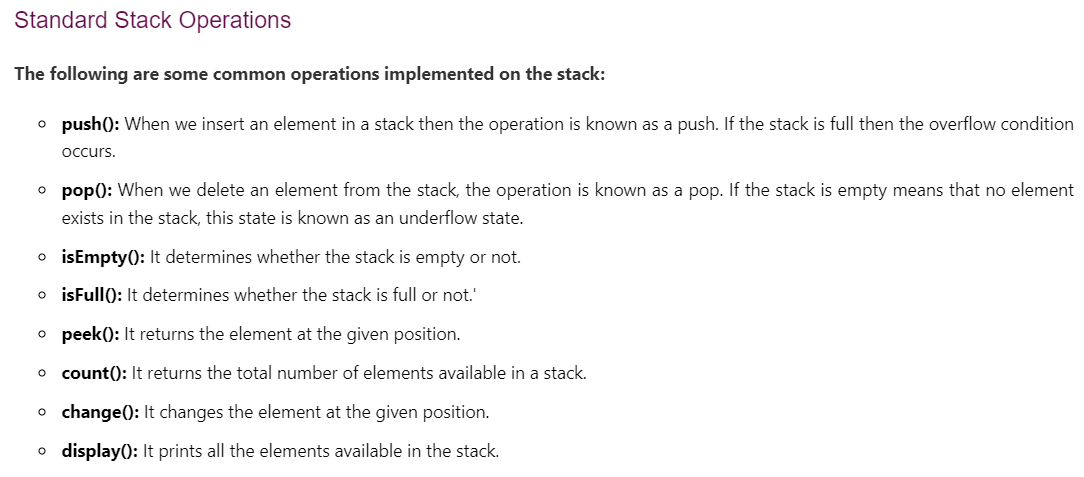
**Theory :**

A Stack is a linear data structure that follows the ****LIFO (Last-In-First-Out)**** principle. Stack has one end, whereas the Queue has two ends (****front and rear****). It contains only one pointer ****top pointer**** pointing to the topmost element of the stack. Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack. In other words, a **stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.**

* A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.



**Implement 2 stack in an array**

**Ap 1:**

Divide the array into two equal halves for two stacks. But it is less efficient. Suppose one stack fill every cell of its own but other stack is empty. So, space wasted for stack2.

**Ap 2 :**

Fill stack1 from left to right and stack2 from right to left.

While pushing, we have to be ensured that there exist at least one empty cell in between top1 and top2 ( top2-top1 > 1)

class twoStacks

{

    int arr[] = new int [100];

    int top1 = -1;

    int top2 = arr.length;

    twoStacks()

    {

        for(int i=0; i<100; i++){

            arr[i] = i;

        }

    }

    //Function to push an integer into the stack1.

    void push1(int x)

    {

        // atleast an empty cell

        if((top2 - top1) > 1){

            arr[++top1] = x;

        }

         }

    //Function to push an integer into the stack2.

    void push2(int x)

    {

       // atleast an empty cell

        if((top2 - top1) > 1){

            arr[--top2] = x;

        }

    }

    //Function to remove an element from top of the stack1.

    int pop1()

    {

        if(top1 >= 0){

            return arr[top1--];

        } else{

            return -1;

        }

    }

    //Function to remove an element from top of the stack2.

    int pop2()

    {

        if(top2<this.arr.length){

            return arr[top2++];

        } else{

            return -1;

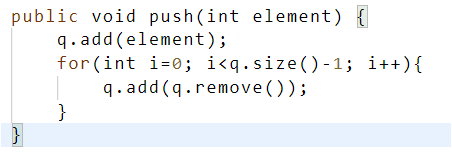
        }

    }

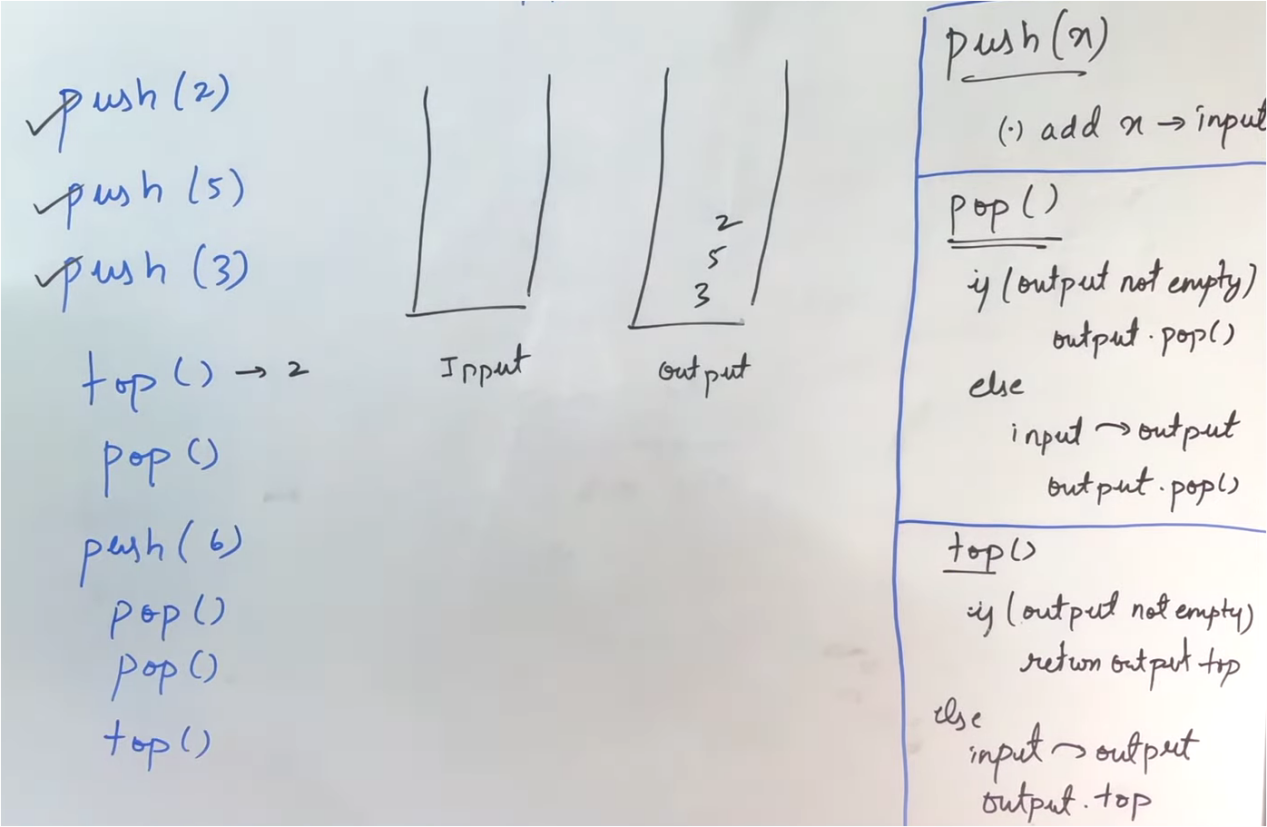
}

# Implement Stack using single Queue

1. Push
   1. Add the element at the end of queue
   2. Run a loop til q.size()-1
      1. Remove from front and add to end
2. Pop, peek is same



# Implement Queueusing single Stack



static class Queue {

        Stack<Integer> input;

        Stack<Integer> output;

        Queue() {

            input=new Stack<>();

            output=new Stack<>();

        }

        void enQueue(int val) {

            input.add(val);

        }

        int deQueue() {

            if(isEmpty()){

                return -1;

            }else{

                if(output.isEmpty()){

                    while(!input.isEmpty()){

                        output.push(input.pop());

                    }

                }

                return output.pop();

            }

        }

        int peek() {

            if(isEmpty()){

                return -1;

            }else{

                if(output.isEmpty()){

                    while(!input.isEmpty()){

                        output.push(input.pop());

                    }

                }

                return output.peek();

            }

        }

        boolean isEmpty() {

            return input.isEmpty()&&output.isEmpty();

        }

    }

**Implement Stack With Linked List**

Push :

When inserting new node, insert at beginning so that pop can be done in O(1)

Pop:

Delete head node

static class Stack {

        Node head;

        int size;

        Stack()

        {

            this.head=null;

            this.size=0;

        }

        int getSize()

        {

            return this.size;

        }

        boolean isEmpty()

        {

            return head==null;

        }

        void push(int data)

        {

            Node newNode=new Node(data);

            // inserting at front

            newNode.next=head;

            head=newNode;

            this.size++;

        }

        int pop()

        {

            if(head==null){

                return -1;

            }

            int val=head.data;

            this.size--;

            head=head.next;

            return val;

        }

        int getTop()

        {

            if(head==null){

                return -1;

            }

            return head.data;

        }

    }

**Implement Queue Using Linked List**

public void enque(int x) {

        Node t=new Node(x);

        if(rear==null){

            front=t;

            rear=t;

        }else{

            rear.next=t;

            rear=t;

        }

    }

    public int dequeue() {

        if(front==null){

            return -1;

        }

        int val=front.data;

        front=front.next;

        if(front==null){

            rear=null;

        }

        return val;

    }

**Balanced parenthesis**

1. Keep a stack of characters.
2. Now iterate over the string
   1. If the current character is a starting bracket that is "(" or "{" or "[" then push it to the stack
   2. If the current character is a closing bracket then check if the stack is empty or not
      1. If the stack is empty return false
      2. If the stack is not empty pop the top character from the stack
      3. Check it with the current closing parenthesis if it is of different nature return false
3. If the stack is empty return true

Time complexity: O( N ) Space complexity: O( N )

public static boolean isValidParenthesis(String s) {

        Stack<Character> st=new Stack<>();

        for(char ch:s.toCharArray()){

            if(ch=='(' || ch=='{' || ch=='['){

                st.push(ch);

            }

            if(st.isEmpty()){

                return false;

            }

            char t=st.peek();

            if(ch==')'&&t=='(' || ch=='}'&&t=='{' || ch==']'&&t=='['){

                st.pop();

            }

        }

        if(st.isEmpty()){

            return true;

        }

        return false;

    }

**302\_Reverse a String using Stack**

Time complexity: O( N ) Space complexity: O( N )

public String reverse(String S){

    StringBuilder ans = new StringBuilder();

    Stack<Character> st = new Stack<>();

    // pushing into stack

    for(int i=0; i<S.length(); i++){

        st.push(S.charAt(i));

    }

    // popping from stack until stack not empty

    while(!st.empty()){

        ans.append(st.pop());

    }

    // can not convert directly from StringBuilder to String

    // so using toString() method

    return ans.toString();

}

**Stack that supports getMin() in O(1) time and O(1) extra space**

Ap 1 :

Push a pair in stack

Pair=element,min so far

But SC would be O(2n)

Ap2 :

****Push:****

Now if there is a push operation just check whether that number is less than the min number. If it is smaller than min we will push a modified value which is a push(2 \* Val – min) into the stack and will update min to the value of the original number. If it’s not then we will just push it as it is.

****Pop :****

While making pop we will check if the top value is lesser than min, If it is then we must update our min to its previous value. In order to do that min = (2 \* min) – (modified value) and we will pop the element.

static class MinStack {

        Stack<Integer> st;

        int min;

        // Constructor

        MinStack() {

            st=new Stack<>();

            this.min=Integer.MAX\_VALUE;

        }

        // Function to add another element equal to num at the top of stack.

        void push(int num) {

            if(st.isEmpty()){

                st.push(num);

                min=num;

            }else{

                if(num<min){

                    st.push(2\*num-min);

                    min=num;

                }else{

                    st.push(num);

                }

            }

        }

        // Function to remove the top element of the stack.

        int pop() {

            if(st.isEmpty()){

                return -1;

            }

            int ele=st.pop();

            if(ele<min){

                int ans=min;

                min=2\*min-ele;

                return ans;

            }else{

                return ele;

            }

        }

        // Function to return the top element of stack if it is present. Otherwise

        // return -1.

        int top() {

            if(st.isEmpty()){

                return -1;

            }

            if(st.peek()<min){

                return min;

            }

            return st.peek();

        }

        // Function to return minimum element of stack if it is present. Otherwise

        // return -1.

        int getMin() {

            if(st.isEmpty()){

                return -1;

            }

            return min;

        }

    }

**304\_Find the next Greater element**

**Ap 1: brute force**

Using nested for loop

Time complexity: O( N\*N ) Space complexity: O( N )

**Ap 2 : using stack**

1. Initialize a empty stack s and vector res of size n.
2. Run a loop from end to start i.e. n-1 to 0 (Right to Left and i-based).
3. In each iteration of loop:
   1. While element at top of stack is less than or equal to current array element, we pop elements from the stack.
      1. Either the stack will be empty(no element greater than current element exists in right side of current element).
      2. The element at top of stack will be greater than current element and closest to the current element in right.
   2. If s is empty then res[i] will be -1 Else it will be top of stack s.
   3. Push current element in stack s.
4. Return res vector as answer.

Time complexity: O( N ) Space complexity: O( N )

public static long[] nextLargerElement(long[] arr, int n)

    {

        Stack<Long> st = new Stack<>();

        // to find next greater of last element

        st.push((long)-1);

        for(int i=n-1; i>=0; i--){

            long cur = arr[i];

            while(st.peek() <= cur && st.peek()!=-1)    {

                st.pop();

            }

            arr[i] = st.peek();

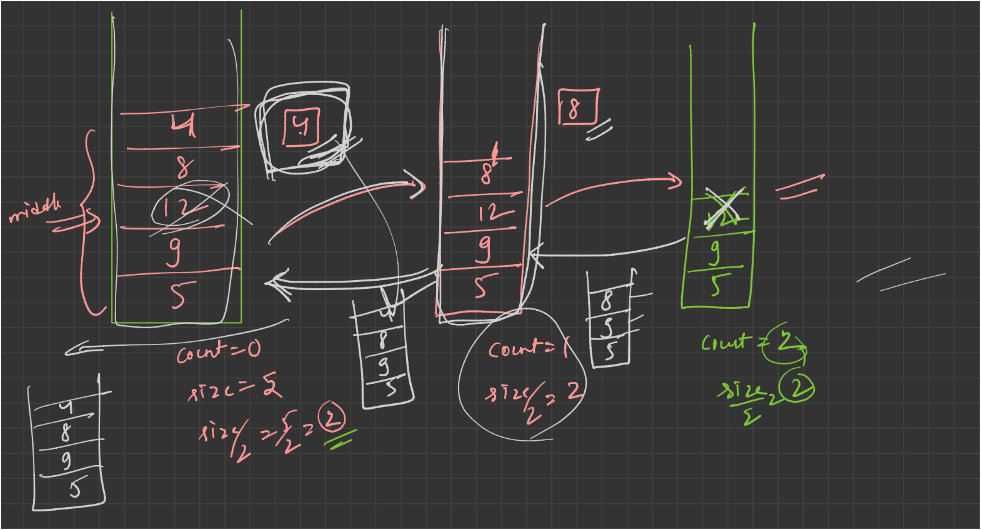
            st.push(cur);

        }

        return arr;

    }

**Delete middle element from stack**



public void dlt(Stack<Integer>s,int sizeOfStack, int count){

    // base case

    if(count==sizeOfStack/2){

        // deleting middle element

        s.pop();

        return;

    }

    // keeping the outstanding element

    int num = s.pop();

    dlt(s,sizeOfStack,count+1);

    // push the outstanding elements when backtracking

    s.push(num);

}

//Function to delete middle element of a stack.

public void deleteMid(Stack<Integer>s,int sizeOfStack){

    int count=0;

    dlt(s,sizeOfStack, count);

}

**308\_Insert an element at bottom**

Same as delete middle element

public void insert(Stack<Integer> St, int X){

if(St.empty()){

St.push(X);

return;

}

int num = St.peek();

St.pop();

insert(St,X);

St.push(num);

}

public Stack<Integer> insertAtBottom(Stack<Integer> St, int X) {

insert(St,X);

return St;

}

**309\_Reverse a stack**

**AP 1: Using recursion**

Same recursive call as deleting middle element. Here, just calling insertAtBottom method recursively.

Time complexity: O( N\*N ) Space complexity: O( 1 )

static void insertAtBottom(Stack<Integer> St, int X){

    if(St.empty()){

        St.push(X);

        return;

    }

    int num = St.pop();

    insertAtBottom(St,X);

    St.push(num);

}

static void reverse(Stack<Integer> s)

{

    // base case

    if(s.empty()){

        return;

    }

    int num = s.pop();

    reverse(s);

    insertAtBottom(s,num);

}

**AP 2: Using Queue (Optimized with extra space)**

1. start extracting the elements from the stack and insert them into a queue.
2. Now the front end to rear end of the queue is containing the reversed elements.
3. So, start extracting the elements from front end and push them into the stack.
4. Return the stack.

Time complexity: O( N ) Space complexity: O( N )

static void reverse(Stack<Integer> s)

    {

        Queue<Integer> q = new LinkedList<>();

        while(!s.empty()){

            q.add(s.pop());

        }

        while(!q.isEmpty()){

            s.push(q.remove());

        }

    }

**310\_Sort a Stack using recursion**

1. The sort function is the entry point. It first checks if the stack is empty. If it is, it simply returns the stack, as an empty stack is already sorted.
2. If the stack is not empty, the top element is removed and stored in the top variable. Then, the sort function is called recursively on the remaining elements in the stack.
3. After the recursive call, the sortedInsert function is used to insert the top element back into the stack at the correct position to maintain the sorted order.
4. Finally, the sorted stack is returned.
5. The sortedInsert function is used to insert the current element into the sorted position of the stack. It uses recursion to find the correct position and shift elements if necessary.

Time Complexity: The time complexity of this algorithm is primarily determined by the sortedInsert function. In the worst case, each element will be inserted in its correct position through a series of recursive calls. Since the recursive calls involve dividing the problem into smaller subproblems, the time complexity is approximately O(n^2), where n is the number of elements in the stack.

Space Complexity: The space complexity is determined by the recursion depth. At any point, there could be at most 'n' recursive calls on the call stack, where 'n' is the number of elements in the stack. Therefore, the space complexity is O(n), considering the space used by the call stack.

public void sortedInsert(Stack<Integer> s, int num){

    if(s.empty() || s.peek()<num){

        s.push(num);

        return;

    }

    int n=s.pop();

    sortedInsert(s,num);

    s.push(n);

}

public Stack<Integer> sort(Stack<Integer> s)

{

    if(s.empty()){

        return s;

    }

    int num=s.pop();

    sort(s);

    sortedInsert(s,num);

    return s;

}

**312\_Largest rectangular Area in Histogram**

1. Iterate through the histogram from left to right (or right to left) and for each bar, find the index of the nearest smaller bar on the left (prevSmaller) and on the right (nextSmaller).
2. For each bar, calculate the area of the rectangle with that bar as the height. The width of the rectangle is the difference between the index of the next smaller bar and the index of the previous smaller bar, minus 1.
3. Keep track of the maximum area encountered during this process.
4. Return the maximum area found.

Time complexity: O( N ) Space complexity: O( N )

public static int[] prevSmaller(ArrayList<Integer> arr, int n) {

        Stack<Integer> st = new Stack<>();

        int[] ans = new int[n];

        for (int i = 0; i < n; i++) {

            while (!st.isEmpty() && arr.get(st.peek()) >= arr.get(i)) {

                st.pop();

            }

            if (st.isEmpty()) {

                ans[i] = -1;

            } else {

                ans[i] = st.peek();

            }

            st.push(i);

        }

        return ans;

    }

    public static int[] nextSmaller(ArrayList<Integer> arr, int n) {

        Stack<Integer> st = new Stack<>();

        int[] ans = new int[n];

        for (int i = n - 1; i >= 0; i--) {

            while (!st.isEmpty() && arr.get(st.peek()) >= arr.get(i)) {

                st.pop();

            }

            if (st.isEmpty()) {

                ans[i] = -1;

            } else {

                ans[i] = st.peek();

            }

            st.push(i);

        }

        return ans;

    }

    public static int largestRectangle(ArrayList<Integer> heights) {

        int n = heights.size();

        int[] ps = prevSmaller(heights, n);

        int[] ns = nextSmaller(heights, n);

        int area = 0;

        for (int i = 0; i < n; i++) {

            int newArea;

            if (ns[i] == -1) {

                newArea = heights.get(i) \* (n - ps[i] - 1);

            } else {

                newArea = heights.get(i) \* (ns[i] - ps[i] - 1);

            }

            area = Math.max(area, newArea);

        }

        return area;

    }

**Maximum Size Rectangle Sub-matrix With All 1's**

1. First finding max area of histogram for 1st row
2. Running nested loops
   1. Assuming current row as base and calculating max area of histogram for that row
   2. area = Math.max(area, largestRectangle(mat[i]));
3. Return area

Time complexity: O( N\*M ) Space complexity: O( N )

public static int maximalAreaOfSubMatrixOfAll1(int[][] mat, int n, int m) {

    // calculate area for 1st row

    int area = largestRectangle(mat[0]);

    for(int i=1; i<n; i++){

        for(int j=0; j<m; j++){

            if(mat[i][j]!=0){

                // assuming current row as base

                mat[i][j] = mat[i][j]+mat[i-1][j];

            }

            else{

                mat[i][j] = 0;

            }

        }

        area = Math.max(area, largestRectangle(mat[i]));

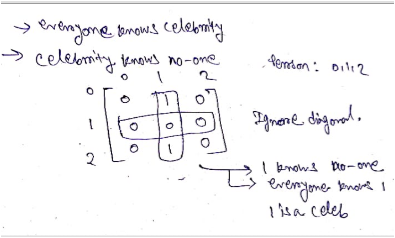
    }

    return area;

}

Methods for prevSmaller, nextSmaller, largestRectangle are same as that of the Largest rectangular Area in Histogram

**305\_The celebrity Problem**



**Ap 1: Brute force**

The idea is to model the solution using graphs. Initialize indegree and outdegree of every vertex as 0. If A knows B, draw a directed edge from A to B, increase indegree of B and outdegree of A by 1. Construct all possible edges of the graph for every possible pair [i, j]. There are NC2 pairs. If a celebrity is present in the party, there will be one sink node in the graph with outdegree of zero and indegree of N-1.

Time complexity: O( N\*N ) Space complexity: O( N )

**Ap 2 : Using Stack**

1. Create a stack and push all the individuals (represented by their indices) onto it.
2. While there are at least two individuals in the stack, pop two individuals.
3. Check if the first individual (a) knows the second individual (b) using the knows function.
4. If a knows b:
   1. push b back onto the stack. Otherwise, push a back onto the stack.
5. After the loop, there will be only one individual left on the stack, which is a potential celebrity candidate.
6. Verify if this potential candidate satisfies the conditions to be a celebrity:
   1. Check each row of the matrix to see if the potential candidate knows anyone. If the candidate knows at least one person in each row, increment the rowCount.
   2. Check each column of the matrix to see if everyone knows the potential candidate. If everyone except the candidate knows them in each column, increment the colCount.
7. If the rowCount equals the total number of individuals and the colCount equals one less than the total number of individuals (indicating that everyone knows the candidate except themselves), the candidate is considered a celebrity.
8. Return the potential candidate if they meet the criteria for being a celebrity. Otherwise, return -1

Time complexity: O( N ) Space complexity: O( N )

boolean knows(int M[][], int a, int b){

        if(M[a][b] ==1 ){

            return true;

        }

        return false;

    }

    //Function to find if there is a celebrity in the party or not.

    int celebrity(int M[][], int n)

    {

        Stack<Integer> st = new Stack<>();

        for(int i=0; i<n; i++){

            st.push(i);

        }

        while(st.size() > 1){

            int a = st.pop();

            int b = st.pop();

            if(knows(M,a,b)){

                st.push(b);

            }

            else{

                st.push(a);

            }

        }

        int potentialCandidate = st.pop();

        boolean celeb = true;

        int rowCount=0, colCount=0;

        // row condition check

        for(int i=0; i<n; i++){

            if(M[potentialCandidate][i]==0){

                rowCount++;

            }

        }

        if(rowCount != n){

            celeb = false;

        }

        // col condition check

        for(int i=0; i<n; i++){

            if(M[i][potentialCandidate]==1){

                colCount++;

            }

        }

        if(colCount != n-1){

            celeb = false;

        }

        if(celeb){

            return potentialCandidate;

        }

        else{

            return -1;

        }

    }

**Ap 3 : optimal without extra space**

1. Create two indices i and j, where i = 0 and j = n-1
2. Run a loop until i is less than j.
3. Check if i knows j, then i can't be a celebrity. so increment i, i.e. i++
4. Else j cannot be a celebrity, so decrement j, i.e. j--
5. Assign i as the celebrity candidate
6. Now at last check whether the candidate is actually a celebrity by re-running a loop from 0 to n-1 and constantly checking that if the candidate knows a person or if there is a candidate who does not know the candidate, then we should return -1. else at the end of the loop, we can be sure that the candidate is actually a celebrity.

Time complexity: O( N ) Space complexity: O( 1 )

//Function to find if there is a celebrity in the party or not.

int celebrity(int M[][], int n)

{

    // initialize two pointers for two corners

    int a=0;

    int b=n-1;

    while(a<b){

        if(M[a][b]==1){

            a++;

        } else{

            b--;

        }

    }

    // checking if a actually celeb or not

    for(int i=0; i<n; i++){

        if( (i!=a) && (M[a][i]==1 || M[i][a]==0) ){

            return -1;

        }

    }

    // celeb is a

    return a;

}

**314\_Expression contains redundant bracket or not**

1. We iterate through the given expression and for each character in the expression
   1. if the character is an open parenthesis ‘(‘ or any of the operators or operands, we push it to the stack.
   2. If the character is close parenthesis ‘)’, then pop characters from the stack till matching open parenthesis ‘(‘ is found.
2. Now for redundancy two conditions will arise while popping.
   1. If immediate pop hits an open parenthesis ‘(‘, then we have found a duplicate parenthesis. For example, (((a+b))+c) has duplicate brackets around a+b. When we reach the second “)” after a+b, we have “((” in the stack. Since the top of the stack is an opening bracket, we conclude that there are duplicate brackets.
   2. If immediate pop doesn’t hit any operand(‘\*’, ‘+’, ‘/’, ‘-‘) then it indicates the presence of unwanted brackets surrounded by expression. For instance, (a)+b contains unwanted () around a thus it is redundant.

Time complexity: O( N ) Space complexity: O( N )

public static boolean findRedundantBrackets(String s)

    {

        Stack<Character> st = new Stack<>();

        for(int i=0; i<s.length(); i++){

            char ch = s.charAt(i);

            if(ch == '(' || ch == '+' ||ch == '-' || ch == '\*' || ch == '/'){

                st.push(ch);

            }

            else{

                if(ch==')'){

                    boolean isRedundant=true;

                    while(st.peek() != '('){

                        char top = st.peek();

                        if(top == '+' ||top == '-' || top == '\*' || top == '/') {

                            isRedundant = false;

                        }

                        st.pop();

                    }

                    if(isRedundant == true)

                        return true;

                    st.pop();

                }

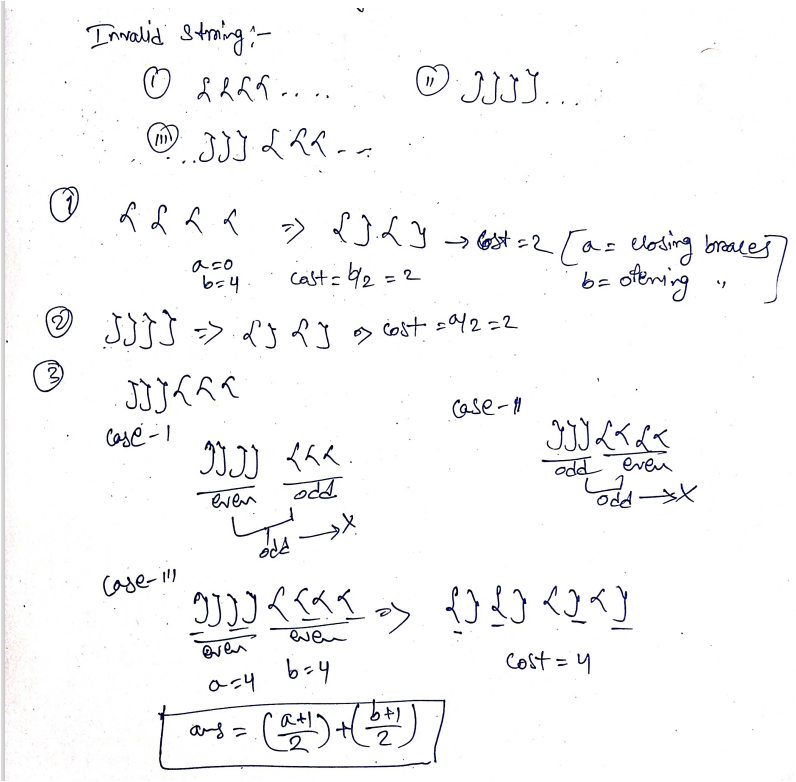
            }

        }

        return false;

}

**Minimum Cost To Make String Valid**



1. If the length of ‘STR’ is odd, it is impossible to make ‘STR’ valid. Because each ‘{’ must have a ‘}’ to pair up. Return -1.
2. Iterate the ‘STR’ and for each index do the following:
   1. If ‘STR[i]’ = ‘}’ and the stack is not empty then do the following:
      1. If the top element of the stack is '{' then we will pop from the stack.
      2. Else we will push the ‘STR[i]’ into the stack.
   2. Else push ‘STR[i]’ into the stack.
3. Now we have removed the valid part from ‘STR’ and we have a string like “}}{{…{“ as explained above. Inside the stack, so we will pop out all the elements and count the number of ‘{’ and ‘}’. Take two variables ‘p’ and ‘q’ where ‘p’ denotes the number of ‘{’ and ‘q’ denotes the number of ‘}’ and do the following steps while the stack not gets empty:
   1. If the top element of the stack is ‘{’ then increase p by 1.
   2. If the top element of the stack is ‘}’ then increase q by 1.
   3. Finally return ((‘p’ + 1) / 2 + (‘q’ + 1) / 2) because half of the ‘{’ can make remaining ‘{’ valid and same for ‘}’

Time Complexity : O(|STR|) where |STR| denotes the length of string ‘STR’.

Because we are traversing each element in the ‘STR’ exactly once and popping out from stack once so we get O(|STR| + |STR|) which is equal to O(|STR|).

Space Complexity : O(|STR|) where |STR| denotes the length of string ‘STR’. Because in the worst case stack will have size equals to the length of ‘STR’.

public static int findMinimumCost(String str) {

        // invalid, if string of odd length

        if(str==null || str.length()%2==1){

            return -1;

        }

        Stack<Character> st = new Stack<>();

        // removing valid parts

        for(char ch : str.toCharArray()){

            if(ch=='{'){

                st.push(ch);

            }

            else{

                if(!st.empty() && st.peek()=='{'){

                    st.pop();

                }

                else{

                    st.push(ch);

                }

            }

        }

        // calculate min cost for invalid part

        int openCount=0;

        int closeCount=0;

        while(!st.empty()) {

            char temp = st.pop();

            if(temp=='{'){

                openCount++;

            }else{

                closeCount++;

            }

        }

        int ans = (closeCount+1)/2 + (openCount+1)/2;

        return ans;

    }