**Queue Using Two Stack**

class StackQueue

{

Stack<Integer> s1 = new Stack<Integer>();

Stack<Integer> s2 = new Stack<Integer>();

//Function to push an element in queue by using 2 stacks.

void Push(int x)

{

// Your code here

s1.push(x);

}

//Function to pop an element from queue by using 2 stacks.

int Pop()

{

// Your code here

if(s1.isEmpty()){

return -1;

}

while(!s1.isEmpty()){

s2.push(s1.pop());

}

int data = s2.pop();

while(!s2.isEmpty()){

s1.push(s2.pop());

}

return data;

}

}

**First non Repeating Character in the stream**

class Solution {

public String FirstNonRepeating(String A) {

Map<Character, Integer> charCounts = new LinkedHashMap<>();

StringBuilder ans = new StringBuilder();

for (char c : A.toCharArray()) {

charCounts.put(c, charCounts.getOrDefault(c, 0) + 1);

char firstNonRepeating = '#';

for (char ch : charCounts.keySet()) {

if (charCounts.get(ch) == 1) {

firstNonRepeating = ch;

break;

}

}

ans.append(firstNonRepeating);

}

return ans.toString();

}

}

**Check for Balanced Tree**

class Tree

{

//Function to check whether a binary tree is balanced or not.

public boolean isBalanced(Node root) {

return checkBalance(root) != -1;

}

private int checkBalance(Node root) {

if (root == null) {

return 0;

}

int leftHeight = checkBalance(root.left);

if (leftHeight == -1) {

return -1;

}

int rightHeight = checkBalance(root.right);

if (rightHeight == -1) {

return -1;

}

int heightDiff = Math.abs(leftHeight - rightHeight);

if (heightDiff > 1) {

return -1;

}

return Math.max(leftHeight, rightHeight) + 1;

}

}

**Diameter of Binary Tree**

class Solution {

// Function to return the diameter of a Binary Tree.

public int diameter(Node root) {

if (root == null) {

return 0;

}

int leftHeight = height(root.left);

int rightHeight = height(root.right);

int leftDiameter = diameter(root.left);

int rightDiameter = diameter(root.right);

int rootDiameter = leftHeight + rightHeight + 1;

return Math.max(rootDiameter, Math.max(leftDiameter, rightDiameter));

}

private int height(Node root) {

if (root == null) {

return 0;

}

return 1+Math.max(height(root.left), height(root.right));

}

}

**Check for BST**

class Solution {

// Function to check whether a Binary Tree is BST or not.

boolean isBST(Node root) {

return isBSTUtil(root, Integer.MIN\_VALUE, Integer.MAX\_VALUE);

}

boolean isBSTUtil(Node root, int min, int max) {

if (root == null)

return true;

if (root.data <= min || root.data >= max)

return false;

return isBSTUtil(root.left, min, root.data) && isBSTUtil(root.right, root.data, max);

}

}

**Top view of Binary Tree**

class Solution

{

//Function to return a list of nodes visible from the top view

//from left to right in Binary Tree.

public ArrayList<Integer> topView(Node root) {

ArrayList<Integer> topViewList = new ArrayList<>();

if (root == null) {

return topViewList;

}

TreeMap<Integer, Integer> horizontalDistances = new TreeMap<>();

Queue<TreeNodeHD> queue = new LinkedList<>();

queue.offer(new TreeNodeHD(root, 0));

while (!queue.isEmpty()) {

TreeNodeHD nodeHD = queue.poll();

Node node = nodeHD.node;

int hd = nodeHD.horizontalDistance;

if (!horizontalDistances.containsKey(hd)) {

horizontalDistances.put(hd, node.data);

}

if (node.left != null) {

queue.offer(new TreeNodeHD(node.left, hd - 1));

}

if (node.right != null) {

queue.offer(new TreeNodeHD(node.right, hd + 1));

}

}

for (Map.Entry<Integer, Integer> entry : horizontalDistances.entrySet()) {

topViewList.add(entry.getValue());

}

return topViewList;

}

}

class TreeNodeHD {

Node node;

int horizontalDistance;

TreeNodeHD(Node node, int hd) {

this.node = node;

horizontalDistance = hd;

}

}

**Find Median in the Stream**

class Solution

{

PriorityQueue<Integer> maxHeap; // Max heap to store the smaller half of the elements

PriorityQueue<Integer> minHeap; // Min heap to store the greater half of the elements

public Solution() {

maxHeap = new PriorityQueue<>(Collections.reverseOrder());

minHeap = new PriorityQueue<>();

}

public void insertHeap(int x) {

if (maxHeap.isEmpty() || x <= maxHeap.peek()) {

maxHeap.offer(x);

} else {

minHeap.offer(x);

}

balanceHeaps();

}

private void balanceHeaps() {

if (maxHeap.size() > minHeap.size() + 1) {

minHeap.offer(maxHeap.poll());

} else if (minHeap.size() > maxHeap.size()) {

maxHeap.offer(minHeap.poll());

}

}

public double getMedian() {

if (maxHeap.isEmpty() && minHeap.isEmpty()) {

return -1; // No elements in the stream

}

if (maxHeap.size() == minHeap.size()) {

return (maxHeap.peek() + minHeap.peek()) / 2.0;

} else {

return maxHeap.peek();

}

}

}

**Kth Largest element in the stream**

class Solution {

public int[] kthLargest(int k, int[] arr, int n) {

int[] result = new int[n];

PriorityQueue<Integer> minHeap = new PriorityQueue<>();

for (int i = 0; i < n; i++) {

if (minHeap.size() < k) {

minHeap.offer(arr[i]);

} else if (arr[i] > minHeap.peek()) {

minHeap.poll();

minHeap.offer(arr[i]);

}

if (minHeap.size() < k) {

result[i] = -1;

} else {

result[i] = minHeap.peek();

}

}

return result;

}

}

**Union of Two Array**

class Solution {

public int doUnion(int a[], int n, int b[], int m) {

Set<Integer> unionSet = new HashSet<>();

for (int i = 0; i < n; i++) {

unionSet.add(a[i]);

}

for (int i = 0; i < m; i++) {

unionSet.add(b[i]);

}

return unionSet.size();

}

}

**Largest Subarray with sum 0**

class GfG

{

public int maxLen(int[] arr, int n) {

int maxLength = 0;

int prefixSum = 0;

HashMap<Integer, Integer> prefixSumMap = new HashMap<>();

for (int i = 0; i < n; i++) {

prefixSum += arr[i];

if (prefixSum == 0) {

maxLength = i + 1;

} else if (prefixSumMap.containsKey(prefixSum)) {

maxLength = Math.max(maxLength, i - prefixSumMap.get(prefixSum));

} else {

prefixSumMap.put(prefixSum, i);

}

}

return maxLength;

}

}