1. Implement the following scenario by **Bridge pattern** in Java :

Add the student names into the class register like “Ajay”, “Bala”, “Cathey”,”Chella”,”Dolly”, ”Ellan”, Francis”, ”Stella”.

Do the following operations on the class register

1. Display all student names
2. Delete “chella” from the register
3. Display previous and next names from the register.
4. Add “Zara” into the register
5. Display all student names after addition of “Zara”
6. Implement BubbleSort and Merge sort sorting algorithms using Strategy pattern

Input: 34,12,67,10,7

Output: 7,10,12,34,67

Ans-

SortingStrategy.java

public interface SortingStrategy

{

int[] sort( int[] inputArray );

}

BubbleSort.java

public class BubbleSort implements SortingStrategy

{

public int[] sort( int[] inputArray )

{

int n = inputArray.length;

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

{

for( int j = 1; j < (n - i); j++ )

{

if( inputArray[j - 1] > inputArray[j] )

{

swap(j - 1, j, inputArray);

}

}

}

System.out.println("Array is sorted using Bubble Sort Algorithm");

return inputArray;

}

private void swap( int k, int l, int[] inputArray )

{

int temp = inputArray[k];

inputArray[k] = inputArray[l];

inputArray[l] = temp;

}

}

public class MergeSort implements SortingStrategy{

**void** merge(**int** a[], **int** beg, **int** mid, **int** end)

{

**int** i, j, k;

**int** n1 = mid - beg + 1;

**int** n2 = end - mid;

**int** LeftArray[n1], RightArray[n2]; //temporary arrays

    /\* copy data to temp arrays \*/

**for** (**int** i = 0; i < n1; i++)

    LeftArray[i] = a[beg + i];

**for** (**int** j = 0; j < n2; j++)

    RightArray[j] = a[mid + 1 + j];

    i = 0, /\* initial index of first sub-array \*/

    j = 0; /\* initial index of second sub-array \*/

    k = beg;  /\* initial index of merged sub-array \*/

**while** (i < n1 && j < n2)

    {

**if**(LeftArray[i] <= RightArray[j])            {

            a[k] = LeftArray[i];

            i++;

        }

**else**

        {

            a[k] = RightArray[j];

            j++;

        }

        k++;

    }

**while** (i<n1)

    {

        a[k] = LeftArray[i];

        i++;

        k++;

    }

**while** (j<n2)

    {

        a[k] = RightArray[j];

        j++;

        k++;

    }

}    }

SortContext.java

import java.util.Scanner;

public class SortContext

{

private SortingStrategy sortingStrategy;

public void setSortingStrategy( SortingStrategy sortingStrategy )

{

this.sortingStrategy = sortingStrategy;

}

private void printArray( int[] inputArray )

{

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

{

System.out.print(inputArray[i] + ",");

}

System.out.println("\n");

}

private int[] readUserInput()

{

System.out.println("Enter array size : ");

Scanner scanner = new Scanner(System.in);

int n = scanner.nextInt();

System.out.println("Enter input array : ");

int[] inputArray = new int[n];

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

{

inputArray[i] = scanner.nextInt();

}

return inputArray;

}

public void sort()

{

int[] inputArray = readUserInput();

inputArray = sortingStrategy.sort(inputArray);

printArray(inputArray);

}

}

StrategyClient.java

import java.util.Scanner;

public class StrategyClient

{

public static void main( String[] args )

{

System.out.println("Please enter Sort Algorithm : 'BubbleSort' or ‘MergeSort’

Scanner scanner = new Scanner(System.in);

String sortAlgorithm = scanner.next();

System.out.println("Sort Algorithm is : " + sortAlgorithm);

SortContext context = new SortContext();

if( "BubbleSort".equalsIgnoreCase(sortAlgorithm) )

{

context.setSortingStrategy(new BubbleSort());

}

else if( "MergeSort".equalsIgnoreCase(sortAlgorithm) )

{

context.setSortingStrategy(new MergeSort());

}

context.sort();

}

}

1. A Simple Scenario: As a developer I want to implement code so that it prints the numbers from 1 to 100.

Given: an input of numbers from 1–100  
When:  
A number is a multiple of ‘3’ return “Fizz”  
A number is a of ‘5’ return “Buzz”  
A number is a of both ‘3’ and ‘5’ return “FizzBuzz”  
A number is not divisible by ‘3’ or ‘5’ return the number itself  
Then: print “Fizz”, “Buzz”, “FizzBuzz” or the number accordingly

**Expected output:** 1, 2, Fizz, 4, Buzz, ……, 14, FizzBuzz, 16, …

Write the test cases and make all to be passed.

Ans-

public class Main

{

public static void main(String args[])

{

for (int i=1; i<=100; i++)

{

// Numbers that are divisible by 3 and 5

// are always divisible by 15

// Therefore, "FizzBuzz" is printed in place of that number

if (i%15==0)

{

System.out.print("FizzBuzz"+" ");

}

else if (i%3==0)

{

System.out.print("Fizz"+" ");

}

// "Buzz" is printed in place of numbers

// that are divisible by 5

else if (i%5==0)

{

System.out.print("Buzz"+" ");

}

// If none of the above conditions are satisfied,

// the number is printed

else

{

System.out.print(i+" ");

}

}

}

}

1. Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:
   1. Only one disk can be moved at a time.
   2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
   3. No disk may be placed on top of a smaller disk

**Sample Inpu**t:

* 3

**Sample Output:**

* Disk 1 moved from A to C
* Disk 2 moved from A to B
* Disk 1 moved from C to B
* Disk 3 moved from A to C
* Disk 1 moved from B to A
* Disk 2 moved from B to C
* Disk 1 moved from A to C

Ans-

import java.io.\*;

import java.math.\*;

import java.util.\*;

class GFG {

static void towerOfHanoi(int n, char from\_rod,

char to\_rod, char aux\_rod)

{

if (n == 0) {

return;

}

towerOfHanoi(n - 1, from\_rod, aux\_rod, to\_rod);

System.out.println("Move disk " + n + " from rod "

+ from\_rod + " to rod "

+ to\_rod);

towerOfHanoi(n - 1, aux\_rod, to\_rod, from\_rod);

}

// Driver code

public static void main(String args[])

{

int N = 3;

// A, B and C are names of rods

towerOfHanoi(N, 'A', 'C', 'B');

}

}

1. There are n gas stations along a circular route, where the amount of gas at the ith station is gas[i].

You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from the ith station to its next (i + 1)th station. You begin the journey with an empty tank at one of the gas stations.

Given two integer arrays gas and cost, return *the starting gas station's index if you can travel around the circuit once in the clockwise direction, otherwise return* -1. If there exists a solution, it is **guaranteed** to be **unique**

**Example 1:**

**Input:** gas = [1,2,3,4,5], cost = [3,4,5,1,2]

**Output:** 3

**Explanation:**

Start at station 3 (index 3) and fill up with 4 unit of gas. Your tank = 0 + 4 = 4

Travel to station 4. Your tank = 4 - 1 + 5 = 8

Travel to station 0. Your tank = 8 - 2 + 1 = 7

Travel to station 1. Your tank = 7 - 3 + 2 = 6

Travel to station 2. Your tank = 6 - 4 + 3 = 5

Travel to station 3. The cost is 5. Your gas is just enough to travel back to station 3.

Therefore, return 3 as the starting index.

Ans-

public int canCompleteCircuit(int[] gas, int[] cost) {

int gassum=0,costsum=0;

int n=gas.length;

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

gassum+=gas[i];

costsum+=cost[i];

}

if(gassum<costsum) return -1;

int total=0;

int pos=0;

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

total+=gas[i]-cost[i];

if(total < 0){

total=0;

pos=i+1;

}

}

return pos;

}

1. Given a string S of parentheses ‘(‘ or ‘)’. The task is to find a minimum number of parentheses ‘(‘ or ‘)’ (at any positions) we must add to make the resulting parentheses string is valid.

**Examples:** 

**Input:** str = "())"

**Output:** 1

One '(' is required at beginning.

Give your explanation for the above problem using Stack or Queue structure.

Find out the minimum number of parentheses needed to complete the input

1. ()[(){()}]
2. []{()()}
3. ((()))()()
4. ())((())

Ans: **public** **class** GFG {

    // Function to return required minimum number

**static** **int** minParentheses(String p)

    {

        // maintain balance of string

**int** bal = 0;

**int** ans = 0;

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

            bal += p.charAt(i) == '(' ? 1 : -1;

            // It is guaranteed bal >= -1

**if** (bal == -1) {

                ans += 1;

                bal += 1;

            }

        }

**return** bal + ans;

    }

**public** **static** **void** main(String args[])

    {

        String p = "())";

        // Function to print required answer

        System.out.println(minParentheses(p));

    }

    // This code is contributed by ANKITRAI1

}

1. Write a program for AVL tree having functions for the following operations:

* Insert an element (no duplicates are allowed),
* Delete an existing element,
* Traverse the AVL (in-order, pre-order, and post-order)

**INPUT:**

Line 1 contains an integer NQ, the number of queries.

The next NQ lines contain queries and are of the form 'i x*x*' (Insert x*x* into an AVL tree) or 'd x*x*' (Delete x*x* from an AVL tree).

**OUTPUT:**

For each query, print value of an unbalanced node (if any) at which rotation is being applied.

The last three lines print 'Preorder traversal', 'Inorder traversal', and 'Postorder traversal' of an AVL tree that results after the execution of all NQ queries.

**SAMPLE INPUT:**

8

i 1

i 2

i 3

i 4

i 5

i 6

d 4

d 5

SAMPLE OUTPUT:

1

3

2

6

2 1 6 3

1 2 3 6

* + 1. 3 6 2

// Java program for deletion in AVL Tree

import java.util.\*;

class Node

{

int key, height;

Node left, right;

Node(int d)

{

key = d;

height = 1;

}

}

class AVLTree

{

Node root;

// A utility function to get height of the tree

int height(Node N)

{

if (N == null)

return 0;

return N.height;

}

// A utility function to get maximum of two integers

int max(int a, int b)

{

return (a > b) ? a : b;

}

Node rightRotate(Node y)

{

Node x = y.left;

Node T2 = x.right;

// Perform rotation

x.right = y;

y.left = T2;

// Update heights

y.height = max(height(y.left), height(y.right)) + 1;

x.height = max(height(x.left), height(x.right)) + 1;

// Return new root

return x;

}

Node leftRotate(Node x)

{

Node y = x.right;

Node T2 = y.left;

// Perform rotation

y.left = x;

x.right = T2;

// Update heights

x.height = max(height(x.left), height(x.right)) + 1;

y.height = max(height(y.left), height(y.right)) + 1;

// Return new root

return y;

}

// Get Balance factor of node N

int getBalance(Node N)

{

if (N == null)

return 0;

int y=height(N.left) - height(N.right);

if(y>1 || y<-1)

{

System.out.println(N.key);

}

return y;

}

Node insert(Node node, int key)

{

/\* 1. Perform the normal BST rotation \*/

if (node == null)

return (new Node(key));

if (key < node.key)

node.left = insert(node.left, key);

else if (key > node.key)

node.right = insert(node.right, key);

else // Equal keys not allowed

return node;

/\* 2. Update height of this ancestor node \*/

node.height = 1 + max(height(node.left),

height(node.right));

/\* 3. Get the balance factor of this ancestor

node to check whether this node became

Wunbalanced \*/

int balance = getBalance(node);

// If this node becomes unbalanced, then

// there are 4 cases Left Left Case

if (balance > 1 && key < node.left.key)

{

//System.out.println(root.key);

return rightRotate(node);

}

// Right Right Case

if (balance < -1 && key > node.right.key){

// System.out.println(root.key);

return leftRotate(node);

}

// Left Right Case

if (balance > 1 && key > node.left.key)

{

//System.out.println(root.key);

node.left = leftRotate(node.left);

return rightRotate(node);

}

// Right Left Case

if (balance < -1 && key < node.right.key)

{

//System.out.println(root.key);

node.right = rightRotate(node.right);

return leftRotate(node);

}

/\* return the (unchanged) node pointer \*/

return node;

}

/\* Given a non-empty binary search tree, return the

node with minimum key value found in that tree.

Note that the entire tree does not need to be

searched. \*/

Node minValueNode(Node node)

{

Node current = node;

/\* loop down to find the leftmost leaf \*/

while (current.left != null)

current = current.left;

return current;

}

Node deleteNode(Node root, int key)

{

// STEP 1: PERFORM STANDARD BST DELETE

if (root == null)

return root;

// If the key to be deleted is smaller than

// the root's key, then it lies in left subtree

if (key < root.key)

root.left = deleteNode(root.left, key);

else if (key > root.key)

root.right = deleteNode(root.right, key);

else

{

if ((root.left == null) || (root.right == null))

{

Node temp = null;

if (temp == root.left)

temp = root.right;

else

temp = root.left;

if (temp == null)

{

temp = root;

root = null;

}

else

root = temp;

}

else

{

Node temp = minValueNode(root.right);

root.key = temp.key;

// Delete the inorder successor

root.right = deleteNode(root.right, temp.key);

}

}

if (root == null)

return root;

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

int balance = getBalance(root);

if (balance > 1 && getBalance(root.left) >= 0){

// System.out.println(root.key);

return rightRotate(root);}

if (balance > 1 && getBalance(root.left) < 0)

{

// System.out.println(root.key);

root.left = leftRotate(root.left);

return rightRotate(root);

}

if (balance < -1 && getBalance(root.right) <= 0){

//System.out.println(root.key);

return leftRotate(root);}

if (balance < -1 && getBalance(root.right) > 0)

{

// System.out.println(root.key);

root.right = rightRotate(root.right);

return leftRotate(root);

}

return root;

}

void preOrder(Node node)

{

if (node != null)

{

System.out.print(node.key + " ");

preOrder(node.left);

preOrder(node.right);

}

}

void inOrder(Node node)

{

if (node != null)

{

inOrder(node.left);

System.out.print(node.key + " ");

inOrder(node.right);

}

}

void postOrder(Node node)

{

if (node != null)

{

// System.out.print(node.key + " ");

postOrder(node.left);

postOrder(node.right);

System.out.print(node.key + " ");

}

}

public static void main(String[] args)

{

Scanner scanner = new Scanner(System.in);

AVLTree avlTree = new AVLTree();

char ch;

int t=scanner.nextInt();

for(int i=0;i<t;i++)

{

ch= scanner.next().charAt(0);

int g=scanner.nextInt();

switch (ch)

{

case 'i':

avlTree.root=avlTree.insert(avlTree.root,g);

break;

case 'd':

avlTree.root= avlTree.deleteNode(avlTree.root,g);

break;

}

}

avlTree.preOrder(avlTree.root);

System.out.println();

avlTree.inOrder(avlTree.root);

System.out.println();

avlTree.postOrder(avlTree.root);

}

}

1. Write a Java program to find the sorted list by using Topological sorting [ Hint: apply DFS concept]

The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no incoming edges).



**Output:**

* + 1. 4 2 3 1 0 **or** 4 5 2 3 1 0

**// A Java program to print topological**

**// sorting of a DAG**

**import java.io.\*;**

**import java.util.\*;**

**// This class represents a directed graph**

**// using adjacency list representation**

**class Graph {**

**// No. of vertices**

**private int V;**

**// Adjacency List as ArrayList of ArrayList's**

**private ArrayList<ArrayList<Integer> > adj;**

**// Constructor**

**Graph(int v)**

**{**

**V = v;**

**adj = new ArrayList<ArrayList<Integer> >(v);**

**for (int i = 0; i < v; ++i)**

**adj.add(new ArrayList<Integer>());**

**}**

**// Function to add an edge into the graph**

**void addEdge(int v, int w) { adj.get(v).add(w); }**

**// A recursive function used by topologicalSort**

**void topologicalSortUtil(int v, boolean visited[],**

**Stack<Integer> stack)**

**{**

**// Mark the current node as visited.**

**visited[v] = true;**

**Integer i;**

**// Recur for all the vertices adjacent**

**// to thisvertex**

**Iterator<Integer> it = adj.get(v).iterator();**

**while (it.hasNext()) {**

**i = it.next();**

**if (!visited[i])**

**topologicalSortUtil(i, visited, stack);**

**}**

**// Push current vertex to stack**

**// which stores result**

**stack.push(new Integer(v));**

**}**

**// The function to do Topological Sort.**

**// It uses recursive topologicalSortUtil()**

**void topologicalSort()**

**{**

**Stack<Integer> stack = new Stack<Integer>();**

**// Mark all the vertices as not visited**

**boolean visited[] = new boolean[V];**

**for (int i = 0; i < V; i++)**

**visited[i] = false;**

**// Call the recursive helper**

**// function to store**

**// Topological Sort starting**

**// from all vertices one by one**

**for (int i = 0; i < V; i++)**

**if (visited[i] == false)**

**topologicalSortUtil(i, visited, stack);**

**// Print contents of stack**

**while (stack.empty() == false)**

**System.out.print(stack.pop() + " ");**

**}**

**// Driver code**

**public static void main(String args[])**

**{**

**// Create a graph given in the above diagram**

**Graph g = new Graph(6);**

**g.addEdge(5, 2);**

**g.addEdge(5, 0);**

**g.addEdge(4, 0);**

**g.addEdge(4, 1);**

**g.addEdge(2, 3);**

**g.addEdge(3, 1);**

**System.out.println("Following is a Topological "**

**+ "sort of the given graph");**

**// Function Call**

**g.topologicalSort();**

**}**

**}**

**// This code is contributed by Aakash Hasija**

1. Write a Java program to find the K-th smallest element in BST.

Sample input: K=3

A picture containing clock

Description automatically generated

Sample Output:

10

// A simple inorder traversal based Java program

// to find k-th smallest element in a BST.

import java.io.\*;

// A BST node

class Node {

int data;

Node left, right;

Node(int x)

{

data = x;

left = right = null;

}

}

class GFG {

static int count = 0;

// Recursive function to insert an key into BST

public static Node insert(Node root, int x)

{

if (root == null)

return new Node(x);

if (x < root.data)

root.left = insert(root.left, x);

else if (x > root.data)

root.right = insert(root.right, x);

return root;

}

// Function to find k'th largest element in BST

// Here count denotes the number of nodes processed so far

public static Node kthSmallest(Node root, int k)

{

// base case

if (root == null)

return null;

// search in left subtree

Node left = kthSmallest(root.left, k);

// if k'th smallest is found in left subtree, return it

if (left != null)

return left;

// if current element is k'th smallest, return it

count++;

if (count == k)

return root;

// else search in right subtree

return kthSmallest(root.right, k);

}

// Function to find k'th largest element in BST

public static void printKthSmallest(Node root, int k)

{

Node res = kthSmallest(root, k);

if (res == null)

System.out.println("There are less than k nodes in the BST");

else

System.out.println("K-th Smallest Element is " + res.data);

}

public static void main(String[] args)

{

Node root = null;

int keys[] = { 20, 8, 22, 4, 12, 10, 14 };

for (int x : keys)

root = insert(root, x);

int k = 3;

printKthSmallest(root, k);

}

}

// This code is contributed by Aditya Kumar (adityakumar129)

1. Given the root of a binary tree and an integer targetSum, return true if the tree has a root-to-leaf path such that adding up all the values along the path equals targetSum.

A leaf is a node with no children.

**Example 1:**

class Solution

{

private boolean flag= false;//we are taking the base case as false, if the sum is not found and if the target sum lies in the path from root to leaf, we make the flag true

public boolean hasPathSum(TreeNode root, int targetSum)

{

int sum = 0;//calculate the downward sum

Helper(root, sum, targetSum);//it helps us calculate the sum of each node for each path from root to leaf and tells us that our desired sum is present or not, by adding the node at that path

return flag;//returns true if sum is present else false is returned

}

public TreeNode Helper(TreeNode root, int sum, int targetSum)

{//postorder traversal + semi preorder traversal(accesing root beforehand once), we want to know about child first and then the parent

if(root == null)//base case when we hit the null node, to tell the parent no node is present we return null

return null;

//ROOT

sum= sum + root.val;//adding the node value to sum at each level//accesing root beforehand once

TreeNode left= Helper(root.left, sum, targetSum);//recursing down the left subtree and knowing about the left child//storring the reference to detect it is leaf or not, if it is leaf then left == null//LEFT

TreeNode right= Helper(root.right, sum, targetSum);//recursing down the right subtree and knowing about the right child//storring the reference to detect it is leaf or not, if it is leaf then right == null//RIGHT

//ROOT

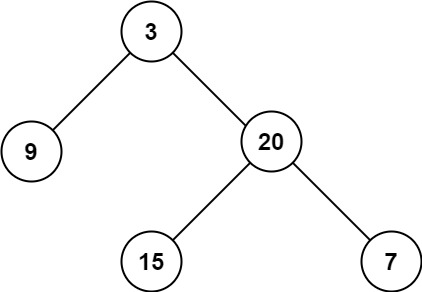
if(sum == targetSum && left == null && right == null) //if we reached to our desired sum and the parent node is a leaf node, we got our path finally

flag= true;//we make the flag to true as the desired path from root to leaf is achived

return root;//returning root, to tell the parent that I am present

}

}



**I**nput: root = [3,9,20,null,null,15,7]  
Output: 3