

Parul University

FACULTY OF ENGINEERING AND TECHNOLOGY

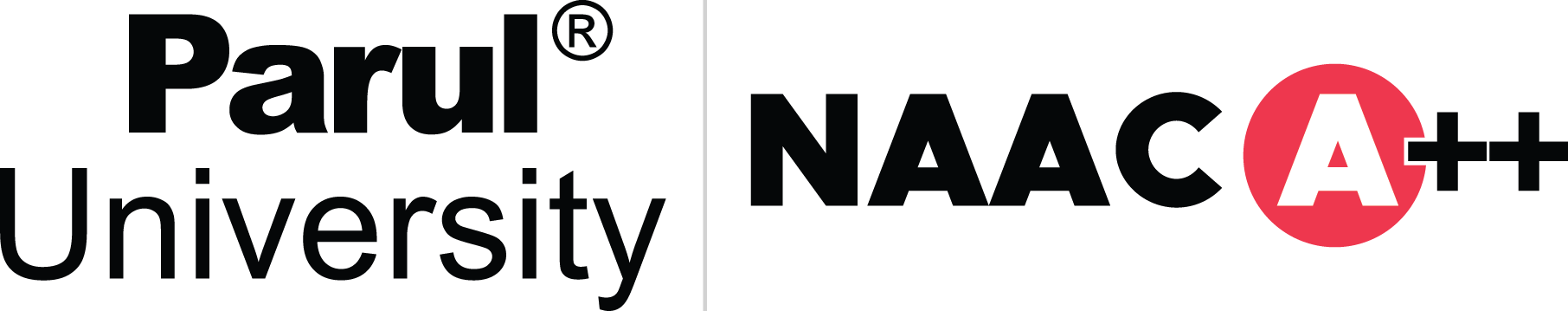
BACHELOR OF TECHNOLOGY

COMPETITIVE CODING-LEVEL 2A

(303105259)

IV SEMESTER

Computer Science & Engineering Department



Laboratory Manual

Session 2023-24

CERTIFICATE

This is to Certify that

Mr./Ms. ㅤㅤㅤㅤㅤ VIJAY KUMAR ㅤㅤㅤㅤㅤㅤㅤㅤㅤㅤㅤㅤㅤㅤㅤㅤㅤㅤㅤ

With enrolment no. ㅤ 2203051050620 has successfully completed his/her

Laboratory experiments in COMPETITIVE CODING (303105259) From the

department of ㅤㅤㅤCOMPUTER SCIENCE AND ENGINEERING ㅤ ㅤㅤduring the academic year 2023 - 2024ㅤㅤ



Date of Submission : ………………………. Staff In Charge: ……………………….

Head of department: ……………………….

**TABLE OF CONTENT**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Sr. No** | **Experiment Title** | **Page No** | | **Date of Completion** | **Sign** | **Marks (out of 10)** |
| **From** | **To** |
| **1.** | Write a program for implementing a MINSTACK which should support operations like push, pop, overflow, underflow, display a. Construct a stack of N-capacity b. Push elements c. Pop elements d. Top element e. Retrieve the min element from the stack |  |  |  |  |  |
| **2.** | Write a program to deal with real-world situations where Stack data structure is widely used Evaluation of expression: Stacks are used to evaluate expressions, especially in languages that use postfix or prefix notation. Operators and operands are pushed onto the stack, and operations are performed based on the LIFO principle. |  |  |  |  |  |
| **3.** | Write a program for finding NGE NEXT GREATER ELEMENT from an array |  |  |  |  |  |
| **4.** | Write a program to design a circular queue(k) which Should implement the below functions a. Enqueue b. Dequeue c. Front d. Rear |  |  |  |  |  |
| **5.** | Write a Program for an infix expression, and convert it to postfix notation. Use a queue to implement the Shunting Yard Algorithm for expression conversion. |  |  |  |  |  |
| **6.** | Write a Program for finding the Product of the three largest Distinct Elements. Use a Priority Queue to efficiently find and remove the largest elements |  |  |  |  |  |
| **7.** | Write a Program to Merge two linked lists(sorted) |  |  |  |  |  |
| **8.** | Write a Program to find the Merge point of two linked lists(sorted) |  |  |  |  |  |
| **9.** | Write a Program to Swap Nodes pairwise |  |  |  |  |  |
| **10.** | Write a Program for Building a Function ISVALID to VALIDATE BST |  |  |  |  |  |
| **11.** | Write a Program to Build BST |  |  |  |  |  |
| **12.** | Write a Program to determine the depth of a given Tree by Implementing MAXDEPTH |  |  |  |  |  |
| **13.** | Write a Program to Understand and implement Tree traversals i.e. Pre-Order Post-Order, In-Order |  |  |  |  |  |
| **14.** | Write a Program to perform Boundary Traversal on BST |  |  |  |  |  |
| **15.** | Write a program for Lowest Common Ancestors |  |  |  |  |  |
| **16.** | Write a Program to verify and validate mirrored trees or not |  |  |  |  |  |
| **17.** | Write a Program for a basic hash function in a programming language of your choice. Demonstrate its usage to store and retrieve key-value pairs. |  |  |  |  |  |
| **18.** | Implement a hash table using separate chaining for collision handling. Perform operations like insertion, deletion, and search on the hash table. |  |  |  |  |  |
| **19.** | Write a Program to Implement Two sums using HASHMAP |  |  |  |  |  |
| **20.** | Write a Program to Implement Search, insert, and Remove in Trie |  |  |  |  |  |
| **21.** | Write a Program to Implement Huffman coding |  |  |  |  |  |
| **22.** | Write a Program to find Distinct substrings in a string |  |  |  |  |  |
| **23.** | Write a Program to find The No of Words in a Trie |  |  |  |  |  |
| **24.** | Write a Program to view a tree from left View |  |  |  |  |  |
| **25** | Write a Program to Traverse a Tree using Level Order Traversal |  |  |  |  |  |

**Q. 1: Write a program for implementing a MINSTACK which should support operations like push, pop, overflow, underflow, display**

a.Construct a stack of N-capacity b.Push elements

c.Pop elements d.Top element

e.Retrieve the min element from the stack

**Solution :**

public class MinStack {

private int[] stack;

private int[] minStack;

private int top;

private int capacity;

public MinStack(int size) {

capacity = size;

stack = new int[capacity];

minStack = new int[capacity];

top = -1;

}

public void push(int x) {

if (isFull()) {

System.out.println("Stack Overflow");

return;

}

stack[++top] = x;

if (top == 0 || x < minStack[top - 1]) {

minStack[top] = x;

} else {

minStack[top] = minStack[top - 1];

}

}

public int pop() {

if (isEmpty()) {

System.out.println("Stack Underflow");

return Integer.MIN\_VALUE;

}

return stack[top--];

}

public boolean isFull() {

return top == capacity - 1;

}

public boolean isEmpty() {

return top == -1;

}

public int top() {

if (isEmpty()) {

System.out.println("Stack is empty");

return Integer.MIN\_VALUE;

}

return stack[top];

}

public int getMin() {

if (isEmpty()) {

System.out.println("Stack is empty");

return Integer.MIN\_VALUE;

}

return minStack[top];

}

public void display() {

if (isEmpty()) {

System.out.println("Stack is empty");

return;

}

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

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

}

System.out.println();

}

public static void main(String[] args) {

MinStack minStack = new MinStack(5);

minStack.push(5);

minStack.push(3);

minStack.push(7);

minStack.push(2);

minStack.display();

System.out.println("Top element: " + minStack.top());

System.out.println("Minimum element: " + minStack.getMin());

minStack.pop();

minStack.display();

System.out.println("Minimum element: " + minStack.getMin());

}

}

**Algorithm :**

* Initialization: Create stack[] and minStack[] with given capacity; set top = -1.
* Push(x):
  + If full, print "Stack Overflow" and return.
  + Insert x into stack[++top].
  + Update minStack[top] with the minimum of x and minStack[top-1] (or x if top == 0).
* Pop():
  + If empty, print "Stack Underflow" and return Integer.MIN\_VALUE.
  + Return and decrease top.
* isFull(): Return top == capacity - 1.
* isEmpty(): Return top == -1.
* top(): Return top element if not empty; else print "Stack is empty" and return Integer.MIN\_VALUE.
* getMin(): Return minimum element (minStack[top]) if not empty; else print "Stack is empty" and return Integer.MIN\_VALUE.
* Display():
  + If empty, print "Stack is empty".
  + Else, iterate from 0 to top, printing each stack[i].

**Output :**

5 3 7 2

Top element: 2

Minimum element: 2

5 3 7

Minimum element: 3

**Q. 2 : Write a program to deal with real-world situations where Stack data structure is widely used Evaluation of expression:**

Stacks are used to evaluate expressions, especially in languages that use postfix or prefix notation. Operators and operands are pushed onto the stack, and operations are performed based on the LIFO principle.

**Solution :**

import java.util.\*;

public class second {

public static int evaluatePostfix(String expression) {

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

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

char c = expression.charAt(i);

if (Character.isDigit(c)) {

stack.push(c - '0');

} else {

int operand2 = stack.pop();

int operand1 = stack.pop();

switch (c) {

case '+':

stack.push(operand1 + operand2);

break;

case '-':

stack.push(operand1 - operand2);

break;

case '\*':

stack.push(operand1 \* operand2);

break;

case '/':

stack.push(operand1 / operand2);

break;

}

}

}

return stack.pop();

}

public static void main(String[] args) {

String expression = "231\*+9-";

System.out.println("Postfix Expression: " + expression);

int result = evaluatePostfix(expression);

System.out.println("Result: " + result);

}

}

# Algorithm :

1. Initialize: Create an empty stack stack to hold integers.
2. Iterate through each character c of the input expression:
3. If c is a digit, push its integer value (c - '0') onto stack.
4. If c is an operator (+, -, \*, /):
5. Pop the top two elements from stack as operand2 and operand1, in that order.
6. Apply the operator c on operand1 and operand2.
7. Push the result back onto stack.
8. After processing all characters, the remaining element in stack is the result.

Return the result by popping it from stack.

# Output :

Postfix Expression: 231\*+9- Result: -4

**Q. 3 : Write a program for finding NGE NEXT GREATER ELEMENT from an array.**

**Solution :**

import java.util.Stack;

public class NextGreaterElement

{

public static void printNGE(int[] arr) { Stack<Integer> stack = new Stack<>(); int[] nge = new int[arr.length];

for (int i = 0; i < arr.length; i++) { nge[i] = -1;

}

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

while (!stack.isEmpty() && arr[stack.peek()] < arr[i]) { nge[stack.pop()] = arr[i];

}

stack.push(i);

}

for (int i = 0; i < arr.length; i++) { System.out.println(arr[i] + " --> " + nge[i]);

}

}

public static void main(String[] args) { int[] arr = {4, 5, 2, 25};

System.out.println("Next Greater Element for each element:"); printNGE(arr);

}

}

# Algorithm ;

1. Initialize:
   * A stack stack to keep track of the indices of elements for which the next greater element (NGE) has not yet been found.
   * An array nge to store the next greater element for each position in the input array arr, initialized with -1 for each element indicating that the NGE has not been found yet.
2. Iterate through each element in the input array arr:
   * For the current element arr[i], repeatedly compare it with the elements corresponding to the indices stored in stack:
   * If arr[i] is greater than arr[stack.peek()] (the element at the index on the top of the stack), it means arr[i] is the NGE for arr[stack.peek()].
   * Pop the index from stack and set arr[i] as the NGE for that index in the nge array.
   * Repeat this process as long as the stack is not empty and arr[i] is greater than the elements corresponding to the indices in the stack.
   * Push the current index i onto the stack to process it in the future steps against the next elements.
3. Print NGE for each element:

Iterate through the arr array, and for each element, print its value and its corresponding NGE from the nge array.

# Output :

Next Greater Element for each element: 4 --> 5

5 --> 25

2 --> 25

25 --> -1

**Q. 4 : Write a program to design a circular queue(k) which Should implement the below functions**

a.Enqueue b.Dequeue c.Front d.Rear

**Solution :**

public class CircularQueue1 { private int[] queue;

private int front; private int rear; private int size; private int capacity;

public CircularQueue1(int k) { capacity = k;

queue = new int[capacity]; front = -1;

rear = -1;

size = 0;

}

public void enqueue(int value) { if (isFull()) {

System.out.println("Queue is full. Cannot enqueue."); return;

}

if (isEmpty())

front = 0;

rear = (rear + 1) % capacity; queue[rear] = value;

size++;

}

public int dequeue() { if (isEmpty()) {

System.out.println("Queue is empty. Cannot dequeue."); return -1;

}

int removedValue = queue[front]; if (front == rear) {

front = -1;

rear = -1;

} else {

front = (front + 1) % capacity;

}

size--;

return removedValue;

}

public int front() { if (isEmpty()) {

System.out.println("Queue is empty."); return -1;

}

return queue[front];

}

public int rear() { if (isEmpty()) {

System.out.println("Queue is empty."); return -1;

}

return queue[rear];

}

public boolean isEmpty() { return size == 0;

}

public boolean isFull() { return size == capacity;

}

public static void main(String[] args) { CircularQueue1 queue = new CircularQueue1(5); queue.enqueue(1);

queue.enqueue(2); queue.enqueue(3); queue.enqueue(4); queue.enqueue(5);

System.out.println("Front element: " + queue.front()); System.out.println("Rear element: " + queue.rear()); System.out.println("Dequeuing: " + queue.dequeue());

System.out.println("Dequeuing: " + queue.dequeue()); System.out.println("Front element: " + queue.front());

System.out.println("Rear element: " + queue.rear()); queue.enqueue(6);

queue.enqueue(7);

System.out.println("Front element: " + queue.front()); System.out.println("Rear element: " + queue.rear());

}

}

# Algorithm :

* 1. Initialization:
     + Allocate an array queue with size k to hold the elements.
     + Initialize front and rear pointers to -1 to indicate the queue is empty.
     + Set size to 0 and capacity to k to track the current number of elements and the maximum queue size, respectively.
  2. Enqueue (value):
     + If the queue is full (size == capacity), print an error message and return.
     + If the queue is empty, set front to 0.
     + Update rear to the next position ((rear + 1) % capacity) to ensure circular motion.
     + Insert value at the rear position in the queue.
     + Increment size.
  3. Dequeue:
     + If the queue is empty, print an error message and return -1.
     + Retrieve the value at front.
     + If front is equal to rear, reset the queue to empty by setting both to -1.
     + Otherwise, update front to the next position ((front + 1) % capacity) for circular motion.
     + Decrement size.
     + Return the dequeued value.
  4. Front:
     + Return the element at the front of the queue. If the queue is empty, print an error message and return -1.
  5. Rear:
     + Return the element at the rear of the queue. If the queue is empty, print an error message and return -1.
  6. isEmpty:
     + Return true if size == 0, indicating the queue is empty; otherwise, return false.
  7. isFull:
     + Return true if size == capacity, indicating the queue is full; otherwise, return false.

# Output :

Front element: 1

Rear element: 5

Dequeuing: 1

Dequeuing: 2

Front element: 3

Rear element: 5

Front element: 3

Rear element: 7

**Q . 5 : Write a Program for an infix expression, and convert it to postfix notation. Use a queue to implement the Shunting Yard Algorithm for expression conversion.**

**Solution :**

import java.util.\*;

public class ShuntingYard {

private static int precedence(char ch) { switch (ch) {

case '+':

case '-':

return 1; case '\*':

case '/':

return 2; case '^':

return 3;

}

return -1;

}

public static String infixToPostfix(String expression) { StringBuilder result = new StringBuilder(); Stack<Character> stack = new Stack<>();

for (int i = 0; i < expression.length(); ++i) { char c = expression.charAt(i);

if (Character.isLetterOrDigit(c)) { result.append(c);

}

else if (c == '(') { stack.push(c);

}

else if (c == ')') {

while (!stack.isEmpty() && stack.peek() != '(') result.append(stack.pop());

if (!stack.isEmpty() && stack.peek() != '(') return "Invalid Expression";

else

stack.pop();

} else {

while (!stack.isEmpty() && precedence(c) <= precedence(stack.peek())) { if (stack.peek() == '(')

return "Invalid Expression"; result.append(stack.pop());

}

stack.push(c);

}

}

while (!stack.isEmpty()) { if (stack.peek() == '(')

return "Invalid Expression"; result.append(stack.pop());

}

return result.toString();

}

public static void main(String[] args) { String expr = "A\*(B+C)/D";

System.out.println("Infix Expression: " + expr); System.out.println("Postfix Expression: " + infixToPostfix(expr));

}

}

# Algorithm ;

1. Create a method named infixToPostfix that takes an infix expression as input and returns the corresponding postfix expression.
2. Initialize an empty StringBuilder called result to store the postfix expression.
3. Initialize an empty Stack called stack to help with the conversion process.
4. Iterate through each character c in the input expression.
5. If c is an alphanumeric character, append it to result.
6. If c is an operator or parenthesis:
   * While the stack is not empty and the precedence of the top operator on the stack is greater than or equal to the precedence of c, pop the top operator from the stack and append it to result.
   * Push c onto the stack.
7. After processing all characters, pop any remaining operators from the stack and append them to result.
8. Return the postfix expression obtained from result.

# Output :

Infix Expression: A\*(B+C)/D Postfix Expression: ABC+\*D/

**Q. 6 : Write a Program for finding the Product of the three largest Distinct Elements. Use a Priority Queue to efficiently find and remove the largest elements.**

**Solution :**

import java.util.PriorityQueue;

public class LargestDistinctProduct {

public static int findLargestDistinctProduct(int[] nums) { if (nums == null || nums.length < 3) {

System.out.println("Array should contain at least three elements."); return -1;

}

PriorityQueue<Integer> maxHeap = new PriorityQueue<>((a, b) -> b - a);

for (int num : nums) { maxHeap.offer(num);

}

int largest1 = maxHeap.poll(); int largest2 = maxHeap.poll(); int largest3 = largest2;

while (!maxHeap.isEmpty() && largest3 == largest2) { largest3 = maxHeap.poll();

}

long product = (long)largest1 \* largest2 \* largest3;

return (int)product;

}

public static void main(String[] args) { int[] nums = {1, 5, 2, 3, 6, 4};

int product = findLargestDistinctProduct(nums);

if (product != -1) {

System.out.println("Product of the three largest distinct elements: " + product);

}

}

}

# Algorithm:

1. Check if the input array nums is valid and contains at least three elements. If not, return an error message or an indicator of invalid input.
2. Use a PriorityQueue (max heap) to keep track of the largest elements, ensuring distinctness by adding elements to a HashSet before the heap.
3. Add all distinct elements from nums to the PriorityQueue.
4. Poll the top three elements from the PriorityQueue to get the three largest distinct elements.
5. Calculate the product of these three elements.
6. Return the product.

# Output :

Product of the three largest distinct elements: 120.

**Q. 7 : Write a Program to Merge two linked lists(sorted).**

**Solution :**

# class Node

# {

# int data;

# Node next;

# Node(int data)

# {

# this.data=data;

# next=null;

# }

# }

# public class MERGELIST

# {

# public static void printlist(Node head)

# {

# Node current=head;

# while(current != null)

# {

# System.out.print(current.data+" ");

# current=current.next;

# }

# System.out.println();

# }

# public static void mergelist(Node list1,Node list2)

# {

# Node list3=new Node(0);

# Node current=list3;

# while(list1 != null && list2 != null)

# {

# if(list1.data<list2.data)

# {

# current.next=list1;

# list1=list1.next;

# }

# else

# {

# current.next=list2;

# list2=list2.next;

# }

# current=current.next;

# }

# current.next=(list1 != null)?list1:list2;

# printlist(list3.next);

# }

# public static void main(String args[])

# {

# Node list1=new Node(2);

# list1.next=new Node(4);

# list1.next.next=new Node(6);

# printlist(list1);

# Node list2=new Node(1);

# list2.next=new Node(3);

# list2.next.next=new Node(5);

# printlist(list2);

# mergelist(list1,list2);

# }

}

# Algorithm:

1. Initialize a dummy node: This acts as a placeholder to simplify operations and helps in returning the head of the merged list easily.
2. Iterate through both lists: Compare the current nodes of both lists, appending the smaller one to the tail of the result list, and advance in the list from which the node was taken.
3. Handle remaining elements: If one list is exhausted before the other, directly link the remainder of the non-exhausted list to the end of the merged list.
4. Return the merged list: The head of the merged list is the next node of the dummy node.

# Output :

Merged Sorted List: 1 1 2 3 4 4.

**Q. 8 : Write a Program to find the Merge point of two linked lists(sorted).**

**Solution :**

class Node {

int data;

Node next;

Node(int data) {

this.data = data;

this.next = null;

}

}

class MergePointLinkedList {

static int findMergePoint(Node head1, Node head2) {

Node current1 = head1;

Node current2 = head2;

while (current1 != null){

while(current2 !=null){

if(current1.data==current2.data){

return current1.data;

}

current2=current2.next;

}

current1=current1.next;

}

return -1; // No merge point found

}

// Helper method to print the linked list

static void printLinkedList(Node head) {

Node current = head;

while (current != null) {

System.out.print(current.data + " ");

current = current.next;

}

System.out.println();

}

public static void main(String[] args) {

// Create linked lists

Node list1 = new Node(1);

list1.next = new Node(2);

list1.next.next = new Node(3);

Node list2 = new Node(5);

list2.next = new Node(3);

list2.next.next = new Node(1);

// Print the linked lists

System.out.println("Linked List 1:");

printLinkedList(list1);

System.out.println("Linked List 2:");

printLinkedList(list2);

// Find the merge point

int mergePoint = findMergePoint(list1, list2);

if (mergePoint != -1) {

System.out.println("Merge Point: " + mergePoint);

} else {

System.out.println("No merge point found.");

}

}

}

# Algorithm:

1. Initialize two pointers, a and b, to the heads of the two lists, head1 and head2, respectively.
2. Traverse the lists with these pointers.
3. When pointer a reaches the end of its list, redirect it to the head of head2. Similarly, when pointer b reaches the end of its list, redirect it to the head of head1.
4. Continue the traversal until the pointers a and b meet. This meeting point is the merge point. If there is no merge point, a and b will eventually both be null.
5. Return the merge point (a or b).

# Output :

Merge Point: 3

**Q. 9 : Write a Program in to Swap Nodes pairwise.**

**Solution :**

# class Node

# {

# int data;

# Node next;

# Node(int data)

# {

# this.data=data;

# next=null;

# }

# }

# class linkedlist

# {

# Node head;

# Node tail;

# linkedlist()

# {

# head=null;

# tail=null;

# }

# 

# void insert(int data)

# {

# Node hello=new Node(data);

# if(head==null)

# {

# head=hello;

# tail=hello;

# System.out.println(data+" is inserted");

# }

# else

# {

# tail.next=hello;

# tail=hello;

# System.out.println(data+" is inserted");

# }

# }

# void swap()

# {

# Node temp=head;

# while(temp!=null )

# {

# int a=temp.data;

# temp.data=temp.next.data;

# temp.next.data=a;

# 

# temp=temp.next.next;

# }

# System.out.println("Swaped");

# }

# void display()

# {

# Node n=head;

# while(n!= null)

# {

# System.out.print(n.data+"->");

# n=n.next;

# }

# }

# }

# public class MAIN

# {

# public static void main(String args[])

# {

# linkedlist list=new linkedlist();

# list.insert(10);

# list.insert(20);

# list.insert(30);

# list.insert(40);

# list.insert(50);

# list.insert(60);

# list.display();

# list.swap();

# list.display();

# }

}

**Algorithm :**

1. Check if the input head is null or has only one node. If so, return the head as there's no need to perform any swaps.
2. Initialize a dummy node and set its next pointer to the head of the list. This dummy node will simplify handling edge cases where the head needs to be swapped.
3. Initialize a prev pointer to the dummy node.
4. Iterate through the list while there are at least two nodes left to swap (head and head.next are not null):
   * Store references to the first (firstNode) and second (secondNode) nodes.
   * Update the next pointers to perform the swap: prev.next points to secondNode, firstNode.next points to secondNode.next, and secondNode.next points to firstNode.
   * Move prev to firstNode.
   * Move head to firstNode.next.
5. Return the head of the modified list.

# Output :

Original List: 10 20 30 40 50 60

List after pairwise swapping: 20 10 40 30 50 60

**Q. 10 : Write a Program for Building a Function ISVALID to VALIDATE BST.**

**Solution :**

class TreeNode { int val; TreeNode left; TreeNode right; TreeNode(int x) {

val = x;

}

}

public class ValidateBST {

public static boolean isValidBST(TreeNode root) {

return isValidBSTHelper(root, Long.MIN\_VALUE, Long.MAX\_VALUE);

}

private static boolean isValidBSTHelper(TreeNode node, long lower, long upper) { if (node == null) {

return true;

}

int val = node.val;

if (val <= lower || val >= upper) { return false;

}

if (!isValidBSTHelper(node.right, val, upper)) { return false;

}

if (!isValidBSTHelper(node.left, lower, val)) { return false;

}

return true;

}

public static void main(String[] args) { TreeNode root = new TreeNode(2); root.left = new TreeNode(1); root.right = new TreeNode(3); boolean isValid = isValidBST(root);

System.out.println("Is the binary tree a valid BST? " + isValid); TreeNode root2 = new TreeNode(5);

root2.left = new TreeNode(1); root2.right = new TreeNode(4); root2.right.left = new TreeNode(3); root2.right.right = new TreeNode(6); isValid = isValidBST(root2);

System.out.println("Is the binary tree a valid BST? " + isValid);

}

}

# Algorithm:

1. Create a method named isValidBST that takes the root of the binary tree as input and returns a boolean indicating whether the tree is a valid BST.
2. Define a helper method named isValidBSTHelper that takes a node, a lower bound (lower), and an upper bound (upper) as input parameters. This method checks if the subtree rooted at node is a valid BST within the given bounds.
3. If node is null, return true because an empty subtree is considered a valid BST.
4. Retrieve the value of the current node (val).
5. Check if the value of the current node (val) is within the bounds (lower and upper). If it's not, return false because the tree violates the BST property.
6. Recursively call isValidBSTHelper for the right subtree (node.right) with the updated lower bound as the value of the current node (val) and the same upper bound.
7. Recursively call isValidBSTHelper for the left subtree (node.left) with the same lower bound and the updated upper bound as the value of the current node (val).
8. If all recursive calls return true, return true to indicate that the subtree rooted at the current node is a valid BST.
9. In the isValidBST method, call isValidBSTHelper with the root node and the minimum and maximum values (Long.MIN\_VALUE and Long.MAX\_VALUE) to validate the entire tree.
10. Return the result obtained from isValidBSTHelper.

**Output :**

Is the binary tree a valid BST? True Is the binary tree a valid BST? False

**Q. 11 : Write a Program to Build BST.**

**Solution :**

# Solution :

# class Node

# {

# int data;

# Node left;

# Node right;

# Node(int data)

# {

# this.data=data;

# left=right=null;

# }

# }

# class BST

# {

# Node root;

# BST(int data)

# {

# root=new Node(data);

# }

# void insert(int data)

# {

# insert(root,data);

# }

# public Node insert(Node root,int data)

# {

# if(root==null)

# {

# return new Node(data);

# }

# if(data<root.data)

# {

# root.left=insert(root.left,data);

# }

# else if(data>root.data)

# {

# root.right=insert(root.right,data);

# }

# return root;

# }

# void inorder(Node root)

# {

# if(root!=null)

# {

# System.out.println(root.left);

# inorder(root.left);

# System.out.print(root.data+" ");

# inorder(root.right);

# }

# }

# }

# public class Main

# {

# public static void main(String args[])

# {

# BST tree=new BST(50);

# tree.insert(tree.root, 20);

# tree.insert(tree.root, 70);

# tree.insert(tree.root, 10);

tree.insert(tree.root, 25);

# tree.insert(tree.root, 60);

# tree.insert(tree.root, 90);

# tree.inorder(tree.root);

# }

# }

# Algorithm:

1. Create a class named TreeNode representing a node in the binary tree. It contains an integer value (val) and references to its left and right child nodes.
2. Create a class named BuildBST to build the BST.
3. Define a method named insert that takes the root of the tree and a value to insert as input parameters. This method inserts a new node with the given value into the BST.
4. If the root is null, create a new node with the given value and return it.
5. If the value is less than the value of the current root node, recursively call insert with the left child of the root and the value to insert. Update the left child of the root with the result of the recursive call.
6. If the value is greater than the value of the current root node, recursively call insert with the right child of the root and the value to insert. Update the right child of the root with the result of the recursive call.
7. Return the root after insertion.
8. Define a method named inorderTraversal to perform an inorder traversal of the BST.
9. If the root is null, return.
10. Recursively call inorderTraversal with the left child of the root.

11. Print the value of the current root node.

12. 1Recursively call inorderTraversal with the right child of the root.

1. In the main method, create an array of values representing the elements to insert into the BST.
2. Initialize the root of the tree to null.
3. Iterate through the array of values and insert each value into the BST using the insert method.
4. Print the inorder traversal of the BST using the inorderTraversal method.

**Output :**

Inorder Traversal of BST:

Node@28a418fc

Node@5305068a

null

10 20 null

25 50 Node@19469ea2

null

60 70 null

90

**Q. 12 : Write a Program to determine the depth of a given Tree by Implementing MAXDEPTH.**

**Solution:**

# class Node

# {

# int data;

# Node left;

# Node right;

# Node(int data)

# {

# this.data=data;

# left=right=null;

# }

# }

# class BST

# {

# Node root;

# 

# BST(int data)

# {

# root=new Node(data);

# }

# void insert(int data)

# {

# insert(root,data);

# }

# public Node insert(Node root,int data)

# {

# if(root==null)

{

# return new Node(data);

# }

# if(data<root.data)

# {

# root.left=insert(root.left,data);

# }

# else if(data>root.data)

# {

# root.right=insert(root.right,data);

# }

# return root;

# }

# void inorder(Node root)

# {

# if(root!=null)

# {

# inorder(root.left);

# System.out.print(root.data+" ");

# inorder(root.right);

# }

# }

# public int depth(Node root)

# {

# if(root==null)

# {

# return -1;

# }

# else

# {

# int leftdepth=depth(root.left)+1;

# int rightdepth=depth(root.right)+1;

# 

# return Math.max(leftdepth,rightdepth);

# }

# }

# }

# public class Main {

# public static void main(String args[])

# {

# BST tree=new BST(50);

# tree.insert(tree.root,20);

# tree.insert(tree.root,10);

# tree.insert(tree.root,6);

# tree.insert(tree.root,5);

# tree.insert(tree.root,70);

# tree.insert(tree.root,80);

# tree.inorder(tree.root);

# System.out.println("TREE HEIGHT " + tree.depth(tree.root));

# }

}

# Algorithm:

1. Define a class named Node to represent a node in the binary tree. Each node contains an integer data, and references to its left and right children (left and right).
2. Define a class named Height to compute the height of the binary tree.
3. Implement a method named maxDepth that takes a node Node as input and returns the maximum depth of the subtree rooted at that node.
4. If the node is null, return 0.
5. Recursively compute the maximum depth of the left subtree and store it in lDepth.
6. Recursively compute the maximum depth of the right subtree and store it in rDepth.
7. Return the maximum of lDepth and rDepth, incremented by 1 to account for the current node.
8. In the main method, create an instance of the Height class.
9. Create the binary tree by instantiating nodes and setting their left and right children.
10. Print the height of the tree using the maxDepth method.

# Output ;

Height of tree is 4.

**Q. 13 : Write a Program to Understand and implement Tree traversals i.e. Pre-Order Post-Order, In-Order.**

**Solution :**

# class Node

# {

# int data;

# Node left;

# Node right;

# Node(int data)

# {

# this.data=data;

# left=right=null;

# }

# }

# class BST

# {

# Node root;

# BST(int data)

# {

# root=new Node(data);

# }

# void insert(int data)

# {

# insert(root,data);

# }

# public Node insert(Node root,int data)

# {

# if(root==null)

# {

# return new Node(data);

# }

# if(data<root.data)

# {

# root.left=insert(root.left,data);

# }

# else if(data>root.data)

# {

# root.right=insert(root.right,data);

# }

# return root;

# }

# void inorder(Node root)

# {

# if(root!=null)

# {

# 

# inorder(root.left);

# System.out.print(root.data+" ");

# inorder(root.right);

# }

# }

# void preorder(Node root)

# {

# if(root!=null)

# {

# System.out.print(root.data+" ");

# preorder(root.left);

# 

# preorder(root.right);

# }

# }

# void postorder(Node root)

# {

# if(root!=null)

# {

# 

# postorder(root.left);

# 

# postorder(root.right);

# System.out.print(root.data+" ");

# }

# }

# 

# } {

# root.right=insert(root.right,data);

# }

# return root;

# }

# void inorder(Node root)

# {

# if(root!=null)

# {

# 

# inorder(root.left);

# System.out.print(root.data+" ");

# inorder(root.right);

# }

# }

# void preorder(Node root)

# {

# if(root!=null)

# {

# System.out.print(root.data+" ");

# preorder(root.left);

# 

# preorder(root.right);

# }

# }

# void postorder(Node root)

# {

# if(root!=null)

# {

# 

# postorder(root.left);

# 

# postorder(root.right);

# System.out.print(root.data+" ");

# }

# }

# 

}}

# Algorithm:

1. Define a class named BinaryTree to represent a node in the binary tree. Each node contains an integer data, and references to its left and right children (left and right).
2. Define a class named Tree with methods for performing different tree traversal operations.
3. Implement methods named inOrderTraversal, preOrderTraversal, and postOrderTraversal, each taking a BinaryTree node as input.
4. In the inOrderTraversal method:
   * Recursively traverse the left subtree.
   * Print the value of the current node.
   * Recursively traverse the right subtree.
5. In the preOrderTraversal method:
   * Print the value of the current node.
   * Recursively traverse the left subtree.
   * Recursively traverse the right subtree.
6. In the postOrderTraversal method:
   * Recursively traverse the left subtree.
   * Recursively traverse the right subtree.
   * Print the value of the current node.
7. In the main method:
   * Create an instance of the BinaryTree class representing the root of the binary tree.
   * Instantiate nodes and set their left and right children to construct the binary tree.
   * Perform in-order, pre-order, and post-order traversals of the binary tree using the respective methods.

# Output ;

In-order Traversal:

10 20 25 50 60 70 90

Pre-order Traversal:

50 20 10 25 70 60 90

Post-order Traversal:

10 25 20 60 90 70 50

**Q. 14 : Write a Program to perform Boundary Traversal on BST.**

**Solution :**

import java.util.\*;

class Node

{

int data;

Node left,right;

Node(int data)

{

this.data=data;

left=right=null;

}

}

class BST

{

Node root;

BST(int data)

{

root=new Node(data);

}

void insert(int data)

{

insert(root,data);

}

Node insert(Node root,int data)

{

if(root==null)

{

return new Node(data);

}

if(data>root.data)

{

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

}

else{

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

}

return root;

}

void inorder(Node root)

{

if(root!=null)

{

inorder(root.left);

System.out.print(root.data+" ");

inorder(root.right);

}

}

void boundarytraversal(Node root)

{

if(root!=null)

{

System.out.print(root.data+" -> ");

leftboundary(root.left);

leaves(root.left);

leaves(root.right);

rightboundary(root.right);

}

}

void leftboundary(Node root)

{

if(root.left!=null)

{

System.out.print(root.data+" -> ");

leftboundary(root.left);

}

}

void leaves(Node root)

{

if(root!=null)

{

leaves(root.left);

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

{

System.out.print(root.data+" -> ");

}

leaves(root.right);

}

}

void rightboundary(Node root)

{

if(root.right!=null)

{

rightboundary(root.right);

System.out.print(root.data+" -> ");

}

}

}

public class Main

{

public static void main(String[] args) {

BST tree=new BST(50);

tree.insert(30);

tree.insert(20);

tree.insert(40);

tree.insert(70);

tree.insert(60);

tree.insert(80);

tree.insert(90);

tree.insert(100);

tree.boundarytraversal(tree.root);

}

}

Algorithm:

TreeNode Class:

• Fields: int val for the node's value, TreeNode left and TreeNode right for the node's left and right children.

• Constructors:

 A default constructor.

 A constructor that accepts an integer value (val) for the node.

 A constructor that accepts an integer value and two TreeNode references to set the current node’s value and its left and right children.

Solution Class :

• Method levelOrder: This method takes the root of a binary tree and performs a level-order traversal to collect the values of the nodes at each level into a list

of lists (List<List<Integer>>).

• It initializes an empty list of lists (ans) to store the result.

• It uses a queue (Queue<TreeNode>) to keep track of nodes to visit next in the traversal.

• Starting with the root node, the method adds nodes to the queue. For each level of the tree, it processes all nodes in the queue (nodes that were at the same depth), adds their values to a temporary list, and enqueues their children.

• After processing all nodes at a given level, it adds the temporary list to the result list.

• This process repeats until there are no more nodes to process (i.e., the queue is empty).

Main Class level:

• In the main method, a binary tree is constructed manually by creating TreeNode instances and linking them as left and right children.

• An instance of the Solution class is created, and the levelOrder method is called with the root of the tree.

• The result of the levelOrder method is a list of lists, where each sublist contains the values of the nodes at a specific depth level of the tree. This result is printed to the console.

**Output :**

50 -> 30 -> 20 -> 40 -> 60 -> 100 -> 90 -> 80 -> 70 ->

**Q. 15 : Write a program for Lowest Common Ancestors.**

**Solution :**

class TreeNode { int val;

TreeNode left, right; TreeNode(int x) {

val = x;

}

}

public class LowestCommonAncestor {

public static TreeNode findLCA(TreeNode root, TreeNode p, TreeNode q) { if (root == null || root == p || root == q) {

return root;

}

TreeNode left = findLCA(root.left, p, q); TreeNode right = findLCA(root.right, p, q); if (left != null && right != null) {

return root;

}

return left != null ? left : right;

}

public static void main(String[] args) { TreeNode root = new TreeNode(3); root.left = new TreeNode(5); root.right = new TreeNode(1); root.left.left = new TreeNode(6); root.left.right = new TreeNode(2); root.right.left = new TreeNode(0); root.right.right = new TreeNode(8);

root.left.right.left = new TreeNode(7); root.left.right.right = new TreeNode(4); TreeNode p = root.left;

TreeNode q = root.left.right.right; TreeNode lca = findLCA(root, p, q);

System.out.println("LCA: " + (lca != null ? lca.val : "None"));

}

}

**Algorithm:**

TreeNode Class:

• Defines a class TreeNode representing a node in the binary tree.

• Each node has an integer value val and references to its left and right children (left and right).

LowestCommonAncestor Class:

• Contains a method findLCA to find the lowest common ancestor of two given nodes p and q in the binary tree.

• The findLCA method:

 Takes the root of the tree and two target nodes p and q as parameters.

 If the root is null or matches either of the target nodes p or q, it returns the root.

 Recursively searches for the target nodes in the left and right subtrees.

 If both target nodes are found in different subtrees (left and right), the current node is the lowest common ancestor, so it returns the root.

 Otherwise, it returns the non-null result from the left or right subtree, whichever is not null.

• In the main method:

 Creates a binary tree by instantiating TreeNode objects and setting their left and right children.

 Defines two target nodes p and q.

 Calls the findLCA method with the root of the tree and the target nodes p and q.

 Prints the value of the lowest common ancestor (or "None" if no common ancestor is found).

**Output :**

LCA: 5

**Q. 16 : Write a Program to verify and validate mirrored trees or not.**

**Solution :**

class Node

{

int data;

Node left;

Node right;

Node(int data)

{

this.data=data;

left=null;

right=null;

}

}

class mirror

{

void inorder(Node root)

{

if(root!=null)

{

inorder(root.left);

System.out.print(root.data+" ");

inorder(root.right);

}

}

boolean mirror(Node root1,Node root2)

{

if(root1==root2)

{

return true;

}

if(root1.data==root2.data)

{

return mirror(root1.left,root2.right)&&mirror(root1.right,root2.left);

}

return false;

}

}

class Main

{

public static void main(String args[])

{

mirror tree=new mirror();

Node tree1=new Node(50);

tree1.left=new Node(60);

tree1.right=new Node(70);

tree1.right.left=new Node(80);

tree1.right.right=new Node(90);

System.out.println("TREE1");

tree.inorder(tree1);

System.out.println();

Node tree2=new Node(50);

tree2.left=new Node(70);

tree2.right=new Node(60);

tree2.left.left=new Node(90);

tree2.left.right=new Node(80);

System.out.println("TREE2");

tree.inorder(tree2);

System.out.println("MIRROR TREE IS"+tree.mirror(tree1, tree2));

}

}

**Algorithm:**

**TreeNode Class:**

**•** Defines a class TreeNode representing a node in the binary tree.

• Each node has an integer value val and references to its left and right children (left and right).

**Solution Class:**

**•** Contains a method isSymmetric to check if a binary tree is symmetric.

• The isSymmetric method:

• Takes the root of the tree as input.

• Calls the helper method, passing both the root's left and right subtrees.

• Returns the result of the helper method, indicating whether the tree is symmetric.

• Contains a private helper method helper that takes two nodes (p and q) as input and recursively checks whether they are symmetric.

• If both nodes are null, they are symmetric, so it returns true.

• If one node is null and the other is not, they are not symmetric, so it returns false.

• If both nodes have different values, they are not symmetric, so it returns false.

• Recursively calls the helper method with the left child of p and the right child of q, and vice versa, to check symmetry for the subtrees.

**symmetric Class:**

**•** In the main method:

 Creates a binary tree by instantiating TreeNode objects and setting their left and right children.

 Creates an instance of the Solution class.

 Calls the isSymmetric method with the root of the tree.

 Prints whether the binary tree is symmetric based on the result returned by the isSymmetric method.

**Output :**

TREE1

60 50 80 70 90

TREE2

90 70 80 50 60

MIRROR TREE IStrue