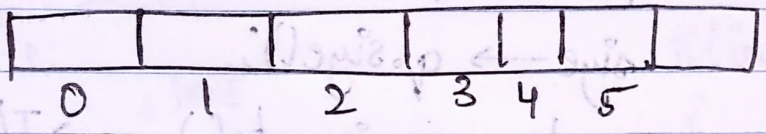
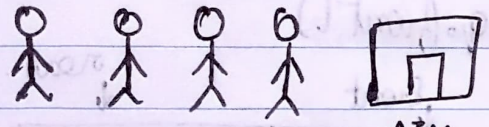
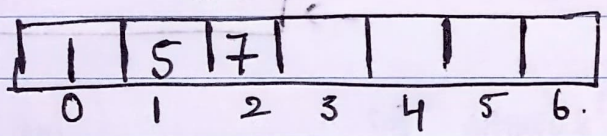
## Lec: 60:- Queue

Queue → Is a type of data structure where FIFO is followed.




FIFO → First in First Out.



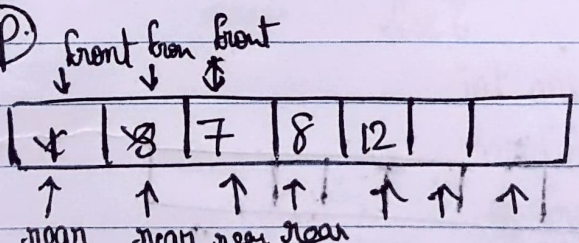




queue.push(1)
.push(5)
.push(8)

FIFO

pop → FIFO

pop




push() → front
pop(), pop() → rear

push() → rear.
pop() → front.

queue → push/insert
pop/remove
→ size
→ isempty
} operation.

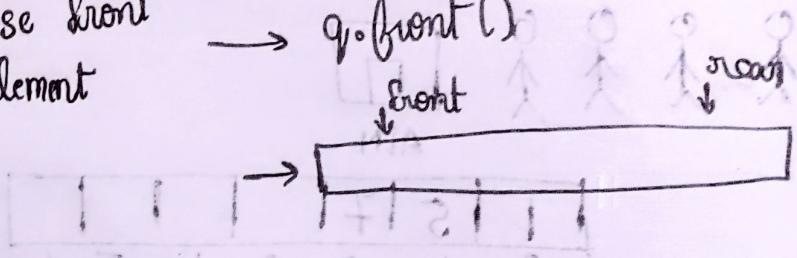
# STL Queue

push → insert
pop → remove.

create → queue<int> q;

push → q.push(17);
pop → q.pop(17);
size → q.size();
isempty → q.isempty() → T/F

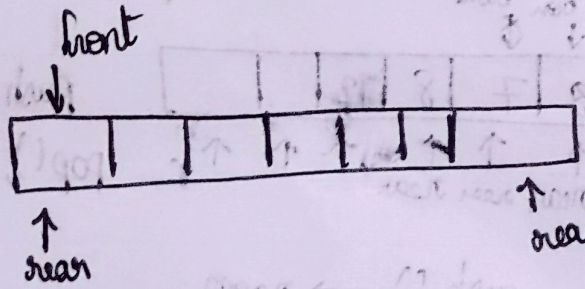
sabse front element → q.front()




## Queue

- Array
- Linked List

Starting.




rear = n-1 → full.

push
if(full)
else arr[rear] = element;
rear++;

empty
front = rear

Pop

if (empty) → Queue is empty.

else

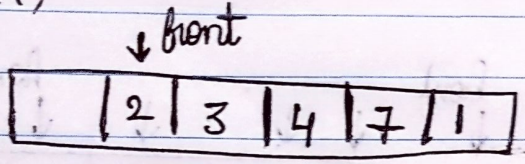
arr[front] = -1;

front++;

front = rear

↓

empty.




isempty → front == rear

```
if (front == rear)
{
    front = 0;
    rear = 0;
}
```

→ Implementation

```
void enque (int data)
{
    if (front == rear)
    {
        cout << "Queue full";
    }
    else
    {
        arr[rear] = data;
        rear++;
    }
```

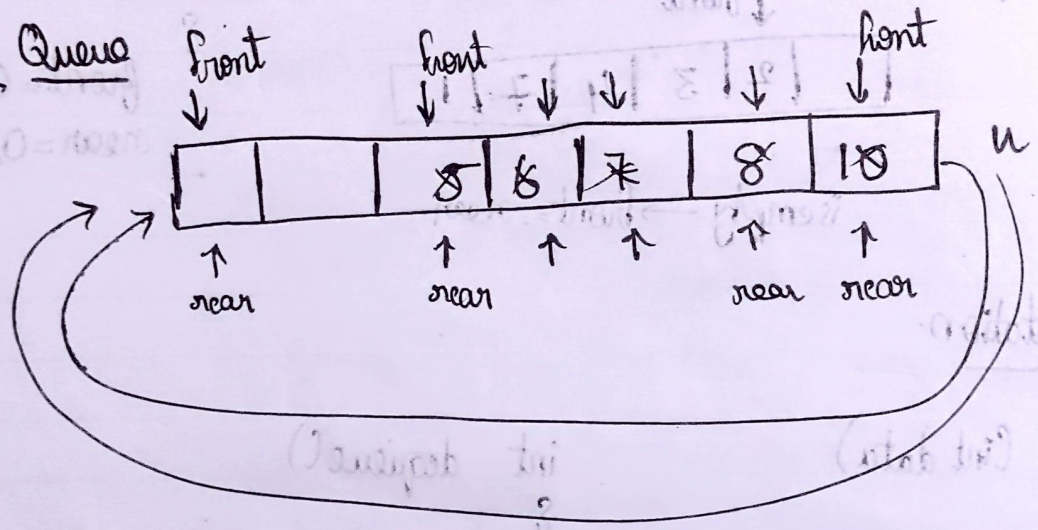
```
int dequeue()
{
    if (front == rear)
    {
        return -1;
    }
    else {
        int ans = arr[front];   // Ele to be removed
        arr[front] = -1;
        front++;
        if (front == rear)
        {
            front = 0;
            rear = 0;
        }
        return ans;
    }
```

```
bool isempty()
{
    if(front==rear)
    {
        return true;
    }
    else
        return false;
```

```
int front()
{
    if(front == rear)
    {
        return -1;
    }
    else
        return arr[front];
```

---

## Circular Queue




queue:

front = -1

rear = -1

push

↳ if (full) ⟶ "Queue is full"

front == 0 && rear == size-1

rear == ((front - 1) % (size - 1))
(a)

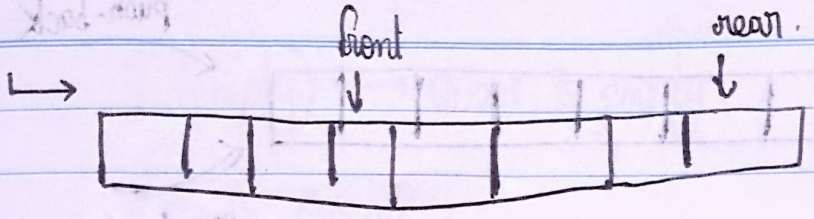
→ if front == -1

{

front = 0

rear = 0

arr[queue] = data;



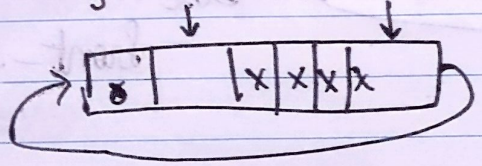

```
if (rear == size-1 && front != 0)
{
    rear = 0;
    arr[rear] = data;
}
else
    rear++;
    arr[rear] = data;
```

**Dequeue**

→ if (empty) ⟶ Queue is empty.
↳ front = -1

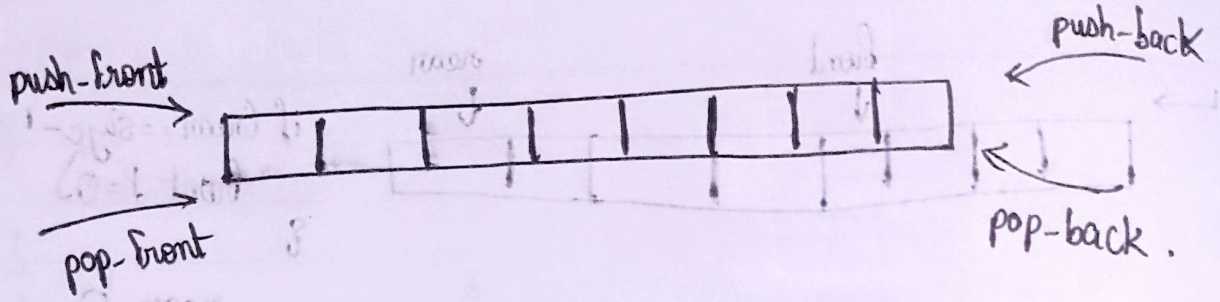
Code implemation → VScode

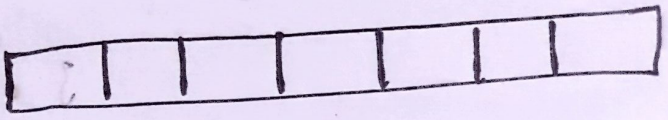
```
//single element ⟶
if (front == rear)
{
    front = rear = -1;
}
```




```
if (front == size-1)
{
    front = 0
}
```

# → Doubly Ended Queue



front = -1
rear = -1

**push front**

→ if full ⟶ Queue is full.

→ // first element

front == -1

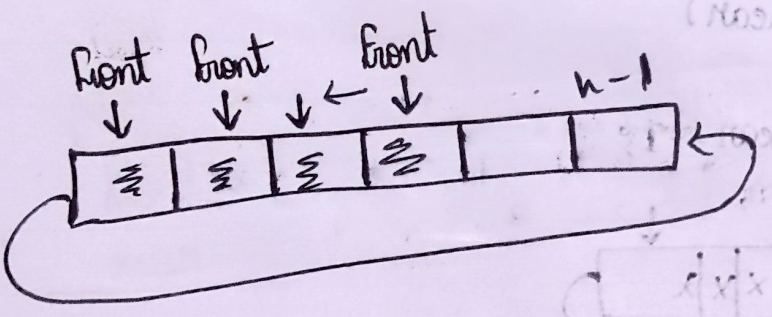
→ front = rear = 0.

if (front == 0)

→ front = n-1

else

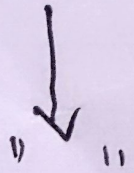
front--;



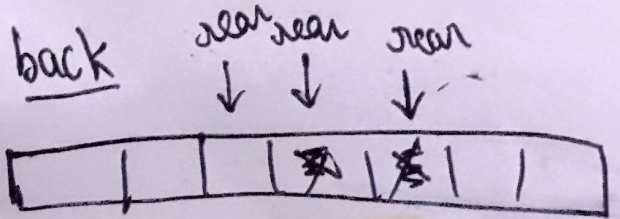

**push_rear**

Same implement as done in queue.

**pop_front**



**rear_pop** **pop_back**



if (empty) ⟶ Queue is empty.

// single element
↳ ⟶ front = rear = -1

// Cyclic Nature.
if (rear == 0)
rear = n - 1
normal flow
rear--;

STL Operations.

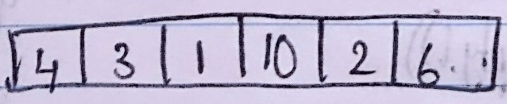
→ push_front()  → pop_front()
→ push_back()  → pop_front()

Array Implementation
↳
→ vscode refer.

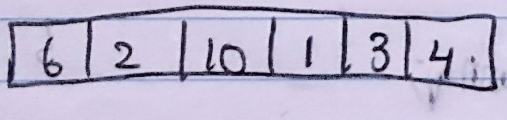
Queue Questions.

Queue reversals

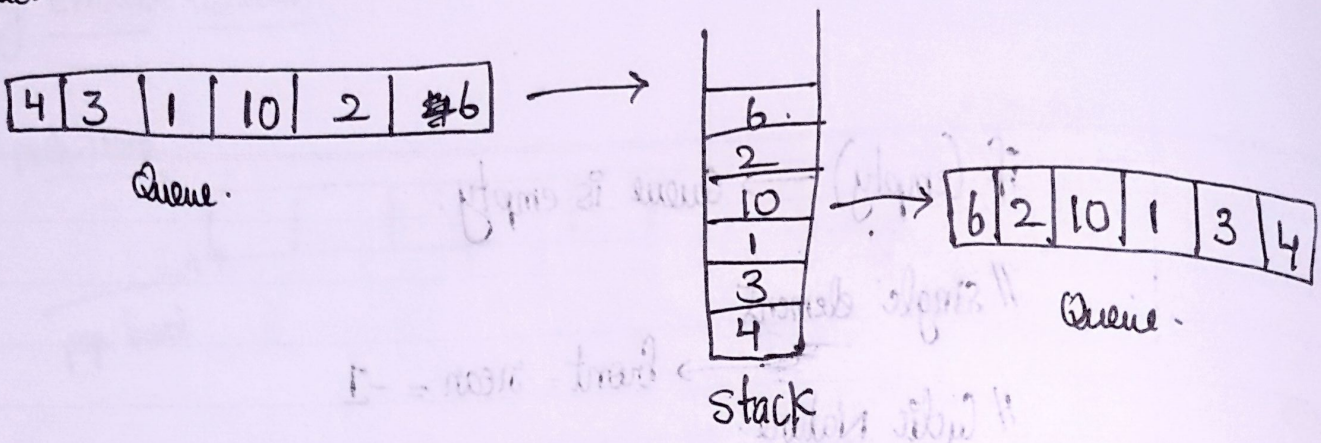
i/p




o/p

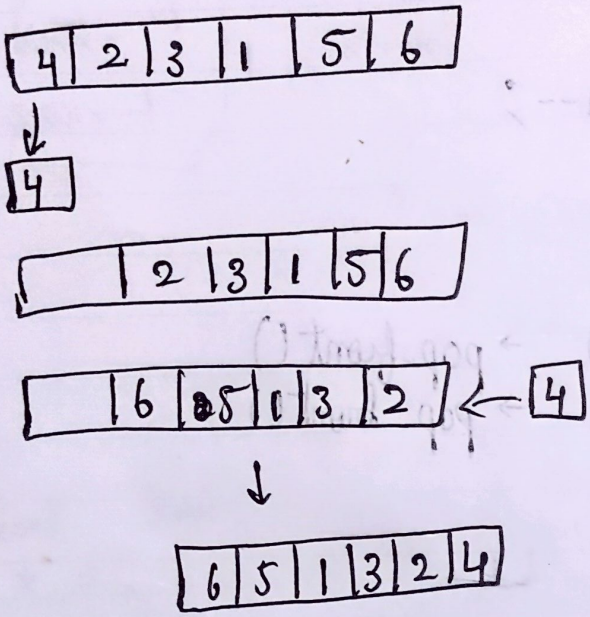


Approach 1




IInd

Recursion.




Code

Recursion.

```
Queue<int> rev(Queue<int> q)
{
    if(q.empty())
    {
        return q;
    }
    int data = q.front();
    q.pop();
    q = rev(q);
    q.push(data);
    return q;
}
```

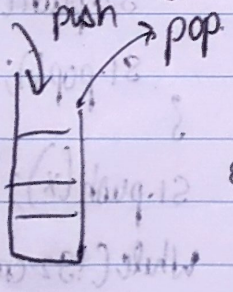
Approach!

```
queue<int> rev(queue<int>q) {
 stack<int>s;
while(!q.empty())
{
   int element = q.front()
   q.pop()
   s.push()
}
while(!s.empty())
{
    int element = s.top();
    s.pop();
    q.push();
}
  return q;
}
```

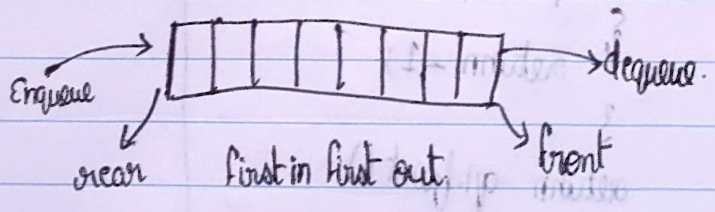
---

Implement Stack using Queue.

Stack




Queue




```
class Stack
{
   queue<int> q1, q2;
```

```
public:
    void push(int x)
    {
        q2.push(x);
        while(!q1.empty())
        {
            q2.push(q1.front());
            q1.pop()
        }
        queue<int>q=q1; //swapping
        q1=q2;
        q2=q;
}
void pop()
{
    if(q1.empty())
    {
        return;
    }
    q1.pop();
}
int top()
{
    if(q1.empty())
    {
        return -1;
    }
    return q1.front();
}
int size()
{
    return q1.size();
}
```

```
int main()
{
    Stack s;
    s.push(1);
    s.push(2);
    s.push(3);
    s.top();
    s.top();
    s.top()
}
```

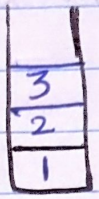
Output:-

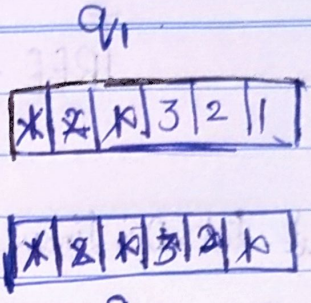
3
2
1

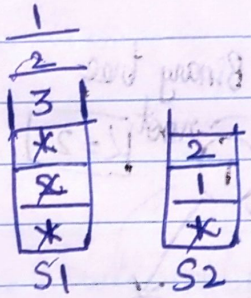
## Implement Queue Using Stack

```
Struct Queue {
    stack<int> s1,s2;
    void enQueue(int x)
    {
        while(!s1.empty())
        {
            s2.push(s1.top());
            s1.pop();
        }
        s1.push(x);
        while(!s2.empty())
        {
            s1.push(s2.top());
            s2.pop();
        }
    }
```

```
int deQueue()
{
    if(s1.empty())
    {
        return -1;
    }
    int x = s1.top();
    s1.pop()
    return x;
}
};
int main()
{
   Queue q;
   q.enQueue(1);
   q.enQueue(2);
   q.enQueue(3);
   cout<< q.dequeue();
   cout<<q.deque();
   cout<< q.dequeue();
   return 0;
}
```



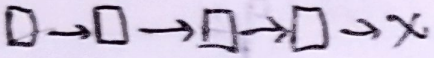



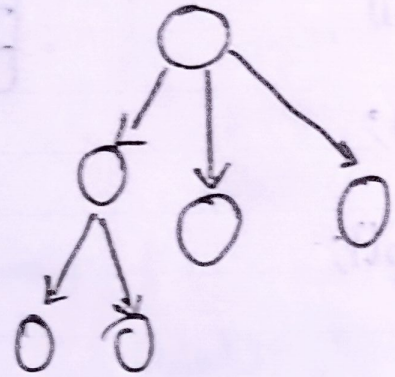



# #62. BINARY TREE

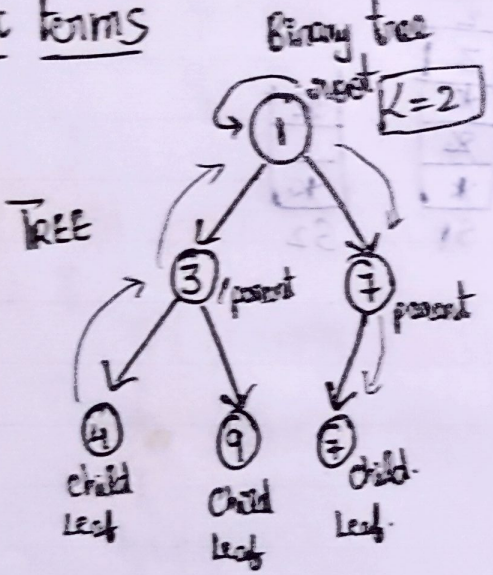
TREE → Non-linear Data Structure

Linked list → DS Linear



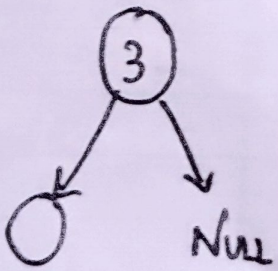


## Basic terms



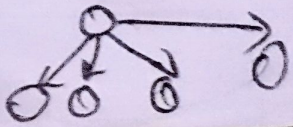

① Node
② Root ✓
③ Children ✓
④ Parent ✓
⑤ Siblings ✓
⑥ Ancestors ✓
⑦ DEScendant ✓
⑧ Leaf → Doesn't have child.

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

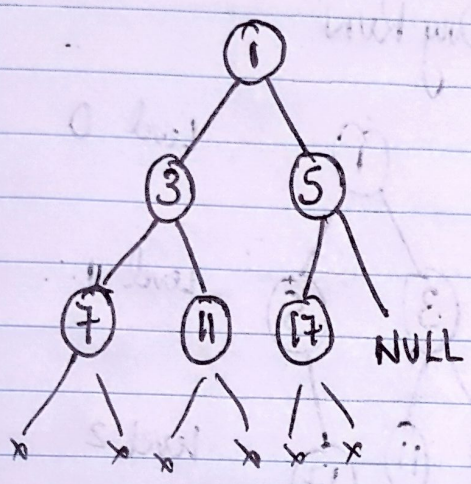


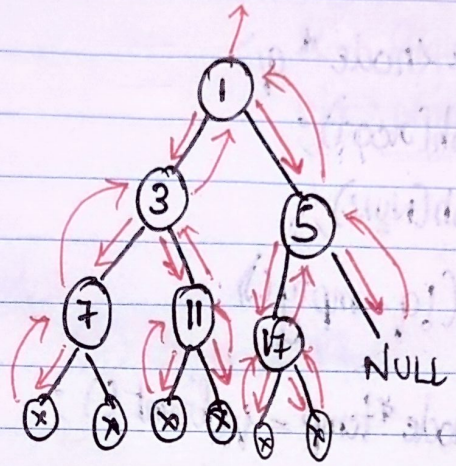

If Multiple Node allowed

```
node
{
    int data
    list <node*> child;
};
```



# Creation Binary Tree



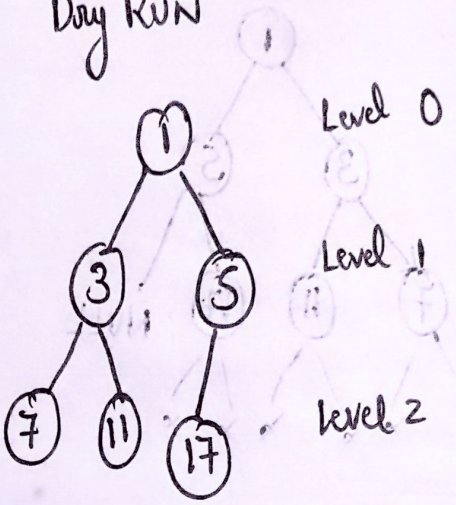

```
class Node
{
  public:
      int data;
      int *left;
      int *right;

  node(int d)
  {
      this->data=d;
      this->left=NULL;
      this->right=NULL;
  }
};
```

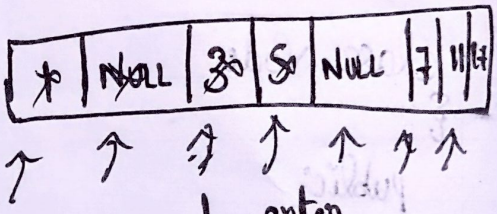
```
node *buildTree(node *root)
{
    cout<<"Enter data"<<endl;
    int data;
    cin>>data;
    root= new node(data);

    if(data==-1)
    {
      return NULL;
    }
    cout<<"Enter data left of "<<data<<endl;
    root->left=buildTree(root->left);
    cout<<"Enter data Right of "<<data<<endl;
    root->right=buildTree(root->right);
    return root;
}
```

# LEVEL Order Traversal

```
void levelOrderTraversal(node *root)
{
    queue<node *> q;
    q.push(root);
    q.push(NULL);
    while(!q.empty())
    {
        node *temp = q.front();
        q.pop();
        if(temp == NULL)
        {
            cout<<endl;
            if(!q.empty())
            {
                q.push(NULL);
            }
        }
        else
        {
            cout<<temp->data<<" ";
            if(temp->left)
            {
                q.push(temp->left);
            }
            if(temp->right)
            {
                q.push(temp->right);
            }
        }
    }
}
```

Dry Run




1 enter

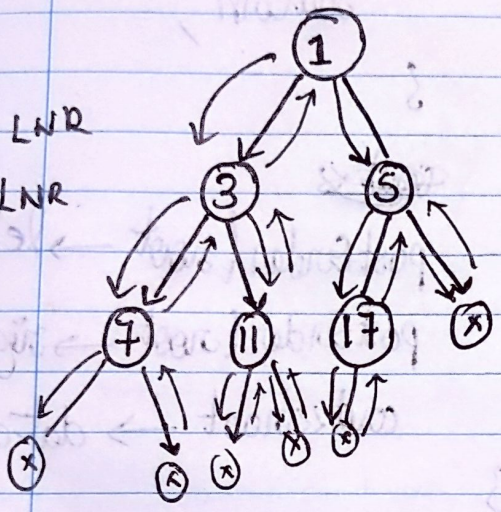
ans 3, 5 enter

7 11 17 ⊙ enter

→ InOrder / PreOrder / PostOrder

LNR NLR LRN

L → Left Part
R → Right Part
N → Node (Print)

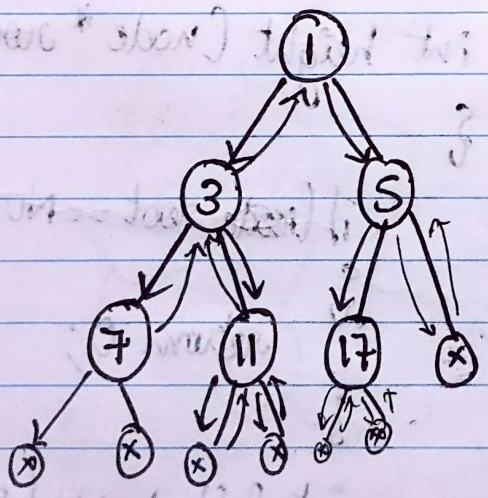



Inorder

7 3 11 1 17 5

PreOrder

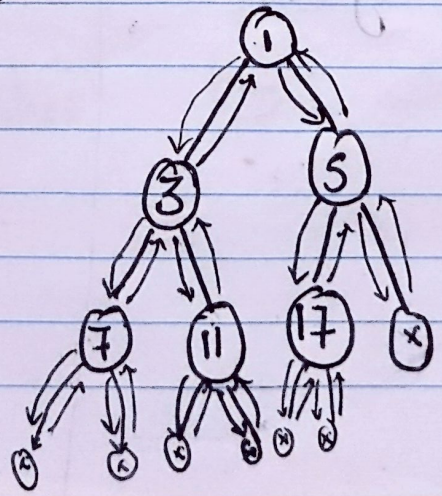
NLR




PreOrder →

1 3 7 11 5 17

Post Order LRN




7 11 3 17 5 1

## Code LNR

```
void Inorder(node * root)
{
    if(root == NULL)
    {
        return;
    }
    inorder(root -> left);
    cout<< root -> data << " ";
    inorder(root -> right);
}
```

LRN

```
void postorder(node * root)
{
    if(root == NULL)
    {
        return;
    }
    postOrder(root -> left);
    postOrder(root -> right);
    cout<<root -> data<<endl;
}
```

---

NLR

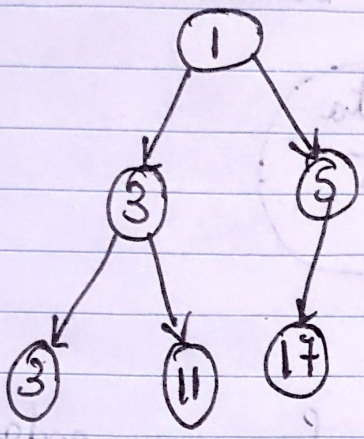
```
void preorder(node * root)
{
    if(root==NULL)
    {
        return;
    }
    cout<< root -> data << " ";
    preorder(root -> left);
    preorder(root -> right);
}
```

---

## Height Tree Code TC → O(n)

```
int height(node * root)
{
    if(root==NULL)
    {
        return 0;
    }
    int left = height(root -> left);
    int right = height(root -> right);
    int ans = max(left, right) + 1;
    return(ans);
}
```

## → Height of A Tree




height → 3

height = ?

↓

Longest Path b/w root node & the leaf node.

↓

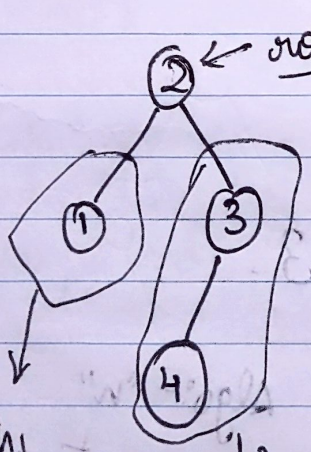
no. of nodes → height

approach

Recursion → rest recursion

↓

1 case solve




h1 left subtree

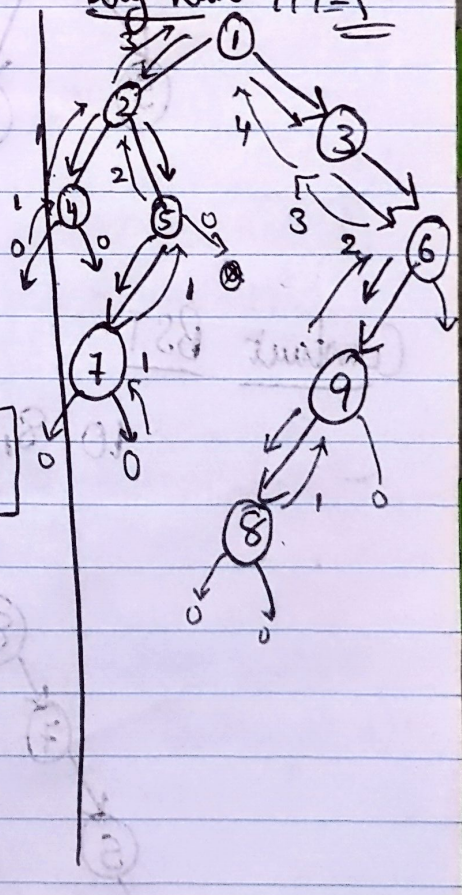
h2 right subtree

max(h1, h2) + 1

↓

height.

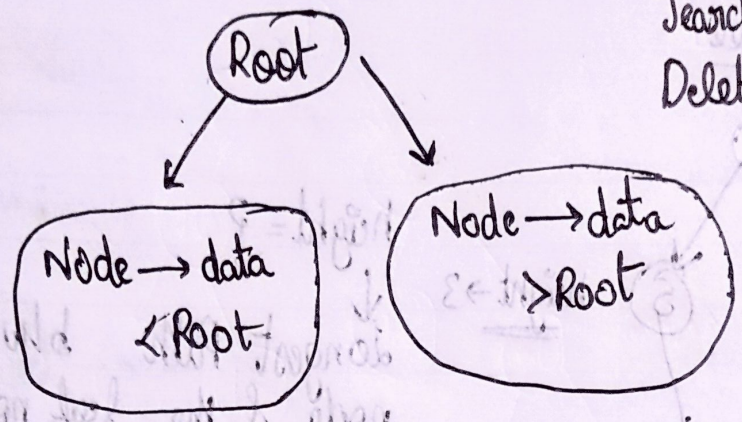
Dry Run 4+1=5



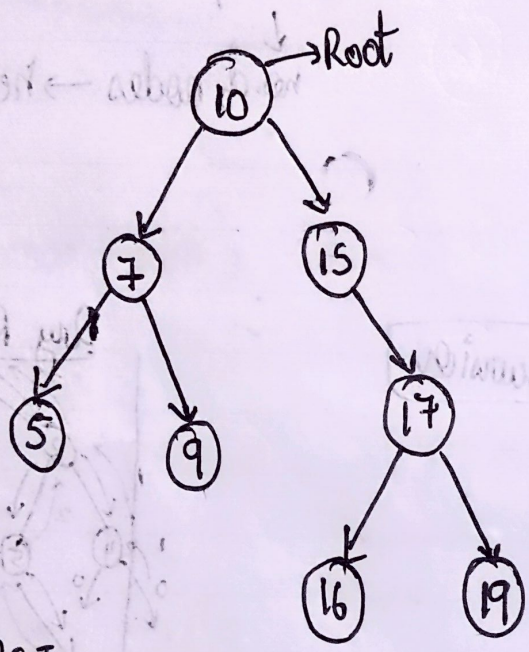
# BINARY SEARCH TREE

T.C → O(log n)

Insertion
Searching
Deletion



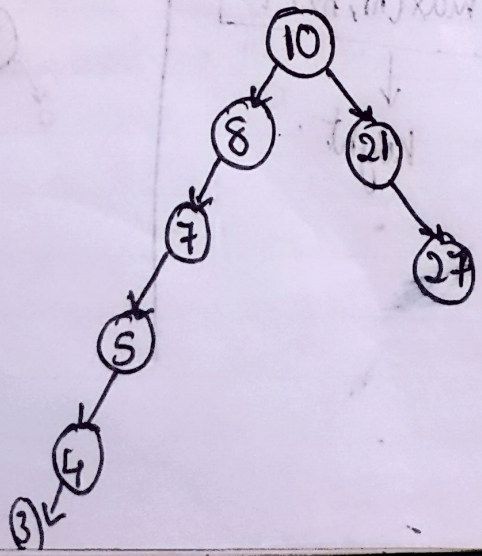

Eg:-




for every node
the condition must satisfy.

Construct BST

10, 8, 21, 7, 27, 5, 4, 3.




Algo:- "n"

root → data

(>) right part

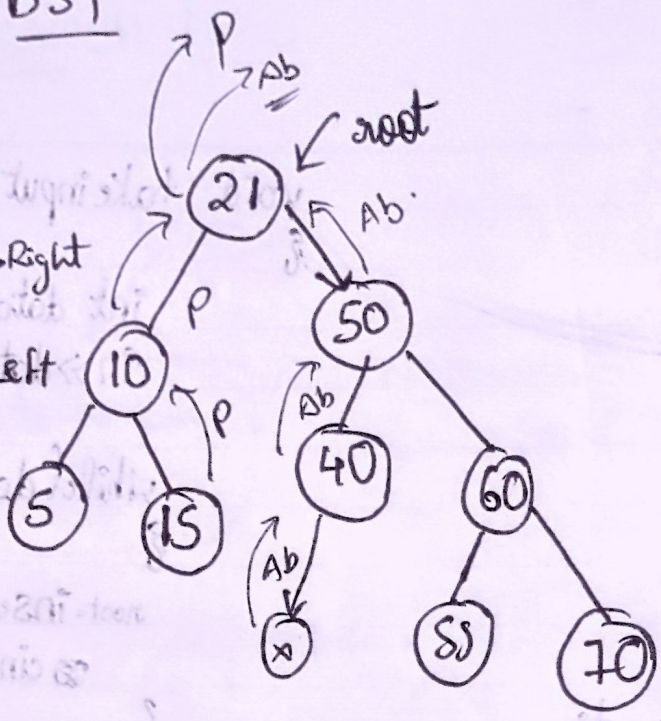
(<) left part

# Insertion

```
class Node
{ public:
    int data;
    Node *left;
    Node *right;

    Node(int d)
    {
        this->data = d;
        this->left = NULL;
        this->right = NULL;
    }
};

int main()
{
    Node *root = NULL;
    cout << "Enter data to create BST" << endl;
    takeinput(root);
}
```

```
void takeinput(Node *root)
{
    int data;
    cin >> data;

    while(data != -1)
    {
        root = insertIntoBst(root, data);
        cin >> data;
    }
}

Node *insertIntoBst(Node *root, int d)
{
    if(root == NULL)
    {
        root = new Node(d);
        return root;
    }

    if(d > root->data)
    {
        root->right = insertIntoBst(root->right, d);
    }
    else{
        root->left = insertIntoBst(root->left, d);
```

# Search in BST




21 < 25 → Right

50 > 25 → Left

40 > 25 → left

X = 27

X = 15

21 > 15 left.

10 < 15 Right.

15 == 15

Base case.

x → return → false / Absent

if root → data > x → left.

else

right part

## Implementation //Recursive

```
bool searchInBST (BinaryTreeNode<int> *root, int x)
{
    if (root == NULL) //Not found
    {
        return false;
    }
    if (root->data == x) //found.
    {
        return true;
    }
    if (root->data > x) //left side
    {
        return searchInBst (root->left, x);
    }
    else { // right side.
        return searchInBst (root->right, x);
    }
```

// Iterative

```
bool searchInBst(BinaryTreeNode<int> *root, int x)
{
    BinaryTreeNode<int> *temp=root;

    while(temp != NULL)
    {
        if (temp->data == x)
        {
            return true;
        }
        if (temp->data > x)
        {
            temp = temp->left;
        }
        else
            temp = temp->right;
    }
    return false;
}
```

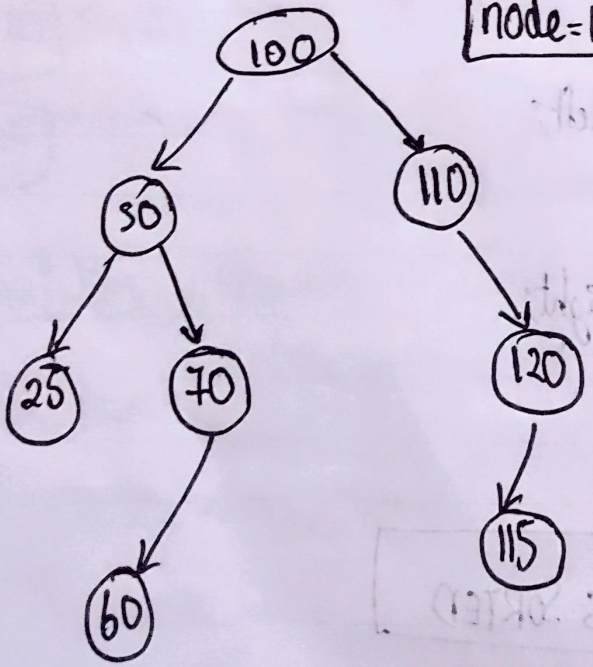
## INORDER OF BST IS SORTED

# MAX / MIN In BST

```
Node * minValue (Node * root)  //MIN
{
    Node * temp = root
    While (temp->left != NULL)
    {
        temp = temp -> left;
    }
    return temp;
}
```

```
Node * maxValue (Node * root)
{
    Node * temp = root
    while (temp->right != NULL)
    {
        temp = temp -> right;
    }
    return temp;
}
```

# Deletion in BST




node = 120

Algo:-

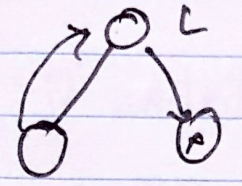
↳ node

```
if (root -> data == x)
{
    ---
    |
    |
    ---
}
else if (root -> data > x)
{
    left part
}
else {
    right part
}
```

Node to delete

0 Child
↳ delete node;
return NULL

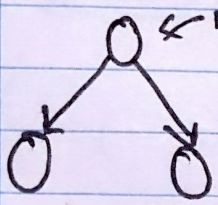
1 Child




Left Child.
temp = root → left
delete (root);
return temp;

Right
temp = root → right
delete (root)
return temp.

↳ 2 Children.




## Implementation

```
Node * deleteFromBst(Node *root, int val)
{
    if(root==NULL)
    {
        return root;
    }
    if(root → data == val)
    {   // 0 child.
        if(root→left==NULL && root→right ==NULL)
        {
            delete root
            return NULL;
        }
```

```
// 1 Child
// left Child
if (root->left != NULL && root->right == NULL)
{
        Node *temp = root->left
        delete (root);
        return temp;
}
if (root->left == NULL && root->right != NULL) // right
{
        Node *temp = root->right;
        delete (root);
        return temp;
}
// 2 Child
if (root->left != NULL && root->right != NULL)
{
        int mini = minValue(root->right)->data;
        root->data = mini;
        root->right = deletefromBst (root->right, mini);
        return root;
}
```

## Question

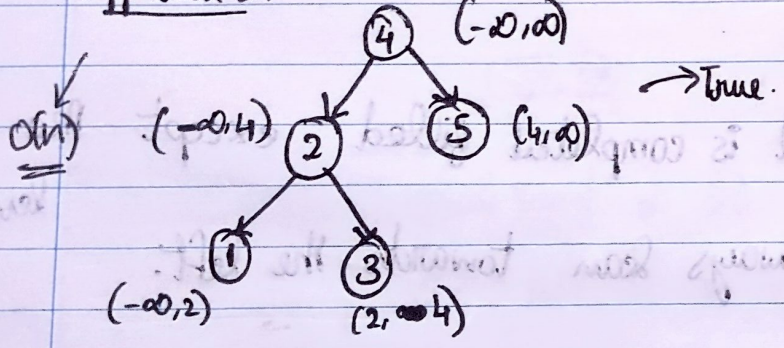
Valid BST (or) NOT

**Approach 1:-**

Inorder traversal if sorted → T then BST
→ F then false.

O(n).

**Approach 2:-**




O(n)

Code // Approach 2

```
bool validate (Node* root, int min, int max)
{
    if(root == NULL)
    { return true; }

    if(root->data > min && root->data < max)
    {
        bool left = validate (root->left, min, root->data);
        bool right = validate (root->right, root->data, max);
        return left && right;
    }
    else
        return false;
}
```

INT_MIN

INT_MAX

// Approach 1 - GeekforGeeks

```
do inorder (node *root, ans)

inorder (root->left, ans)
ans.push_back(root->data)
inorder (root->right)
```

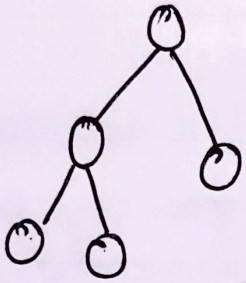
Check arr whether sorted (or) not.

# #74 Heaps

What is Heap?

↳ is a Complete Binary Tree that comes with a heap Order property.

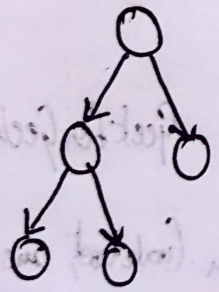
What's Complete Binary Tree?



atmost 2 Childs

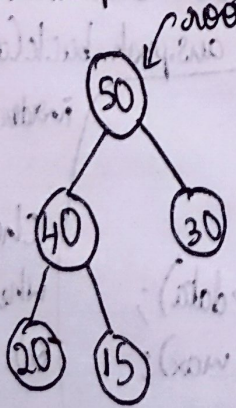
CBT → Every level is completed filled except the last level

→ Nodes always lean towards the left.



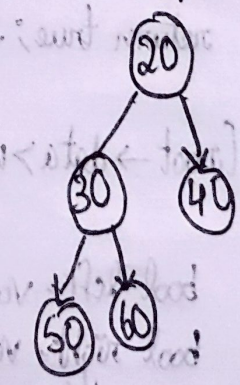
→ Complete Binary Tree

Max Heap:



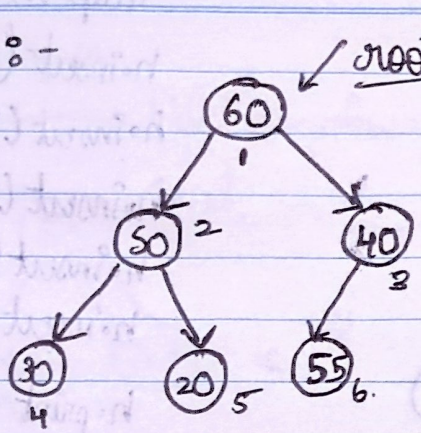

All Child nodes are smaller

Min Heap:




All Child nodes are bigger.

## Insertion :-




Max heap.

| x | 60 | 50 | 40 | 30 | 20 | 55 |
|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |

Node → $i^{th}$ index

left Child → $2*i^{th}$ index.

right Child → $(2*i+1)$ index.

parent = $(i/2)$

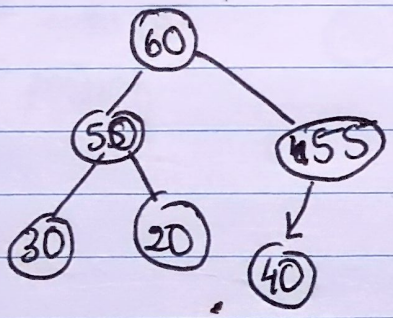
Eg :- $(5/2) = 2^{nd}$ index

val = 55

↓

① → insert at end.

② → take it to correct position.




→ Becomes likes this

## Implementation Insertion

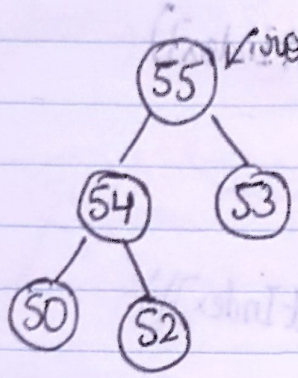
```
class Heap.
{
  public:
    int arr[100];
    int size;

    // Constructor Heap()
    {  arr[0]=-1;
       size=0;
    }
```

```
void insert (int val)
{
    Size=Size+1;
    int index=Size;
    arr[index]=val;

    while(index>1)
    {
        int parent = index/2;
        if(arr[parent]< arr[index])
        {
            swap(arr[parent], arr[index])
            index=parent;
        }
        else
        {
            return;
        }
    }
}
void print()
{
    for(int i=1; i<=Size; i++)
    {
        cout<<arr[i]<<" ";
    }
    cout<<endl;
}
};
```

```
int main()
{
    Heap h;
    h.insert(50);
    h.insert(55);
    h.insert(53);
    h.insert(52);
    h.insert(54);

    h.print();
}
```

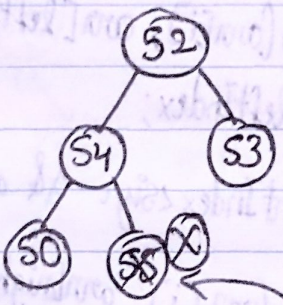
## Deletion in Heaps

⟶ Deletion always takes place for root node.



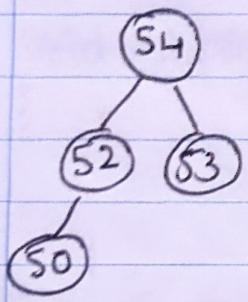

① swap first node with last node.

⇒



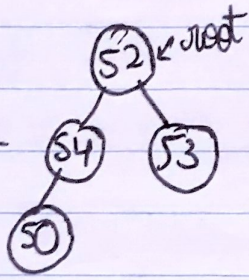

a[i]=a[n]

II Remove last node




III propagate root node to correct position.

←




## Implentation-

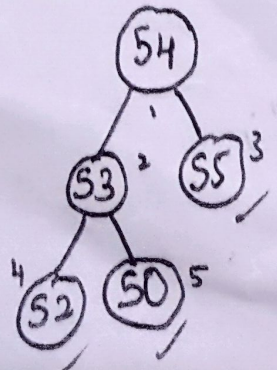
```
void deleteFromHeap()
{
    if(size==0)
    {
        cout<<"Nothing to delete";
        return;
    }
arr[1] = arr[size];
size--;
```

```
int i=1;
While(i<Siye)
{
    int leftIndex = 2*i;
    int rightIndex = 2*i+1;
    if (leftIndex < Siye && arr[i] < arr[leftIndex])
    {
        swap(arr[i], arr[leftIndex])
        i = leftIndex;
    }
    else if (rightIndex < Siye && arr[i] < arr[rightIndex]).
    {
        swap(arr[i], arr[rightIndex]).
        i = rightIndex;
    }
    else
    {
        return;
    }
}
}
```

## Heapify Algo

i/p → | 54 | 53 | 55 | 52 | 50 |
1 2 3 4 5

Convert into Heap.



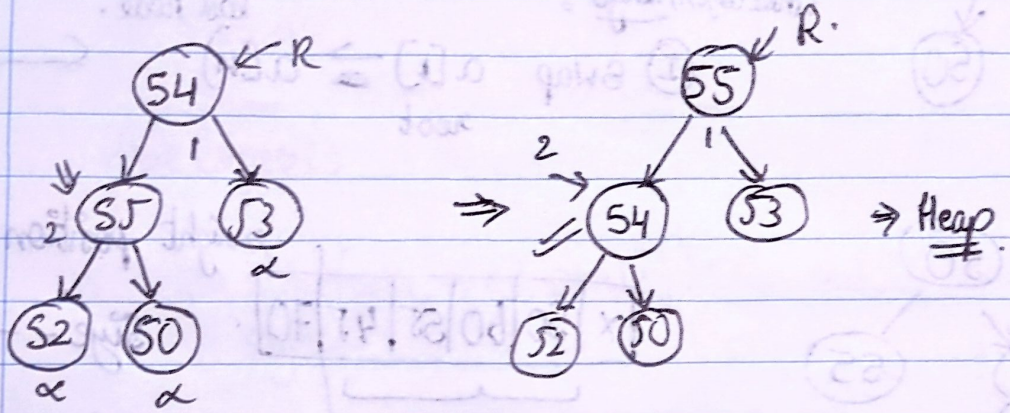

CBT ⇒ leaf node
↳ $(n/2 + 1)$

$5/2 + 1$

$= 3 \rightarrow 5 \rightarrow$ leaf node

Focus. $1 \rightarrow n/2$

```
for (n/2 -> >0)
{
    heapify (arr, size, i);
}
```

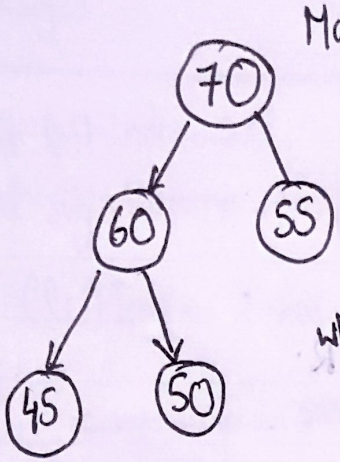



Implementation:

```
void Heapify (int arr[], int n, int i)
{
    int largest = i;
    int left = 2*i;
    int right = 2*i+1;
    if (left < n && arr[largest] < arr[left])
    {
        largest = left;
    }
    if (right < n && arr[largest] < arr[right])
    {
        largest = right;
    }
    if (largest != i)
    {
        swap (arr[largest], arr[i]);
        heapify (arr, n, largest);
    }
}
```

```
int main()
{
    int arr[6] = {-1, 54,
    53, 55, 52, 50};
    int n = 5
    for (int i = n/2; i > 0; i--)
    {
        Heapify (arr, n, i)
    }
// printing
for (int i = 1; i <= n; i++)
{
    cout << arr[i] << " ";
}
}
```

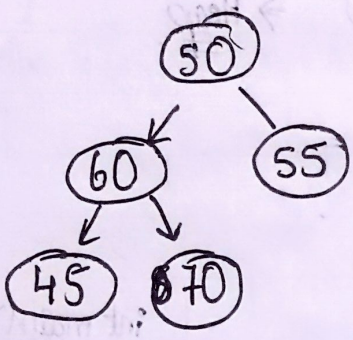
# Heap Sort:




| x | 70 | 60 | 55 | 45 | 50 |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

while(size>1) Algo :-

(I) swap a[i] ⇄ a[n]
root → last node.

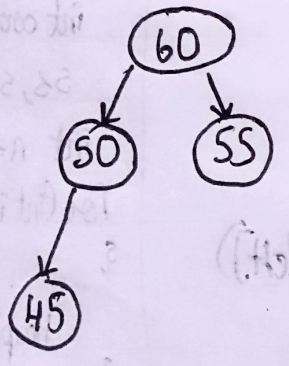



| x | 50 | 60 | 55 | 45 | 70 |
|---|---|---|---|---|---|

↗ right position.

size--;

(II) root node correction position




60|50|55|45

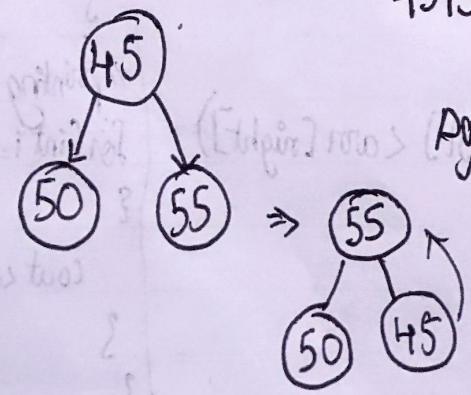
Again (I)

45|50|55|60,70

↳ sorted

Again (II)

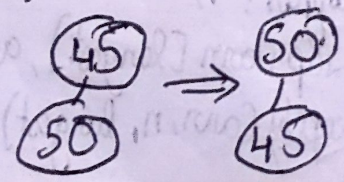
size--;




55|50|45|60|70

Again (I)

↳ 45|50|55,60




↳ 50|45|55,60

↳ 45,50,55,60

## Implementation

```
void Heapsort(int arr[], int n)
{
    int size=n;

    while(size>1)
    {
        swap(arr[size], arr[1]);
        size--;

        Heapify(arr, size, 1);
    }
}
```

## PRIORITY QUEUE STL.

// Max heap

```
priority_queue<int> pq;
```

pq.push
top()
push()
empty()
pop.

// min heap.

```
priority_que<int, vector<int>, greater<int>>minheap;
```

# Hashmaps

↓

Data Structure

↓

Type of D.S

↓

Ins / Del / Sear

⟶ operations → T.C is very good → O(1)

## Question

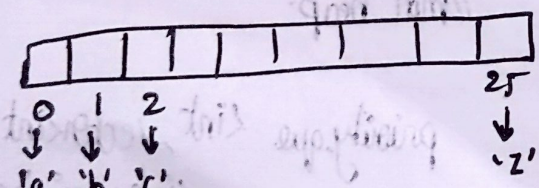
string str = "abcbdeaaf"

"Maximum Occuring Character"

a → 3
b → 2
c → 1
d → 1
e → 1
f → 1

Possible Normally.

```
for (i=0; i<str.size; i++)
{
    char ch = str[i];
    count[ch-'a']++;
}
```




'd' - 'a' → 0

'b' - 'a' → 1

if str = "My name is Ganesh name is GaneshGanesh";

My → 1
name → 2
Ganes → 3

Not possible

So, hashmaps are used.

Max occuring word.

As we want to store like this

<name, 2>

<My, 1>

<Ganesh, 3>

## Inbuilt Stuff

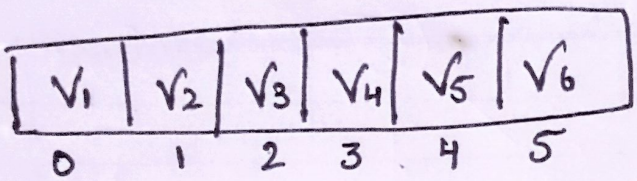
→ map → O(logn)

→ unordered_map → O(1)

```
int main()
{
    map<string, int> m;

    //1
    pair<string, int> p = make_pair("ganesh", 3);
    //2
    pair<string, int> pair2("name", 2);
    //3
    m["My"] = 2;
    // Size
    cout << m.size();  ⟶ gives size.
    // Count
    cout << m.count("ganesh") ⟶ Returns 1
    //Erase
    m.erase("name") ⟶ removes key
    // iterator.
    map<string, int>:: iterator it = m.begin();
    while (it != m.end())
    {
      cout << it->first << " " << it->second;
      it++;
    }
}.
```

# Bucket Array

<Key, value>




Hash functions.

Conversion to int
Uniformation.

↑

→ Hash Code

Hash Code ← Conversion to int → Comparesion function.

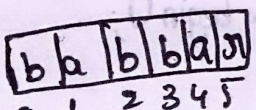
→ identity function.

int hashcode
↓
brings into range

i/p [H.C] → [23] → o/p.

i/p "babbar" → [H.C] → 23 → o/p.

b + a + b + b + a + r
↓ ↓ ↓ ↓ ↓
97 96 97 97 96 → 606. → 23.




b×0 + a×1 + b×2 + b×3 + a×4 + r×5

= 1283 % N → 22

## Collision

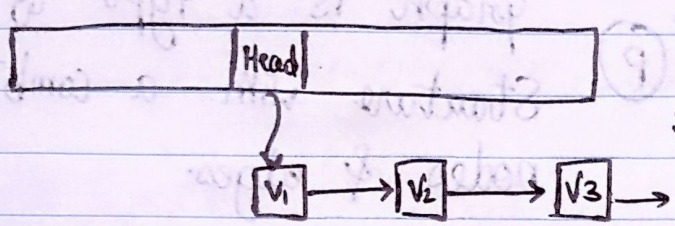
"babbar"
"babbar"


Collision Handling.

→ Open Hashing → same place put

→ closed Addressing.

## Open Hashing



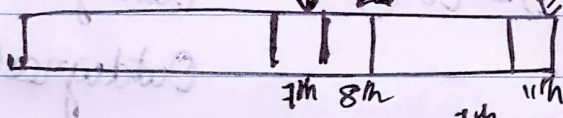

Separate Chaining.

## Closed Addressing

'A' → 7th index

'B' → 7th index

Search for another space.




7th + 1

→ Hi(a) = h(a) + Fi(a)

f(i) → is actually "i".

Linear Probing happens.

f(i) → i

f(1) → 1

f(2) → 2.

7 + 4 + 1

Hi(a) = h(a) + fi(a)

Quadratic Probing

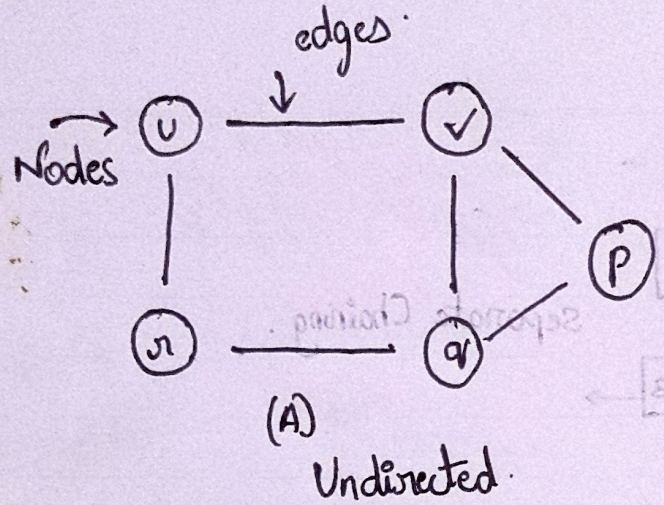
f(i) → i

f(1)² → 1

f(2)² → 4.

Complexity → O(k) → ~ O(1) if (n >> k).

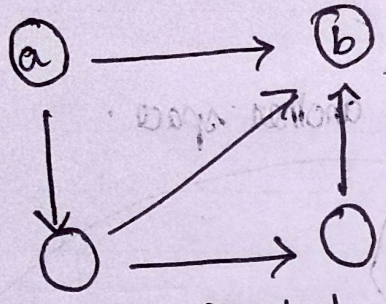
# Graphs #85




(A) Undirected

What-?

Graph is a type of data Structure with a combination of nodes & edges.




Directed

a → b ✓

b — a ✗

Indegree (b) = 3

Outdegree (b) = 0

Outdegree (a) = 2

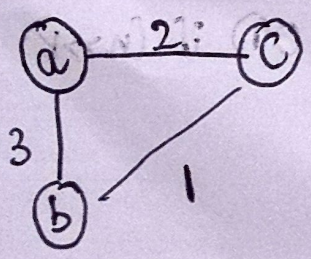
→ Node → Entity to store data

→ Edge → Connecting Nodes

→ weighted graph

Degree(v) = 3

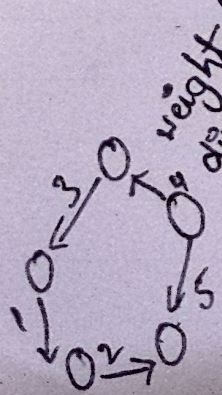
Degree (u) = 2




a — c → 2

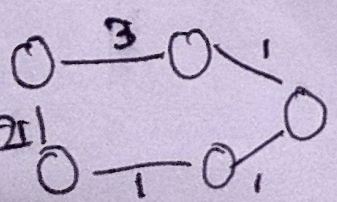
a — b → 3

b — c → 1




weighted directed graph

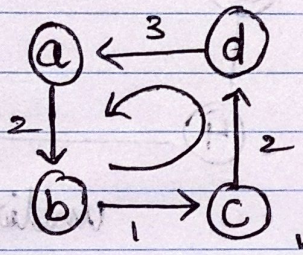
weighted Undirected graph →

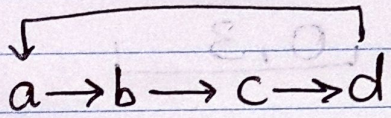



→ safe to assume 1

→ Path = u-v-p
= p-q-v
- p-q-v-p ⊗ Not a path.

Cyclic Graph.

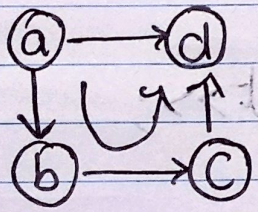



→ Cyclic Graph

weighted Cyclic directed Graph

Acyclic Graph




a → b → c → d
Acyclic

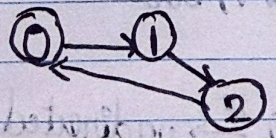
weighted acyclic directed graph

Graph
↓
→ Adjacency Matrix
→ Ajacency List

① → Ajacency Matrix

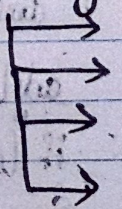
| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 2 | 1 | 0 | 0 |

S.C → O(n²)



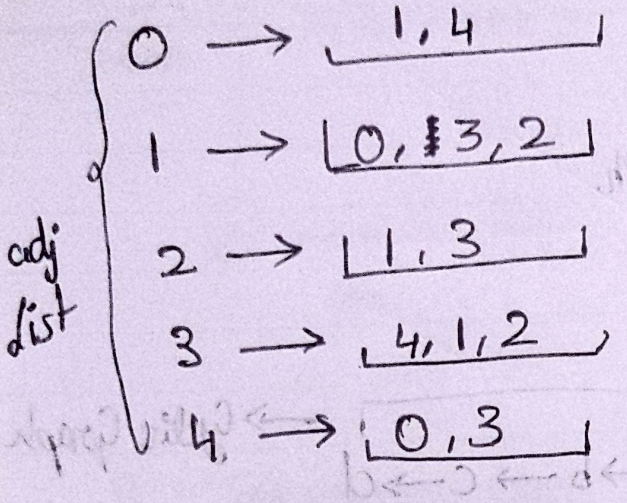

i/p →
No of nodes (n)
No of edges (m)

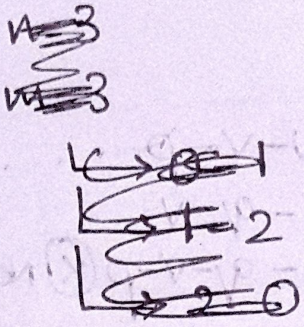
n=3, m=3
0 → 1
1 → 2
2 → 0

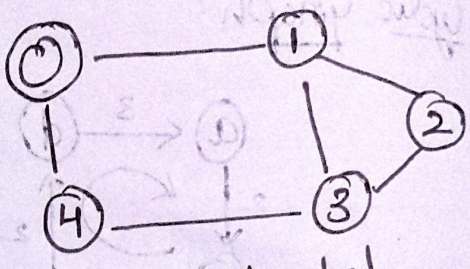


} edges.

## ② Adjacency List






undirected graph.

Implementation

adj List →

```
map<int, list<int>> ;
```

Code :-

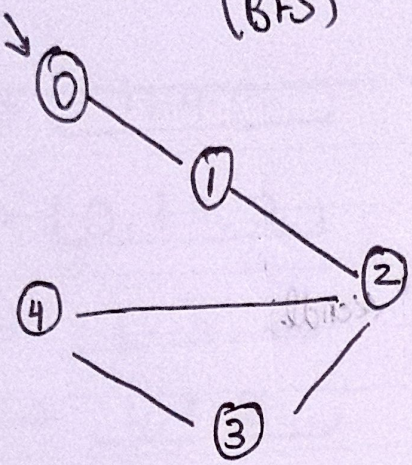
```
class graph
{
    public:
        unordered_map<int, list<int>> adj;

    void addEdge(int u, int v, bool direction)
    {
        // direction = 0 → undirected
        // direction = 1 → directed
        adj[u].push_back(v);
        if(direction == 0)
        { adj[v].push_back(u); }}
```

```
void printAdjList()
{
    for(auto i: adj)
    {
        cout << i.first << "->";
        for(auto j: i.second)
        {
            cout << j << ",";
        }
        cout << endl;
    }
}
```

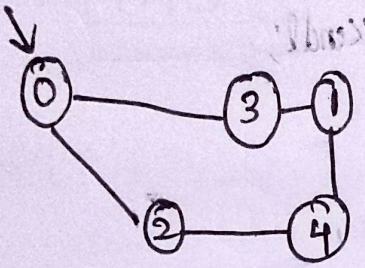
```
int main()
{
    int n;
    cout << "Enter the number of nodes" << endl;
    cin >> n;

    int m;
    cout << "Enter the number of edges" << endl;
    cin >> m;

    graph g;

    for(int i=0; i<m; i++)
    {
        int u, v;
        cin >> u >> v;
        // Creating an Undirected graph
        g.addEdge(u, v, 0);
    }
    g.printAdjList();
}
```

# Breadth First Search #86

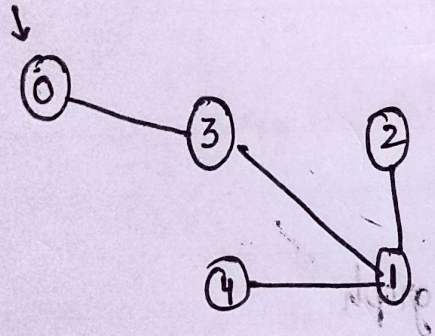
(BFS)



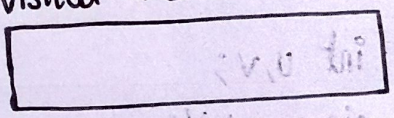

point → 0, 1, 2, 3, 4




→ 0, 2, 3, 4, 1



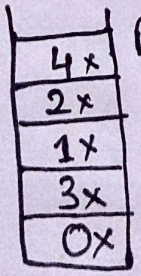

visited → D.S



unordered_map <node, bool>;

↓ int

queue → FIFO




D.S.

q

adj → list

0 → 3

1 → 2, 3, 4

2 → 1

3 → 0, 1

4 → 1

visited

0 → F → T

1 → F → T

2 → F → T

3 → F → T

4 → F → T

frontnode = 0x  frontNode = 3x

frontNode = 1x

frontNode = 2x

frontNode ≥ 4

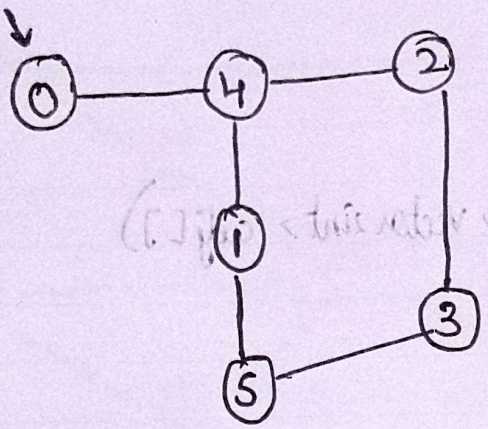
ans → 0, 3, 1, 2, 4

```
if (!visited[node])
{
    bfs();
}
```

## BFS Code Implementation

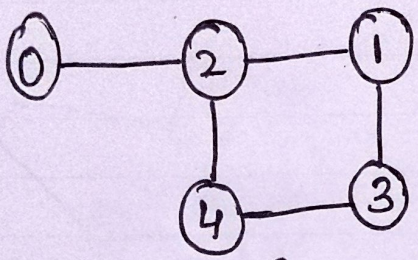
```
vector<int> bfsOfGraph(int V, vector<int> adj[])
{
    vector<int> ans;
    vector<int> vis(V,0);
    queue<int> q;
    q.push(0);

    vis[0] = 1

    while(!q.empty())
    {
        int front = q.front();
        q.pop();
        ans.push_back(front);

        for(auto &it: adj[front])
        {
            if(!vis[it])
            {
                q.push(it);
                vis[it] = 1;
            }
        }
    }
return ans;
}
```

# Depth First Search (DFS) #87 T.C → O(n)




BFS - 0, 4, 1, 2, 5, 3

DFS - 0, 4, 2, 3, 5, 1




C1

visited
im → T/F

for ( ____ )
{ if (!visited[i])
{ dfs()
}
}

dfs()

vist

0 → ~~F~~ x → T
2 → F → T
1 → F → T
3 → F → T
4 → F → T

dfs(0)
↓
dfs(2)
↓
dfs(1)
↓
dfs(3)
↓
dfs(4)

## Implementation

```
void dfs(int node, vector<int>& ans , vector<int> &visited , vector<int>adj[])
{
    ans.push_back(node);
    visited[node] = 1;

    for(auto it: adj[node])
    {
        if(visited[it] == 0)
        {
            dfs(it, ans, visited, adj)
        }
    }
}
vector<int> dfsofGraph(int V, vector<int> adj[])
{
      vector<int>visited(V,0);
      vector<int>ans;
       dfs(0, ans , visited, adj)
       return ans;
}
```