4) $: (power)

A $ B = A^B

A ^ B = AB

power (A, B) = A^B

5) A + (B - C) * D $ (E * F)



6) ~~correct~~

A + (B - C) * D $ (E * F)

Pre-Order : + A * - B C $ D * E F

Post order : A B C - D E F * $ * +

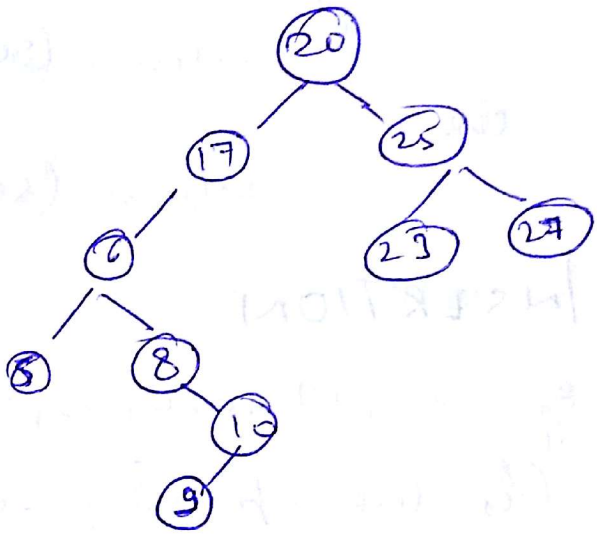
The Pre-Order traversal of an expression tree always results in its prefix expression.

BINARY SEARCH TREE

Operation

1) Search:

~~BST Nodes~~



Time complexity = O(h)
h = height of tree = n-1

a) Iterative

```
struct Bnode * search_Bst (struct Bnode *tree, int key)
{
while (tree != NULL && key != tree->data)
{ if (key < tree->data)
    tree = tree->left;
  else
    tree = tree->right;
}
return tree;
}
```

the item which we want to search.

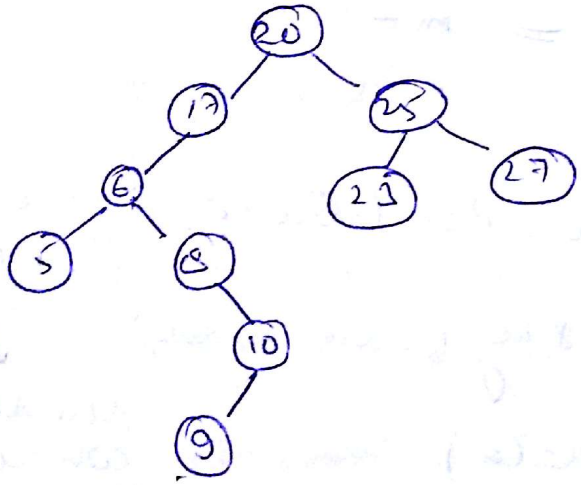
b) Recursive

```
struct Bnode *Search_BST1(struct Bnode *tree, int key)
{
  if (tree==NULL || key==tree->data)
      return tree;
  if (key<tree->data)
      return (Search_BST1(tree->left, key));
  else
      return (Search_BST1(tree->right, key));
```

2 INSERTION

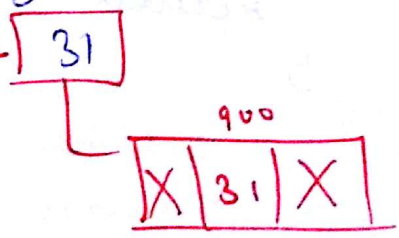
Eg 20, 17, 6, 8, 25, 10, 23, 5, 27, 9.

(for the upcoming value, compare that value with the parent root)



a) Iterative

```
void insert_BST (struct Bnode *tree, int key)
{
  struct Bnode *z, *y, *x;
  z = (struct node*) malloc (sizeof(struct node))
  y = NULL;
  x = tree;
  while (x != NULL)
  {
```



```
if (key < x->data)
    x = x->left
else
    x = x->right;
}
if (y == NULL)
    tree = z;
else if (key < y->data)
    y->left = z;
else
    y->right = z;
}
```

```
z = ( ) malloc(
z->left = NULL
z->right = NULL
z->data = key;
);
```

5) Recursive

```
void insert_bst1 (
{
    if (tree == NULL)
        tree = z;
    else if (key < tree->data)
        insert_bst_1 (tree->left, key)
    else (key > tree->data)
        insert_bst_1 (tree->right, key)
}
```

→ Time complexity of search func^n depends on ht. of tree.

→ Shape of the bst is dependent on the order of values or data items given in the input sequence.

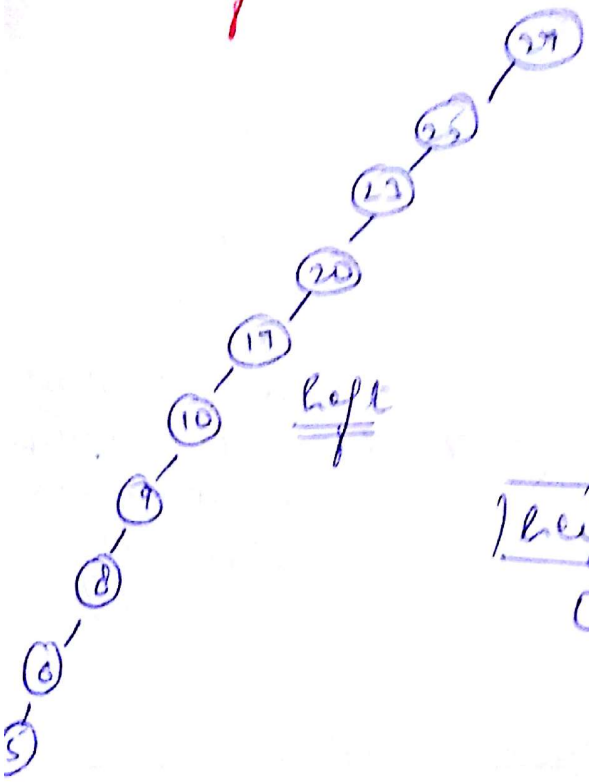
→ 5, 6, 8, 9, 10, 17, 20, 23, 25, 27

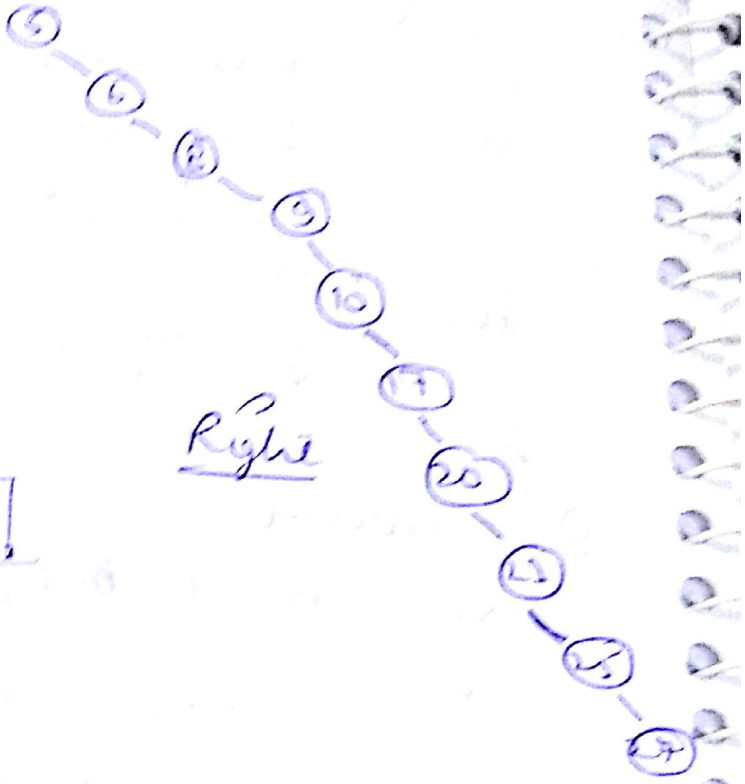
Right skewed tree

] 10 elements

→ 27, 25, 23, 20, 17, 10, 9, 8, 6, 5

Left skewed tree





Height = 9

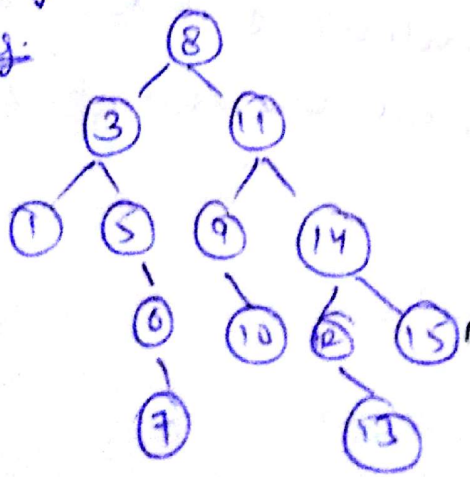
O(n)

3 DELETION

* The inorder traversal of a BST result in the ascending order sequence.

→ If the node to be deleted has no child then it may be deleted without further adjustment to the tree..

Eg:



→ leaf node (just simply disconnect)

→ 1/ the node to be deleted has only one subtree then it's only child can be moved up to take it's place.

eg.