Data structures through C

Algorithm
- Arrays
- Stack
- Queue
- Linked lists
- Trees
- Graphs

Abstract data types

Set of rules.

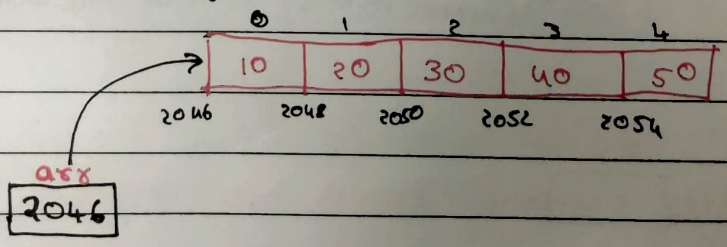
Data Structures / Algorithm

Linear
- Arrays
- Stacks
- Queues
- Linked lists

Non-linear
- Trees
- Graphs

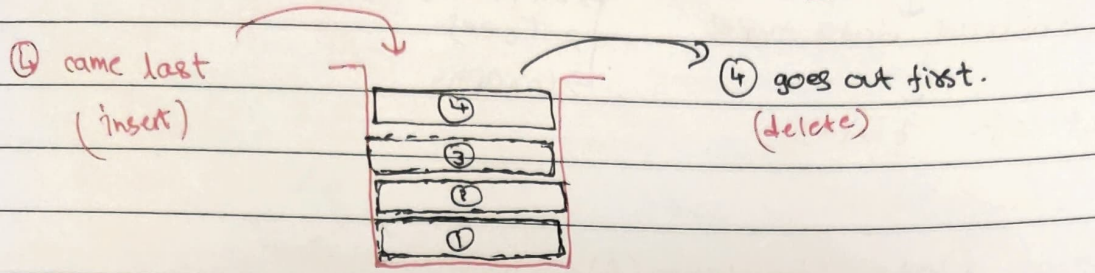
Array:
- It holds more than '1' elements
- It holds homogenious elements only (same data type)
- Index based.
- Contiguous Memory Allocation.
- Derived datatype
- It doesn't follow any algorithm.

```
int arr[5] = { 10, 20, 30, 40, 50 }
```



STACK: • It is an algorithm to store the information

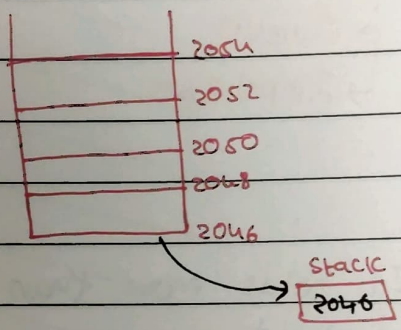
• Last In First Out [LIFO]



• Every stack should have a particular amount of memory block (capacity of stack) → static Allocation
→ Dynamic Allocation

↳ static Allocation:-

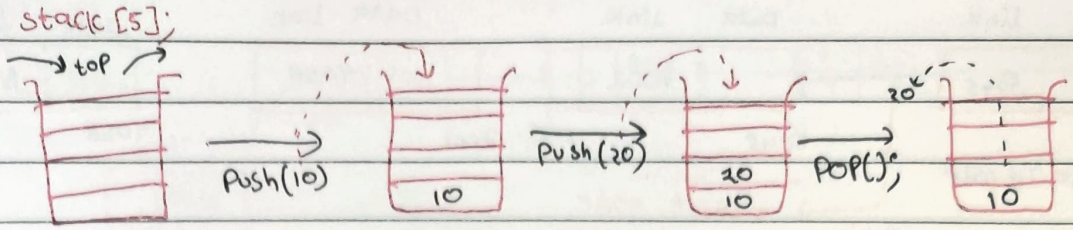
```
int stack[5] ⟶
```



↳ Dynamically Allocation:-

```
int* stack;
    = (int*)malloc(capacity, sizeof(int));
        ↓ returns
    (void*) / Generic Pointer → can be type casted to any
```

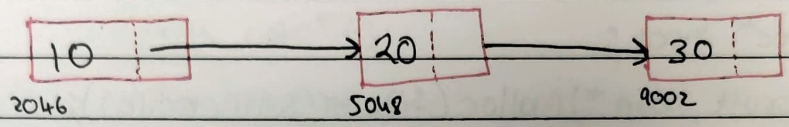
Stack operations:- ① creation of stack
② Push the elements into the stack (int element);
③ Pop();
④ traverse(); → Display all the elements in stack
⑤ isEmpty(); → is it empty or does it contains elements
⑥ isFull(); → is it full or not
⑦ Size(); → Size/elements in stack.



Linked List → always dynamic

- Dynamic memory allocation collection

→ Insertion and Deletion will be easy Here compared to Arrays stack



Types of linked lists
- Single linked list
- Double linked list
- Circular linked list.

Single linked list

node → 2 fields

Data	*link

Double linked list

node

*Llink	Data	*Rlink

Left side link. Right side link.

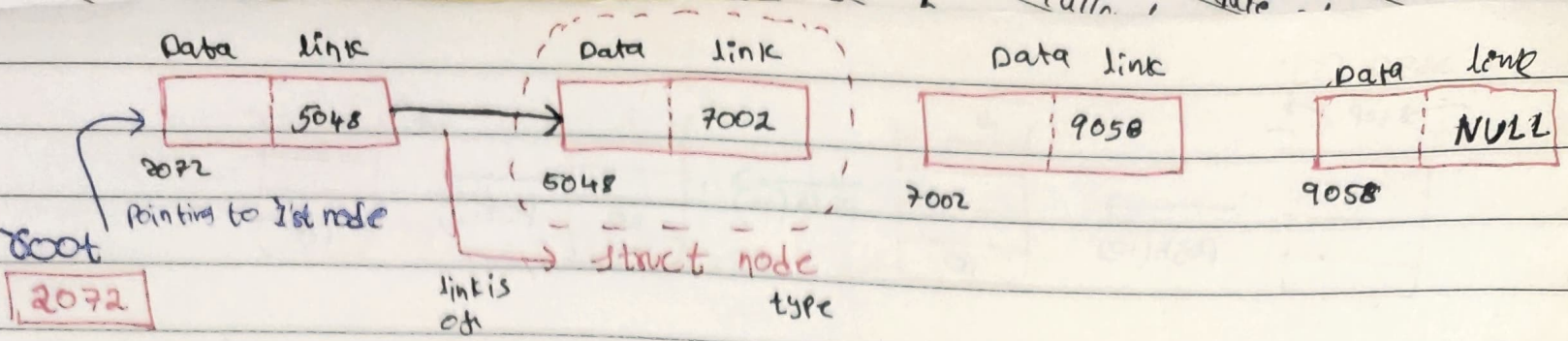
3 Fields

circular linked list

node

*left	Data	*Right

3 Fields.



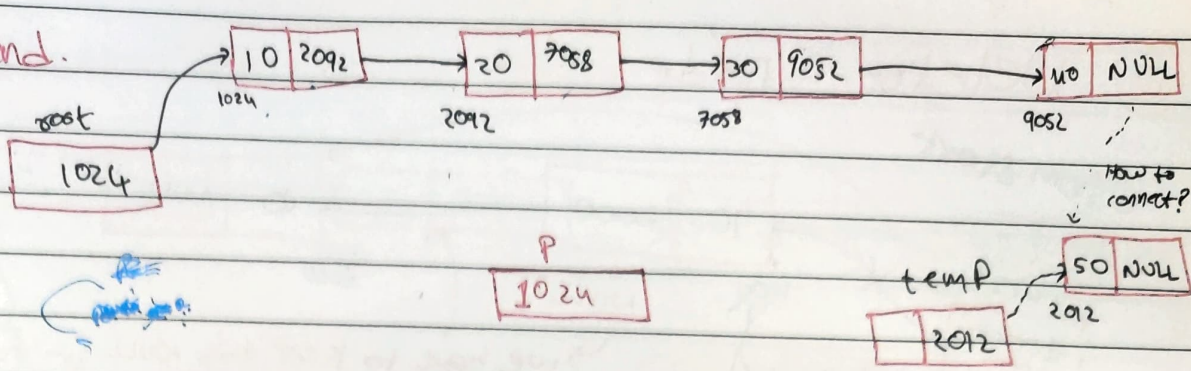
16 bit Pointer size → 2 bytes
32 bit Pointer size → 4 bytes

```
Struct node
{
    int data;
    struct node *link;
};
Struct node* root;
root = (struct node*) malloc(sizeof(Struct node));
```

Single linked list operations.

① → Append [add at end]
② → Add at begin
③ → Add at After (in the middle)
④ → Delete first node
⑤ → Delete specified node
⑥ → Display elements
⑦ → Length
⑧ → How to Reverse the list
⑨ → Swap 2 nodes
⑩ → Sort elements

① Append.

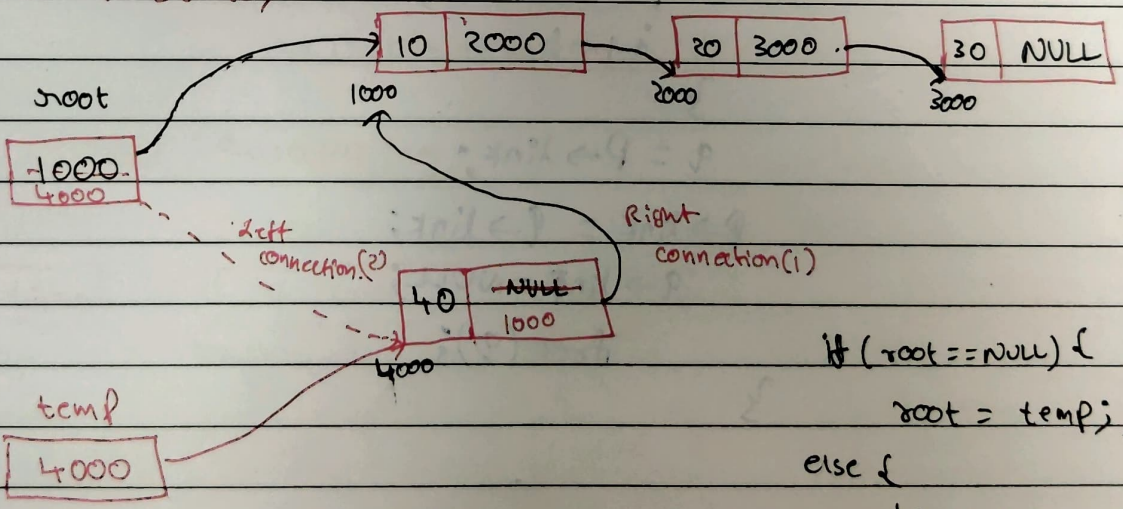



```
if (root = NULL) {
        root = temp;
    }
else {
        struct node* p;
        p = root;
        while (p->link != NULL)
            {
                p = p->link;
            }
        p->link = temp;
```

NOTE: Here temp is pointer pointing to node

→ p->link (2092)
nxt iteration p->link (7058)
nxt iterate p->link (9052)

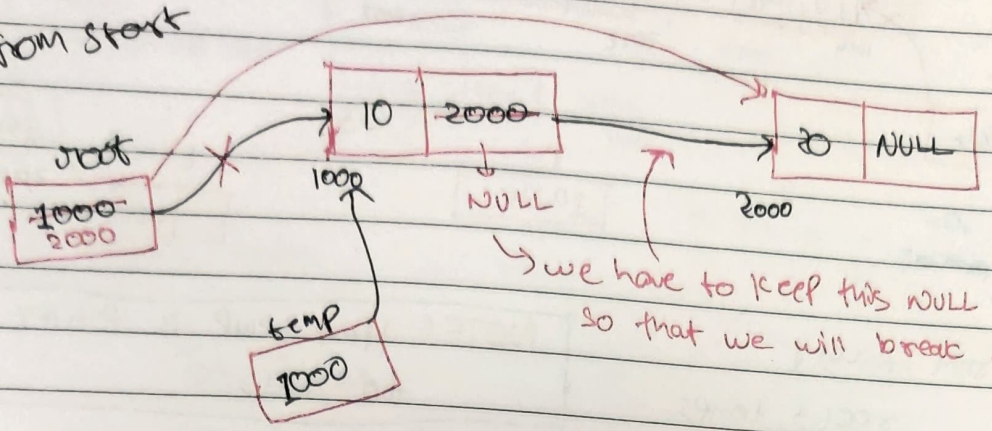
② Append at Start,



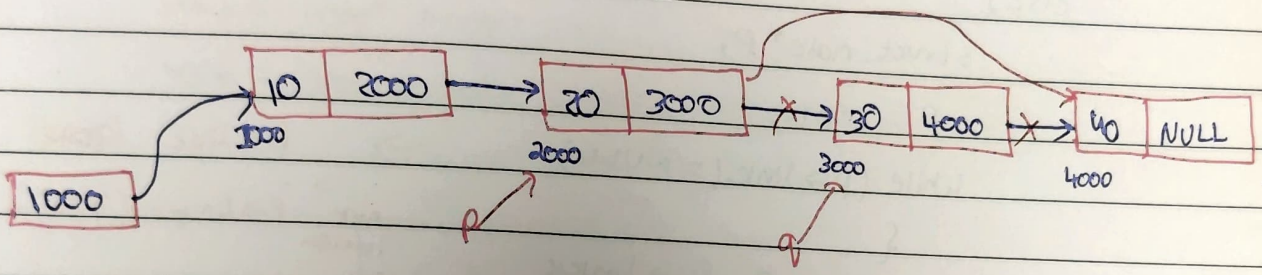

```
if (root == NULL) {
    root = temp; }
else {
    temp->link = root; (1)
    root = temp; (2)
}
```

③ Delete a node

① from start

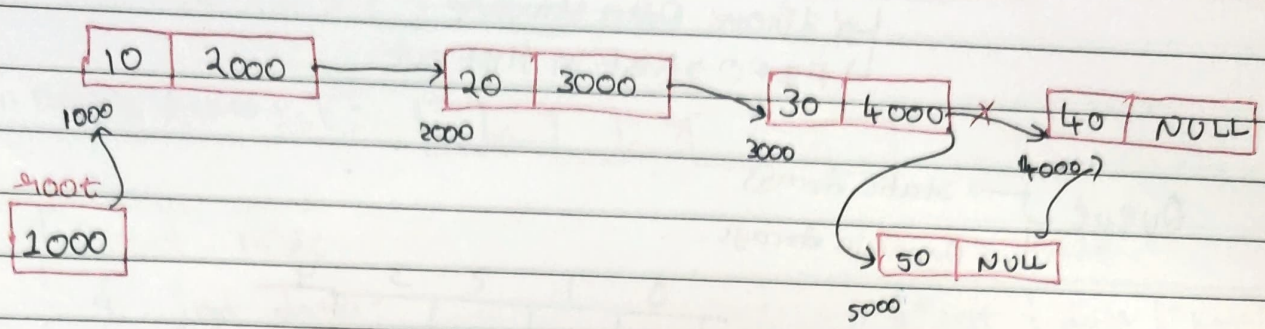


② from middle (or) end.



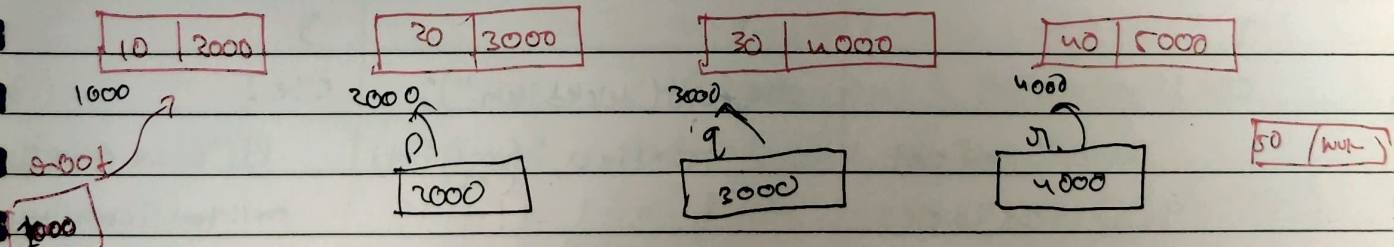

```
else {
    struct node *p = root, *q;
    int i = 1;
    while (i < loc-1)
    { p = p->link;
      i++;
    }
    q = p->link;
  p->link = q->link;
    q->link = NULL;
      free(q);
}
```

④ Insertion of Node in Single Linked List




```
Void addafter( ){
 Struct node *temp,*P;
 inc loc,len,i=1;
 Printf("Enter location : ");
 scanf("%d", loc);
 len = length();
 if(loc > len) { Pf("Invalid location \n"); }
else
{ P = root;
 while(i<loc) { P = P->link; i++; }
 temp = (struct node*) malloc(sizeof(struct node));
 → (Read node data & store NULL value)
 temp->link = P->link;  (Right)
   P->link = temp;      (left)
```

Node Swapping



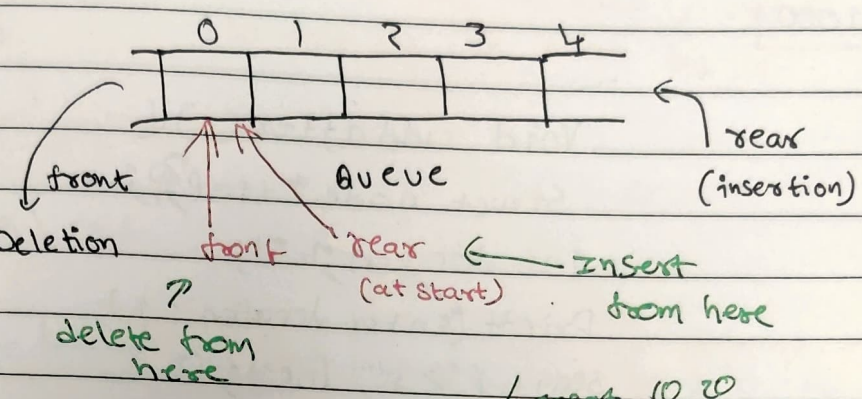

```
q->next = r->next;
r->next = q;
P->next = r;
```

Queue

↳ Linear Data structure
↳ FIFO → first in first out

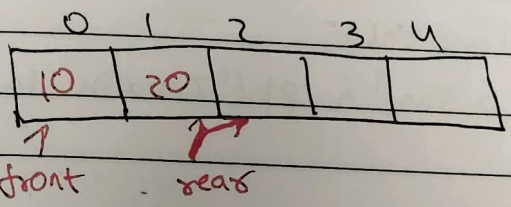
Queue ↳ Static Arrays
↳ Dynamic Arrays

int queue [5];

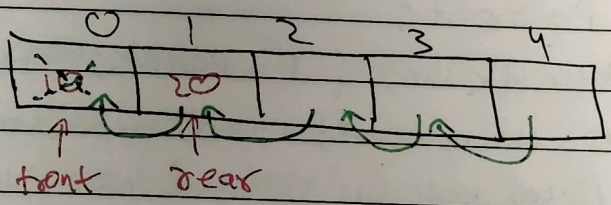


insert 10, 20

Insert →



Delete →
(10)



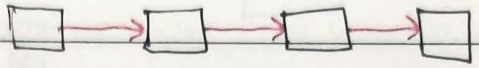

```
Void insert()
{
   if (CAPACITY==rear) {
     Pf("Queue is full \n");
   }
   else{
        int ele; (read ele)
      queue[rear]=ele;
        rear++;
    }
}
```

```
Void delete()
{
if (front==rear) {
    Pf ("Queue is empty \n");
  }
else{ Pf("deleted item");
    for(i=0;i<(rear-1);i++)
    { q[i]=q[i+1];
    } } rear--;
}
```

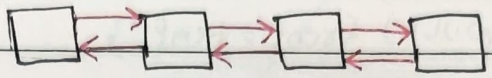
```
Void traverse()
{
if (front==rear){
Pf("Queue is empty \n");
}
else{
   Pf("Queue elements");
for(i=front;i<rear;i++)
{ Pf("%d", queue[i]);
}
```

Double Linked List

In single linked list :-



In Double linked list :-

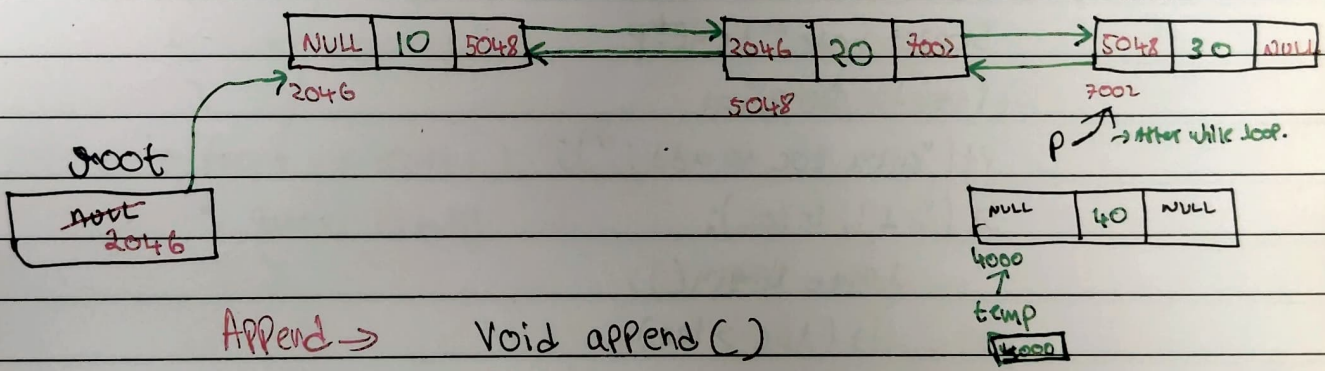


```
struct node
{ int data;
  struct node *left;
  struct node *right;
};
  struct node *root = NULL;
```

node

*left	data	*Right

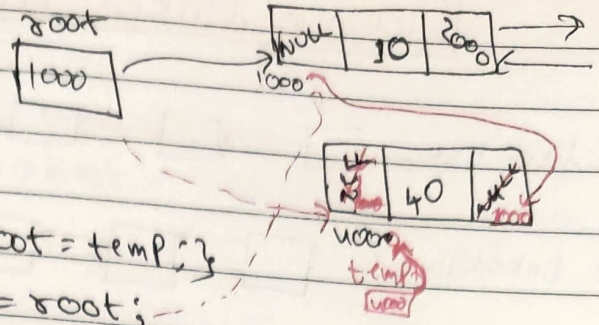
→ first node left link is NULL & last node Right link is NULL



Append → Void append()

```
{ struct node *temp;
    temp = (struct node*) malloc (size of(struct node));
  Pf("enter node data :");
  sf ("%d", &temp→data);
  temp→left = NULL; temp→Right = NULL;
  if (root == NULL) { root = temp; }
  else { struct node *p; p = root;
    while (p→right != NULL) { p = p→right; }
    p→right = temp;
    temp→left = p;
  }
}
```

AddAtStart ()




```
if(root=NULL) {root=temp;}
else { temp->right = root;
       root->left = temp;
       root = temp;
     }
}
```

Add → After a node/at end.

```
Void addAtafter() {
  Struct node *temp,*p
  int loc,leng,i=1
  Pf("enter loc to add:");
  Sf("%d", &loc);
     leng = length();
     if(loc > len) {
       Pf("Invalid location"); }
     else {
       temp = (struct node*) malloc(size of (struct node));
       ->(read data)
       temp->left = NULL;
       temp->right = NULL; p = root
       while (i<loc) {
       { p=root;
           p = p->right; i++;
         }
       temp = p->right;
```

★ `p->right->left = temp;`

`temp->left = p; . p->right = temp;`

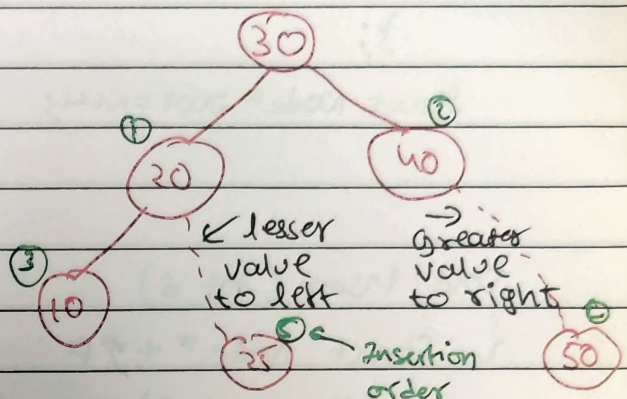
Binary Search Tree

↳ non-linear data structure

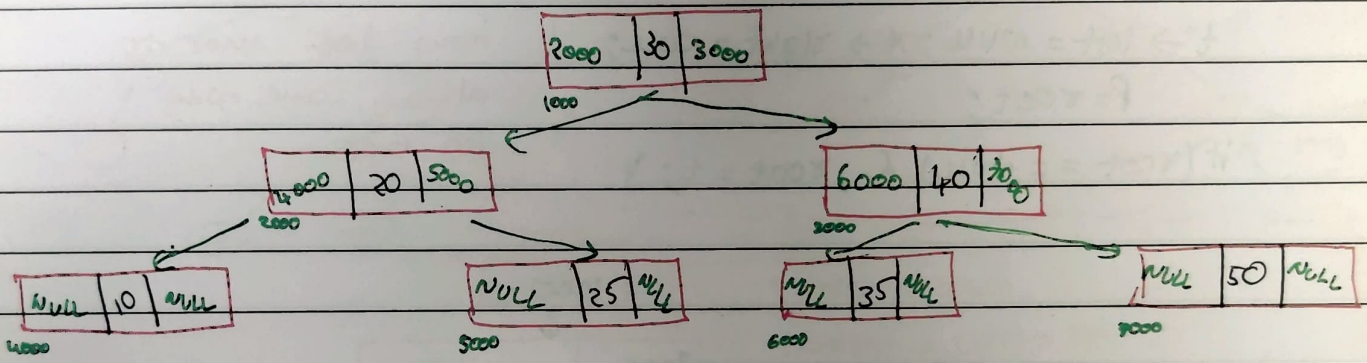
↳ node based data structure

Left child	data	Right child

```
Struct Node {
    int data;
    Struct Node * left;
    Struct Node * right;
};
```



- A BST having 'n' nodes contains (n+1) null nodes



7 → nodes, (n+1) → 8 null nodes

Operations → ① Insert
② delete
③ Traverse → In-order Traversal
→ Pre-order Traversal
→ Post-order Traversal

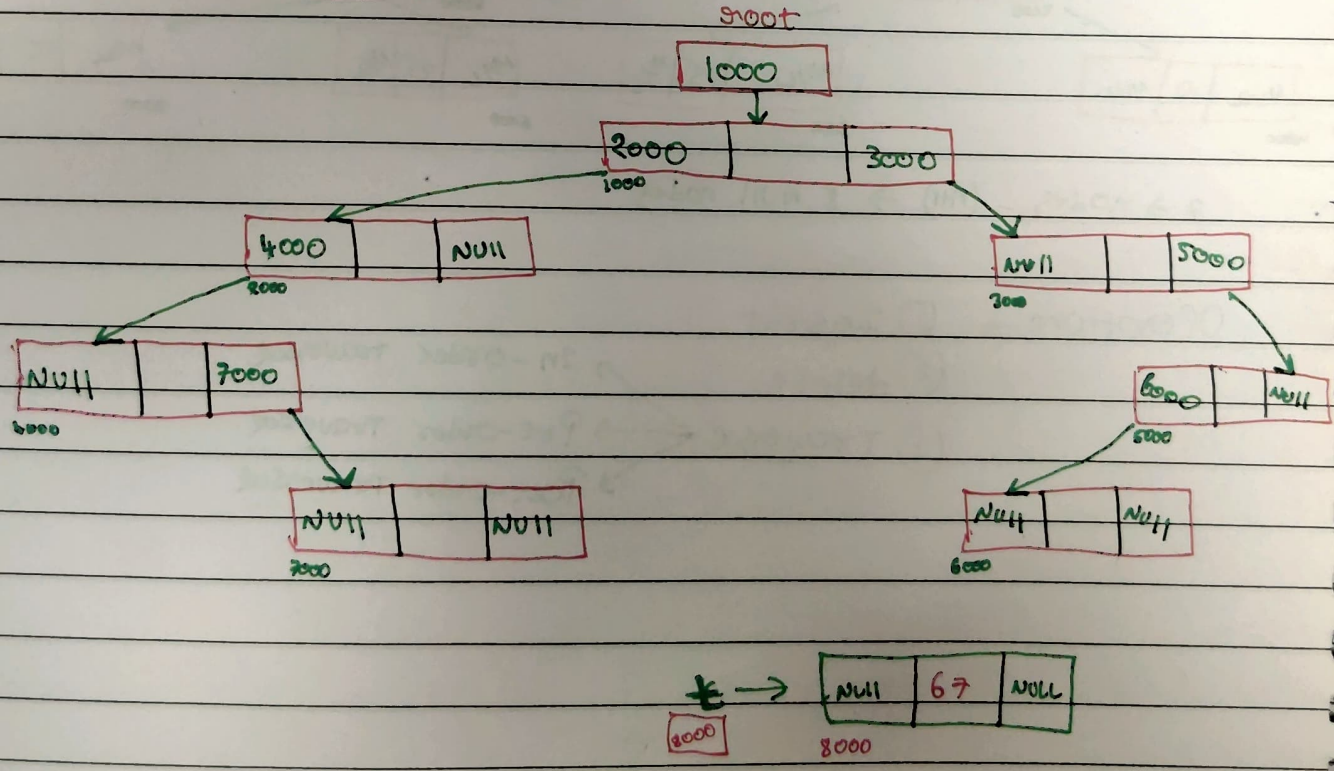
Insert Elements

```
Struct Node {
    int data;
    Struct Node *left;
    struct node *right;
};
Struct Node* root = NULL;
```

root
NULL

```
Void insert (int d)
{ Struct node* t,*p
  t = (Struct node*) malloc (sizeof(Struct node));
  t.data = d;
  t->left = NULL; t->right = NULL;
     P = root;
  if (root == NULL) { root = t; }
```

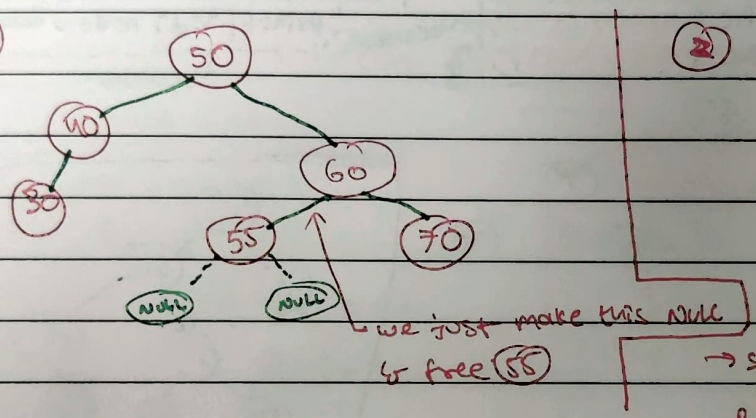
now left then is already some node in tree



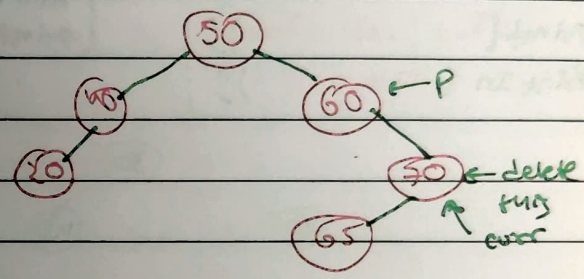
```
else {
struct node * curr;
   curr = root;
   while (curr) { P = curr;
   if (t -> data > curr -> data)
     { curr = curr -> right; }
   else { curr = curr -> left; } }
if ( t -> data > P -> data)
   { P -> right = t; }
else { P -> left = t; }
```

Deletion in BST

(1)



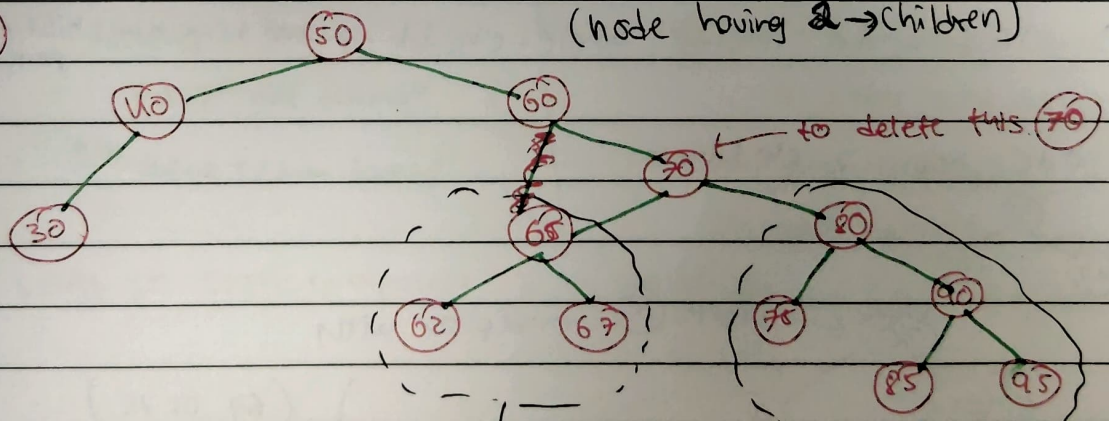
(2)



→ since 65 > 60

P->right = curr -> left

(3) (node having 2 → children)



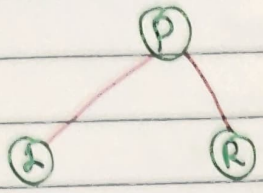
Highest value (or) of left sub tree (67)

least value of right sub tree (75)

Should replace the (70)

Displaying Elements in BST

In-order:



Print:-

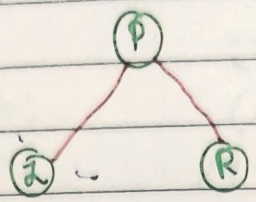
Left → Parent → right

→ 20, 30, 40, 50, 60, 70, 90

↳ Ascending order

```
Void Print In (struct node * node)
{ if (node = NULL)
    return ;
  PrintIn ( node → left);
  Printf("%d", node → data);
  Print In ( node → right);
```

Pre-order



Print:-

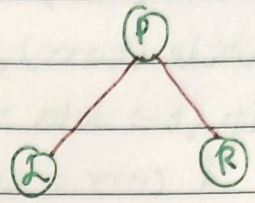
Parent → Left → right

→ 50, 30, 20, 40, 70, 60, 90

```
Void Print Pre ( struct node* node)
{ if (node = = NULL)
    return ;
  Printf ("%d", node → data);
  Print Pre ( node → left);
  PrintPre (node → right);
}
```

recursion

Post-order



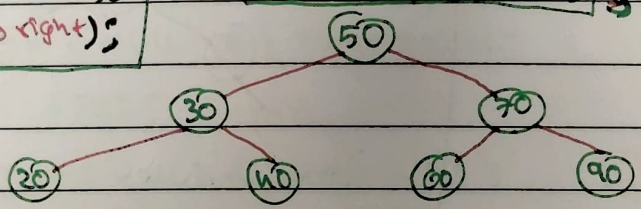
Print:-

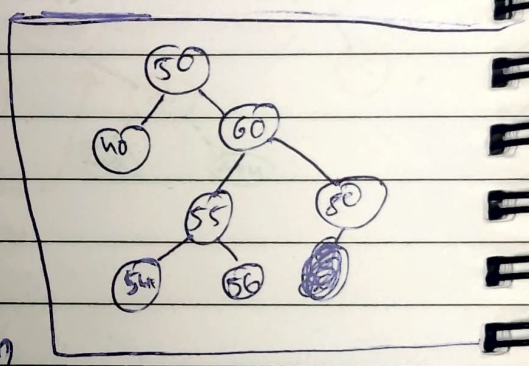
Left → right → Parent

→ 20, 40, 30, 60, 90, 70, 50

```
void Print Post ( struct node *node);
  if (node = = NULL)
    return ;
  Print Post (node → left);
  PrintPost (node → right);
  Printf("%d", node → data);
```

recursion.



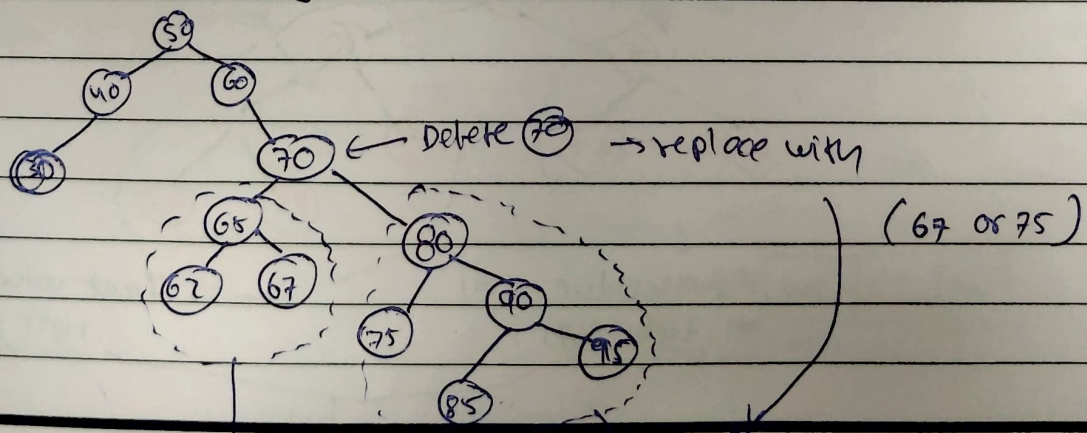


Deletion

(1) → leaf node having NO-children

(2) → leaf node having single-child → add remaining child to parent

(3) → leaf node having 2-children



Searching & Sorting

Time complexity

Searching
- Linear search ---------- O(n)
- Binary search
  - recursive ---------- O(log₂n)
  - non-recursive ----------

Sorting
- Bubble sort
- Selection sort
- Insertion sort
  - Time complexity: O(n²)
- Quick sort
- merg sort
- ~~Radix sort~~
- Heap sort
  - O(nlogn)
- ~~Best sorting technique~~
- Count sort
  - O(n)
- Radix sort
  - O(d*(n+k))
  - max place
  - length of array

Radix sort is the Best sorting technique

Search → Linear search.

```
flag = 0;
for i = 1 to 7
{  if (A[i] == Element)
     { loc = i;
       flag = 1 & Break;
     }
}
if flag = 0, then
   Printf" not found"
else Print "Item found"
```

* works on static & dynamic

Binary search

* condition elements must be sorted
* works only on static

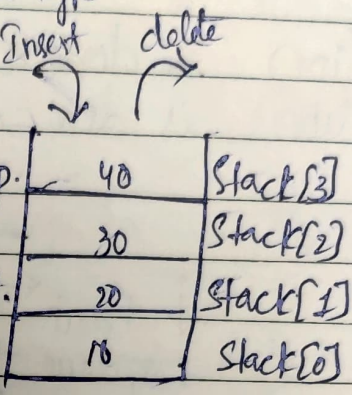
Recursive:

```
void BinSrch(A, ITEM, Beg, End){
① { if Beg > End then
      → Item not found
② → mid = (int)((Beg + end)/2)
if(ITEM == A[mid]) then
      → Item found → exit
else if (ITEM < A[mid]) then
      BinSrch (A, ITEM, Beg, MID-1)
else  BinSrch (A, ITEM, MID-1, End)
```

* Stack

↳ It is a linear Data Structure
- It Follows LIFO (last In First Out)
- Insertion and Deletion takes place from the top
- All elements should be of same type

- * Major Operations



- Push() → Used to insert element at top.
- Pop() → Removes the top most element.
- Peek() → Retuen the top most elemnt without removing.
- isEmpty() → Returns if stack is Empty
- isFull → Returns if stack is Full.

* Push
↳ increase top (top++)
→ Add the element

* Pop
↳ Delete the element
- Decrease top (top--)

* is Empty
↳ no elements in stack (UNDERFLOW)

* isFull
↳ If the Stack is Full (OVERFLOW)

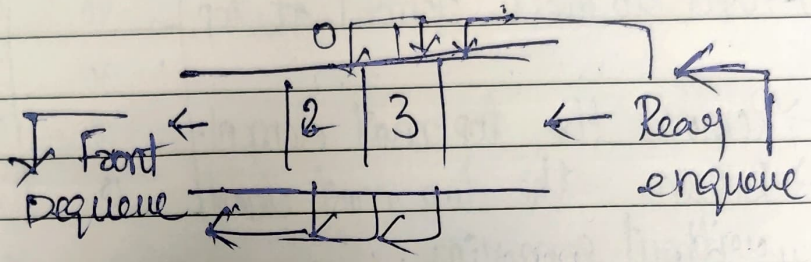
Applications
↳ Reverse a String
- Undo
- etc.

→ O(1)

Queue

↳ Linear Data Structure.
- Abstract Data type
→ Follows FIFO (First In First out)

Rules:
Insertion is done at Rear (or) tail → enqueue
Deletion is done at Front (or) head → dequeue.



Operations

- Enqueue(2) → Adding data to Queue
- Dequeue() → Deleting data from Queue.
- front() / Peek() → Element Front of the Queue.
- IsFull() → Returns True if Queue is full.
- isEmpty() → If Queue is empty it returns true.

front = rear = -1 → Queue is Empty → Underflow

* Enqueue() → Increase the rear (rear = rear+1)
And Insert the element

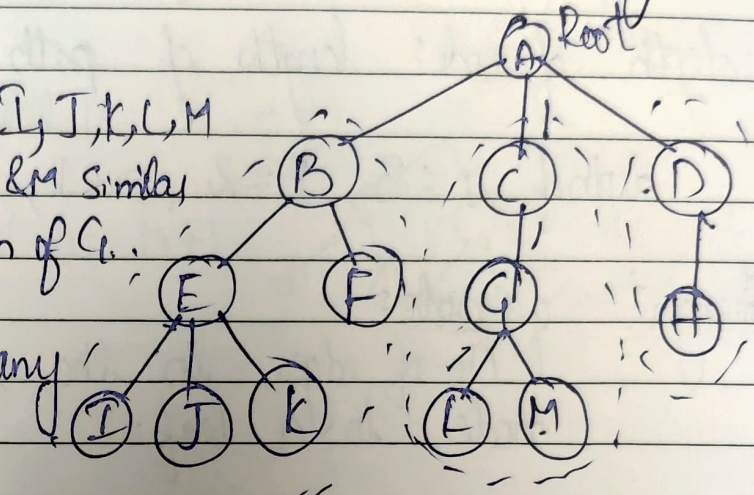
* dequeue() →
front = front + 1

* Rear = Max-1 → isFull. → Overflow

* Trees

↳ It is a non-linear Data Structure.

→ Tree can be defined as a collection of entities (nodes) linked together to simulate a hierarchy.



→ Root = A

→ nodes = A, B, C, D, E, F, G, H, I, J, K, L, M

→ Parent nodes = G is parent of L & M similar

→ Child node = L & M are children of G.

→ external node

→ Leaft node → not having any child.

I, J, K, F, L, M, H

→ Non-leaf node → Having atleast one child. (Internal nodes) A, B, C, D, E, G.

→ Path : Sequence of consecutive edges from source node to destination node.

→ Ancestor : Any predecessor node on the path from root to that node.

ancestor of L → A, C, G

H → A, D

→ decendant : Any successor node on the path from that node to leaf node.

decendant of C is

C = G, L, M.

B = E, F, I, J, K

→ Sibling: Children of Same Parent

→ degree: degree of node is no. of children of that node.

degree of tree: Max degree among all nodes

→ depth of node: length of path from root to that node.

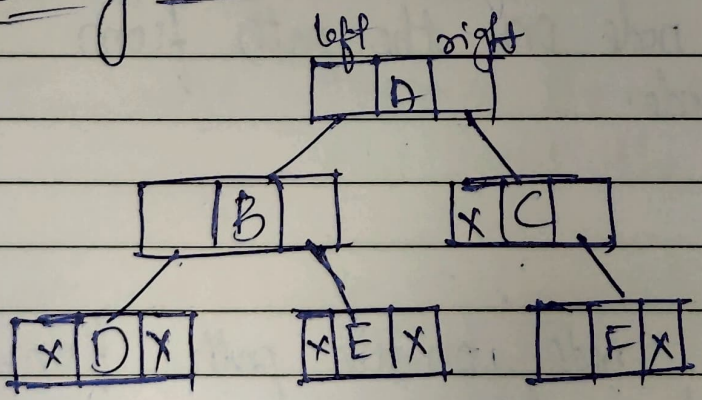
depth of J: 3, F=2, D=1, A=0

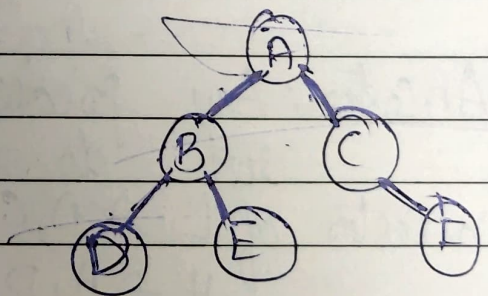
→ Height of node:
no. of edges in the longest path from that node to a leaf.

→ Level of node:
It is same as the height of a tree.

* nodes:
n nodes - (n-1) edges → Condition of a tree.

* Binary Trees






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

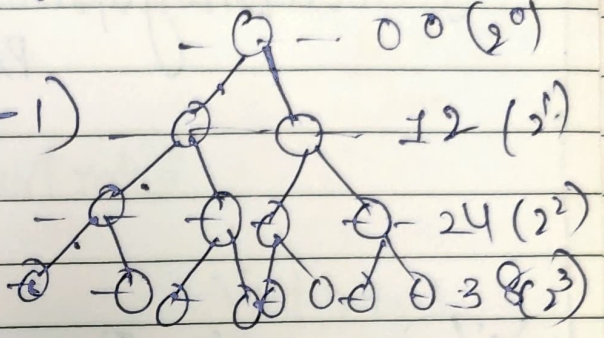
* Binary Tree

- It is a tree each node can have ~~max~~ atmost of 2 children.

→ Max no. of nodes possible at any level (i) is 2^i

max no. of nodes of height h = $(2^{h+1} - 1)$

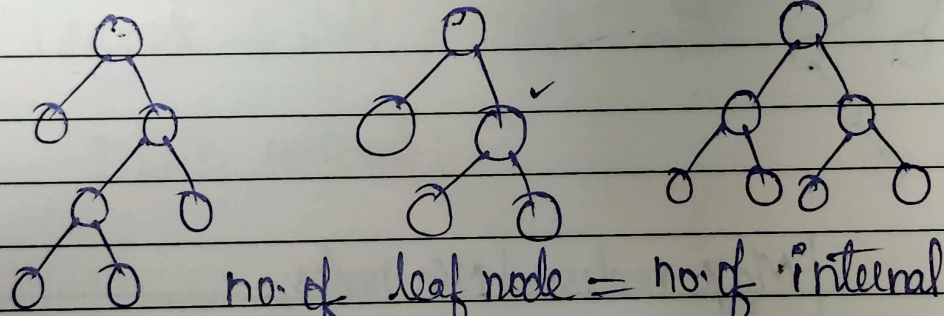
min no. of nodes of height h = h+1



Types of Binary Tree

→ Full/Proper/Strict binary Tree

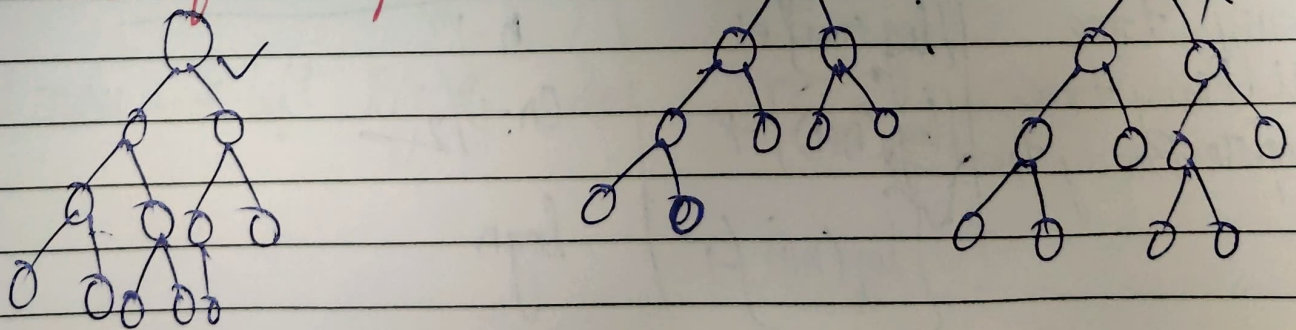
each node have either "0" (or) "2" children



no. of leaf node = no. of internal node + 1

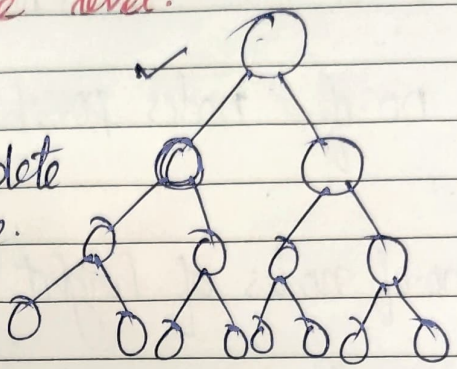
→ Complete Binary Tree

all levels are completely filled (except possibly the last node level) and the last level has nodes as left as possible..



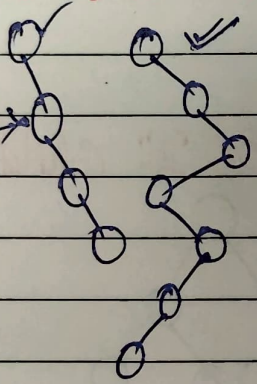
→ Perfect Binary tree
all internal nodes have 2 children & all leaves are at same level.

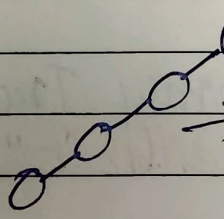
Every perfect Binary Tree is a Full Binary Tree and Complete Binary tree.



→ Degenerate Binary tree
all the internal nodes are having only 1 child.

Right skewed binary tree



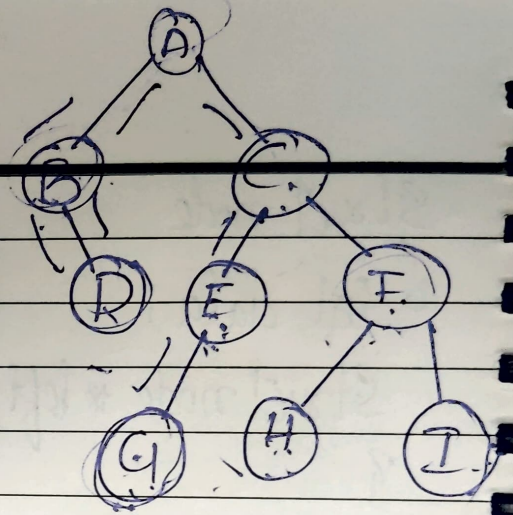


→ Left skewed binary tree.

	Max nodes	Min nodes
Binary tree	$2^{h+1}-1$	$h+1$
Full Binary tree	$2^{h+1}-1$	$2h+1$
Complete Binary tree	$2^{h+1}-1$	2^h

	Max height	Min nodes
Binary tree	$\lceil \log_2(n+1) \rceil + 1$	$h-1$
Full Binary tree	$\lceil \log_2(n+1) \rceil - 1$	$(n-1)/2$
Complete Binary tree	$\lceil \log_2(n+1) \rceil - 1$	$\log n$

*InOrder: BDAGECHFI Left Root Right



* Pre Order:
ABDCEGFHI Root Left Right

* Post Order
DBGEHIFCA Left Right Root