Time & Space Complexity :-

$O(1)$, $O(\log N)$, $O(\sqrt{N})$, $O(N)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$, $O(n!)$, $O(n^n)$

↓ Least Complex

↓ Most Complex

Array :-

- **Declaration :-** `int arr[5] = {1, 2, 3, 4, 5};`

6	2	3	12	2	1
0	1	2	3	4	5

$\Rightarrow$ arr[0] = 6 $\Rightarrow$ arr[2] = 3 $\Rightarrow$ arr[4] = 2

$\Rightarrow$ arr[1] = 2 $\Rightarrow$ arr[3] = 12 $\Rightarrow$ arr[5] = 1

$$\therefore \text{Size of array} = \frac{\text{size of (arr)}}{\text{size of (arr[0])}}$$

- **Access the elements of array :-**

Using Loop :-

```
for (int i=0 ; i<arr.size(); i++){
        cout << arr[i] ;
        cout << endl;
}
```

n = size of the array

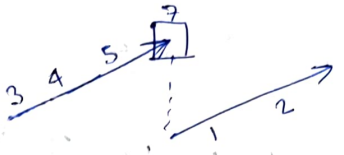
```
while (i ≠ n){
        cout << arr[i] << " " ;
        i++ ;
}
```

Find Peak Element :-

Binary Search

{ Never forget to write m = s + (e-s)/2 }

3	4	5	7	1	2




```
s = 0 ; e = arr.size() -1 ; m = s + (e-s)/2 ;
while (s<=e){

    if (m+1 < arr.size() && arr[m] > arr[m+1])
        return m;

    if (m-1 >=0 && arr[m-1] > arr[m])
        return m-1;

    if (arr[s] >= arr[m])
        e = m-1;
    else
        s = m+1;
```

m > m+1

3	4	5	7	1	2	3

m-1 > m

3	5	8	7	2	1

s e m

15	9	7	5	3	2

Left Search

Right Search

Find Pivot :-

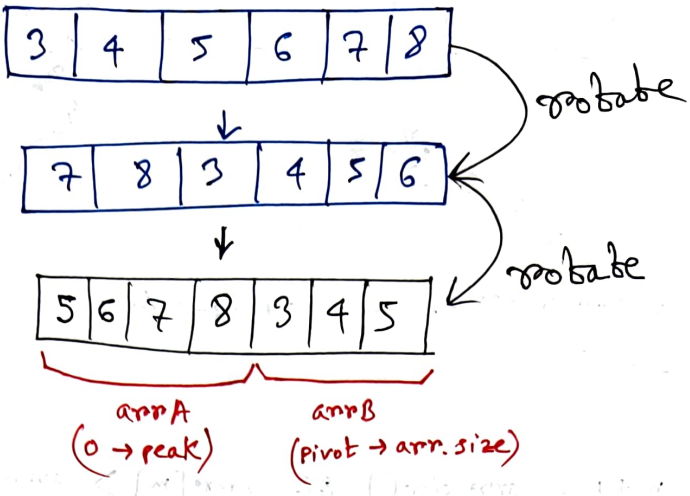
```
while (s<e){
    if (arr[m] >= arr[0]){
        s = m+1;
    }else{
        e = m;
    }
    m = s + (e-s)/2;
}
return s;
```

m s=m+1

7	9	11	1	5

return s;

Rotated Array :



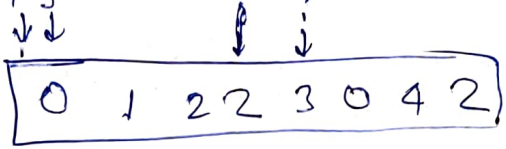

```
if (target < peak && target > arr[0])
        search (arrA);
else
        search (arrB);
```

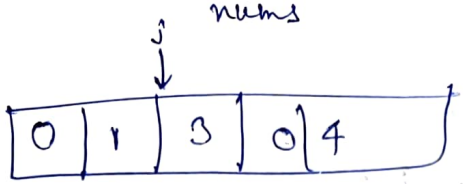
Remove Element

0	1	2	2	3	0	4	2	2

remove val = 2 ;

```
int j=0;
for(int i=0; i<nums.size(); i++){
        if (nums[i] != val){
                nums[j++] = nums[i]
        }
}
return j;
```



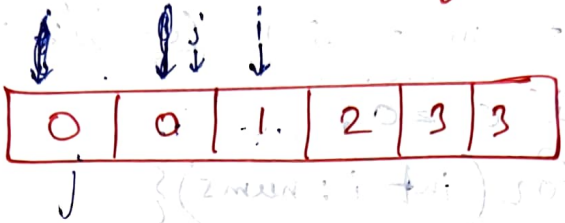


0 ! = 2

nums [0] = nums [0]

num [1] = nums [1]

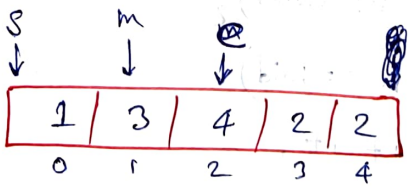
Remove duplicates in sorted Array:




```
int j = 0;
for (int i = 1; i < nums.size; i++) {
        if (nums[i] != nums[j]) {
            nums[j+1] = nums[i]; j++;
        }
}
```

Find duplicate number *(IMPORTANT)

```
int s = 1;
int e = nums.size() - 1;

while (s<e){

    int m = s + (e-s)/2;
    int c =0;
    for(int i : nums){
        if (i<=m){
            ++c;
        }
    }
    if (c>m){
        e = m;
    }
    else{
        s = m+1;
    }

}
return s;
```



i=0 c=0

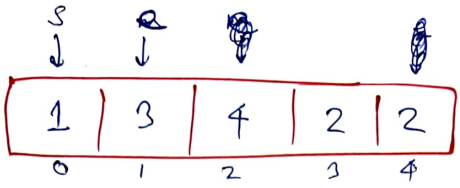
⇒ i=0, c=1

⇒ i=1, c=2

⇒ i=2, c=3

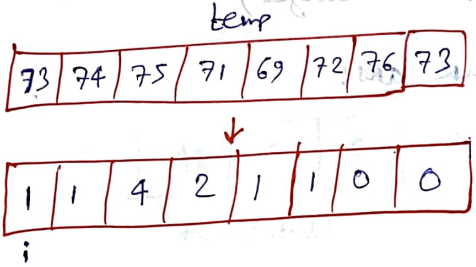
2>3

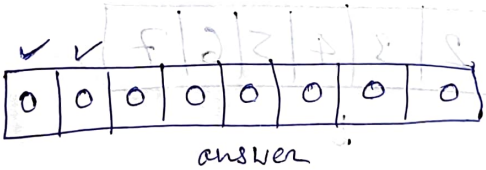
e=2

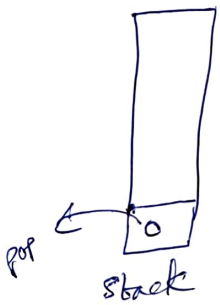



```
m = 1
c = 3
for (i = 2 ){   i=3
    2 <= 2
        c=3
}
3>1
    e = 1;
```

Daily Temperatures (739):





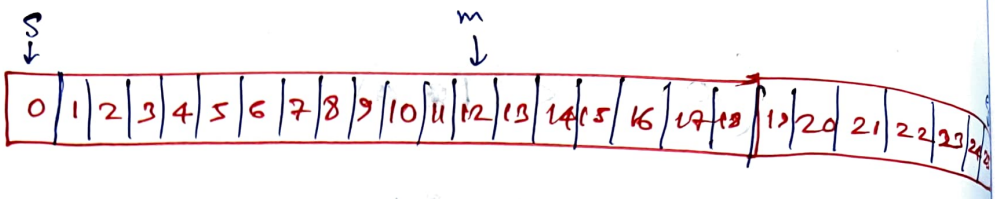



```
for (i<n;i++) {
    while(!s.empty && temp[s.top()]
                        < temp[i]){
        answer[s.top()] = i - s.top();
        s.pop();
    }
    s.push(i);
}
return answer;
```

Find sqrt :-

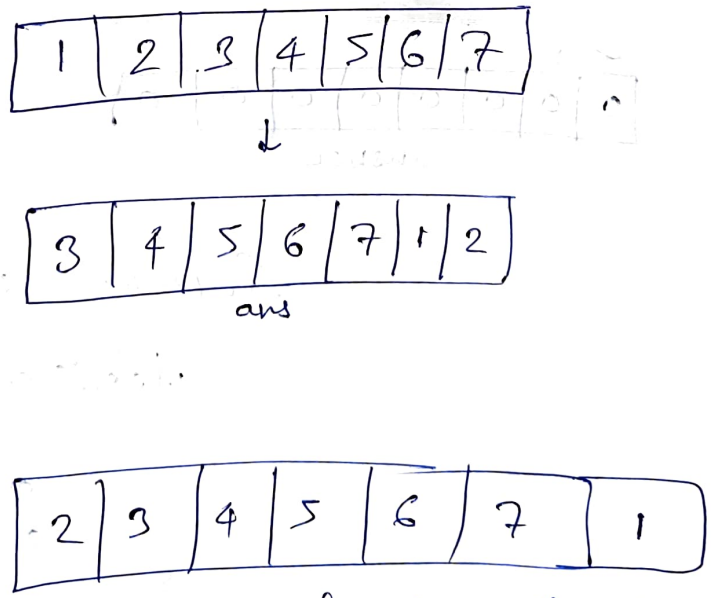
m = 12

Find sqrt of n = 25



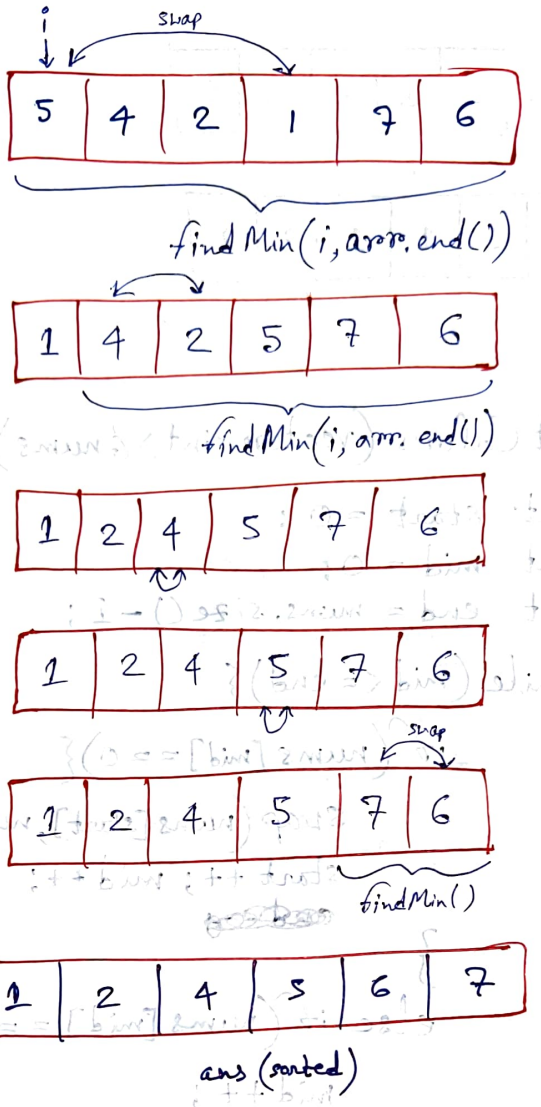

```
if (mid * mid > target)
    L.S.
if (mid * mid < target)
    R.S.
    store ans;
if (mid * mid == target)
    return ans;
```

Rotate Array

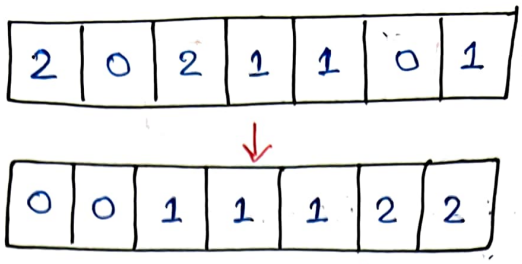


pushing back the 0th element to the end and erasing the first element

Selection Sort :-



75. Sort Colors

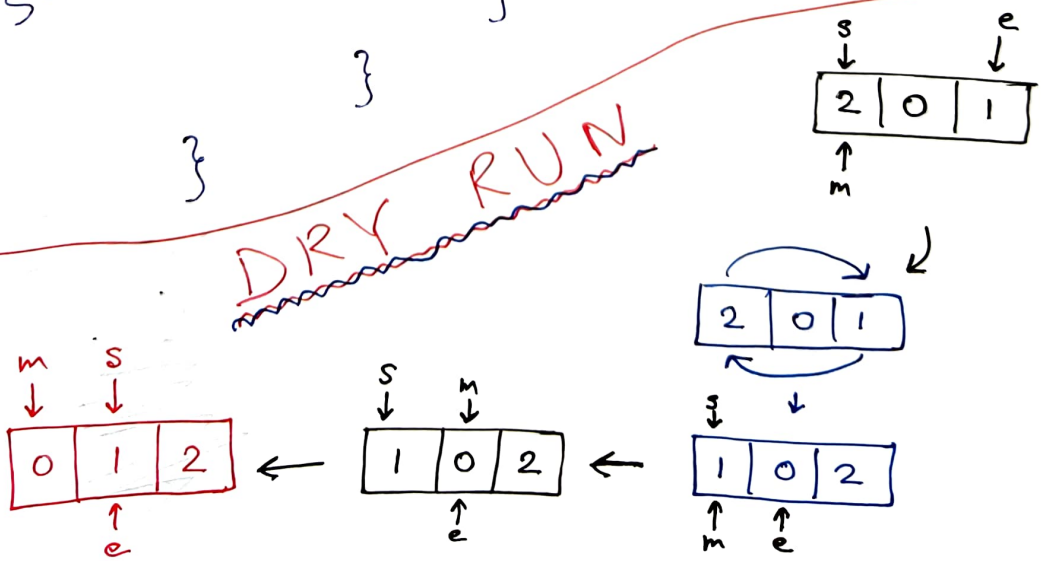


Code :-

```
void sortColors(vector<int>& nums){
    int start = 0;
    int mid = 0;
    int end = nums.size()-1;
    while(mid <= end){
        if(nums[mid] == 0){
            swap(nums[start], nums[mid]);
            start++; mid++;
        }
        else if(nums[mid] == 1){
            mid++;
        }
        else{
            swap(nums[mid], nums[end]);
            end--;
        }
    }
}
```

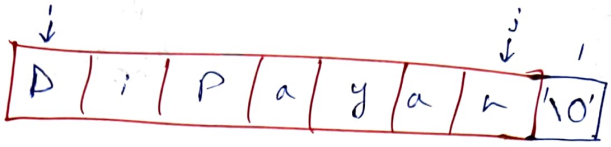
T.C = O(n)
S.C = O(1)

DRY RUN



Char Arrays :-

```
char name[100];
cin.getline(name, 50);
cout << name;
```



- Length of array → strlen(name)
- Reverse array → swap(name[i], name[j]); i++; j--;
- Palindrome → "noon", "lol", "racecar"

```
if(ch[i] != ch[j])
    return false
else
    i++; j--;
```

String :

```
string s = "hello";
```

- Length of string → s.length();
- string is Empty → s.isEmpty();
- Append in String → s.push_back('A');
- Pop from String → s.pop_back();
- Sub String → s.substr(0, 3); → hel
- Palindrome Check →


```
s = "A man, a plan, a canal: Panama"
    i                          j
```

Leetcode : 125

```
if(!isalnum(s[i])) i++;        (Ignores chars other than alphanumeric)
else if(!isalnum(s[j])) j--;
else
    if(tolower(s[i++]) != tolower(s[j--])) return false;
```

680. Valid Palindrome II

Given a string s, return true if the s can be palindrome after deleting at most one character from it.


```
bool checkPalindrome(string s, int i, int j){
    while(i<=j){
        if(s[i] != s[j]){
            return false;
        }
        i++;
        j--;
    }
    return true;
}

bool validPalindrome(string s){
    int i = 0;
    int j = s.length()-1;
    while(i<=j){
        if(s[i] != s[j]){
            return checkPalindrome(s, i+1, j) ||
                   checkPalindrome(s, i, j-1);
        }
        else{
            i++; j--;
        }
    }
    return true;
}
```

539. Minimum Time Difference

"23:59"	"00:00"

→ Step 1

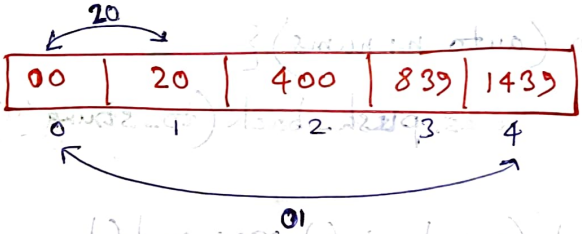
(i) convert the string into Integer
(ii) convert hrs & min into minutes

→ Step 2

(i) Sort the array
(ii) Find the difference between first 2 elements

→ Step 3

(i) Find the last difference between the first and the last elemet and return that



"00:00"	"00:10"	"12:04"	"23:59"

↓ convert stoi

"00:00"	00:10	12:04	23:59

```
int hrs = stoi(time[i].substr(0,2));
int min = stoi(time[i].substr(3,2));
ans[i] = hrs*60 + min;
```

0	10	724	1439

{ Sort the array if not sorted }

mini = min(ans, ans[1] - ans[0]); → 10 ✗ Wrong

mini = min(ans, ans[n-1] - ans[0] + 1440); → 1 ✓ Right

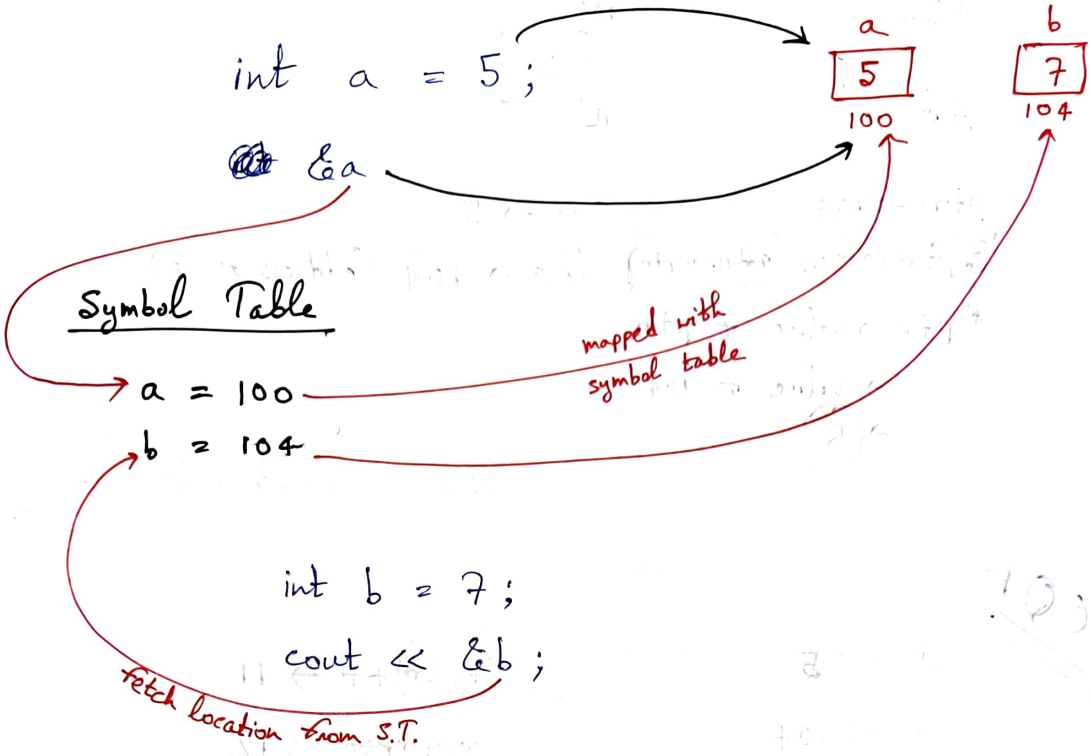
179. Largest Number :-

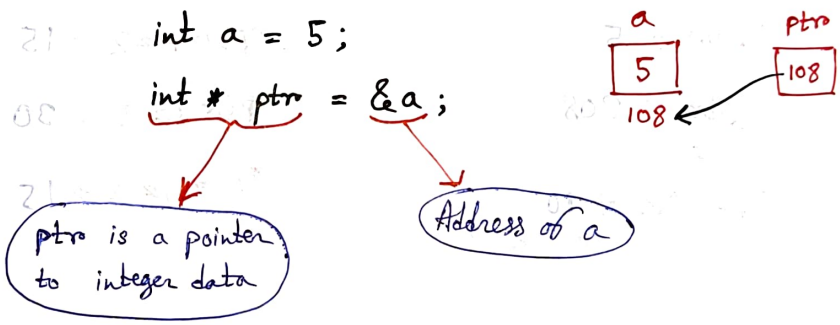
Input: nums = [10, 2]
Output: "210"

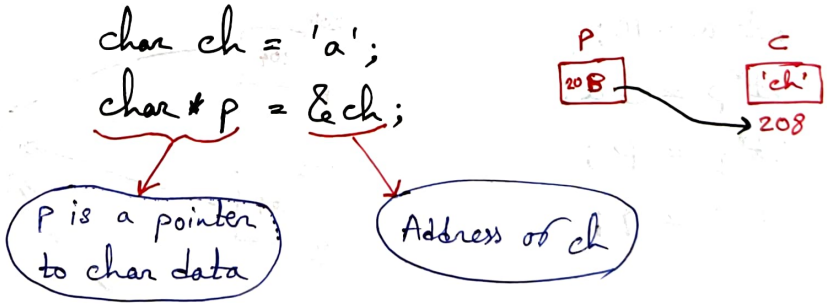
Input: nums = [3, 30, 34, 5, 9]
Output: "9534330"

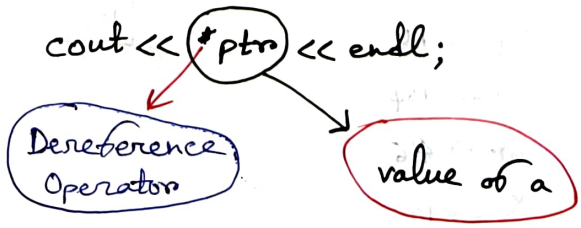
```
static bool comp(string a, string b){
        string s1 = a + b;
        string s2 = b + a;
        return s1 > s2;
}
string largestNumber(vector<int>& nums){
        vector<string> res;
        for(auto n: nums){
                res.push_back(to_string(n));
        }
        sort(res.begin(), res.end(), comp);
        if(res[0] == "0") return "0";
        string ans = "";
        for(auto str: res){
                ans += str;
        }
        return ans;
}
```

Pointers



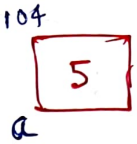






Notes

Output: 5



ptr → 104

&ptr → 208 (address of ptr)

*ptr → value of ptr

→ value of 104

→ 5

a → 5

&a → 104 (address of a)

Size of (pointer) is always 8

MCQ:

a → 5

&a → 104

ptr → 104

*ptr → 5

&ptr → 208

*ptr *2 → 10

(*ptr)++ → 11

++(*ptr) → 12

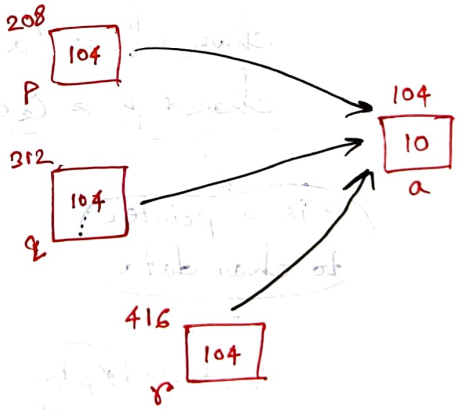
a = a+1 → 13

*ptr += 2 → 15

*ptr *= 2 → 30

*ptr /= 2 → 15

```
int a = 10;
int* p = &a;
int* q = p;
int* r = q;
```



a → 10

&a → 104

p → 104

&p → 208

*p → 10

q → 104

&q → 312

*q → 10

r → 104

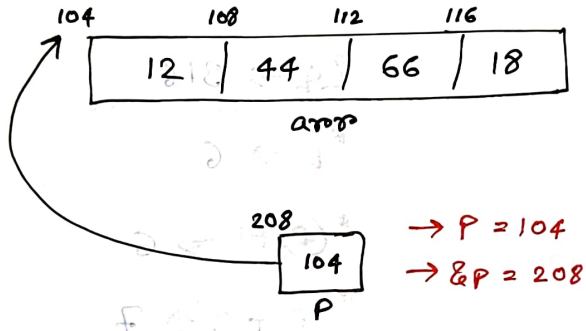
&r → 416

*r → 10

*p + *q + *r → 30

Pointer in Array :-

```
int arr[4] = {12, 44, 66, 18}

cout << arr;     → Base Address → 104
cout << arr[0]; → 12
cout << &arr;    → Base Address → 104
cout << &arr[0]; → Base Address → 104

int* p = arr;
cout << p ;  → Base Address of arr[0] → 104
cout << &p; → Address of p. → 208
```



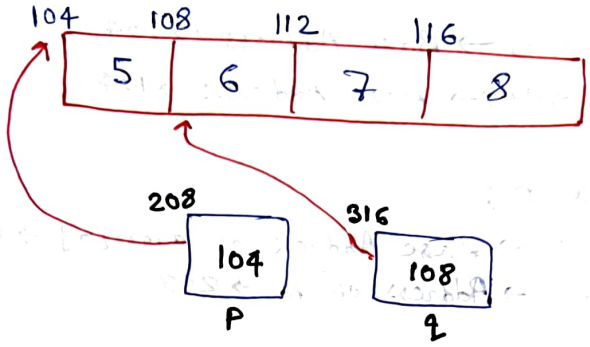
Dereference Operator :-

```
cout << *arr ;     → 12 (Value of base address)
cout << *arr +1 ; → 13 (0th element +1)
cout << *(arr) + 1 ; → 13 (      "      )
cout << *(arr +1) ; → 44 (1st element)
cout << *(arr +2) ; → 66 (2nd element)
cout << *(arr +3) ; → 18 (3rd element)
```

IMPORTANT :-

MCQ:- These are same, {arr[i] == i[arr] == *(arr+i)}

```
int arr[4] = {5, 6, 7, 8};
int *p = arr;
int *q = arr+1;
```



arr → 104
&arr → 104
arr[0] → 5
&arr[0] → 104
p → 104
&p → 208
*p → 5
q → 108
&q → 316
*q → 6
*(p)+1 → 6
*(p)+2 → 7
*(q)+3 → 9
*(q)+4 → Error

Array	Pointer
(i) sizeof(arr) → 40	(i) sizeof(p) → 8/4
(ii) arr = arr+1 can't be used to jump to the next element. Code: int arr{2,6,8,12}; ~~cout<<~~ arr = arr+1; cout<< arr; o/p: Error Reason: You can't change the value inside symbol table.	(ii) p = p+1 can be used to jump to the next element of arrray. Code: int arr{2,6,8,12}; int* p = arr; p = p+1; cout << *p; o/p: 6

① `char ch[10] = "Babbar";`

`char* c = ch;`

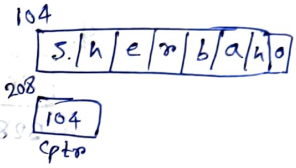
`cout << ch ;` → Babbar

`&ch` → Address of first char `&c` → Address of pointer `c` → Babbar

`ch[0]` → B `*c` → B

② `char name[10] = "SherBano";`

`char* cptr = &name[0];`



`name` → SherBano

`&name` → 104

`*(name +3)` → 'r'

`cptr` → "SherBano"

`&cptr` → 208

`*(cptr + 3)` → 'r'

`cptr +2` → "erBano"

`*cptr` → 'S'

`cptr +8` → "s"

Pointer with Function :-

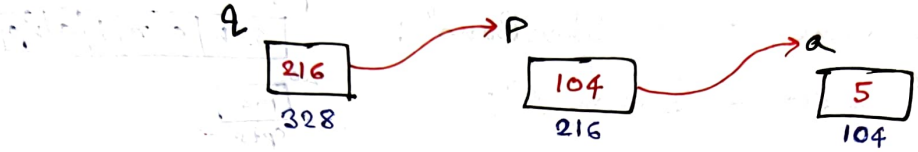
```
solve(int arr[]){
    sizeof(arr); → 8
}
main(){
    int arr[10] = {1,2,3,4,_};
    sizeof(arr); → 40
    solve(arr); → prints the sizeof (pointer)
}
```

pass by reference
↓
pointer passes
↓
creates block in memory
↓
prints memory address

Double Pointers :-

```
int a = 5;
int* p = &a;
int** q = &p;
```

int** → Double Pointer



a → 5	*p → 5	**q → 5
&a → 104	q → 216	
p → 104	&q → 328	
&p → 216	*q → 104	

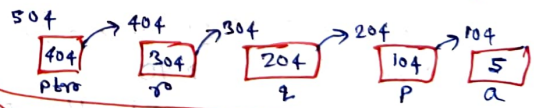
Double Pointer in Function :-

```
void solve(int** ptr){
    **ptr = **ptr + 1;
}
int main(){
    int x = 12;
    int* p = &x;
    int** q = &p;
    solve(q);
    cout << x << endl;   → 13
    return 0;
}
```

***ptr → value stored in location **ptr

**ptr → value stored in location *ptr

*ptr → value stored in location ptr



ptr → 404

*ptr → 304

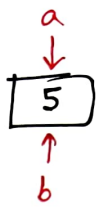
**ptr → 204

***ptr → 104

****ptr → 5

Solved

Reference Variable :- Same memory location but different names! Alternative of pointers__




```
main(){
    int a = 5;
    int& b = a;   ---> Creating a reference variable
    cout << a <<endl;  -> 5
    cout << b <<endl;  -> 5

    a++;
    cout << a ; -> 6
    cout << b ; -> 6
}
```

{ Update in one variable causes update in another variable also }

* Pass by Value → func(a){ a++;} → copy → 6 / 5 ; 5
* Pass by Reference → func(int& a){ a++;} → 6 / 6 [no copy]

```
void solve(int& num){
    num++;
}
main(){
    int a = 5;
    solve(a);
    cout << a <<endl;  -> 6
}
```

pass by reference → { no copy of variable }

Recursion

What is Recursion? → When a function calls itself

↓

Bigger Problem — depend → Small & Same Problem

sol^n

```
void solve(){
    solve();
}
```

Factorial Ka Example

Bigger Problem = 5!

5! → 5 × 4! → 4 × 3! → 3 × 2!

5 × 4! → 4! : Small Problem

Fibonacci Series

0 1 1 2 3 5 8 13 21 ----

n^{th} term = $(n-1)^{th}$ term + $(n-2)^{th}$ term

B.P | S.P | S.P

Recursion Code

- Base Condition (Rukna kab hai?)
- Recursive Relation / Recursive Call ($f(n) = n \times f(n-1)$)
- Processing (Optional)

Factorial Code:

```
int factorial(int n){
    if(n==1)
        return 1;
    cout<<"Function is called for n:"<<n<<endl;
    int ans = n* factorial(n-1);
    return ans;
}
int main(){
    int n;
    cout<<"Enter n: "<<endl;
    cin>>n;
    int ans = factorial(n);
    cout<<"Answer is "<<ans<<endl;
}
```

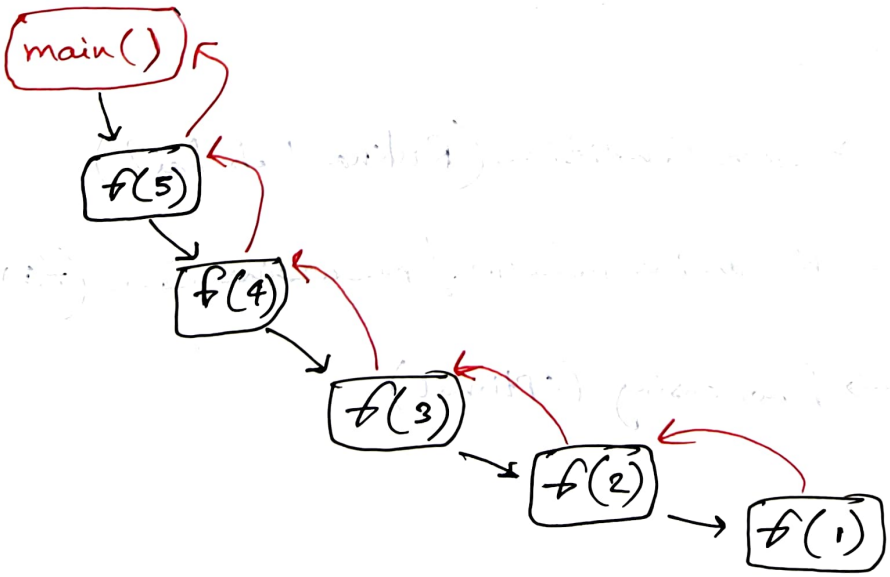
Base Condition

Processing

Recursive Relation

Output:

```
Enter the n: 5
Function is called for n:5
                        :4
                        :3
                        :2
Answer is 120
```



```
void solve(){
    // Base Case
    // Processing
    // R.R
}
```

TAIL Recursion

```
void solve(){
    // Base Case
    // R.R
    // Processing
}
```

HEAD Recursion

```
void print(n){
    if(n==0)
        return;
    cout << n;
    print(n-1);
}
```

Tail

5 4 3 2 1

```
void print(n){
    if(n==0)
        return;
    print(n-1);
    cout << n;
}
```

Head

1 2 3 4 5

Fibonacci

```
int fib(int n){
    if(n==0 || n==1){
        return n;
    }
    return fib(n-1) + fib(n-2);
}
int main(){
    int n;
    cin>>n;
    int ans = fib(n);
    cout << ans;
    return 0;
}
```

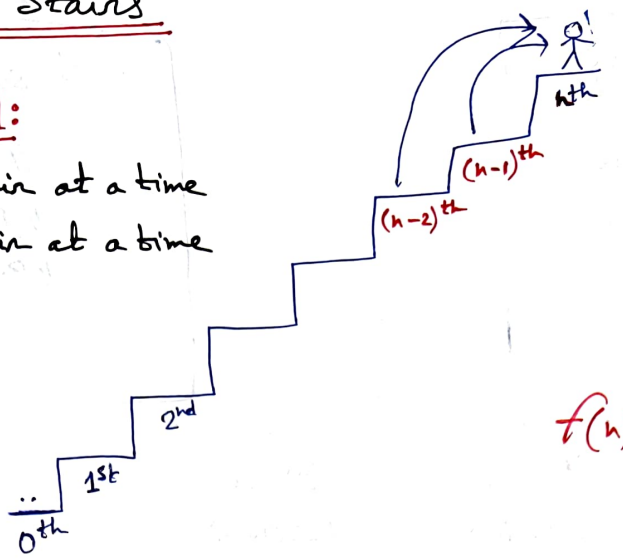
$\therefore T.C = O(2^n)$ — Exponential Time Complexity

$\therefore S.C = O(n)$ — Linear Space Complexity

70. Climbing Stairs

Steps allowed:

- 1 stair at a time
- 2 stair at a time



$f(n)$ ↓

Total no. of ways to reach n^{th} stair

$$f(n) = f(n-1) + f(n-2)$$

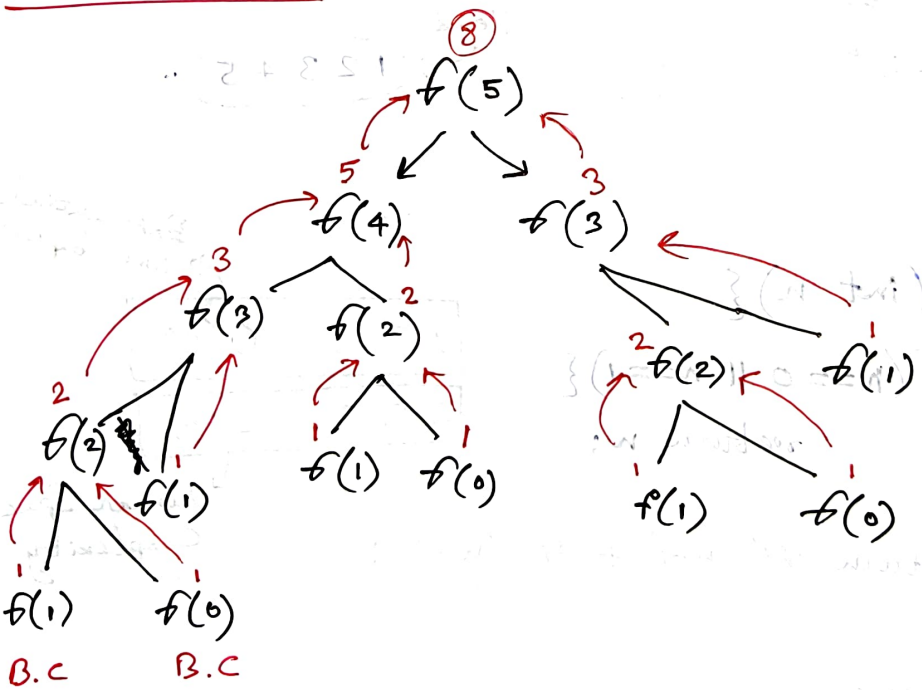
Code:

```
int climbStairs(int n){
    if (n<=1)
        return 1;
    return climbStairs(n-1) + climbStairs(n-2);
}
```

$T.C = O(2^n)$

$S.C = O(n)$

Recursive Tree:



Print/Traverse the Array using Recursion:-

```
int printarr(int arr[], int n, int i){
        if(n<=i)
            return false;
        cout << arr[i] << " ";
        printarr(arr, n, i+1);
}
int main(){
    int arr[5] = {10, 20, 30, 40, 50};
    int n = 5;
    int i = 0;
    printarr(arr, n, i);
    return 0;
}
```

Find Max element using Recursion:-

```
int findMax(int arr[], int n, int i, int& maxi){
        if(i>=n){
            return false;
        }
        if(arr[i] > maxi){
            maxi = arr[i];
        }
        findMax(arr, n, i+1, maxi);
}
int main(){
        int arr[7]= {1, 2, 3, 5, 7, 19, 25};
        int n = 7;
        int i = 0;
        int maxi = INT_MIN;
        findMax(arr, n, i, maxi);
        cout << maxi;
        return 0;
}
```

Binary Search using Recursion:

```
bool binarySearch(arr, s, e){
    // Base Case
    if (arr[mid] == target)
        return true;

    // Processing
    while(s<=e){
        int mid = s+(e-s)/2;
        if (arr[mid] > target)
            e = mid-1; binarySearch(arr, s, m-1);
        else
            s = mid+1; binarySearch(arr, m+1, e);
        mid = s + (e-s)/2;
    }
}
```

Total Calls = $\log n$

Total time/Space =

$\Rightarrow$ (1 call ka time) $\times$ (total calls)

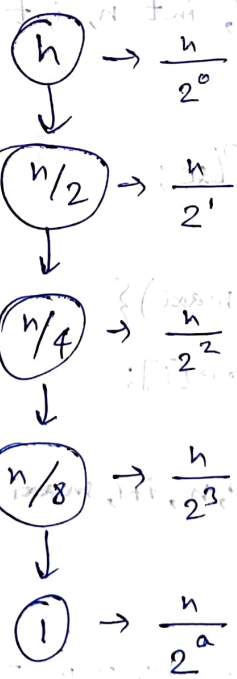
$\Rightarrow K \times \log n$

$\Rightarrow K \log n$

$\Rightarrow O(K \log n)$

$\Rightarrow O(\log n)$

a times



$\therefore \frac{n}{2^a} = 1$

$\Rightarrow n = 2^a$

$\Rightarrow \log n = a$

$\therefore a = \log n$

$\therefore T.C = O(\log n)$

$\therefore S.C = O(\log n)$

881. Boats to Save People:- (IMPORTANT)

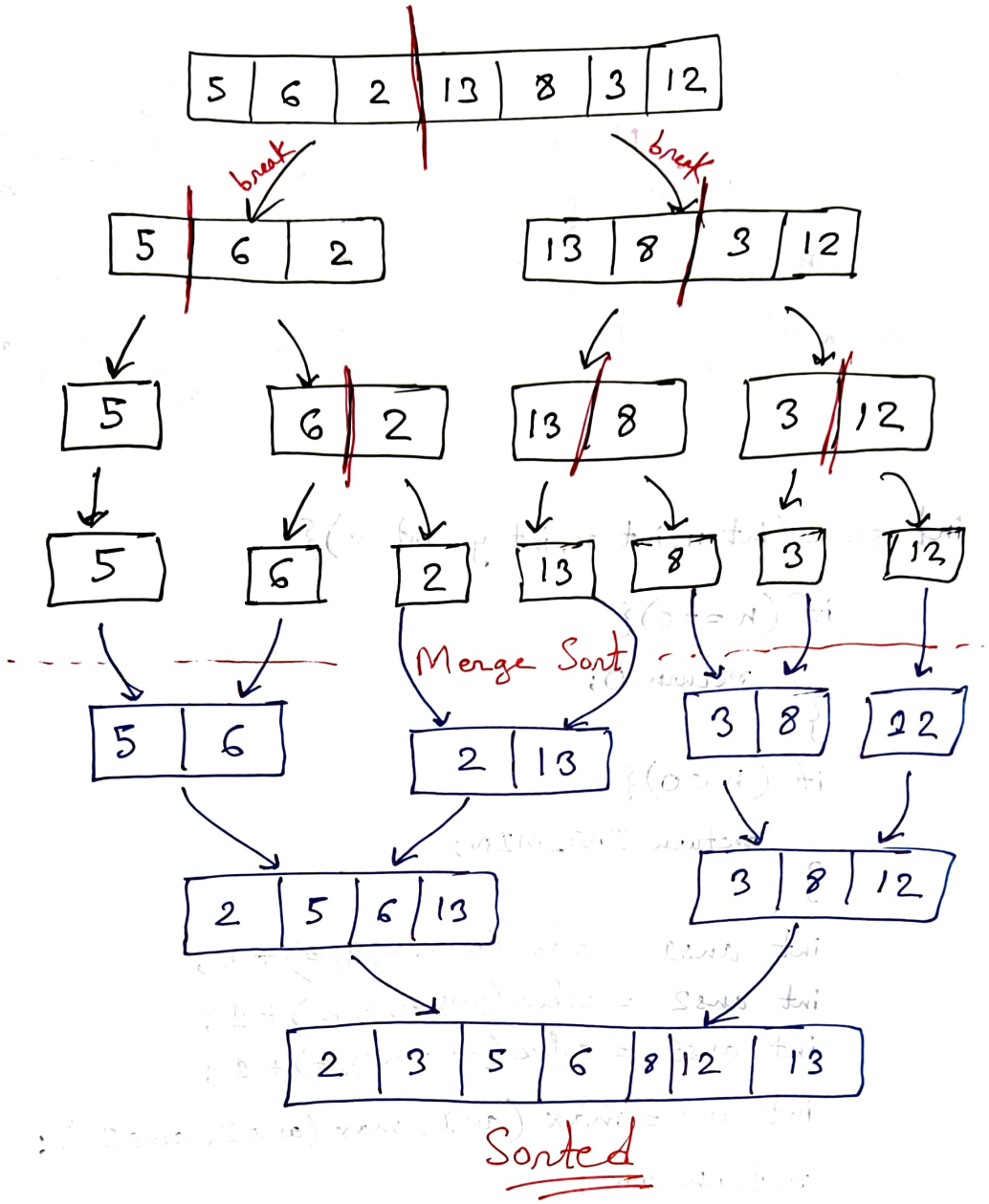
people = [1, 2] limit = 3

Output = 1 Explanation = 1 boat (1, 2)

Given an array People where People[i] represents the weight of ith person.
Each boat can carry a max weight of Limit.
Return the minimum number of boats to carry every given person.

```
int numRescueBot(vector<int>& people, int limit){
    sort(people.begin(), people.end());
    int boat = 0, left = 0, right = people.size()-1
    while(left <= right){
        if(people[left] + people[right] <= limit)
            left++;
            right--;
        }
        else{
            right--;
        }
        boat++;
    }
    return boat;
}
```

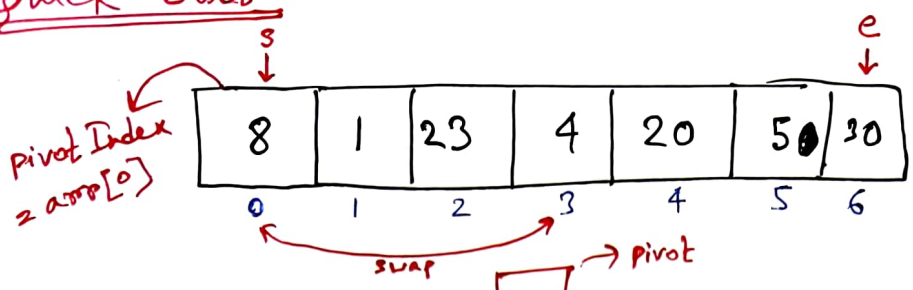
Merge Sort using Recursion :-

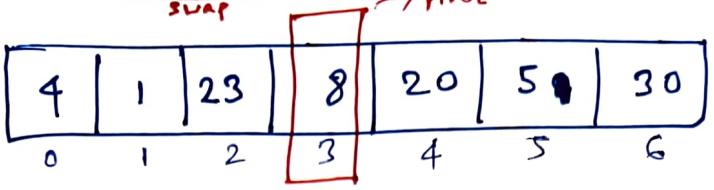


Time Complexity : $O(n \log n)$ Space Complexity : $O(n)$

Code : Sorting → mergeSort.cpp

Quick Sort :-





Step 1 : Partition Logic

(i) pivotIndex = s = 0

(ii) rightIndex = s + count = 3

(iii) swap(arr[pivot], arr[rightIndex])

Kithe elements pivot se chote hain

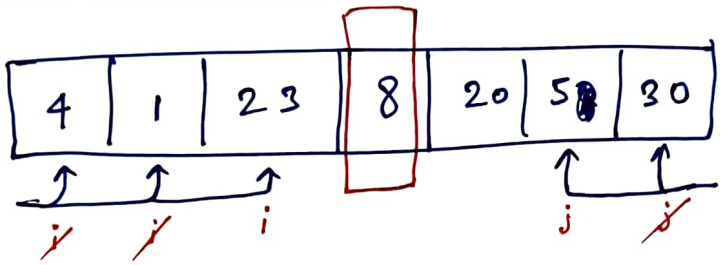
Step 2 : Wrong Element ?

swap (arr[i], arr[j])

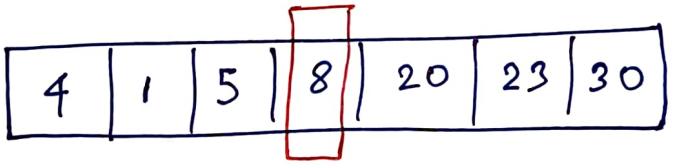
: i++ ; j--;

Left array should be less than Pivot

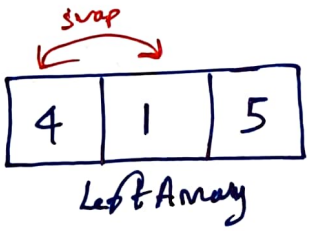
Right array should be greater than Pivot

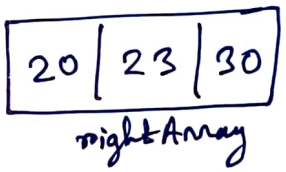


Step 2 completed



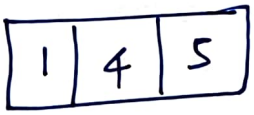
Recursion ka kaam

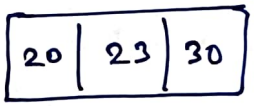




Step 1
Step 2

Step 1
Step 2





Sorted

Time Complexity :- Avg & Best Case → $O(n \log n)$

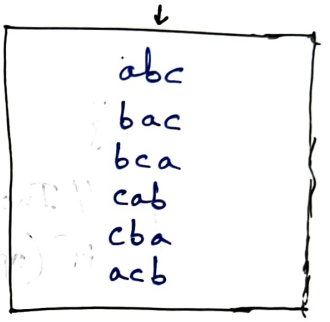
Worst Case → $O(n^2)$

Back Tracking :-

- Explore all possible solutions
- if any of the possible solution discarded / returned False then we will not check that again!

→ Specific form of Recursion ☺

Example : i/p → string → "abc" → All Permutations



Code :-

```
void permutation (string &str, int i){

    //base case
    if(i >= str.length()){
        cout << str << " ";
        return;
    }

    //Swapping
    for(int j=i; j<str.length(); j++){
        //swap
        swap(str[i], str[j]);
        //recursive call
        permutation(str, i+1);
        //backtracking
        swap(str[i], str[j]);
    }
}
int main(){

    string str = "abc";
    int i = 0;
    permutation(str, i);
    return 0;

}
```

} This is the extra part in Recursion / Backtracking

∴ T.C = O(n!)

Output :- abc acb bac bca cba cab

22. Generate Parentheses :-

I/P : n = 3

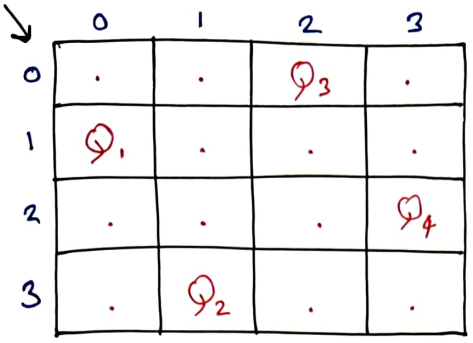
O/P : ["((()))", "()()()", "()(())", "(())()", "(()())"]

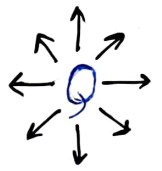
Code :-

```
void Solve(vector<string> &ans, int n, int open, int close, string output) {

        // Base Case
        if (open == 0 && close == 0) {
                ans.push_back(output);
                return;
        }
        // Include Open Bracket
        if (open > 0) {
                // Processing
                output.push_back('(');
                // Recursive Call
                solve(ans, n, open-1, close, output);
                // Backtracking
                output.pop_back();
        }
        // Include Close Bracket
        if (close > open) {
                output.push_back(')');
                solve(ans, n, open, close-1, output);
                output.pop_back();
        }
}
vector<string> generateParentheses(int n) {

        vector<string>ans;
        int open = n, close = n;
        string output = "";
        solve(ans, n, open, close, output);
        return ans;

}
```

N-Queen: Place the queens in such a manner such that no queen can attack the other Queen.

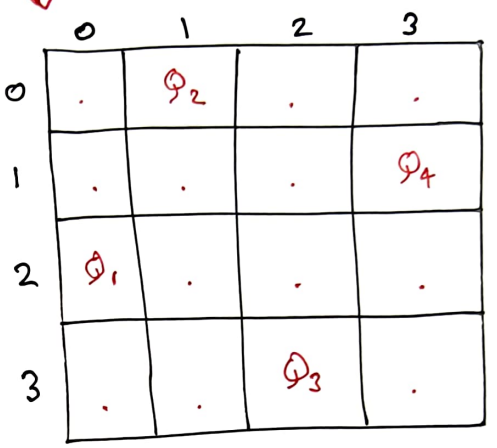
board





8-type Movement

i/p → n → n×n board
→ n queen



2 Solutions for 4×4

Code: C++/Backtracking/N-Queen
or
C++/Backtracking/N-queen-Map.cpp

Leetcode:- N-Queen

T.C = O(n!) → Avg Case

T.C = O(n^n) → Worst Case

S.C = O(n×n) → Space Complexity

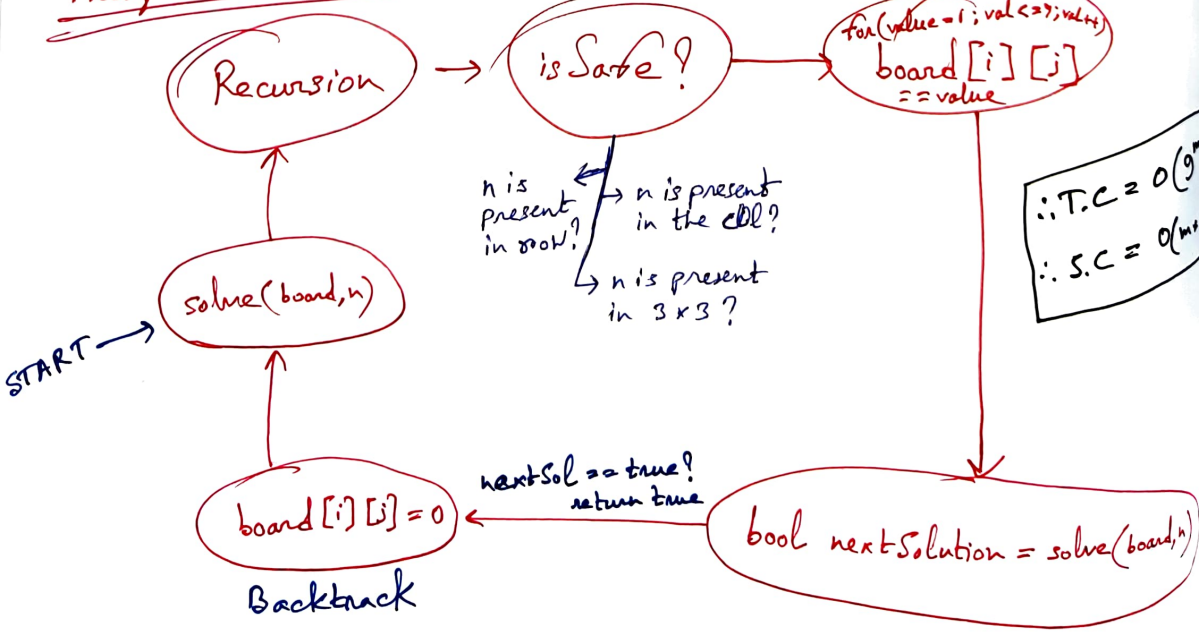
37. Sudoku Solver :-

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

Rule :-

1 - 9 → Should be present
- → Each Row (No Repeat)
- → Each Col (No Repeat)
- → Each 3x3 (No Repeat)

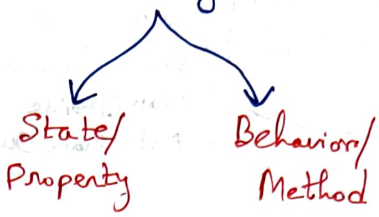
Thought Process :-



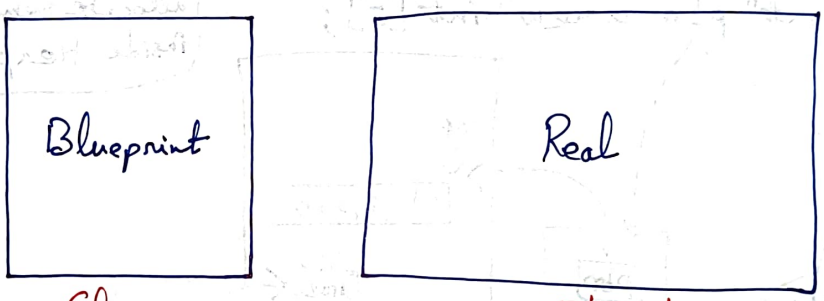
Object - Oriented Programming :-

OOPs is a programming technique where everything revolves around Objects, Classes.

⇒ **Object :-** Object is a entity which have 2 things,



⇒ **Class :-** Class is a set of object which shares common behavior and properties.




```
class Animal{
public:
// state or properties
int age;
string name;

//behaviors
void eat(){

}
void sleep(){

}

};
```

```
int main(){
    //Object creation

    //static
    Animal Dog;
    cout << Dog.age <<endl;
    cout << Dog.name <<endl;

    //Dynamic

}
```

Access Modifiers :-

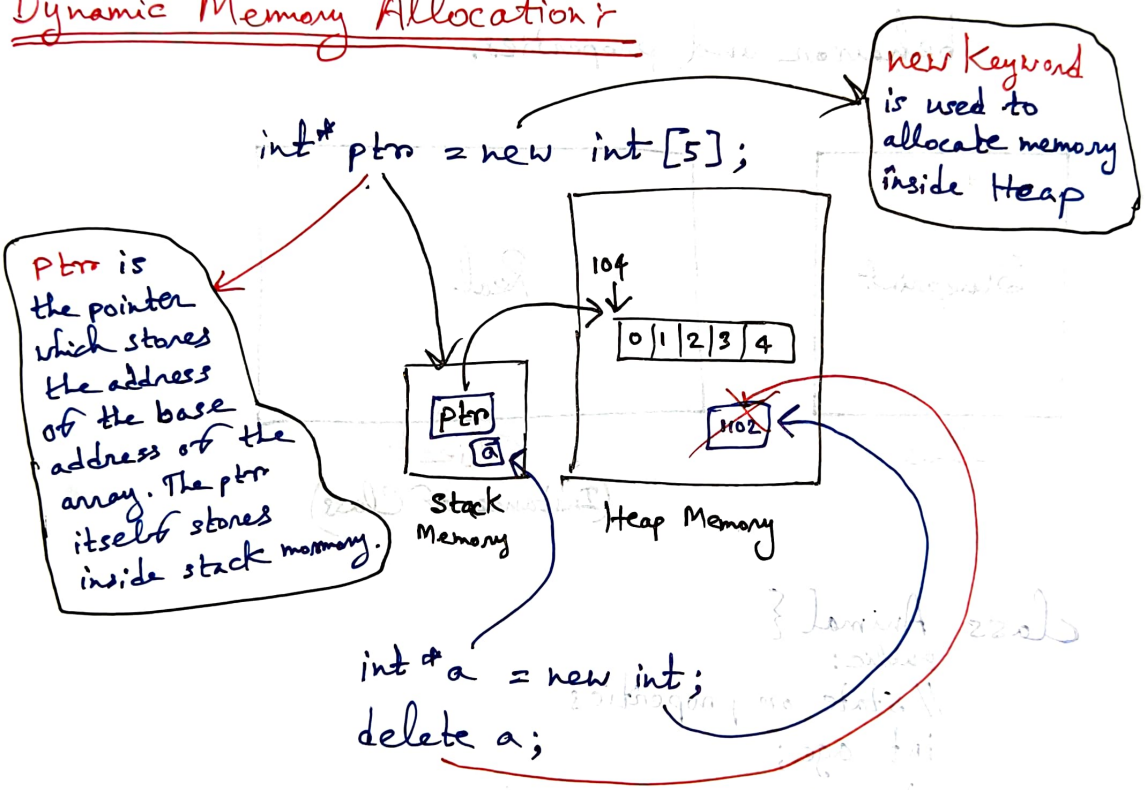
Default

public → Can be accessed from inside class & outside the class

private → Can be accessed from the same class / inside class. Not from Outside.

protected

Dynamic Memory Allocation :-



new ⇒ Allocation of memory inside Heap.

Delete ⇒ De-Allocation of memory inside Heap.

```
int main() {
    //Dynamic memory
    Animal* Billi = new Animal;
    (*Billi).age = 15;
    (*Billi).name = "Meow";
```

or

```
Billi->age = 17;
Billi->name = "piu";
```

```
Billi -> eat();
Billi -> sleep();
cout << "Age of " << Billi -> name << is << Billi -> age;
}
```

THIS Keyword :-

This is a pointer to current object.

```
class xyz {
    int weight;
    void func(int weight){
        this->weight = weight;
    }
}
```

Constructor :-

```
class Animal{
    Animal(){
        cout << "Constructor Called";
    }
}
int main(){
    Animal a;
    Animal* b = new Animal;
}
```

Default Constructor

Output: Constructor Called.
Constructor Called.

Parameterized Constructor:-

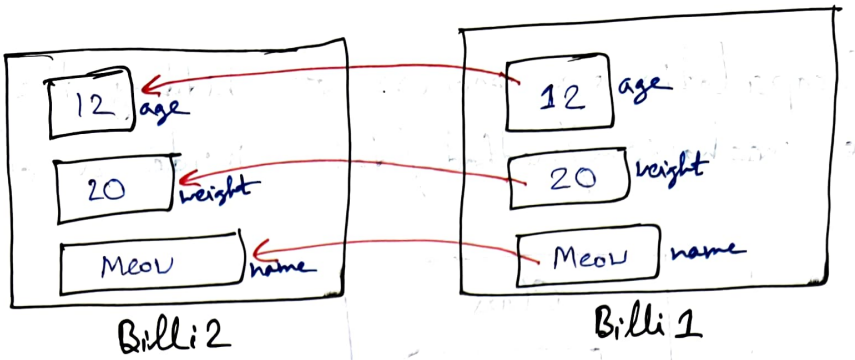
```
class XYZ{
        XYZ(){
            cout<<"Default Constructor";
        }
        XYZ(int x){
            cout<<"Parameterized Constructor";
        }
}
int main(){
    XYZ e(10);
}
```

Output:- Parameterized Constructor.

Copy Constructor:-

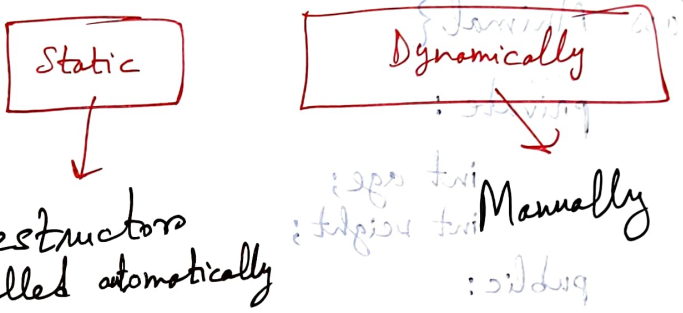
```
class Animal{
        Animal(Animal &obj){
            cout<<"Copy Constructor";
        }
}
int main(){
    Animal a;
    Animal c = a; or Animal b(c); [Object Copy]
}
```

Object Copy



Animal a = b; // Animal a(b);

Destructor (FREE-UP Space)



Syntax:

```
~Animal(){
    cout<<"Destructor Called";
}
```

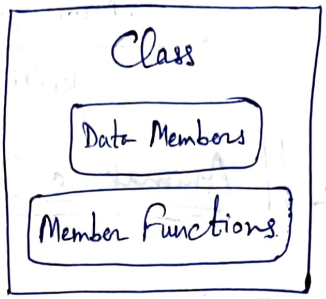
Code:

```
class Animal{
    ~Animal(){
        cout<<"Destructor called";
    }
};
int main(){
    Animal* a = new Animal();
    delete a;
}
```

Output: Destructor Called.

Encapsulation :- (Data Hiding)

Encapsulation is a process of hidding the underline mechanism/method/data inside a class.



Code :-

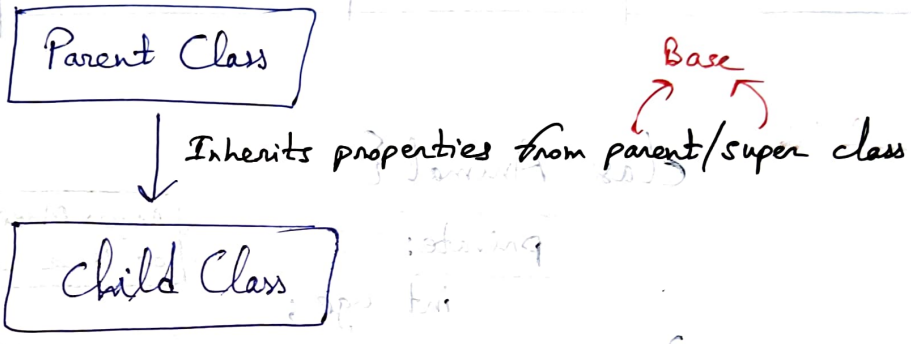
```
class Animal{
    private:
        int age;
        int weight;
    public:
        void eat(){
            cout << "Eating" << endl;
        }
        int getAge(){
            return this->age;
        }
        void setAge(int age){
            this->age = age;
        }
};
int main(){
    Animal* a = new Animal;
    a->setAge(20);
    cout << "Age is: " << a->getAge() << endl;
    return 0;
}
```

Output: Age is : 20

⇒ Full Encapsulation: Jab sara ka sara code private access modifier ke andar rakhenge, we call it Full Encapsulation.

Inheritance :-

The capability of a class to derive properties and characteristics from another class is called Inheritance.

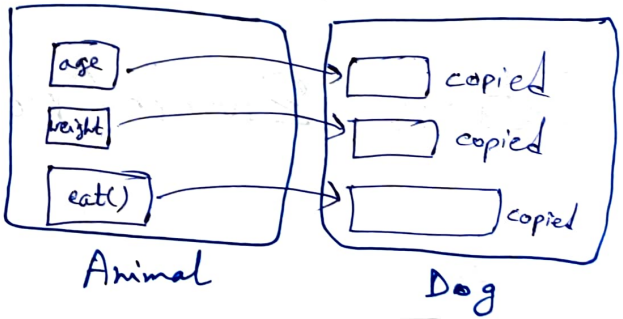


Code :

```
class Animal {
    public:
        int age;
        int weight;
        void eat(){
            cout << "Eating" << endl;
        }
};
class Dog : public Animal {
};
int main(){
    Dog d1;
    d1.eat();
    return 0;
}
```

Mode of Inheritance

Output: Eating



Base Class (parent) Ka Access Modifier	Mode of Inheritance		
	Public	Protected	Private
public	Public	Protected	Private
protected	Protected	Protected	Private
Private	NA	NA	Not Accessable

Code:

```
class Animal {
    private:
        int age;
};
class Dog : public Animal {
    public:
        void print(){
            cout << this -> age;
        }
};

int main(){
    Dog d1;
    d1.print();
    return 0;
}
```

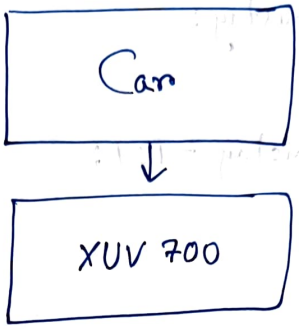
private: → Access Modifier of Base Class

public → Mode of Inheritance

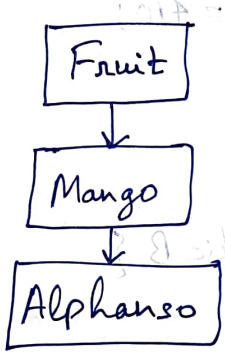
Output: Error (Not Accessible)

Types of Inheritance :-

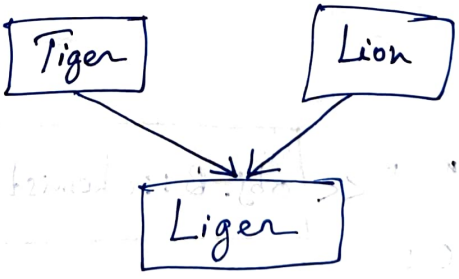
⇒ **Single-Level Inheritance :-** The inheritance in which a derived class is inherited from the only one base class.



⇒ **Multi-Level Inheritance :-** The inheritance in which not only you can derive a class from the base class but you can also derive a class from the derived class.



⇒ **Multiple Inheritance :-** The inheritance in which a class can be derived from more than one parent.



Code :-

```
class A{
    public:
        int physics;
};
class B{
    public:
        int chemistry;
}
```

```
class C : public A, public B{
    public:
        int maths;
}
int main(){
    C obj;
    cout << obj.physics<<obj.chemistry<<obj.maths();
}
```

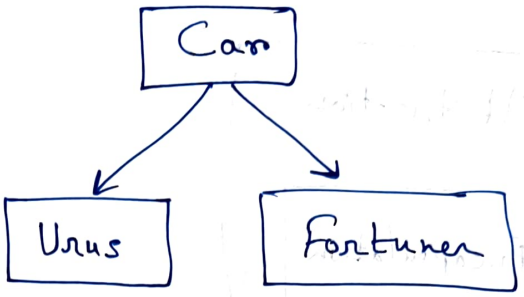
Diamond / Ambiguity Problem :

```
class A{
    public:
        int chemistry;
        A(){
            chemistry = 101;
        }
};
class B{
    public:
        int chemistry;
        B(){
            chemistry = 410;
        }
};

class C : public A, public B {
    public:
        int maths;
};

int main(){
    C obj;
    cout << " " << obj.B::chemistry << endl;
    return 0;
}
```

→ Conflict b/w A and B on Ambiguity can be fixed

→ Scope Resolution Operator

⇒ **Hierarchical Inheritance:-** If more than one class is inherited from the base class, then it's Hierarchical Inheritance.



⇒ **Hybrid Inheritance:-** Combining various types of inheritance like multiple, simple and hierarchical is known as Hybrid Inheritance.

Polymorphism:- Polymorphism means "many forms", and it occurs when we have many classes that are related.

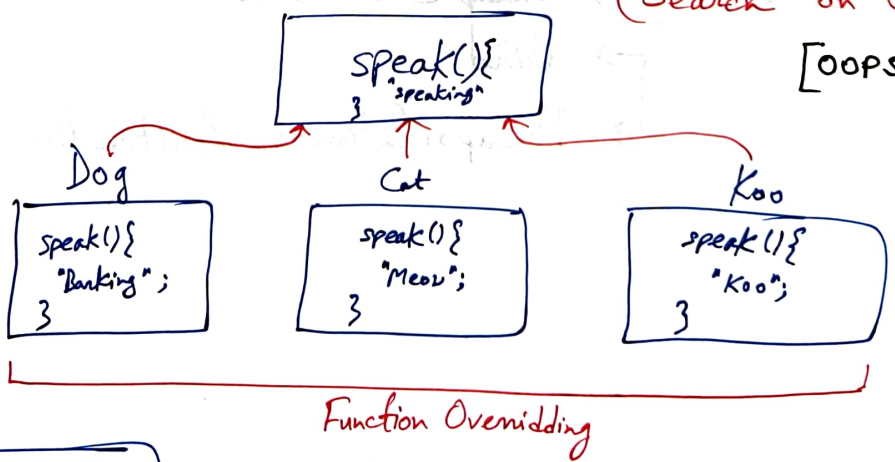
① **Compile-time Polymorphism:-** This type of polymorphism is achieved by Function Overloading & Operator overloading.

⇒ **Funtion Overloading:** Search on GFG

⇒ **Operator Overloading:** Search on GFG

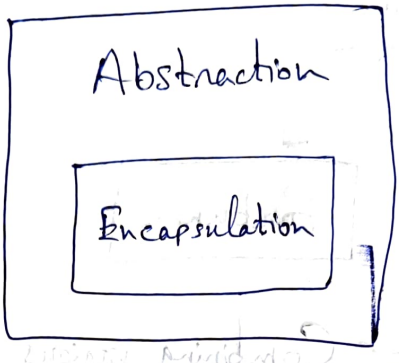
② **Runtime Polymorphism:-** Custom/Virtual Functions. (Search on GFG)

[OOPS Lecture-3]



`virtual` Keyword is important for MCQs

Abstraction :- Abstraction is used to describe things in simple terms. It's used to create a boundary between the application and the client programs.

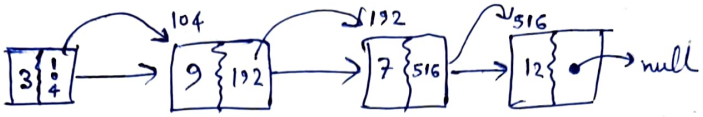


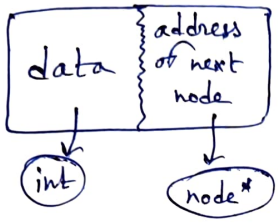
Abstraction is used to achieve generalization of the mechanisms.

⇒ Important Interview Questions :-

1. Encapsulation
2. Inheritance
   - Diamond Problem
3. Polymorphism
   - Basics
   - Compile-Time vs Run-Time
4. Abstraction
   - Example of Sort
   - Bind
   - Encapsulation vs Abstraction

Linked List :- LL is a data-structure that stores data in Non-Linear / Non-Continous memory location by which we can store same type of data in different pieces.



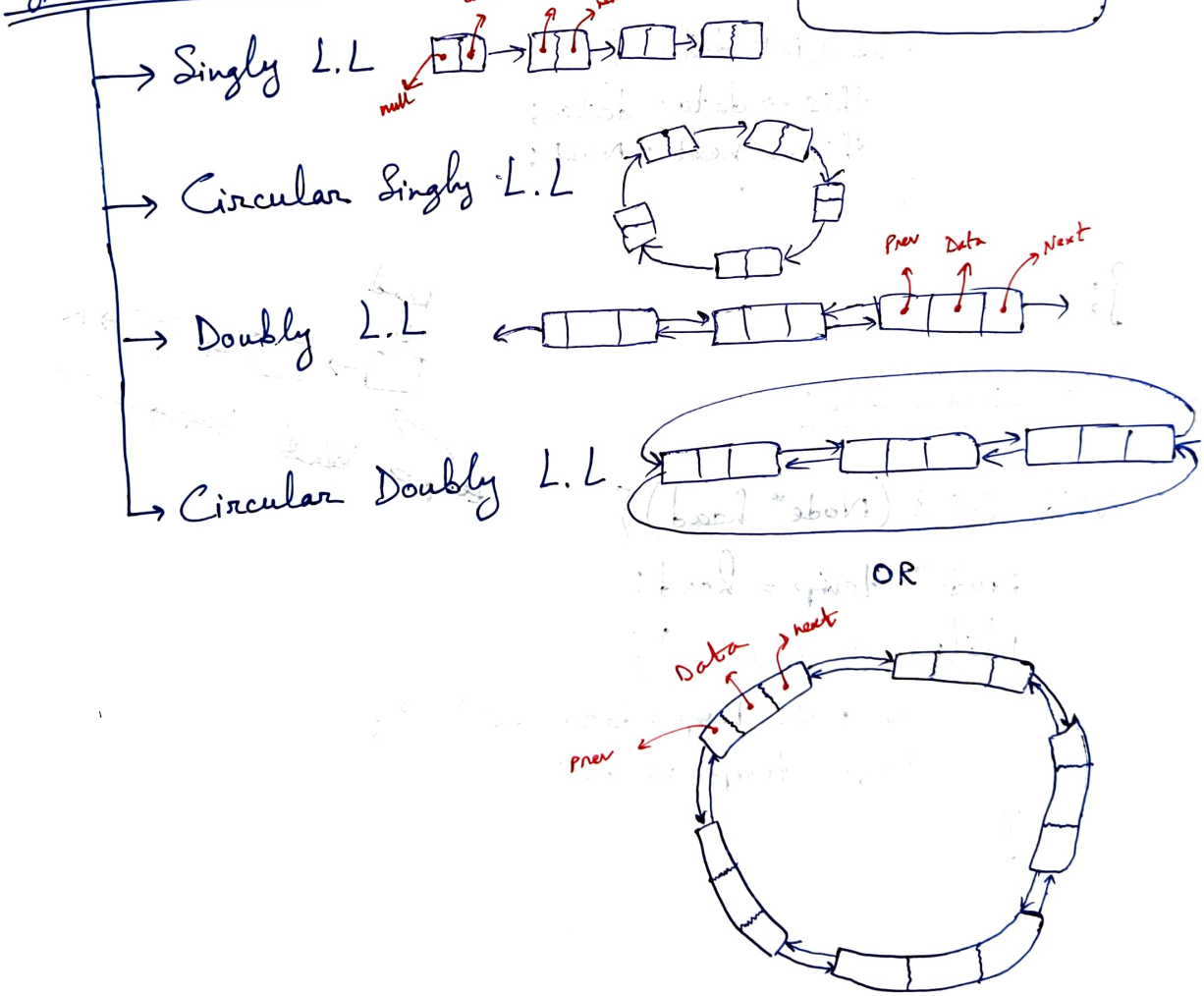


→ Collection of Nodes

Singly Linked list

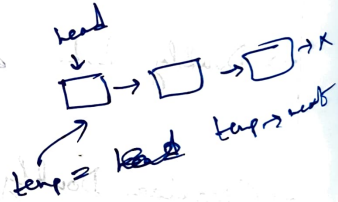
```
class Node {
    int data;
    Node* next;
};
```

Types of L.L :-



Create Node of Linked-List:

```
#include <iostream>
using namespace std;

class Node {

    public:

        int data;
        Node* next;

        Node(){
            this -> data = 0;
            this -> next = NULL;
        }

        Node(int data){
            this -> data = data;
            this -> next = NULL;
        }

};
```



Print Linked List:

```
void print(Node* head){
    Node* temp = head;
    while(temp != NULL){
        cout << temp -> data << " ";
        temp = temp -> next;
    }
}
```

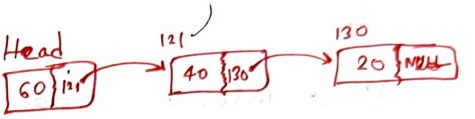
Insert At Head :-

```
void InsertAtHead(Node* &head, Node* &tail, int data){
```

Step 1: `Node* newNode = new Node(data);`

Step 2: `newNode->next = head;`

```
if(head == NULL){
    tail = newNode;
}
```



Step 3: `head = newNode;`

```
}
```

Insert At Tail :-

```
void InsertAtTail(Node* &head, Node* &tail, int data){
```

Step 1: `Node* newNode = new Node(data);`

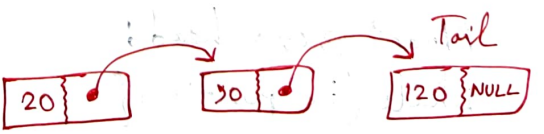
```
if(tail == NULL){
    tail = newNode;
    head = newNode;
}else{
```

Step 2: `tail->next = newNode;`

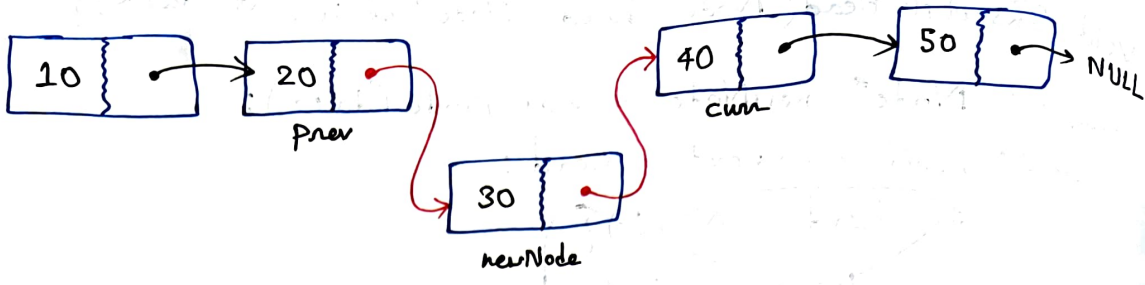
```
}
if(head == NULL){
    head = newNode;
}
```

Step 3: `tail = newNode;`

```
}
```



Insert At Position :-



Code:

```
void insertAtPosition(int data, int position, Node* &head, Node* &tail){
    if (head == NULL){
        Node* newNode = new Node(data);
        head = newNode;
        tail = newNode;
        return;
    }
Insert At Head → if (position == 0){
        insertAtHead(head, tail, data);
        return;
    }
Insert At Tail → int len = findLength(head);
    if (position >= len){
        insertAtTail(head, tail, data);
        return;
    }

    # insert at Middle
    int i = 1;
    Node* prev = head;
    while (i < position){
        prev = prev->next;
        i++;
    }
```

```
    Node* curr = prev -> next;

    Node* newNode = new Node(data);

    newNode -> next = curr;

    prev -> next = newNode;

}

int findLength(Node* &head){

    int len = 0;
    Node* temp = head;
    while(temp != NULL){
        temp = temp -> next;
        len++;
    }
    return len;

}
```

Delete Node :-

```
void deleteNode(int position, Node* head, Node* tail){

    if(head == NULL){
        cout<< "Can't delete, LL is empty"; return;
    }
    // Delete First Node
    if(position == 1){
        Node* temp = head;
        head = head -> next;
        temp -> next = NULL;
        delete temp;
        return;
    }
```

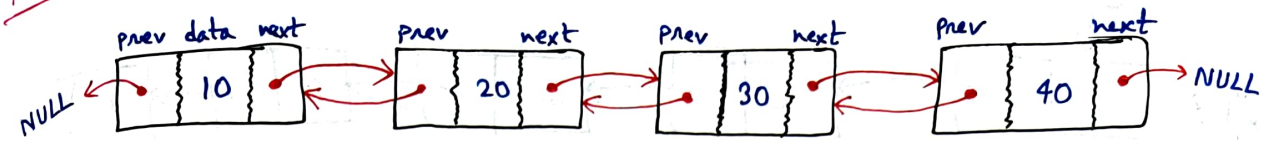
Delete Last Node

```
int len = findLength(head);
if (position == len){
        int i = 1;
        Node* prev = head;
        while (i < position - 1){
            prev = prev -> next;
            i++;
        }
        prev -> next = NULL;
        Node* temp = tail;
        tail = prev;
        delete temp;
        return;
}
```

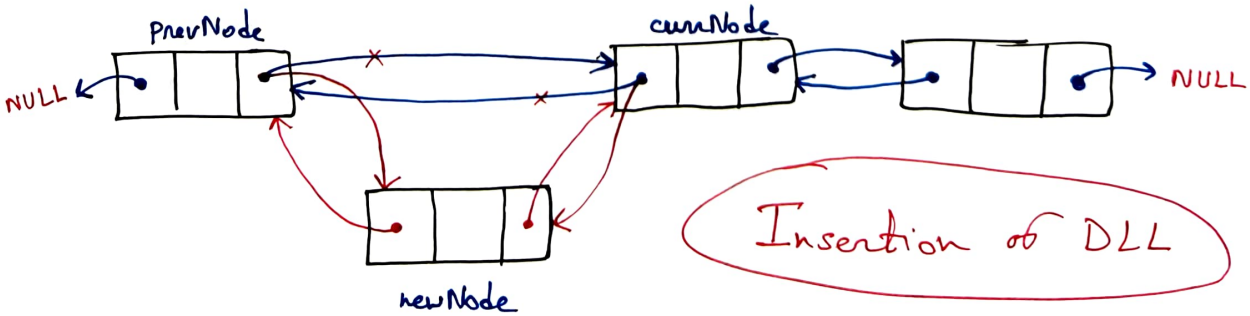
// Delete Middle Node

```
int i = 1;
Node* prev = head;
while (i < position - 1){
    prev = prev -> next;
    i++;
}
Node* curr = prev -> next;

prev -> next = curr -> next;
curr -> next = NULL;
delete curr;

}
```

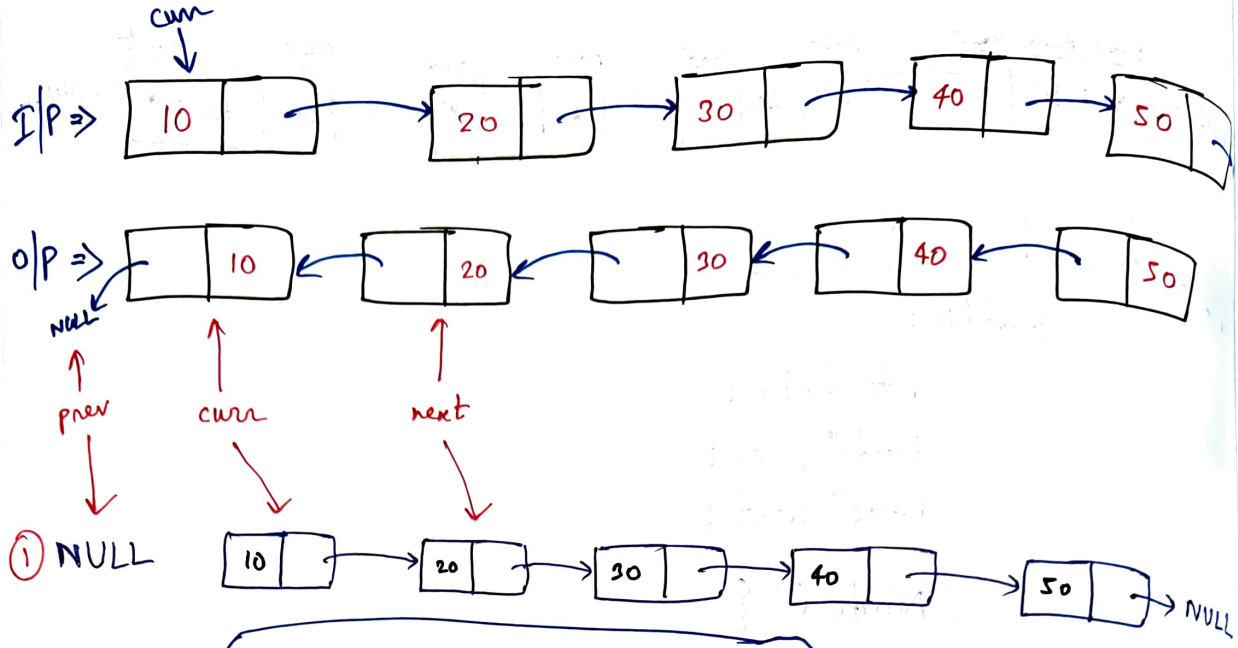
Doubly Linked List




```
class Node {
    public:
        int data;
        Node* prev;
        Node* next;

        Node() {
            this -> data = 0;
            this -> prev = NULL;
            this -> next = NULL;
        }
        Node(int data) {
            this -> data = data;
            this -> prev = prev;
            this -> next = next;
        }
        ~Node() {
            cout << "Node Deleted" << endl;
        }
};
```



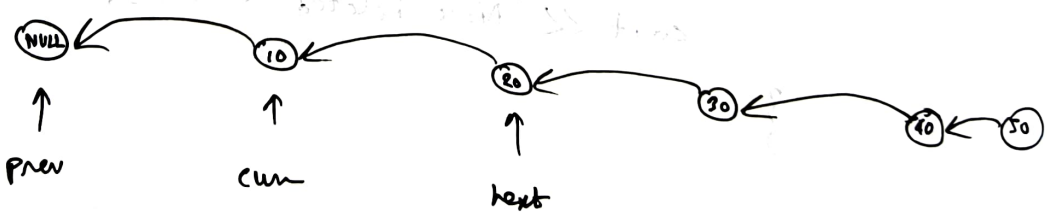
Reverse a LL :-



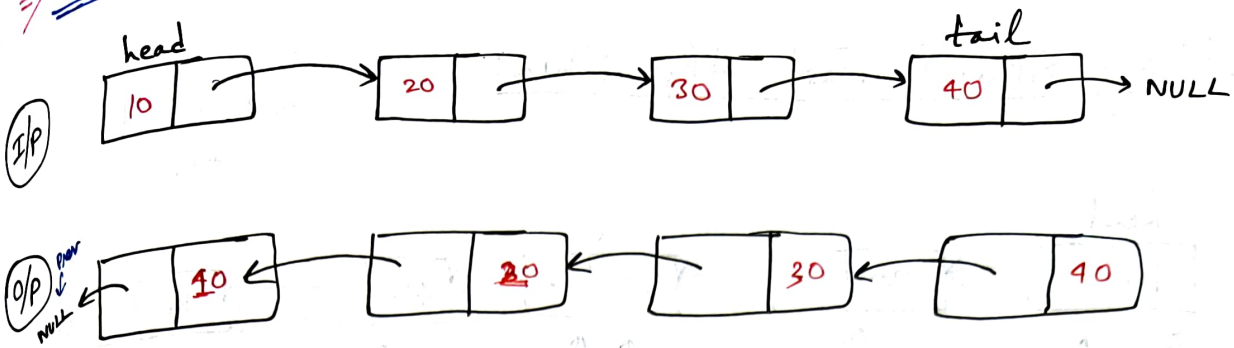

```
while (curr != NULL) {
    next = curr -> next;
    curr -> next = prev;
    prev = curr;
    curr = next;
}
return prev;
```

NULL
10
NULL

②



⇒ Reverse a Linked List :-



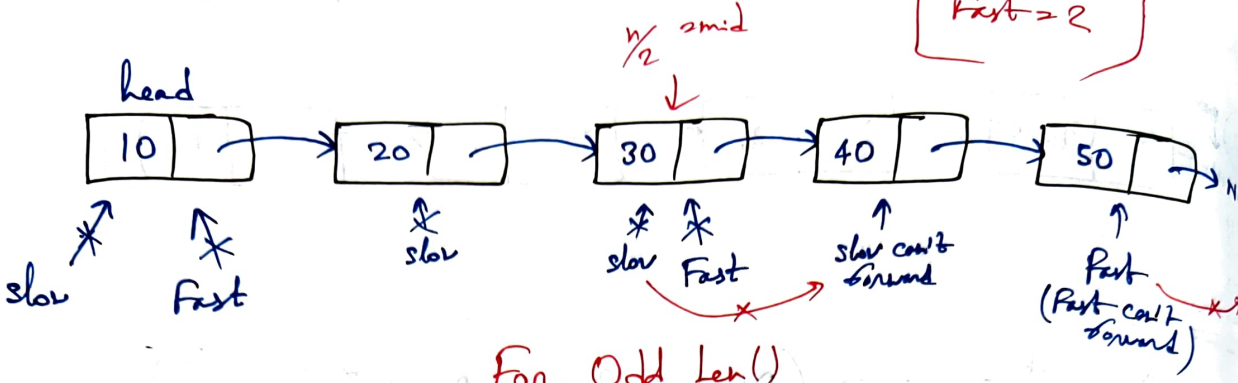
Code:

```
Node* reverse (Node* &prev, Node* &curr){
        //Base Case
        if (curr == NULL){
            return prev;
        }

        //Processing
        Node* temp = curr -> next;
        curr -> next = prev;
        prev = curr;
        curr = temp;

        //recursive call
        return reverse (prev, curr);

}
int main (){
        Node* head = new data (10);
        Node* second = new data (20);
        Node* third = new data (30);
        Node* tail = new data (40);

        Node* prev = NULL;
        Node* curr = head;
        head = reverse (prev, curr);
        Print (head);
}
```

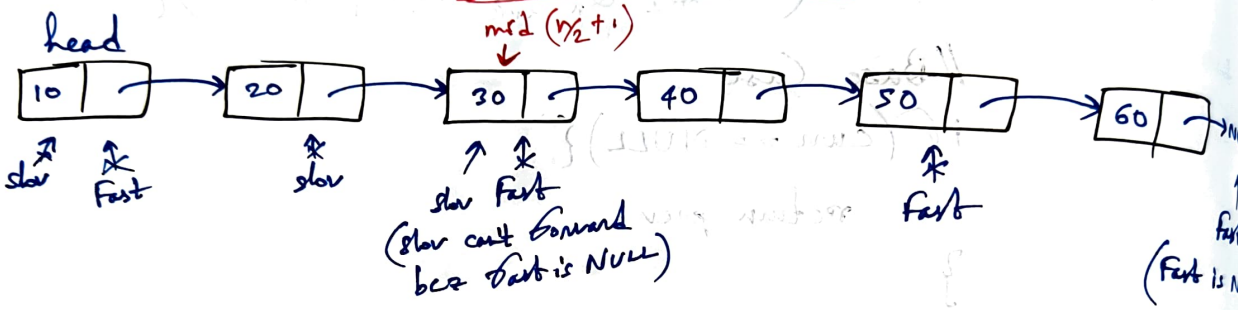
Find Middle Node of Linked List :-

⇒ Using Slow & Fast Algorithm:

slow = 1
Fast = 2



For Odd Len()



For Even Len():

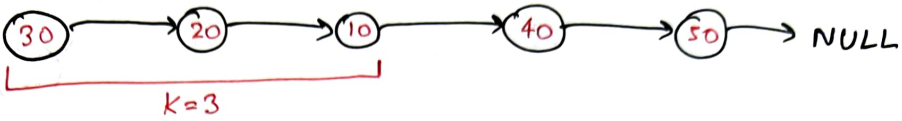
```
Node* getMid (Node* &head){

    Node* slow = head;
    Node* fast = head->next;
    while(fast != NULL){
        fast = fast->next;
        if(fast != NULL){
            fast = fast->next;
            slow = slow->next;
        }
    }
    return slow;

}
```

⇒ Reverse K Nodes :-

I/p

K = 3

O/p



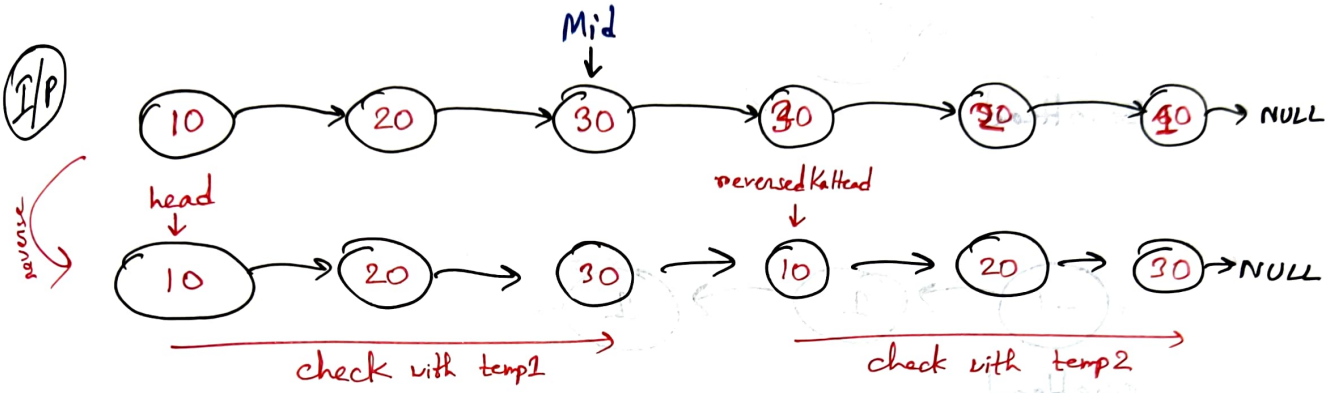

```
Node* reversekNodes (Node* &head, int k){

    Node* prev = NULL;
    Node* curr = head;
    Node* forward = curr -> next;
    int count = 0;
    while (count < K){
            forward = curr -> next;
            curr -> next = prev;
            prev = curr;
            curr = forward;
            count ++;
    }
    if (forward ≠ NULL){
        head -> next = reversekNodes (forward, k);
    }
    return prev;
}
```

⇒ Check Palindrome :-

 # Step 1 :- Find Mid of LL

 # Step 2 :- Reverse the Nodes after mid

 # Step 3 :- Check with two pointers whether the head & ReversedKaHead LL are same or not




```
if (temp1 -> data == temp2 -> data)
```

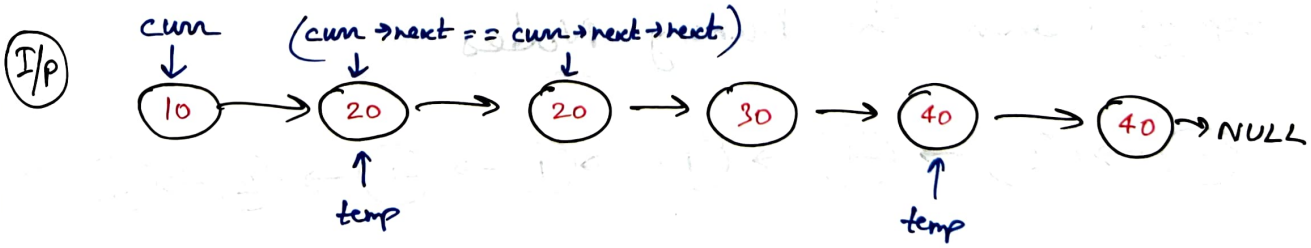
↓

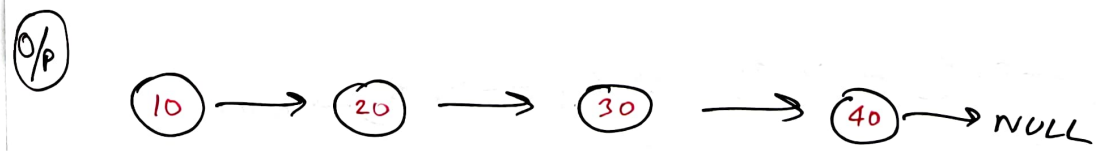
Valid Palindrome

⇒ Remove Duplicates :-

 # Step 1 :- Check whether curr/head is NULL, if not then check (curr -> next ≠ NULL) && (curr -> data == curr -> next -> data) then

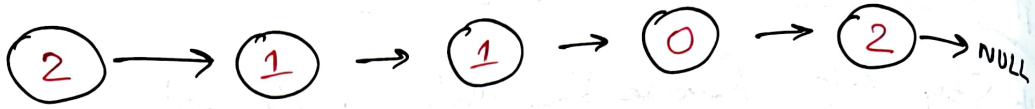
 # Step 2 :- Create temp Node, map it to curr -> next -> next, then delete temp otherwise curr = curr -> next



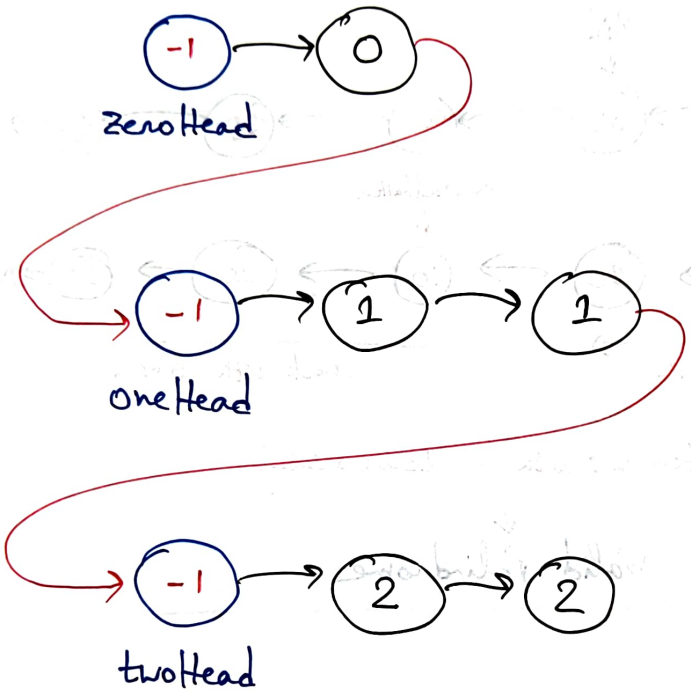


Sort 0 1 2 s :-

I/p

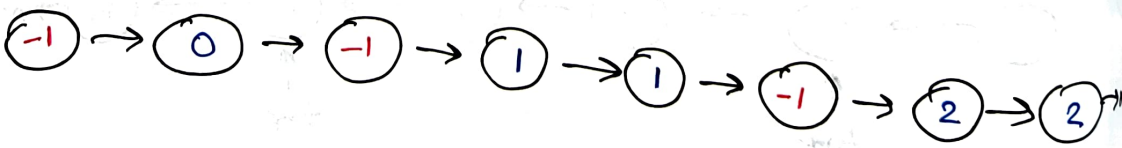


Step 1: Create Dummy Nodes & append to related nodes.

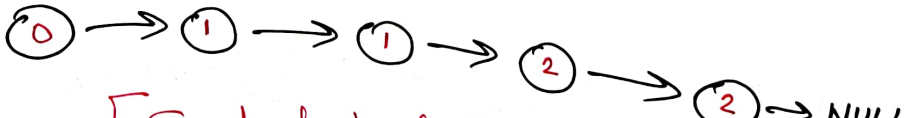


Step 2: Add the Nodes to each other if the LL is not NULL

Step 3: Remove the Dummy Nodes



Ans:



[Sorted Linked List]

Add Two Numbers :-

Step 1: Reverse the both LLs

Step 2: Add both LLs

```
while (head1 ≠ NULL && head2 ≠ NULL){
    int sum = carry + head1->data + head2->data;
    int digit = sum % 10;
    carry = sum/10;
    Node* newNode = new Node(digit);
    if (ansHead == NULL){
        ansHead = newNode;
        ansTail = newNode;
    }else{
        ansTail->next = newNode;
        ansTail = newNode;
    }
    head1 = head1->next;
    head2 = head2->next;
}
```

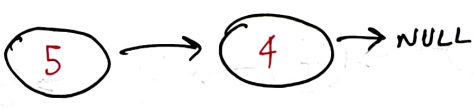
Case 1:
```
while(head1 ≠ NULL){
    ____
}
```

Case 2:
```
while(head2 ≠ NULL){
    ____
}
```

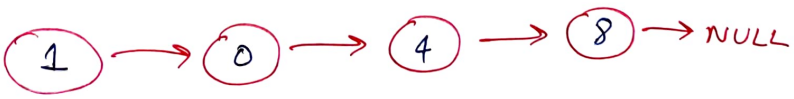
Case 3:
```
while (carry ≠ 0) {
    ____
}
```

Step 3: Reverse the ans LL

I/p

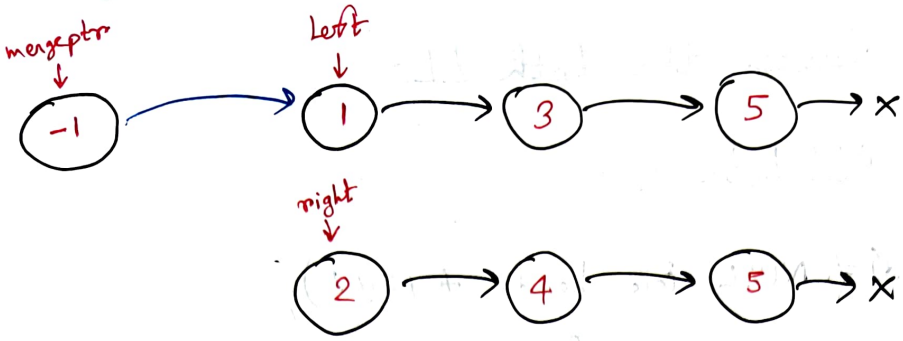


O/p



Merge Two Sorted List :-

LeetCode : 21



Step 1 : Create Merge Pointer (mptr), left & Right pointer to keep track the value of numbers.

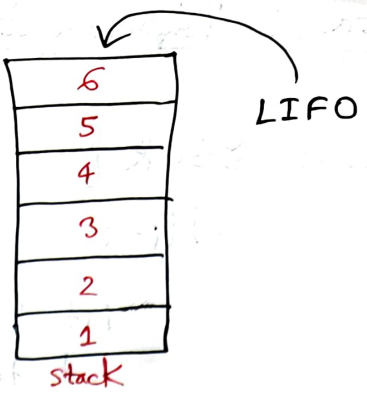
Step 2 :

```
if (left -> value <= right -> value){
        mptr -> next = left;
        mptr = left;
        left = left -> next;
}
else {
        mptr -> next = right;
        mptr = right;
        right = right -> next;
}
```

Step 3 : ~~Remove mergeptr~~ Return ans -> next

Output LL

Stack (Data Structure)

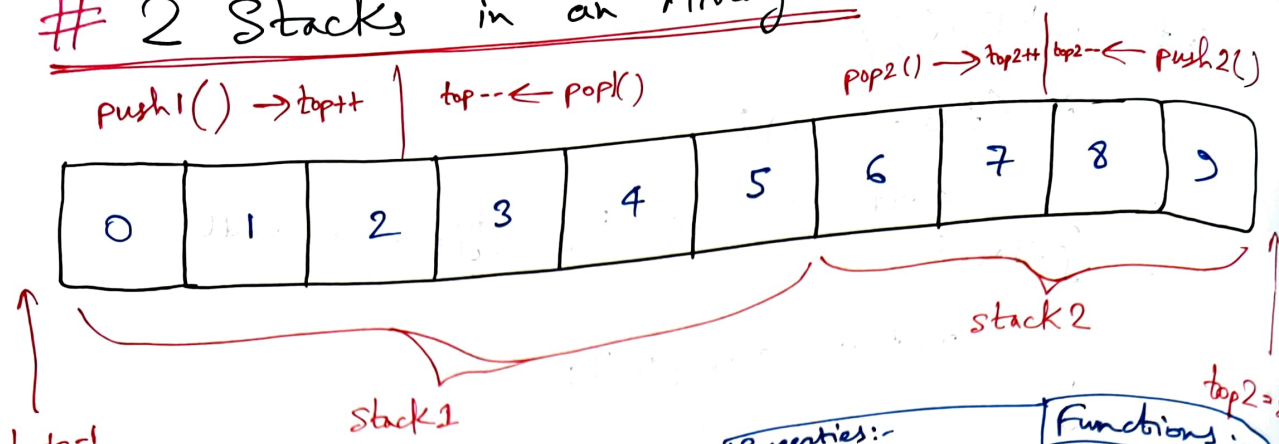


- ⇒ s.push(6)
- ⇒ s.pop(6)
- ⇒ s.top() → 5
- ⇒ s.empty() → False
- ⇒ s.size() → 5 [no. of elements]

Code:

```
#include <stack>
int main(){
    #Creation
    Stack<int> s;

    #insertion
    s.push(10);
    s.push(20);

    #remove
    s.pop();

    #check top element
    s.top();

    #Size of stack
    s.size();

    #Empty?
    s.empty()? cout "yes" : cout "No";

    return 0;

}
```

2 Stacks in an Array :-




```
if (top2 - top1 == 1){
    cout << "No space" << endl;
}
```

Properties :-	Functions :-	
int *arr; int size; int top1; int top2;	push1()	pop1()
	push2()	pop2()

Reverse String :-

using Stack

Code:

```
int main(){
    string str = "dipayan";
    stack<char> s;
    for (int i=0; i<str.length(); i++){
        s.push(str[i]);
    }
    while (!s.empty()){
        cout << s.top() << " ";
        s.pop();
    }
    cout << endl;
    return 0;
}
```

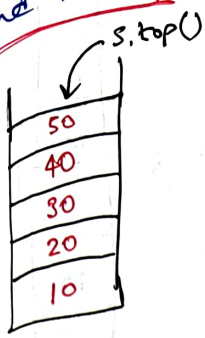
Create Stack

Insert in Stack

Print the stack elements

Nayapid

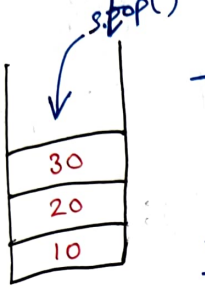
Find Mid :-



 # Step 1: Find S.size()

 # Step 2:

```
if (totalSize/2 + 1 == s.size()){
    return s.top();
}
```



 # Step 3:

```
int temp = s.top();
s.pop()
```

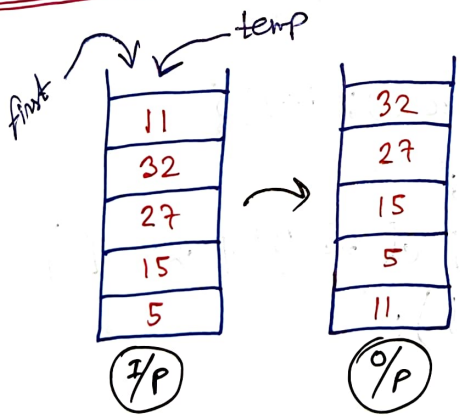
} Create temp

 # Step 4: Recursive Call : `findMin(s, totalSize);`

 # Step 5: Backtracking : `s.push(temp);`

I/p 10 20 30 40 50 o/p 30 Ans

Insert At Bottom :-



 # Step 1: Store the top/first element and Pop that

 # Step 2: Create temp and traverse to the end while !s.empty()

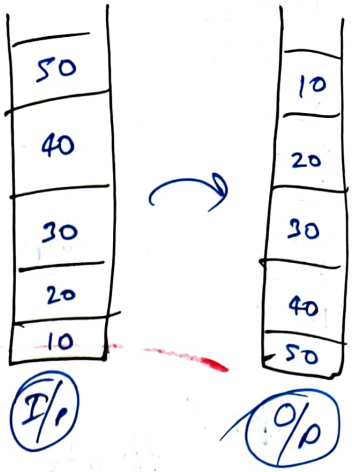
 # Step 3:

```
if (s.empty()){
    s.push(first);
}
```

 # Step 4: Recursive Call : solve(s, first)

 # Step 5: Backtracking : s.push(temp)

#Reverse a Stack:



#Step1: Find first element using S.top() and create a temp variable on the same

#Step2: Check if the stack is empty the return otherwise temp = S.top(); S.pop(

#Step 3: Recursive call: reverse(S, first);

#Step 4: Backtracking: S.push(temp);

=> Valid Parenthesis:

IMPORTANT!

```
bool isValid(string s){
    stack<char> st;
    for(int i=0; i<size(); i++){
        char ch = s[i];
        if(ch == '(' || ch == '{' || ch == '[')
            st.push(ch);
        else{
            if(!st.empty()){
                char topCh = st.top();
                if(ch == ')' && topCh == '('){
                    st.pop();
                }else if(ch == '}' && topCh == '{'){
                    st.pop();
                }
```

```
else if (ch == ']' && ch == '[') {
        st.pop();
    }
    else {
        return false;
    }
}
else {
    return false;
}
}
return st.empty();
}
```

Similar Question:

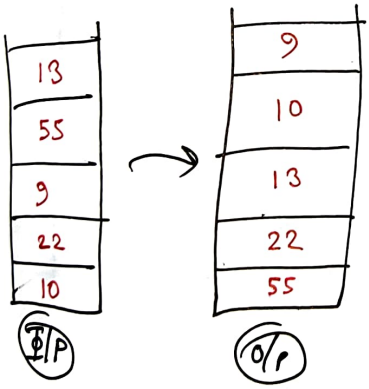
① Remove Redundant Parenthesis

$((a+b)) \rightarrow (a+b)$

$(\{\}(a+b)\} \rightarrow (a+b)$

$(a+(b*c+\{d\})) \rightarrow a+(b*c+d)$

Sort a Stack :-



 # Step 1: If the stack is empty then push the first/top element into the stack

 # Step 2:
```
if (s.top() >= first) {
    s.push(first);
    return;
}
```

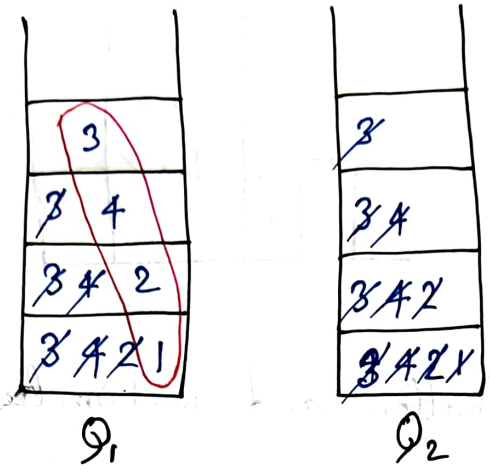
 # Step 3: `int temp = s.top(); s.pop();`

 # Step 4: Recursive call: `insertSorted(s, first);`

 # Step 5: Backtrack: `s.push(temp);`

#Implement Stack using Queues :

push(3)
push(4)
push(2)
push(1)
top() → 1
pop()
top() → 2



T.C → O(n) | S.C → O(n)

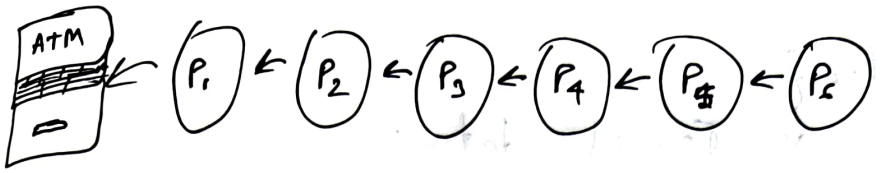
Steps :

push (x)
↳ Add x → Q2
↳ Q1 → Q2
↳ swap (Q1, Q2)

pop()
↳ Remove the top of Q1

Queue

→ FIFO (Fast In Fast Out)



Using STL :-

```
#include <queue>
int main(){
    queue<int> q; → creation
    q.push(10);
    q.push(25);
    q.push(40);
    q.push(65);
```

} Insertion

```
    q.size(); → return size of q
    q.pop(); → pop the first element
    q.empty(); → bool function
    return 0;
}
```

Without STL :-

using Array pointer

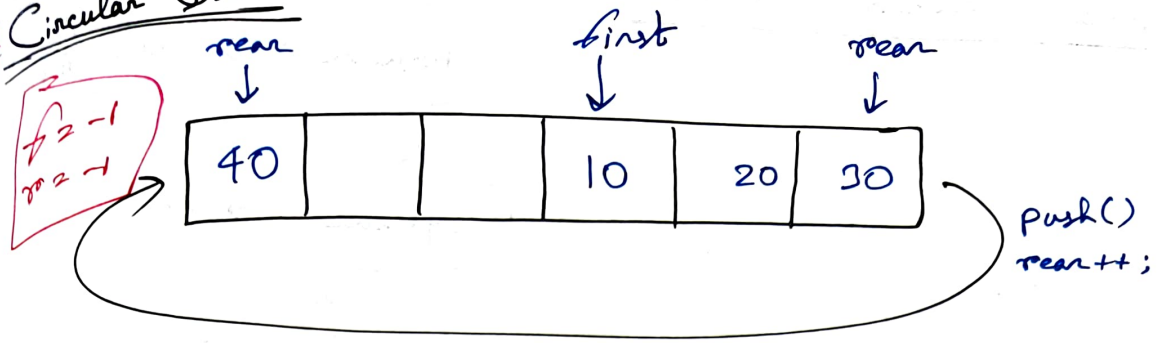
```
class Queue{
  public:
    int *arr; int size, front, rear;
    Queue(int size){
        this->size = size;
        arr = new int[size];
        front = 0, rear = 0;
    }
```

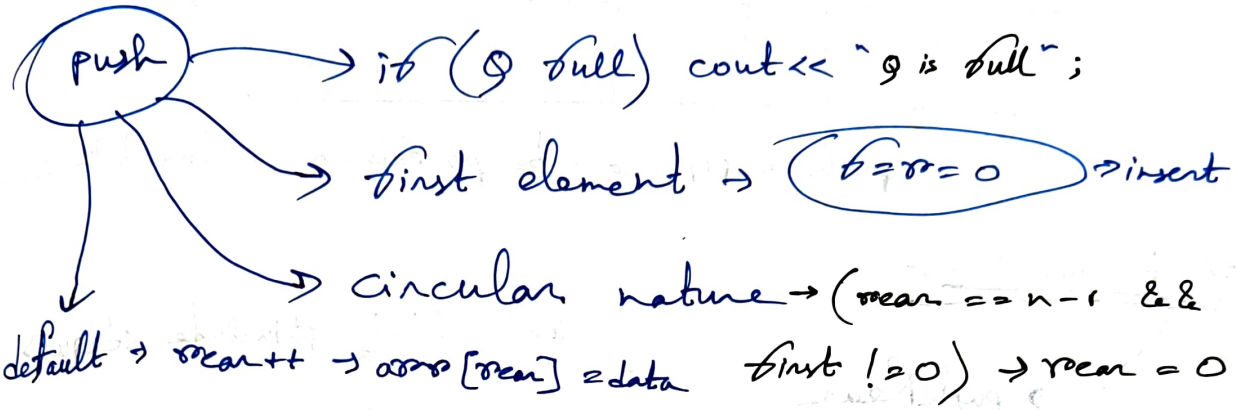
Parameterized Constructor

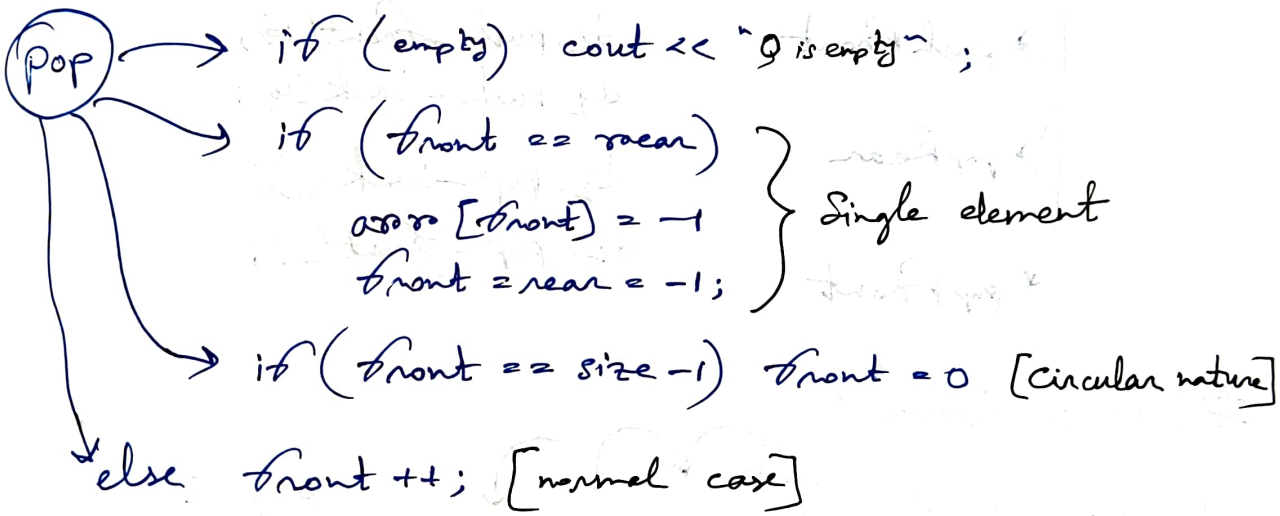
```
void push(int data){
    if (rear == size){
        cout << "Q is full";
    }else{
        arr[rear] = data;
        rear++;
    }
}

void pop(){
    if (front == rear){
        cout << "Q is empty!";
    }
    else{
        front++;
        if (front == rear){
            front = 0;
            rear = 0;
        }
    }
}

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

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

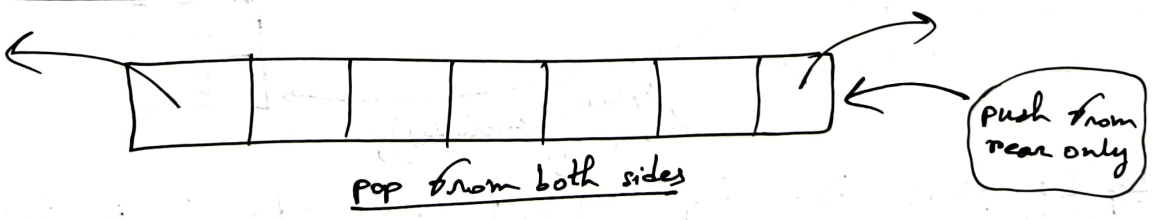
Circular Queue



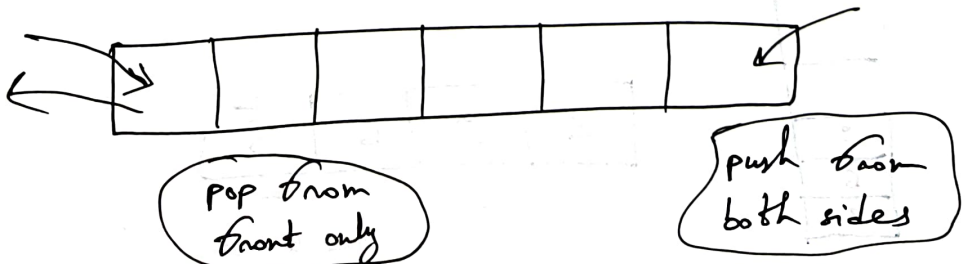




i/p Restricted Queue :



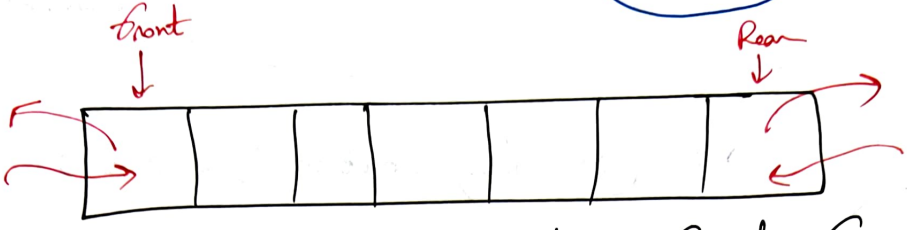
o/p restricted Queue :-



Doubly Ended Queue :-

enqueue → push / insert

dequeue → pop



push & pop can be performed from both sides

#4 Functions :-

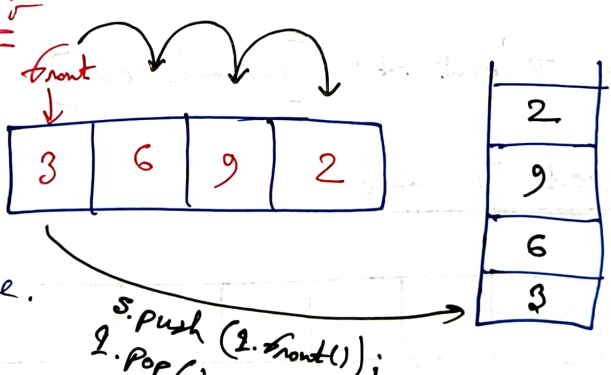
- pushRear
- pushFront
- popRear
- popFront

Using STL : #include <deque>

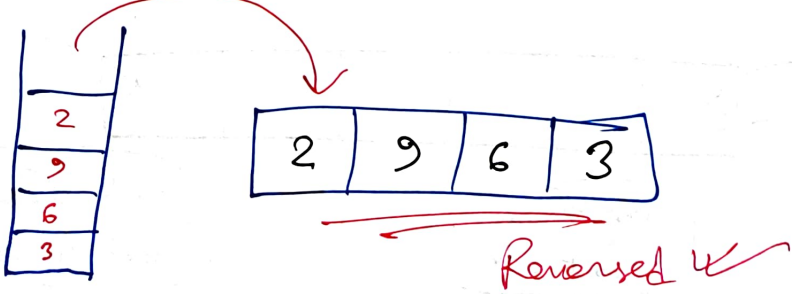
```
deque <int> dq;
dq.push_front(10);
dq.push_back(20);
dq.pop_back();
dq.pop_front();
```

Reverse Queue :-

Step 1: Push the front element of queue to the stack and pop and traverse.



Step 2: Push the top element of stack in the queue and pop that, traverse through the stack.

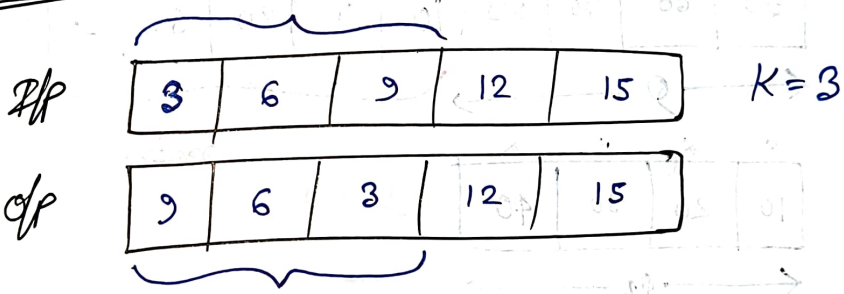


Reverse Queue using Recursion :-

```
void reverse (Queue<int> &q) {
    if (q.empty) return;          <-- Base Case
    int temp = q.front();         } Logic
    q.pop();
    reverse (q);                  <-- recursive call
    q.push (temp);                <-- back tracking
}
```

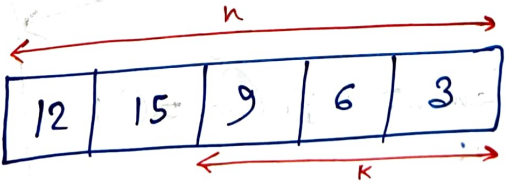
Reverse-K-Elements :-

Iterative Approach [Best]

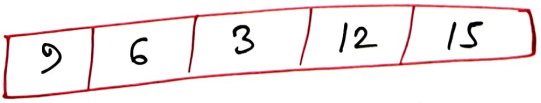


Step 1: Push elements of Queue to Stack upto K.

Step 2 :- Push elements of stack into Queue again upto k.

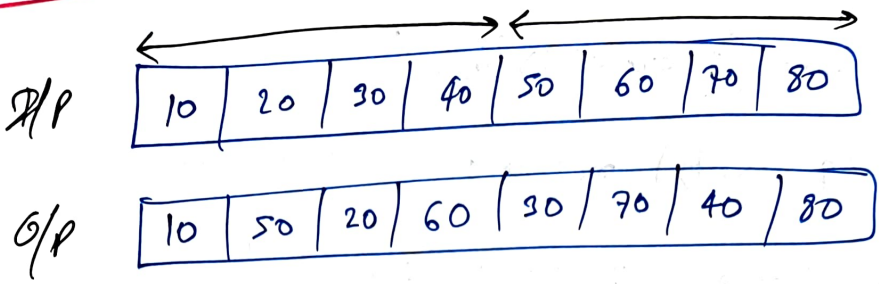


Step 3 :- Pop n-k elements from queue and push to the back

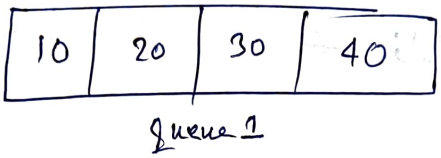


Reversed K number of elements of Queue

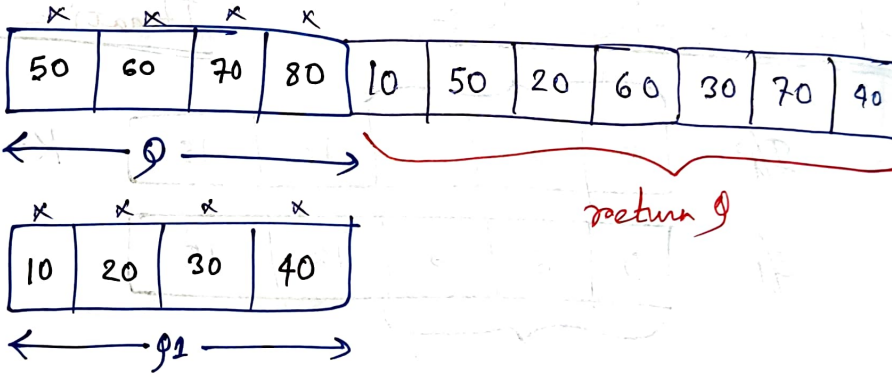
Interleave Queue :



Step 1: Take first half of Q into a new Queue



Step 2:



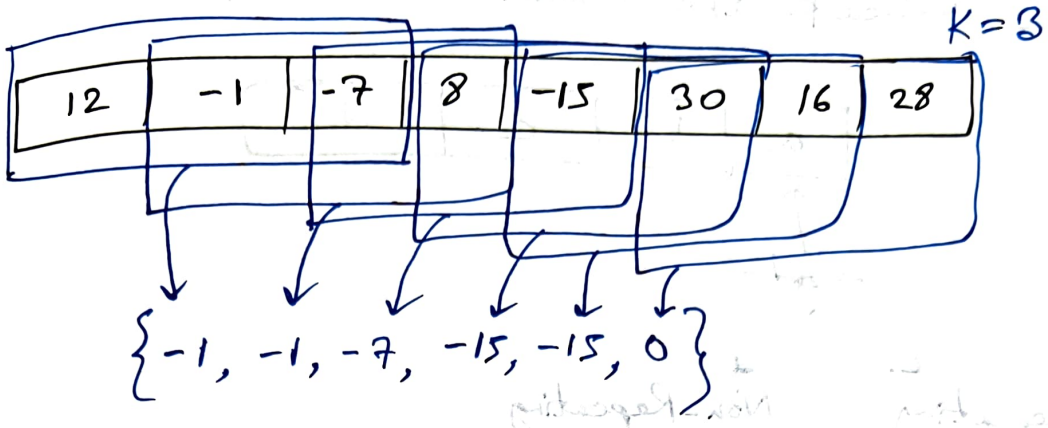
If the n is odd, Step 3:

```
if (n & 1){
    int element = q.front();
    q.pop();
    q.push(element);
}
```

Fastest way to check if the n is odd

#Sliding Window

Q1) First -ve Integer in every window size of K



Step 1: Process first K size [-ve no → index push to Queue]

Step 2:
slide the window

(i) ans = arr[q.front()]
or
ans = 0

(ii) remove out of window elements

(iii) New element insertion

Loop (K+1 → n)

#First Non Repeating Character :

i/p →	a	a	b	c	i/p →	a	a	c
o/p →	a	#	b	b	o/p →	a	#	c

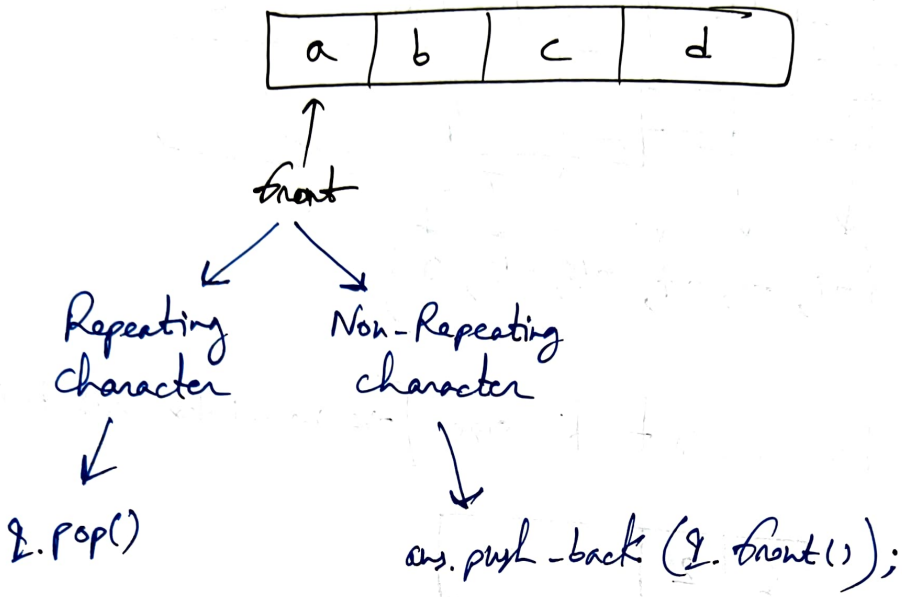
Step 1: Keep a track of characters that how many times it repeated in the stream using array [frequency]

```
int freq[26] = {0};  freequency++;
```

Step 2: Push the character inside Queue.

```
q.push(ch);
```

Step 3: Check the front of the Queue.



Step 4: If `q.empty()` {

```
    ans.push_back('#');
}
```

#Gas Station :- {Leetcode :- 134}

gas → | 1 | 2 | 3 | 4 | 5 |

cost → | 3 | 4 | 5 | 1 | 2 |

Step 1: Find out the starting point where the whole circle movement is possible.

(petrol - distance >= 0)

petrol = gas[i]
distance = cost[i]

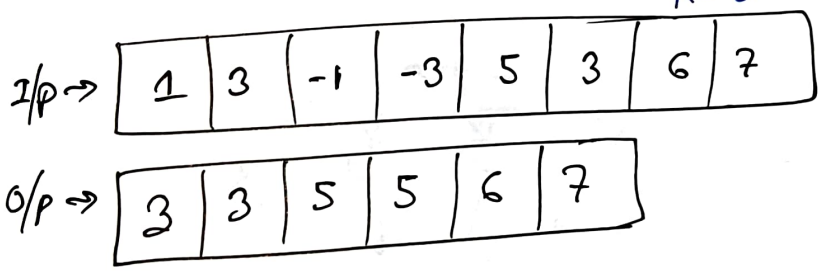
Step 1: Calculate Deficit and Balance

Step 2:
```
if (balance < 0){
    deficit += balance;
    start = i+1;
    balance = 0;
}
```

Step 3:-
```
If (deficit + balance >= 0){
    return start;
} else { return -1; }
```

Sliding Window Maximum :-

K = 3



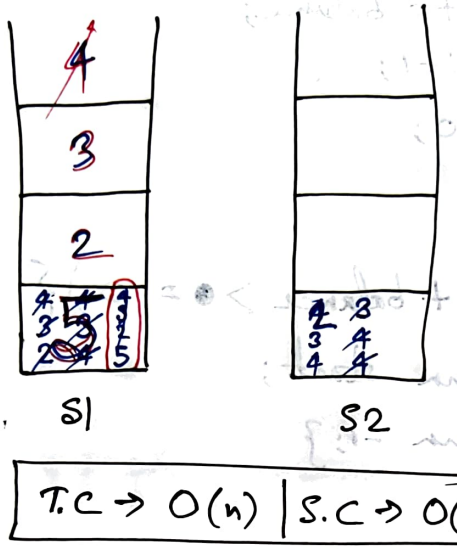
Step 1: Return the maximum number inside the window, push the index of sabse bada element

Step 2: Store answer for the first window, in ans []

Step 3: Remaining window process karo, return ans

Implement Queue using Stack :-

push(4)
push(3)
push(2)
push(5)
top() → 4
pop() →
top() → 3



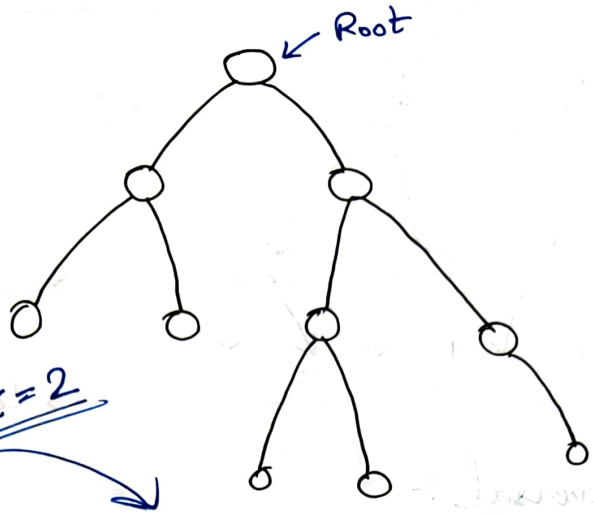
T.C → O(n) | S.C → O(2N)

Rules:

Push(x)
- → S1 ↷ S2
- → x → S1
- → S2 ↷ S1

Pop(x)
- → S1.pop()

⇒ Trees :-

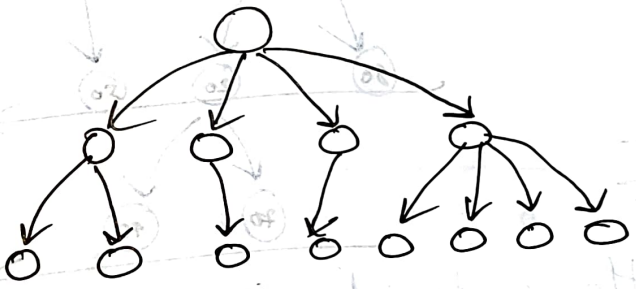


[Non-Linear DS]

no. of childs <= 2

[Binary Tree Exp]

N-ary tree ⇒ no. of childs can be (n) :-



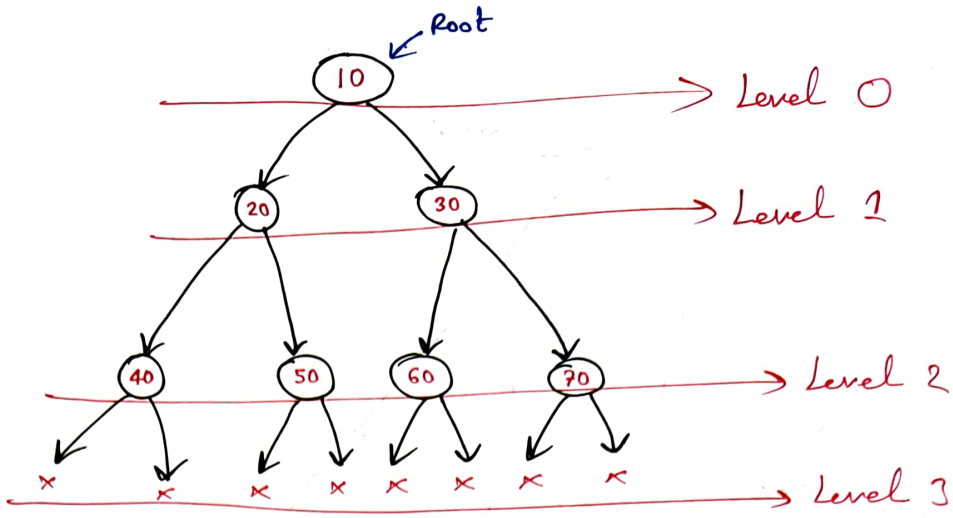
⇒ Code / Structure of Tree :-

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

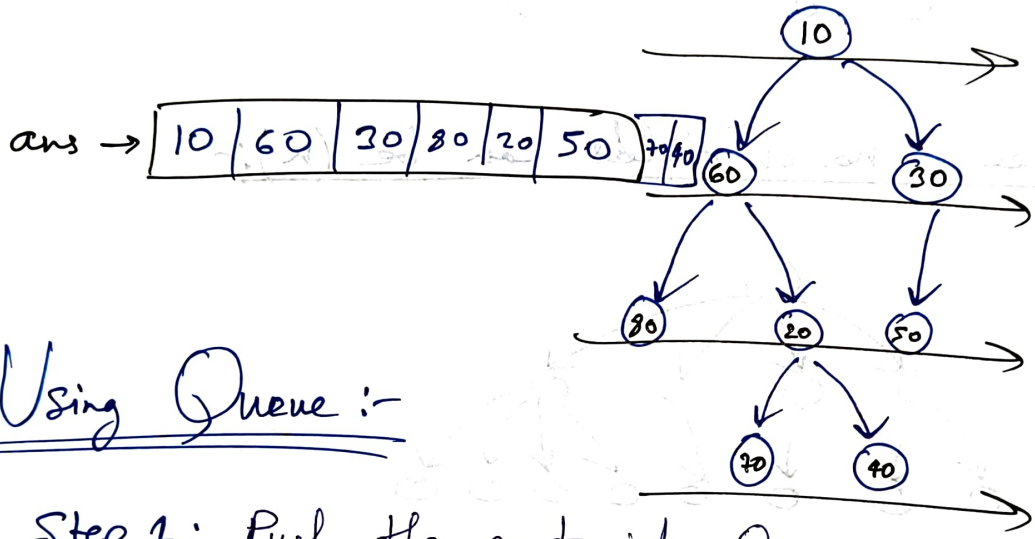
→ Binary Tree

```
class Node {
    int data;
    vector<node*> childs;
}
```

← N-ary Tree



Level Order Traversal :-



Using Queue :-

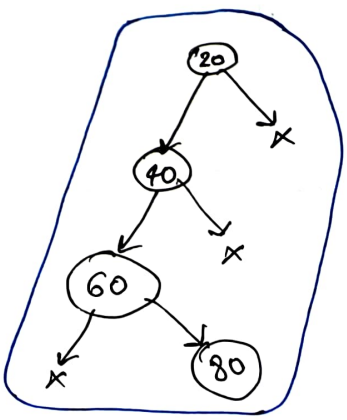
Step 1: Push the root into Queue.

Step 2: Storee q.front() in Node* temp and pop it

Step 3: Print temp → data and left and right also if any.

Example:-

I/p :- 20 40 60 -1 80 -1 -1 -1 -1

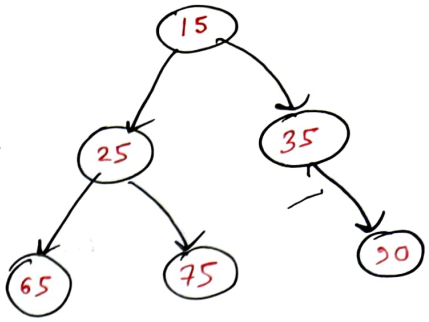


O/p :- 20 40 60 80

Level Order Traversal in next Line :-

I/P: 15 25 65 -1 -1 75 -1 -1 35 -1 90 -1 -1

O/P: 15
25 35
65 75 90



Inorder / Preorder / Post order :-

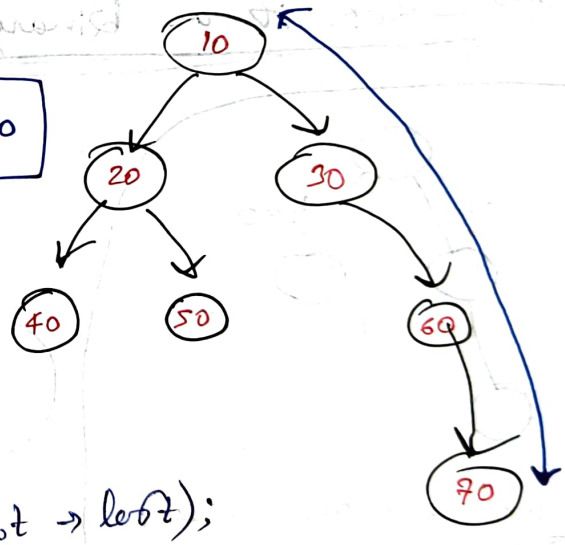
- LNR
- NLR
- LRN

LNR : 65 25 75 15 35 90
NLR : 15 25 65 75 35 90

LNR
- Left
- current
- Right

⇒ Height / Max Depth of Tree :-

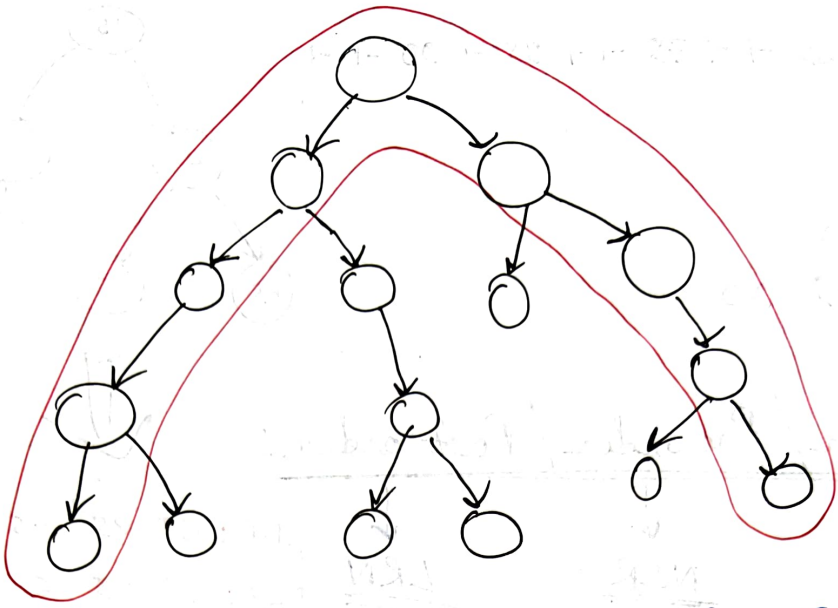
Height = 4 → 10 → 30 → 60 → 70




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

Leet Code : 104

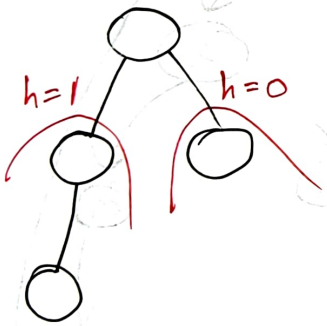
⇒ Diameter of Tree :



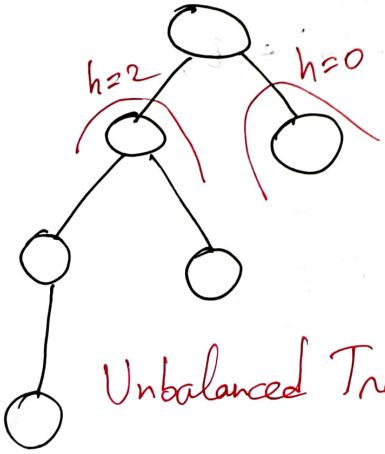
Diameter of the Tree = 8 [no. of Edges]

ans = max (left, right, height)

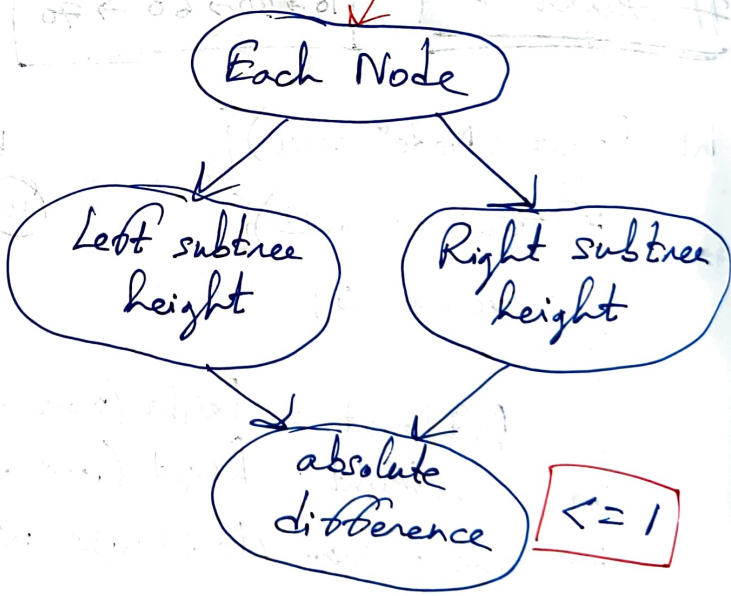
⇒ Check if a Binary Tree is Balanced or not :-



absolute diff <= 1

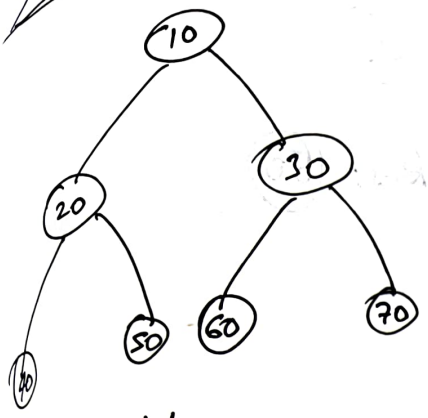


Unbalanced Tree

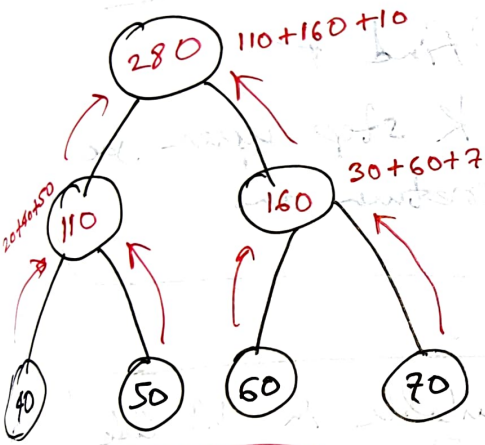


LeetCode : 110

⇒ Convert Into SumTree :




```
convertIntoSumTree (root) {
    if (root == NULL)
        return 0;
    int left = convertIntoSum (root->left);
    int right = convertIntoSum (root->right);
    root->data = left + root->data + right;
    return root->data;
}
```

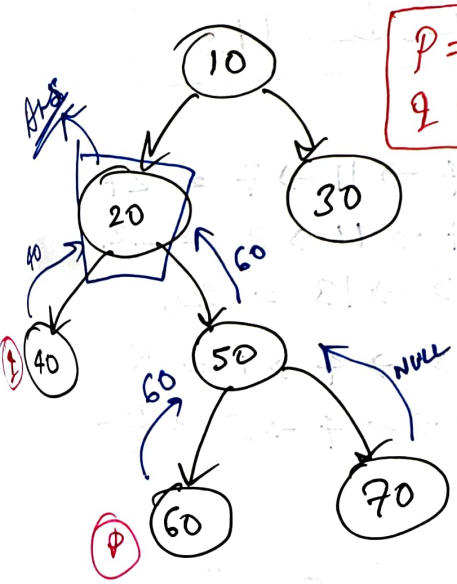


SumTree

GFG Pe Hain!

⇒ Lovest Common Ancestors :-

IMPORTANT



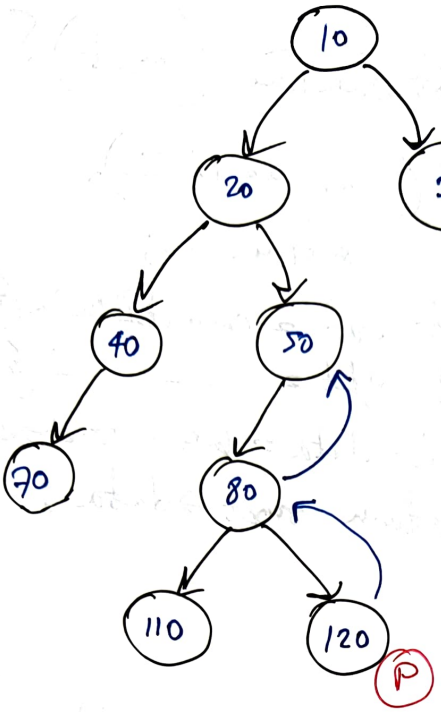
p = 60
q = 40

⇒ return lovest Common Ancestor

Answer: 20

LeetCode: 236

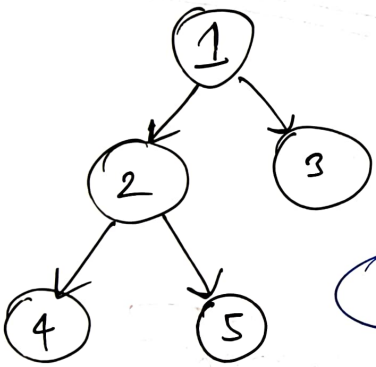
K^{th} Ancestor :-



K = 2

Ans: 50

① find P
② K step upar jao
③ return ans



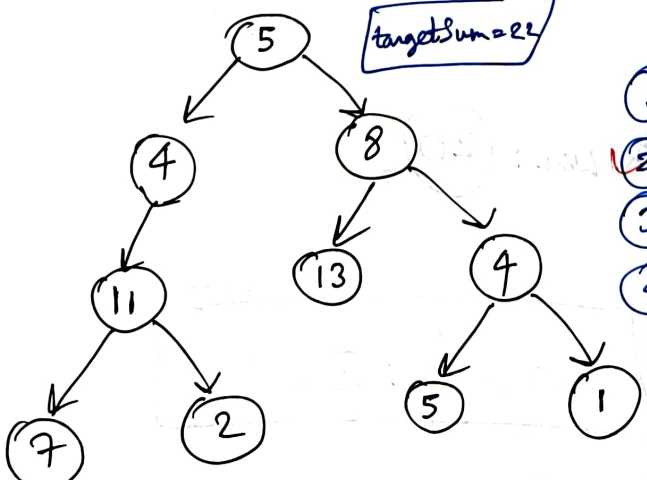
K = 1
P = 4

ans = 2

Available in VS Code
[CODE]

Path Sum II :

Leet Code : 113



targetSum = 22

① 5 → 4 → 11 → 7 = 27
② 5 → 4 → 11 → 2 = 22
③ 5 → 8 → 13 = 26
④ 5 → 8 → 4 → 1 = 18
⑤ 5 → 8 → 4 → 5 = 22

ans: 5 → 8 → 4 → 5
5 → 4 → 11 → 2

[vector.push()]

*** Create a tree using Inorder & Preorder Traversal =>

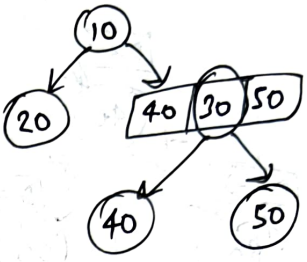
i/p:

L N R

inorder → 20 10 40 30 50

preOrder → 10 20 30 40 50

Root

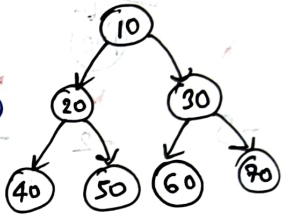


i/p:

inorder → 40 20 50 10 60 30 70

preoreder → 10 20 40 50 30 60 70

Root



Step 1: Find Root from Preorder, then find L, N, and R in inorder to get the Tree

Step 2: Recursion kar lega

*** Create a tree using Inorder & Post Order Traversal:-

L N R

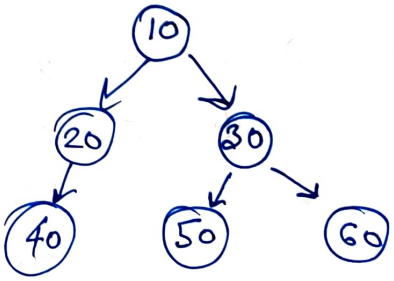
i/p: inorder → 40 20 10 50 30 60 LNR

postorder → 40 20 50 60 30 10 LRN

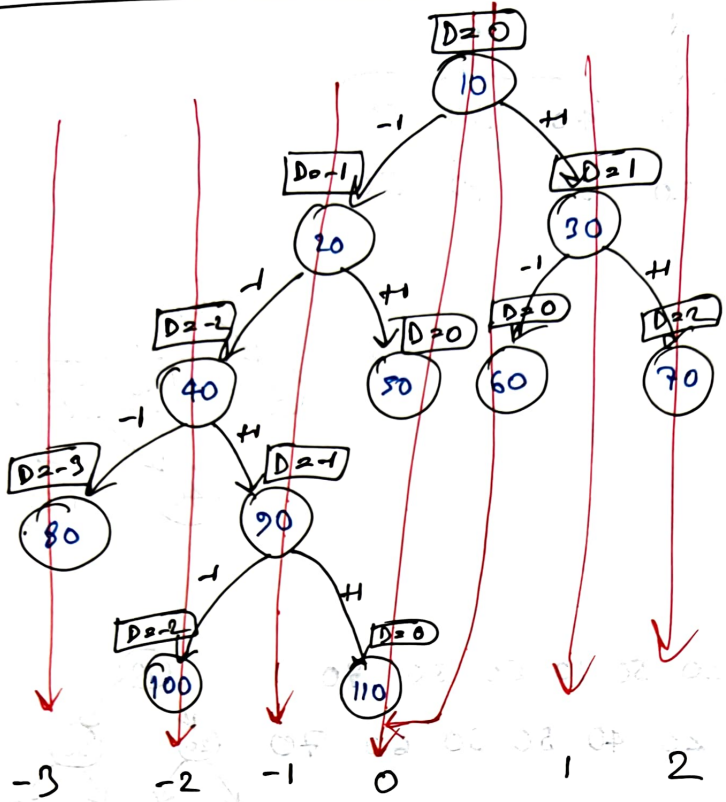
Root

Step 1: Find Root from post Order, the search it on inorder, get the left and right nodes

Step 2: Recursion kar lega

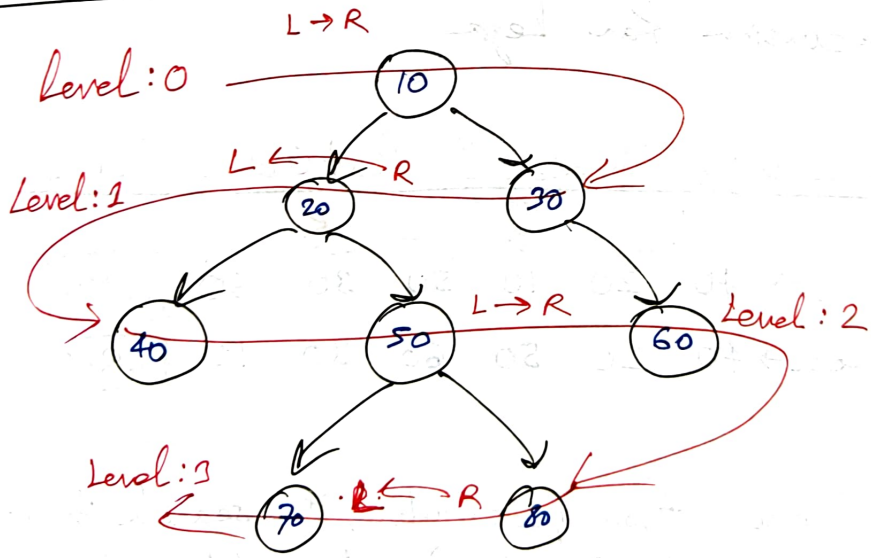


Vertical Order Traversal :-



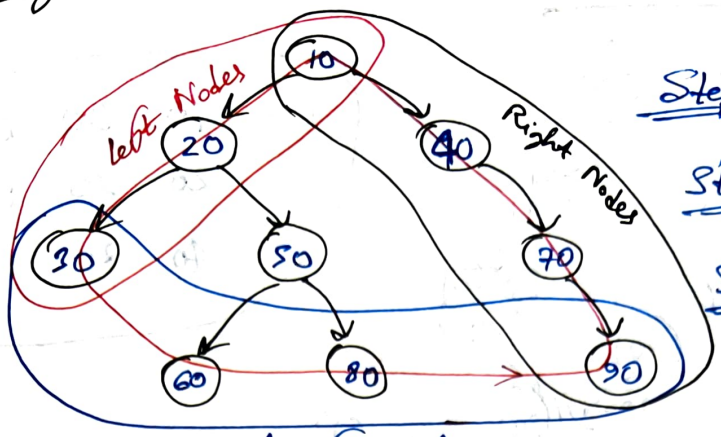
ans → 80
40 100
20 90
10 50 60 110
30
70

Zig-Zag Traversal :-



ans → 10 30 20
40 50 60
80
70

Boundary Traversal :-

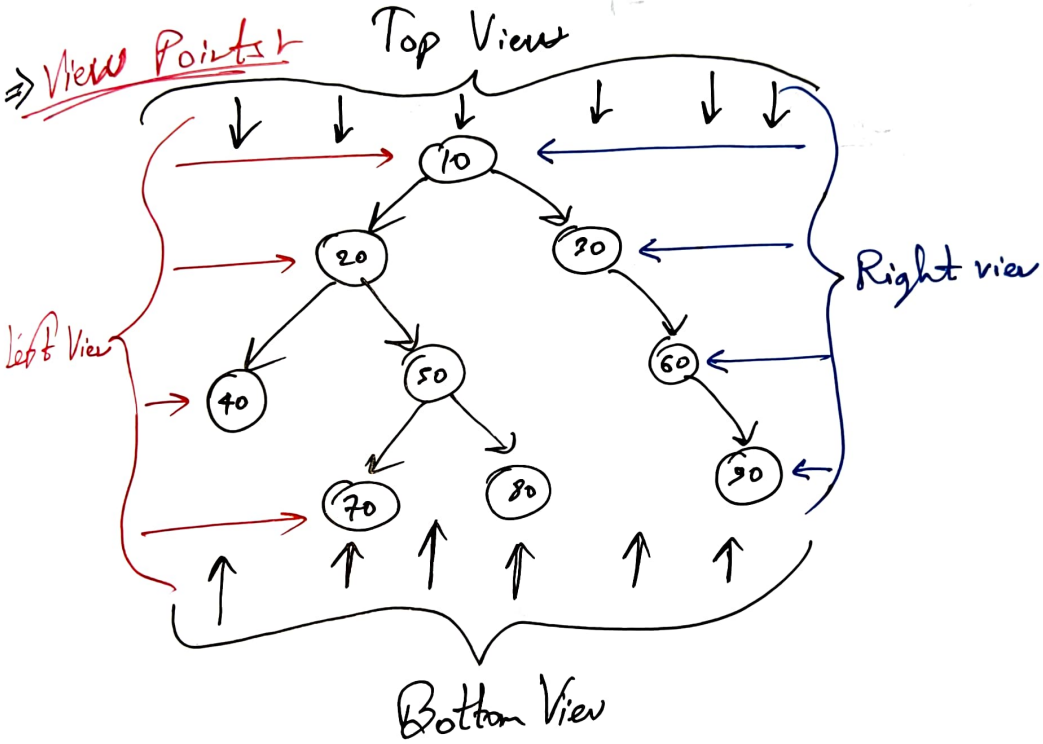


Step 1 : Print Left view

Step 2 : Print Bottom views

Step 3 : Print Right view

⇒ View Points :-



1) Top View : | 40 | 20 | 10 | 30 | 60 | 70 |

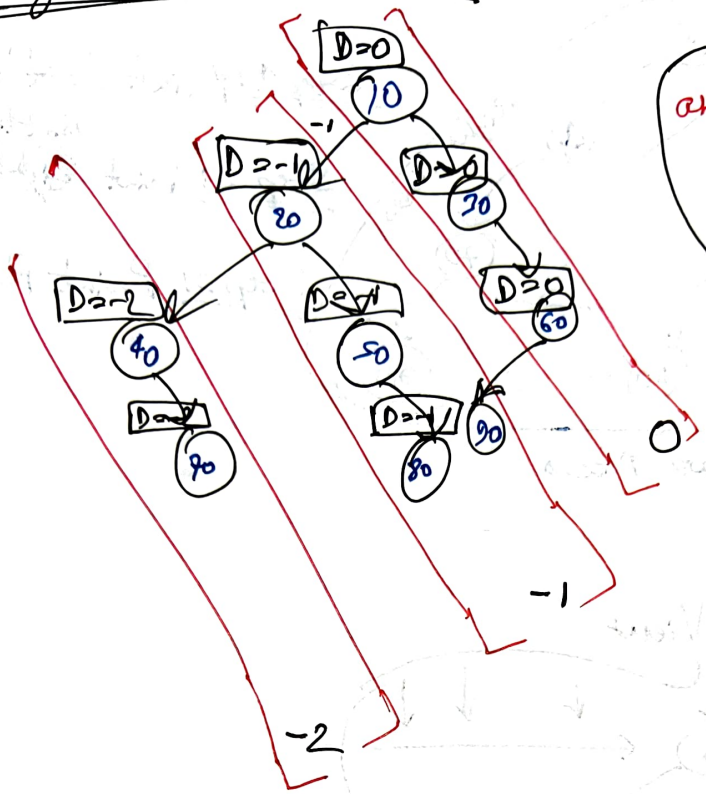
2) Left View : | 10 | 20 | 40 | 70 |

3) Right View : | 10 | 30 | 60 | 70 |

4) Bottom view : | 40 | 70 | 50 | 80 | 60 | 90 |

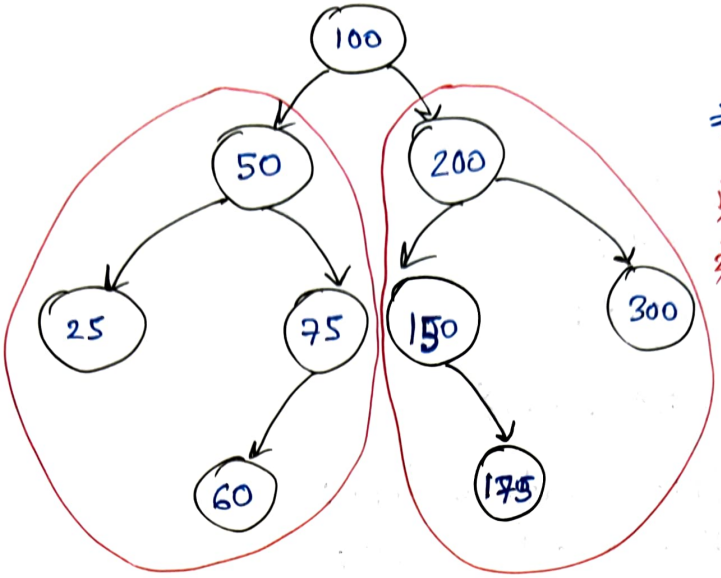
⇒ Code is available on VS Code → Tree → 4_..cpp

Diagonal Traversal :-



Ans → 10 30 60
20 50 80 90
40 70

Binary Search Tree :-



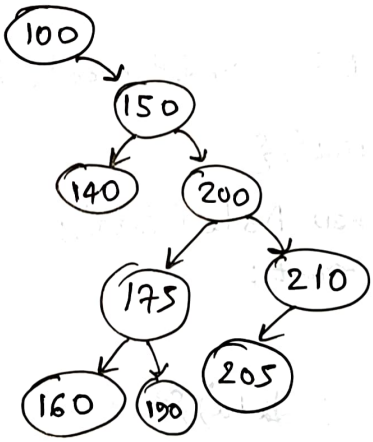
BST

⇒ For each node

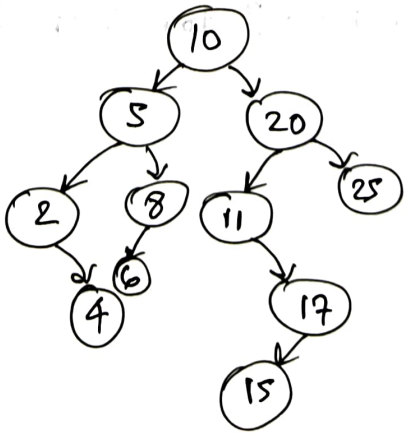
1) root → data > root → left → data

2) root → data < root → right → data

i/p : 100 150 200 175 160 140 210 205 190



i/p : 10 20 5 11 17 2 4 8 6 25 15

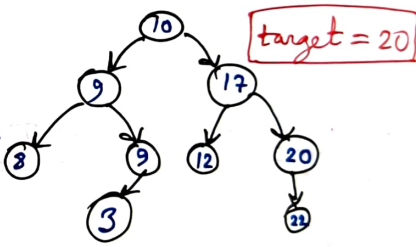


⇒Create BST :

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

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

Node* insertIntoBST(Node* root, int data){
    if (root == NULL){
        root = new Node(data);
        return root;
    }
    if (root -> data > data){
        root -> left = insertIntoBST (root->left, data);
    }else{
        root -> right = insertIntoBST (root -> right, data);
    }
    return root;
}
```

```
void takeInput(Node* &root){
    int data;
    cin>>data;
    while(data != -1){
        root = insertIntoBST(root, data);
        cin>>data;
    }
}

int main(){
    Node* root = NULL;
    cout << "Enter the data for Node: " << endl;
    takeInput(root);
    cout<<"printing the tree" <<endl;
    levelOrderTraversal(root);
    return 0;
}
```

⇒ Searching :-

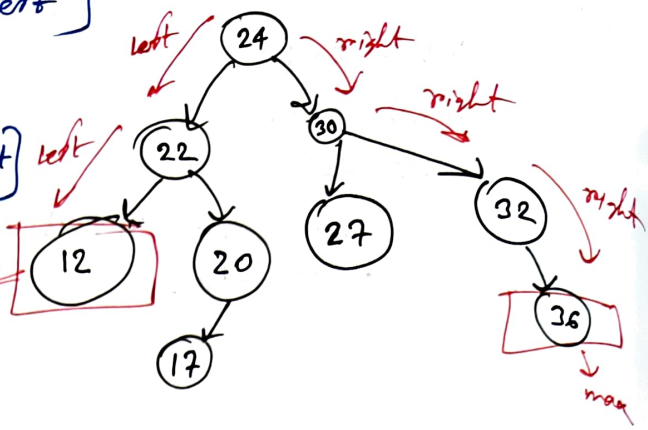



```
bool findInBST(Node* root, int target){
    if(root == NULL){
        return false;
    }
    if(root->data == target){
        return true;
    }
    if(root->data > target){
        return findInBST(root->left, target);
    }else{
        return findInBST(root->right, target);
    }
```

⇒ MinVal :- 12 [root → left]

⇒ Max Val :- 36 [root → right]

[Code available in VS Code]



⇒ Inorder Predecessor / Successor :-

inorder :-

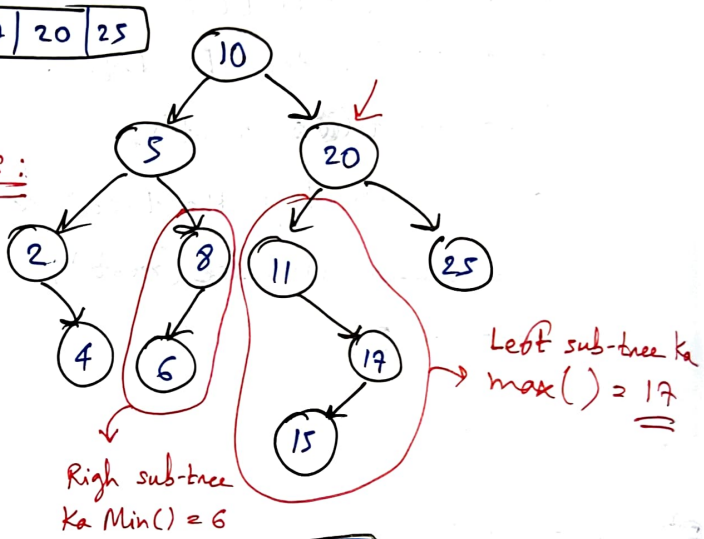
2	4	5	6	8	10	11	15	17	20	25

Predecessor of 20 is :

⇒ 17

Succesor of 5 is :

⇒ 6



⇒ Deletion in BST :-

LeetCode : 450

Steps : if(root → data == Target){

→ left && right == NULL

→ Left == NULL && right != NULL

→ Left != NULL && right == NULL

→ Left && right != NULL

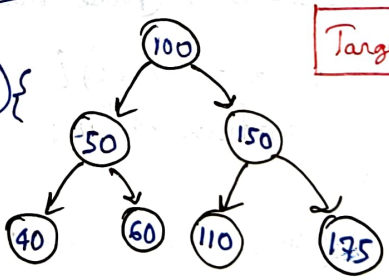
}

else if(target > root → data){

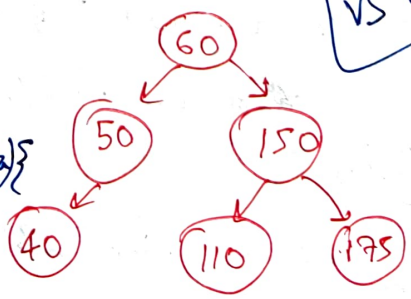
}

else {

}



VS Code



Validate BST :-

Leetcode : 98

Step 1: Check if the root node is in range $-\infty$ to ∞

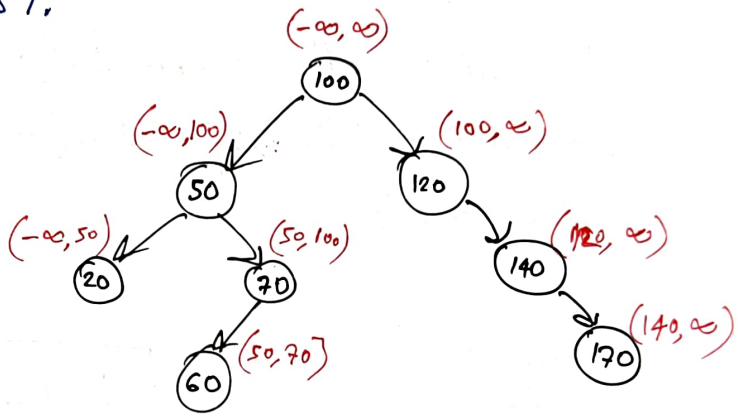
Step 2: Check right & left nodes if their range is correct

Step 3: If both is correct, return true, else false

OR

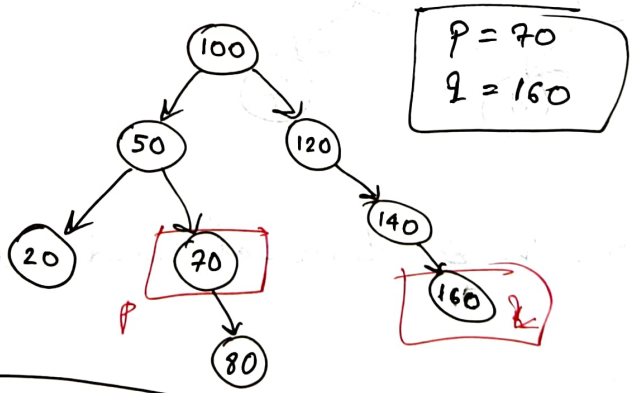
Step 1: Check the inorder

Step 2: If the inorder traversal values are sorted then it is a valid BST.



Approach: [Range]

LCA of BST :-



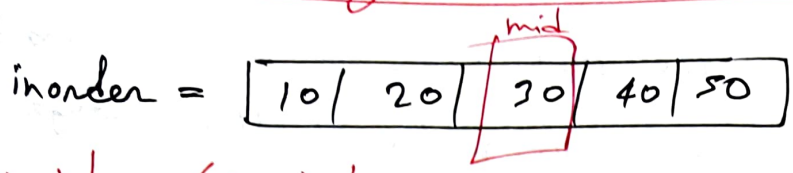
p = 70
q = 160

Kth smallest value :-

Leetcode: 230

Leet Code : 235

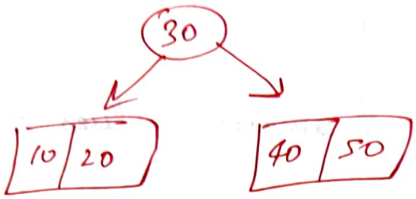
Create BST using inorder traversal :-



mid = (s+e)/2

int element = inorder[element]

Node* root = new Node(element);



root → left = BSTusingInorder(inorder, s, mid-1);

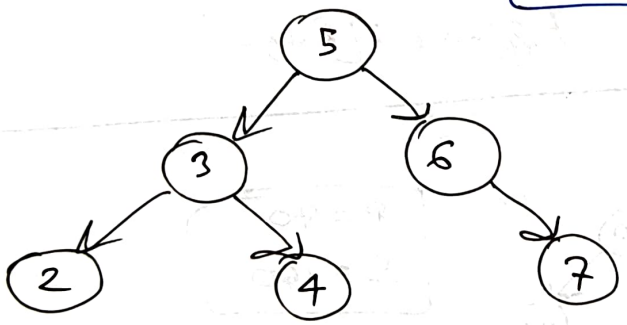
root → right = BSTusingInorder(inorder, mid+1, e);

return root;

Two Sum :-

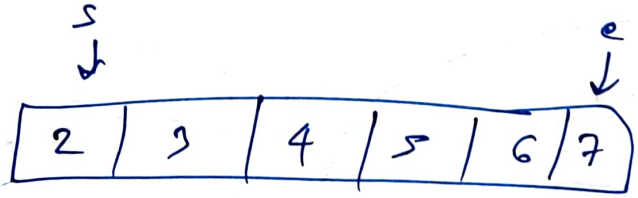
Leetcode : 653

K = 9



Step 1: Store inorder traversal and search the K

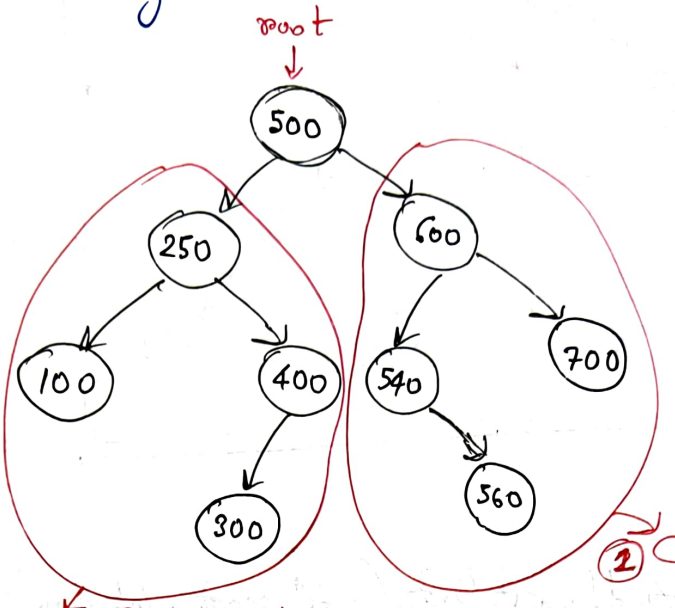
Step 2



if(s+e = K) return true; else false;

BST → Sorted DLL :-

Convert Binary Search Tree to Sorted Doubly-LL



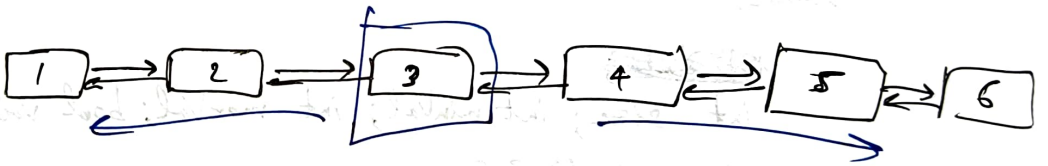
② Create LL of this sub tree using Recursion

② Create LL of this sub-tree Using Recursion

③ Left LL → root → Right LL

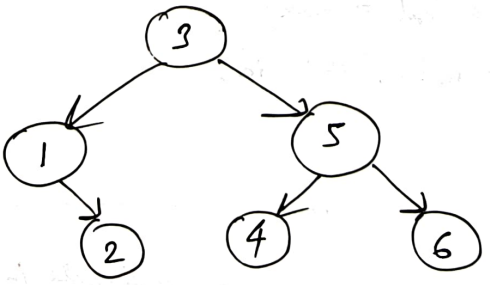
④ return ans;

Sorted LL → BST :-



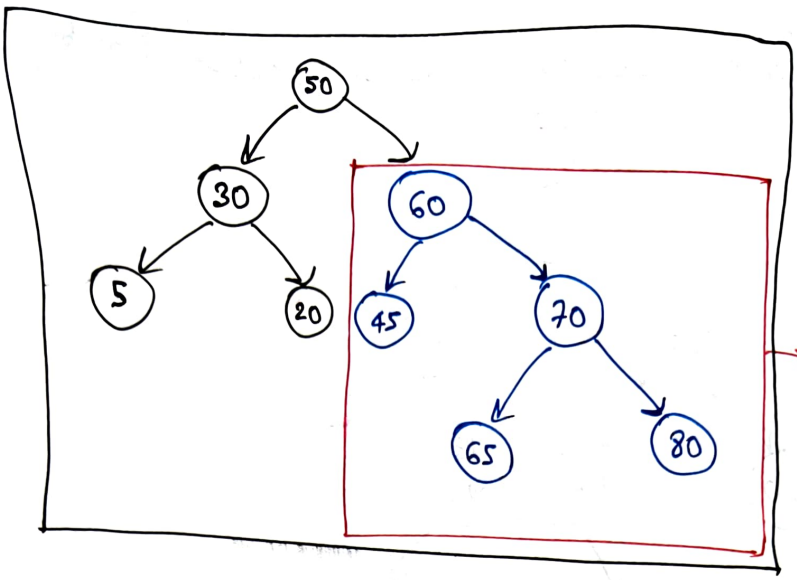
Step 1: find Mid and create root Node

Step 2: Recursively make tree → left, → right



BST

Largest BST in a Binary Tree :-



→ Largest BST
⇓
return
no. of node

Ans: 5

⇒ NodeData findLargestData(Node* root, int &ans){

——— Logic here [VS code]

}

⇒ class NodeData {

public:

int size; int minVal; int maxVal; bool validBST;

NodeData() { }

NodeData(int size, int max, int min, bool valid){

this → size = size;

minVal = min;

maxVal = max;

validBST = valid;

}

};

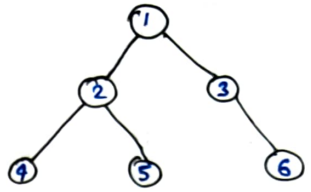
Code available in VS Code

Heap

Heap → CBT + that holds a heap property (max / min)

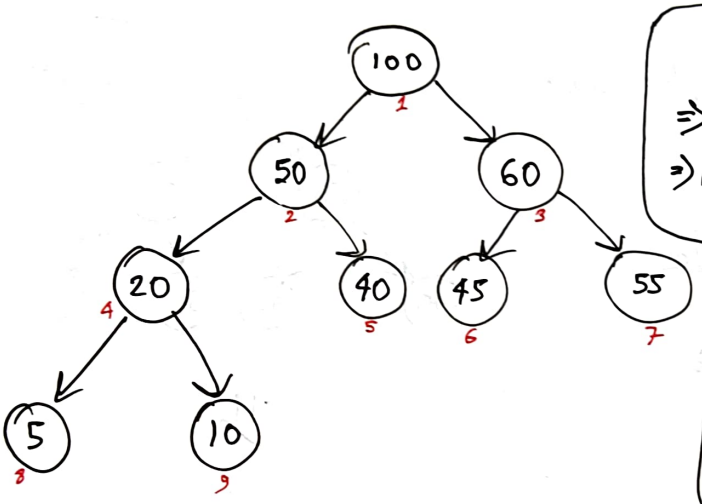
Complete Binary Tree

- All levels are completely filled except last one
- → filling → (L → R)



Example of CBT

→ Heap (using Arrays):-



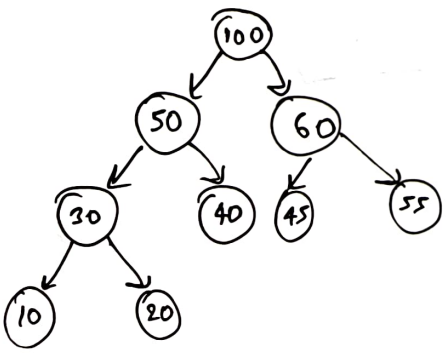
if, parent = i
⇒ Left child = 2i
⇒ Right child = 2i + 1

indexing starts with 1

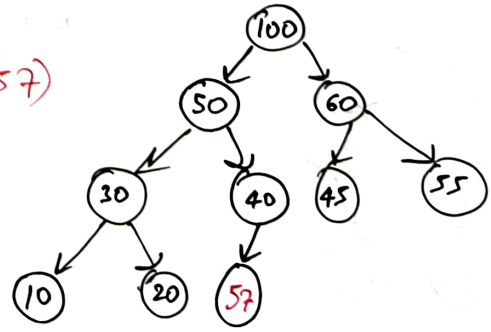
if, child = i
∴ parent = i/2

-1	100	50	60	20	40	45	55	5	10
0	1	2	3	4	5	6	7	8	9

⇒ Insertion in a Heap:-

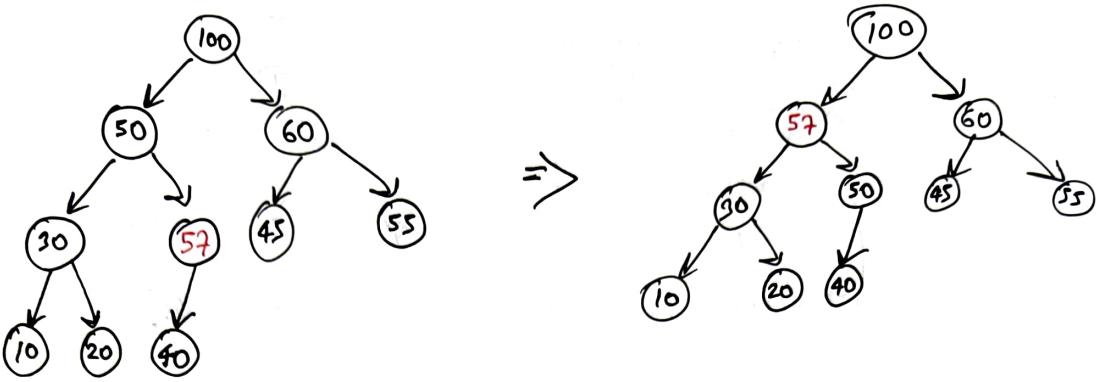


insert (57) ⇒



Step 1: Add node at the end of the array

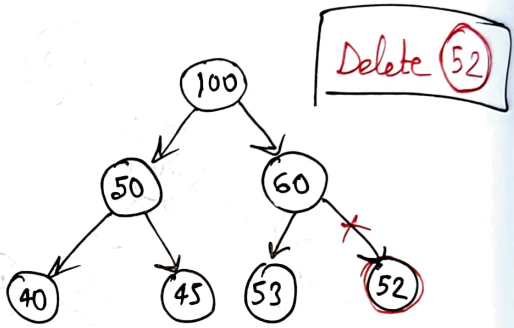
Step 2: Compare i with i/2 position, if i/2 is smaller than child (i), then swap

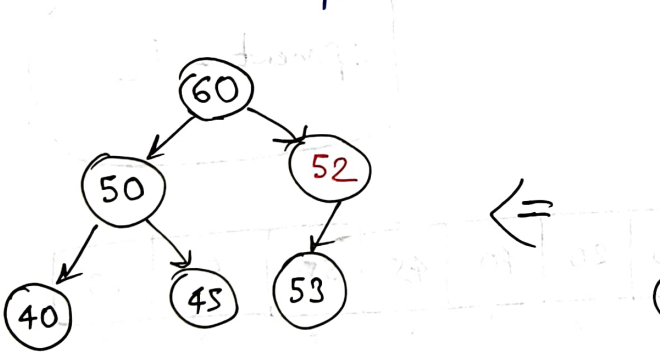


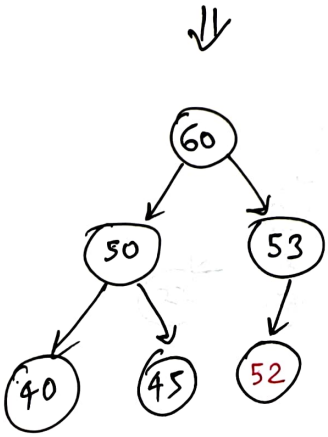
⇒ Deletion :-

Step 1: Replace last value with root node

Step 2: Root node → correct position



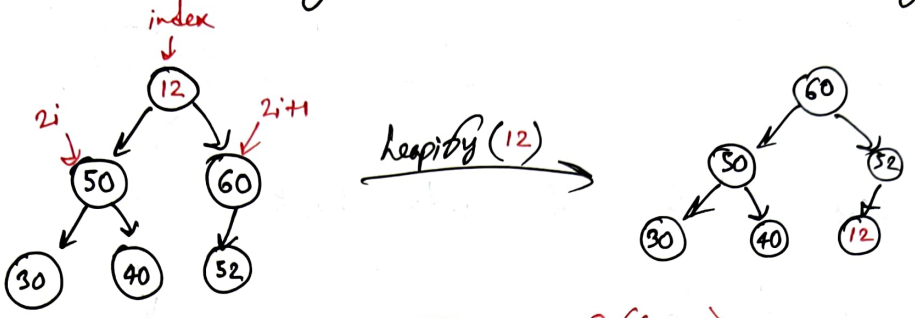




Code in VS Code

→ Heapify :-

⇒ Create array and convert it to Heap using Heapify



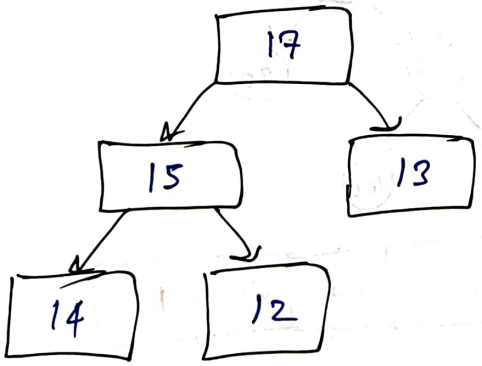
T.C = O(log n)

Step 1: Compare the index with the left and right child, if true then swap each other

Step 2: place the index and largest pointer

Step 3: If the condition false, returns

→ Build a Heap from Arroay :-

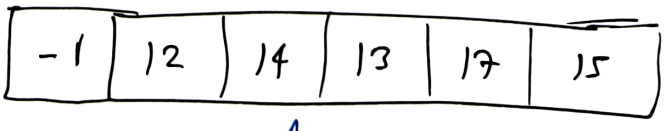


Valid Heap

Build Heap ()

[Code available in VS code]

T.C = O(n)



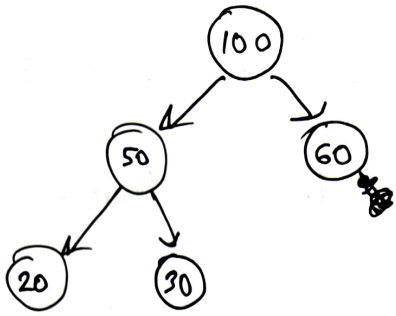
Arroay

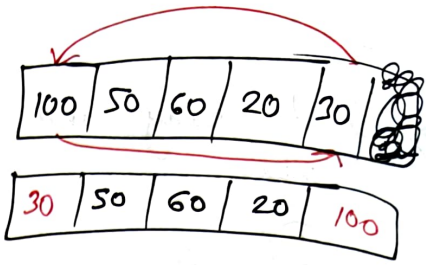
→ No need to heapify Leaf nodes

→ $(\frac{n}{2}+1)$ to n nodes are leaf nodes in CBT

→ Heapify $\frac{n}{2}$ to >0 nodes

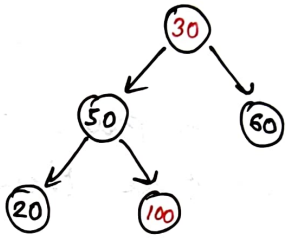
→ Heap Sort :-

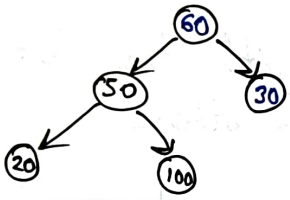


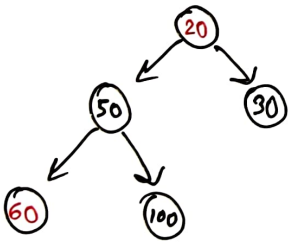


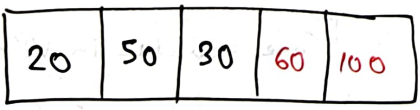
Step 1: Swap first node with last

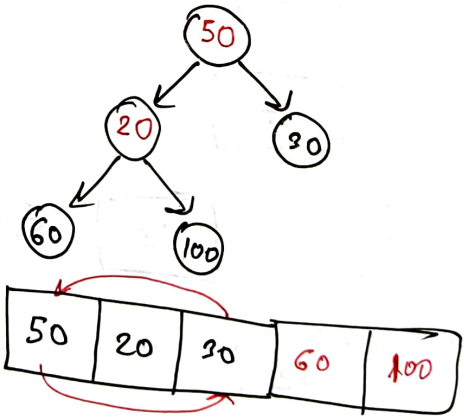
Step 2: Heapify Karlo

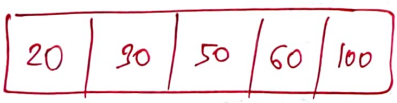


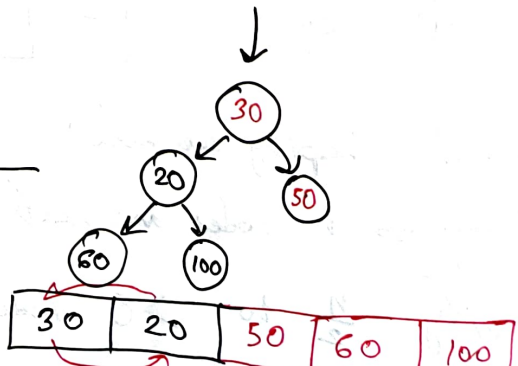












Priority Queue :-

MaxHeap / MinHeap using STL Library

```
#include <queue>
int main(){
```

max

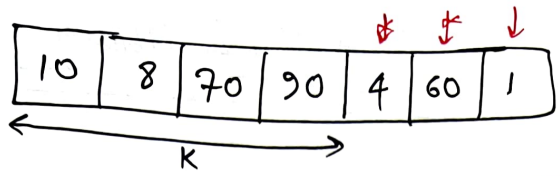
```
    priority_queue <int> pq;   ]→ It is a maxHeap creation
    pq.push(3);
    pq.push(6);
    pq.push(9);
    pq.top(); → 9
    pq.pop();
    pq.top(); → 6
    pq.size(); → 2
    pq.empty(); → false
```

min

```
    // Min Heap Syntax
    priority_queue <int, vector<int>, greater<int>> pq;
    pq.push(4);
    pq.push(8);
    pq.push(3);
    pq.top(); → 3
    pq.pop();
    pq.top(); → 4
    pq.size(); → 2
    pq.pop();
    pq.empty(); → false
    pq.pop();
    pq.empty(); → True
}
```

Code available in VS code
16.Heaps → 5_priority_queue.cpp

K^{th} Smallest Element :-

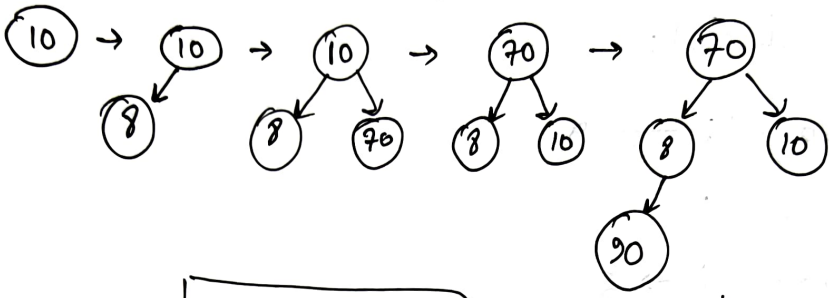


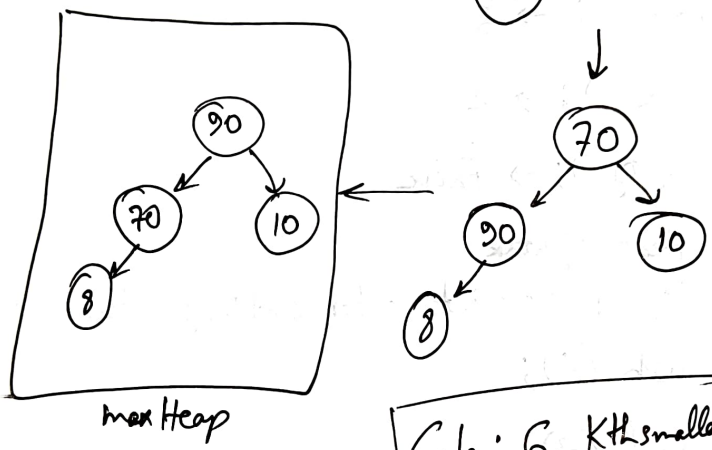
K = 4

Using MaxHeap :-

T.C → O(n) S.C → O(k)

① Create a maxHeap till K elements and Heapify then

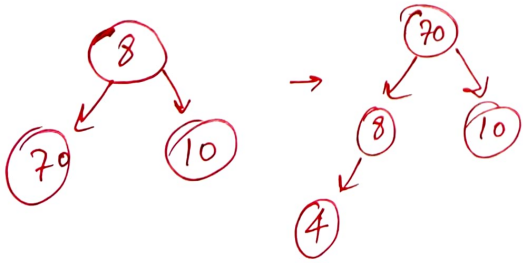




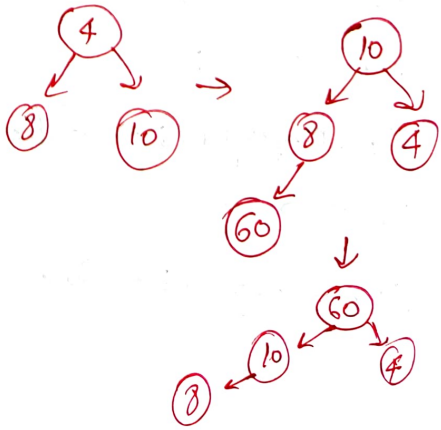
Code : 6_Kthsmallest.cpp

② Compare the K+1, K+2, K+n elements with the top element. If topelement > K+n element then remove the top element and add the K+n element.

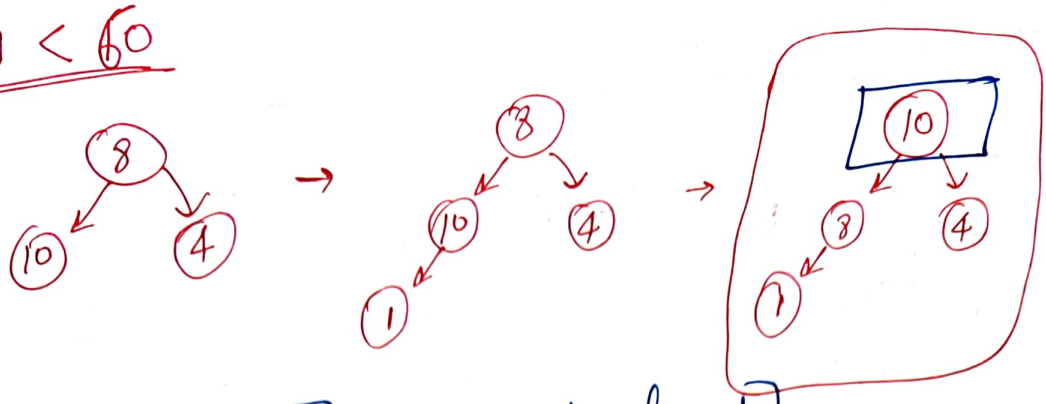
4 < 90



60 < 70

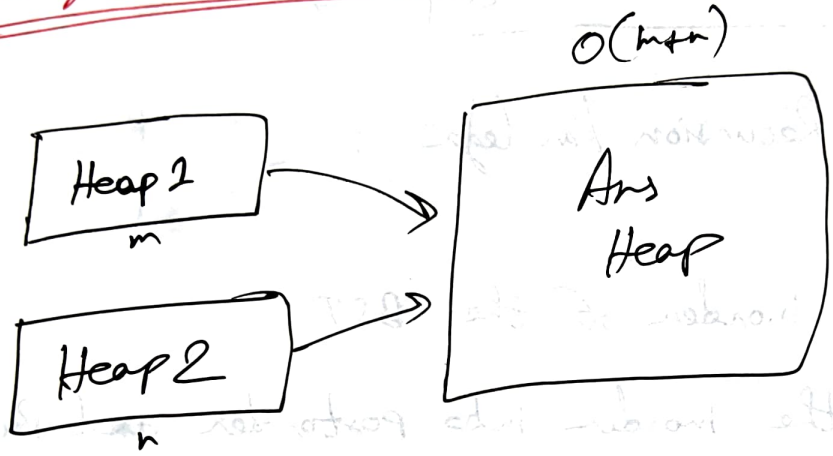


1 < 60

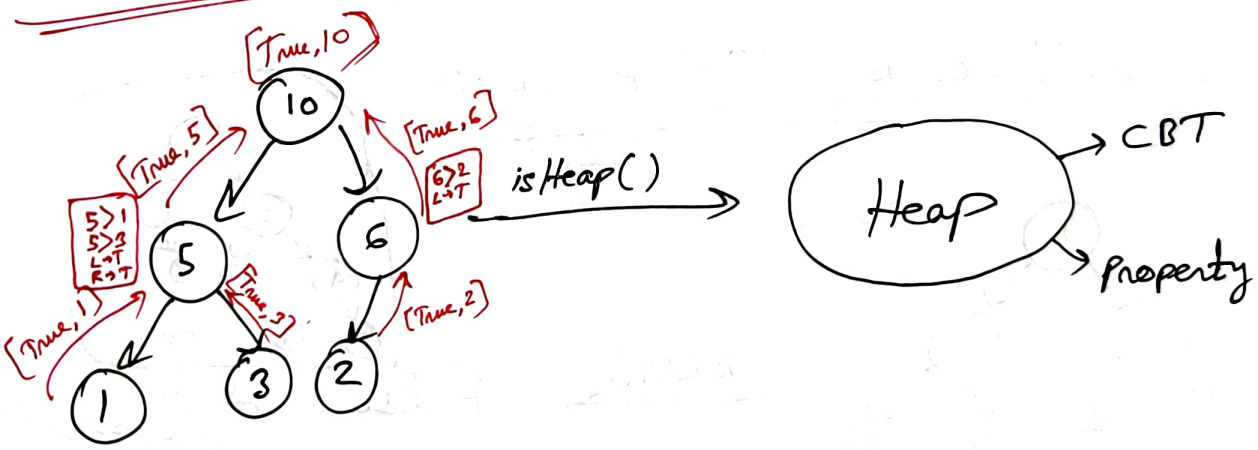


Ans is 10 [4th smallest element]

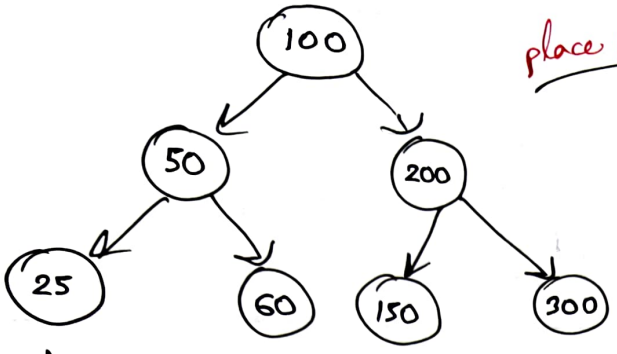
#Merge 2-MaxHeap :



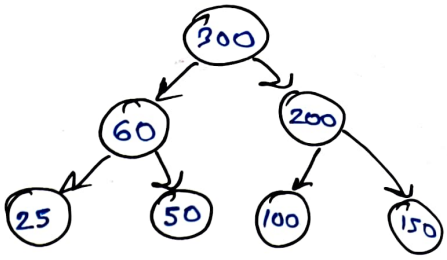
#CBT is a Heap or not :



Convert BST to Heap :-



place Postorder →



Heap (Max)

inorder L R N

25	50	60	100	150	200	300

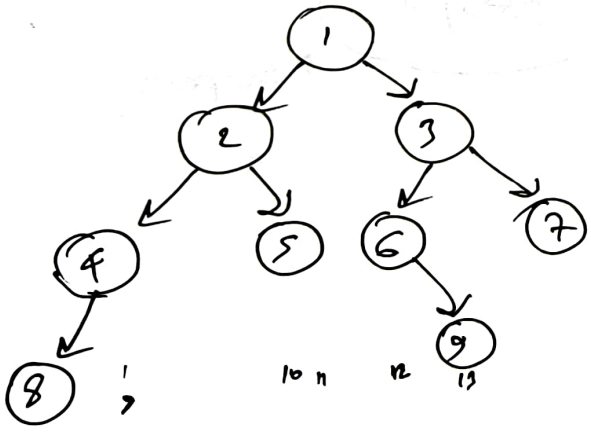
LRN
Postorder + Recursion Kar lega

Step 1: Get the inorder of the BST

Step 2: Convert the inorder into postorder and Build

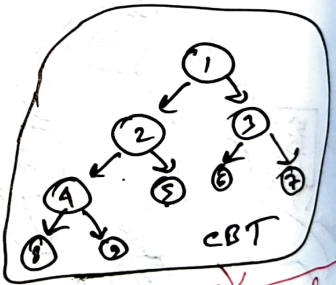
~~# BST Code #~~.

Check Tree is CBT or not :-



Tree

is CBT() →



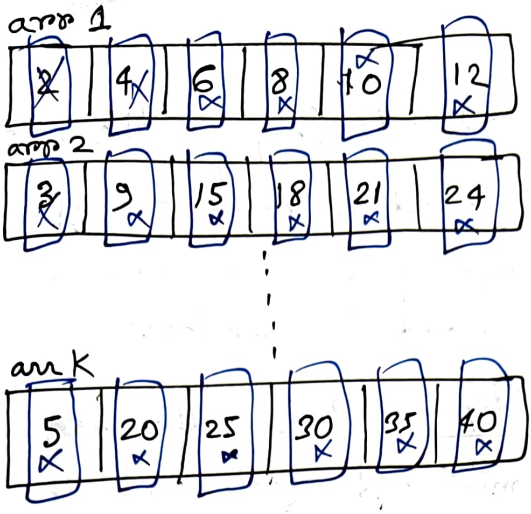
False

Total == last return True

Total Nodes = 9
Last Node = 13
Not a CBT

```
if (Total Node == Last Node)
        return True;
else
        return False;
```

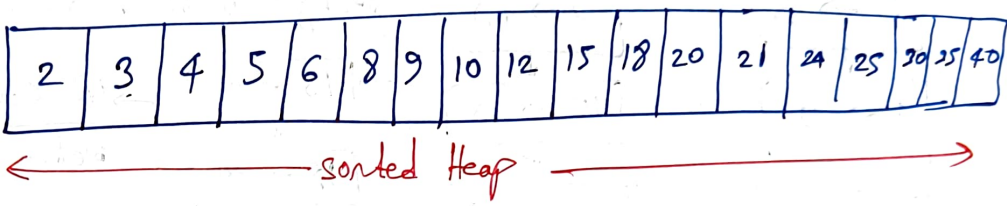
Merge K-Sorted Arrays :-



Step 1: Start from the first index of each array and push the smallest element one by one

Step 2: pointer ++

Heap ans :

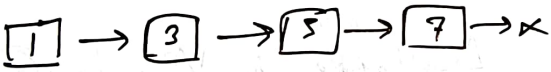


Merge K-sorted Linked List :-

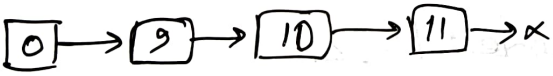
LeetCode : 23

Head 1

Head 2



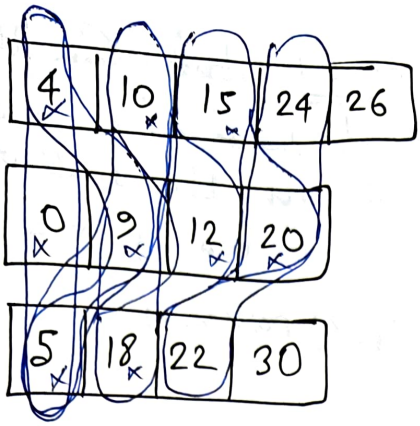
Head 3



Logic same as the previous one (Merge K-sorted arrays)

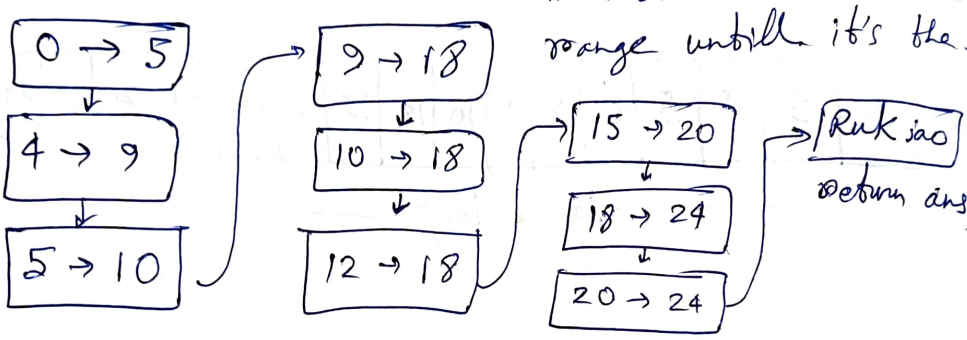
#Smallest range in K list:-

Leetcode: 632



Step 1: Compare the first element of each array. the range of mini → maxi should be smallest

Step 2:- Remove the ~~max~~ mini and recalculate the range untill it's the smallest.



Step 3:- Calculate minimum distance from all these ranges.

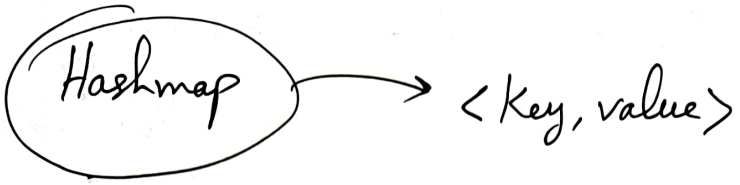
#Median of a stream:- Available in VS code → (8)

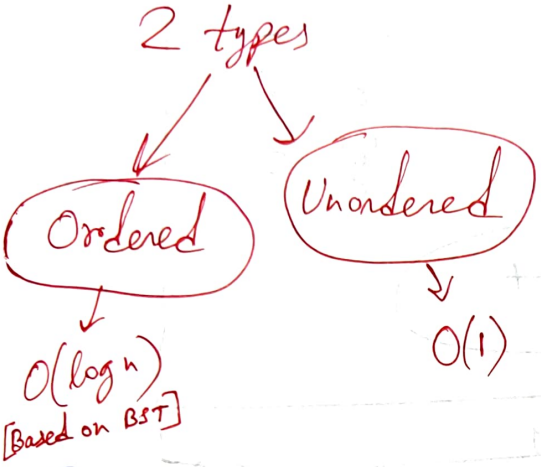
#Remove stones:- Leetcode: 1962

#Reorganize string:- Leetcode: 767

#Longest happy string:- Leetcode: 1405

Hashmaps & Tries :-





Scorpio → 9
Balleno → 3
Fortuner → 9.1
Thar → 10
RollesRoyes → 10

Imp: Create a data structure that completes insertion, deletion, searching & getRandom in O(1) time.

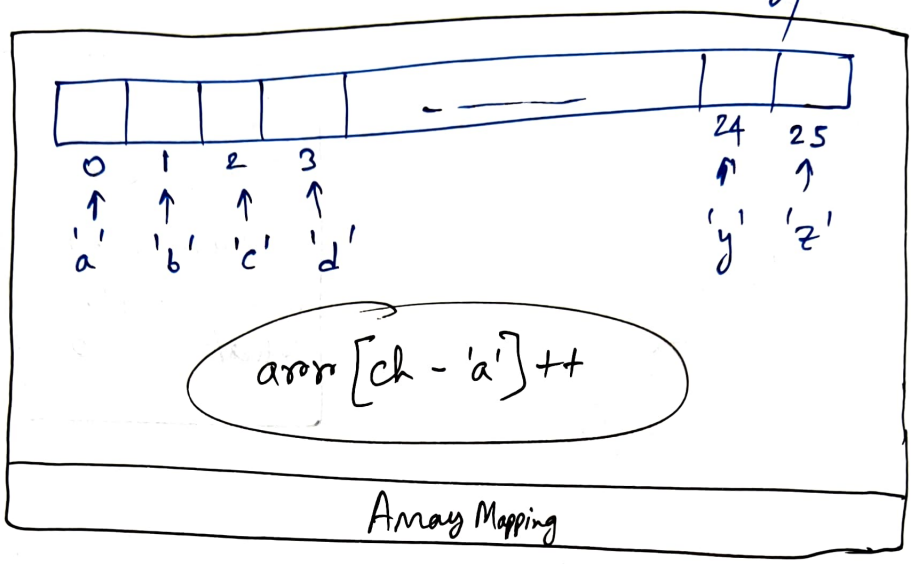
Ans: Unordered Map.

VS code @17.1

```
#include <unordered_map>
int main(){
    unordered_map<string, int> m; → Creation of Map

Insertion {
    pair<string, int> p("Scorpio", 1);
    pair<string, int> p = make_pair("Scorpio", 10);
    m["fortuner"] = 10;

access {
    cout << m["XUV"] << endl;
    cout << m["Scorpion"] << endl;
    cout << m.at("Suzuki") << endl;
```

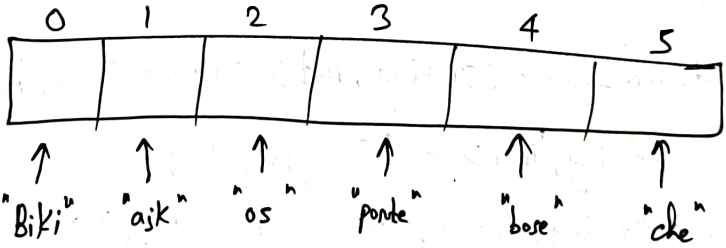
Ordered Map → O(logn) → BST based

Unordered Map → O(1) → Bucket Array/ Array/Hashtable



Bucket Array Mapping :-

Using Hash Function :-



Maps the string with the index → Hash( );

Hash code (conversion of key to num)

Compression function (num ko range pe laao...)

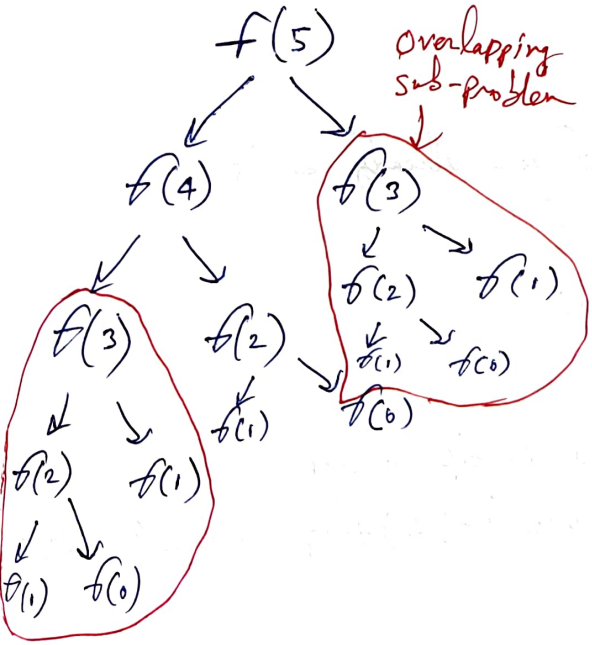
Dynamic Programming (DP) :- is a programming technique in which we don't calculate anything which already calculated.

- Top-down { Recursion + Memoisation } — If it is already calculated, never calculate again
- Bottom-up { Tabulation method → O(n)
- Space Optimization { → O(1) if possible

When to apply DP?

Overlapping Sub-problems + Optimal sub-structure

⇒ Fibonacci using recursion :-



0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55

Rule ⇒ $f(n) = f(n-1) + f(n-2)$

Base case: $f(0) = 0$, $f(1) = 1$

Steps of DP:-

↳ **Memoization:-**

1. Create DP array
2. Store ans in DP
3. Base case ke baad
   ↓
   check if answer is stored?

Recursive Sol^n for Fibonacci:-

T.C : Exponential

```
if (n == 1 || n == 0)
    return n;
int ans = solve(n-1) + solve(n-2);
return ans;
```

Top down sol^n for Fibonacci:-

```
int topDownSolve(int n, vector<int>&dp){
    if (n == 0 || n == 1){
        return n
    }
    //Step 3: check if ans already exists
    if (dp[n] != -1){
        return dp[n];
    }

    //Step 2: Store ans in dp array
    dp[n] = topDownSolve(n-1, dp) + topDownSolve(n-2,
    return dp[n];
}
```

```
int fib(int n){
    //Step 1: Create DP array
    vector<int> dp(n+1, -1);
    int ans = topDownSolve(n, dp);
    return ans;
}
```

BottomUp sol^n for Fibonacci :-

```
int fib(int n){
    //Step 1: Create an DP array
    vector<int> dp(n+1, -1);

    //Step 2: Base case
    dp[0] = 0;
    if(n == 0) return dp[0];
    dp[1] = 1;

    //Step 3: 2 to n [Bottom-up]
    for(int i=2; i<=n; i++){
        dp[i] = dp[i-1] + dp[i-2];
    }
    return dp[n];
}
```

$T.C = O(n)$

$S.C = O(n)$

By using Space Optimization solution, we can reduce the space complexity to $O(1)$

Space Optimization :-

```
int spaceOpt(int n){
    int prev2 = 0;
    int prev1 = 1;
    if(n==0) return prev2;
    if(n==1) return prev1;

    int curr;
    for(int i=2; i<=n; i++){
        curr = prev1 + prev2;

        prev2 = prev1;
        prev1 = curr;
    }
    return curr;
}
```

T.C = O(n)
S.C = O(1)