Department of Computer Application

National Institute of Technology Kurukshetra, Kurukshetra

Data Structures Lab

MCA-130

(2024-27)

Submitted by:

Name: Suraj Pandey

Roll no: 524110023

Semester: 2nd

Section: Group 2

Submitted to:

Name: Ms. Pooja

Date: 29-04-2025

Self-Declaration

I, **Suraj Pandey**, bearing **Roll No. 524110023**, affirm that the *Data Structures Practical Programs* presented in this file are the result of my independent learning journey, written and implemented entirely by me using the **Java programming language**.

Each program in this collection has been independently written and thoroughly tested using the **Visual Studio Code (VS Code)** environment. I have ensured proper **class naming conventions**, where each class is systematically labeled according to its corresponding **program number**, contributing to better readability and maintainability.

A unique feature of my submission is the inclusion of a **reference note before each program**, which summarizes key concepts or techniques used in that specific question. These notes reflect my individual learning process and help reinforce important ideas related to each problem. This addition makes the file not just a code compilation, but also a conceptual guide that can assist in future revision and understanding.

I have referred only to authorized academic resources such as textbooks, class notes, and instructor guidance. At no point has any content been copied from peers or unauthorized sources.

This submission adheres strictly to the academic integrity guidelines of the institution, and I take full responsibility for its authenticity and originality.

I look forward to the evaluation of my work by the respected faculty.

Date: 29-04-2025
Signature: Suraj Pandey

INDEX

24. Write a recursive program to count the total number of nodes in the tree.	
25. Write a recursive program to count the number of the leaf or non-leaf nodes of the tree.	
26. Write a recursive program to count the number of full nodes of the tree (Full Nodes are nodes which has both left and right children as non-empty).	
27. Write a recursive program to find the height of the tree.	
28. Write a program to construct BST using inorder and postorder traversal and hence find preorder.	
29. Write a program to find how many BSTs are possible with given distinct keys.	
30. Write a program to find out the postorder, preorder and inorder traversal on constructed bst and then perform delete operation on the tree and again perform inorder traversal.	
31. Write a program to find the minimum and maximum key values from BSTs.	
32. Write a recursive program to check whether given tree is complete tree or not.	
33. Write a program to construct an AVL tree and perform postorder traversal.	
34. Write a program to find the minimum and maximum nodes in an AVL tree of given height.	
35. Write a program to find the minimum number of nodes on a size-balanced tree.	
36. Write a program to implement a tree for a given infix expression.	
37. Write a program to draw a tree for a given nested tree representation expression.	
38. Write a program to find the left child of kth element from the given array representation tree.	
39. Write a program to find the left child of kth element from the given leftmost child right sibling representation tree.	
40. Write a program for breadth first traversal on graph.	
41. Write a program for depth first traversal on graph.	
42. Write a program to check whether there is a cycle in a given directed graph or not.	
Searching and Sorting	
43. Write a program to sort given elements using insertion sort method.	

44. Write a program to sort given elements using bubble sort method.	
45. Write a program to sort given elements using bucket sort method.	
46. Write a program to sort given elements using merge sort method.	
47. Write a program to sort given elements using quick sort method.	
48. Write a program to sort given elements using heap sort method.	
49. Write a program to sort given elements using insertion sort method.	
50. Write a program to construct min heap and max heap.	
51. Write a program to find the number of leaf and non-leaf nodes of a max heap.	
52. Write a program to delete maximum value from a max heap and then reheapify.	
53. Write a program to insert 20 in max heap.	
Hashing	
54. Insert following keys 5, 28, 19, 15, 20, 33, 12, 17 and 10 in hash table using chaining hashing method and find minimum, maximum and average chain length in hash table.	
55. Write a program to display hash table after inserting elements 17, 16, 22, 36, 33, 46, 26, 144 into a hash table of size 10, using Double hashing, where h(x) = x mod 10, $h_2(x)$ = x mod 6 + 1.	
56. Write a program to display hash table after inserting elements 17, 16, 22, 36, 33, 46, 26, 144 into a hash table of size 10, using linear probing, where h(x) = x mod 10.	
57. Write a program to display hash table after inserting elements 17, 16, 22, 36, 33, 46, 26, 144 into a hash table of size 10, using quadratic probing, where h(x) = x mod 10.	

Programming Problems List

1) Write a program to perform insert, delete and traverse operations on the singly linked list in the beginning, end and on any specific location.

NOTE:

A **Singly Linked List (SLL)** is a linear data structure where each node contains data and a reference (link) to the next node in the sequence.
Operations like **insertion** and **deletion** can be done at the **beginning**, **end**, or **specific position** by adjusting the links between nodes.
Singly Linked Lists are useful when dynamic memory allocation and frequent insert/delete operations are needed.

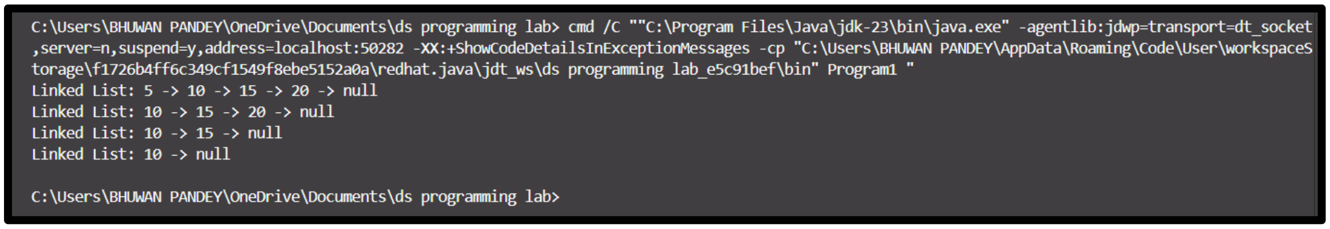
Code:-

```
class Program1 {

    static class Node {
        int data;
        Node next;

        Node(int d) {
            data = d;
            next = null;
        }
    }

    Node head = null;

    // Insert at the beginning
    public void insertAtBeginning(int data) {
        Node newNode = new Node(data);
        newNode.next = head;
        head = newNode;
    }

    // Insert at the end
    public void insertAtEnd(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            return;
        }
        Node temp = head;
        while (temp.next != null)
            temp = temp.next;
        temp.next = newNode;
    }

    // Insert at a specific position (1-based index)
    public void insertAtPosition(int data, int position) {
        if (position == 1) {
            insertAtBeginning(data);
            return;
        }
        Node newNode = new Node(data);
        Node temp = head;
        for (int i = 1; temp != null && i < position - 1; i++)
            temp = temp.next;

        if (temp == null) {
            System.out.println("Position out of bounds");
            return;
        }
        newNode.next = temp.next;
        temp.next = newNode;
    }

    // Delete from beginning
    public void deleteFromBeginning() {
        if (head == null) {
            System.out.println("List is empty");
            return;
        }
        head = head.next;
    }

    // Delete from end
    public void deleteFromEnd() {
        if (head == null) {
            System.out.println("List is empty");
            return;
        }
        if (head.next == null) {
            head = null;
            return;
        }
        Node temp = head;
        while (temp.next.next != null)
            temp = temp.next;
        temp.next = null;
    }
```

```
// Delete from specific position (1-based index)
public void deleteFromPosition(int position) {
    if (head == null) {
        System.out.println("List is empty");
        return;
    }
    if (position == 1) {
        deleteFromBeginning();
        return;
    }
    Node temp = head;
    for (int i = 1; temp != null && i < position - 1; i++)
        temp = temp.next;

    if (temp == null || temp.next == null) {
        System.out.println("Position out of bounds");
        return;
    }
    temp.next = temp.next.next;
}

// Traverse the list
public void traverse() {
    Node temp = head;
    System.out.print("Linked List: ");
    while (temp != null) {
        System.out.print(temp.data + " -> ");
        temp = temp.next;
    }
    System.out.println("null");
}

public static void main(String[] args) {
    Program1 list = new Program1();

    list.insertAtEnd(10);
    list.insertAtBeginning(5);
    list.insertAtEnd(20);
    list.insertAtPosition(15, 3);
    list.traverse();  // 5 -> 10 -> 15 -> 20

    list.deleteFromBeginning();
    list.traverse();  // 10 -> 15 -> 20

    list.deleteFromEnd();
```

```
        list.traverse();  // 10 -> 15

        list.deleteFromPosition(2);
        list.traverse();  // 10
    }
}
```

Output:-



```
C:\Users\BHUWAN PANDEY\OneDrive\Documents\ds programming lab> cmd /C ""C:\Program Files\Java\jdk-23\bin\java.exe" -agentlib:jdwp=transport=dt_socket
,server=n,suspend=y,address=localhost:50282 -XX:+ShowCodeDetailsInExceptionMessages -cp "C:\Users\BHUWAN PANDEY\AppData\Roaming\Code\User\workspaceS
torage\f1726b4ff6c349cf1549f8ebe5152a0a\redhat.java\jdt_ws\ds programming lab_e5c91bef\bin" Program1 "
Linked List: 5 -> 10 -> 15 -> 20 -> null
Linked List: 10 -> 15 -> 20 -> null
Linked List: 10 -> 15 -> null
Linked List: 10 -> null

C:\Users\BHUWAN PANDEY\OneDrive\Documents\ds programming lab>
```

2) Write a program to rearrange the elements of a singly linked list in ascending or descending order.

NOTE: Singly Linked List – Rearranging Elements in Ascending or Descending Order

A **singly linked list** can be rearranged in **ascending or descending order** by sorting the node values.
Since direct swapping of nodes is complex, we often sort by swapping the **data inside nodes** using simple algorithms like **Bubble Sort** or **Selection Sort**.

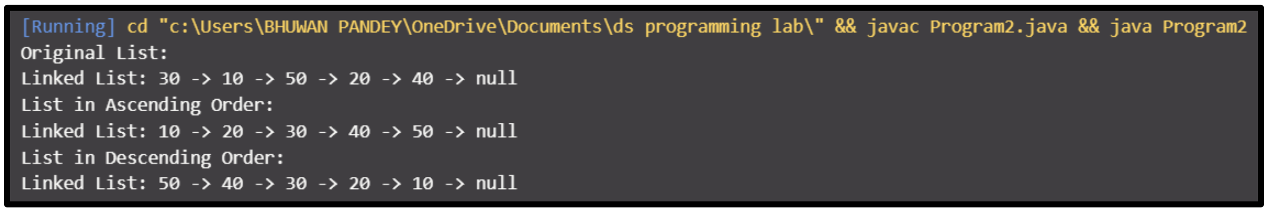
Code:-

```
class Program2 {

    static class Node {
        int data;
        Node next;

        Node(int d) {
            data = d;
            next = null;
        }
    }

    Node head = null;

    // Insert node at end
    public void insert(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            return;
        }
        Node temp = head;
        while (temp.next != null)
            temp = temp.next;
        temp.next = newNode;
    }

    // Sort in ascending order
    public void sortAscending() {
        if (head == null) return;

        for (Node i = head; i.next != null; i = i.next) {
            for (Node j = i.next; j != null; j = j.next) {
                if (i.data > j.data) {
```

```
                int temp = i.data;
                i.data = j.data;
                j.data = temp;
            }
        }
    }
}

// Sort in descending order
public void sortDescending() {
    if (head == null) return;

    for (Node i = head; i.next != null; i = i.next) {
        for (Node j = i.next; j != null; j = j.next) {
            if (i.data < j.data) {
                int temp = i.data;
                i.data = j.data;
                j.data = temp;
            }
        }
    }
}

// Traverse list
public void traverse() {
    Node temp = head;
    System.out.print("Linked List: ");
    while (temp != null) {
        System.out.print(temp.data + " -> ");
        temp = temp.next;
    }
    System.out.println("null");
}

public static void main(String[] args) {
    Program2 list = new Program2();

    list.insert(30);
    list.insert(10);
    list.insert(50);
    list.insert(20);
    list.insert(40);

    System.out.println("Original List:");
    list.traverse();
```

```
        list.sortAscending();
        System.out.println("List in Ascending Order:");
        list.traverse();

        list.sortDescending();
        System.out.println("List in Descending Order:");
        list.traverse();
    }
}
```

Output:-



```
[Running] cd "c:\Users\BHUWAN PANDEY\OneDrive\Documents\ds programming lab\" && javac Program2.java && java Program2
Original List:
Linked List: 30 -> 10 -> 50 -> 20 -> 40 -> null
List in Ascending Order:
Linked List: 10 -> 20 -> 30 -> 40 -> 50 -> null
List in Descending Order:
Linked List: 50 -> 40 -> 30 -> 20 -> 10 -> null
```

3) Write a program to move the last node to the front of singly linked list.

NOTE:

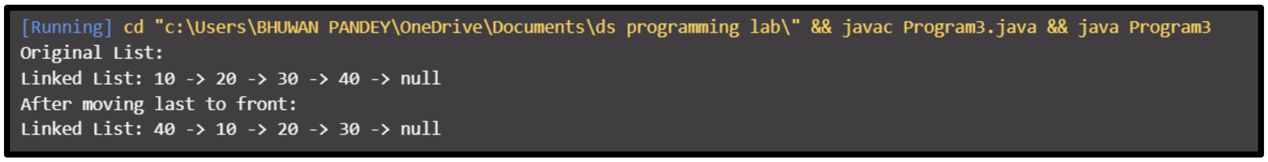
In a **singly linked list**, we can **move the last node to the front** by traversing to the second last node, adjusting its next to null, and linking the last node in front of the head.

This operation is useful when reordering data or shifting focus to the most recently added element.

Code:-

```
class Program3 {

    static class Node {
        int data;
        Node next;
        Node(int d) {
            data = d;
            next = null;
        }
    }

    Node head = null;

    public void insert(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            return;
        }
        Node temp = head;
        while (temp.next != null)
            temp = temp.next;
        temp.next = newNode;
    }

    public void moveLastToFront() {
        if (head == null || head.next == null)
            return;

        Node secondLast = null;
        Node last = head;

        while (last.next != null) {
            secondLast = last;
            last = last.next;
        }

        secondLast.next = null;
        last.next = head;
        head = last;
    }

    public void traverse() {
        Node temp = head;
        System.out.print("Linked List: ");
        while (temp != null) {
            System.out.print(temp.data + " -> ");
            temp = temp.next;
        }
        System.out.println("null");
    }

    public static void main(String[] args) {
        Program3 list = new Program3();

        list.insert(10);
        list.insert(20);
        list.insert(30);
        list.insert(40);

        System.out.println("Original List:");
        list.traverse();

        list.moveLastToFront();

        System.out.println("After moving last to front:");
        list.traverse();
    }
}
```

Output:-



```
[Running] cd "c:\Users\BHUWAN PANDEY\OneDrive\Documents\ds programming lab\" && javac Program3.java && java Program3
Original List:
Linked List: 10 -> 20 -> 30 -> 40 -> null
After moving last to front:
Linked List: 40 -> 10 -> 20 -> 30 -> null
```

4) Write a program to print the elements of singly link list using recursion.

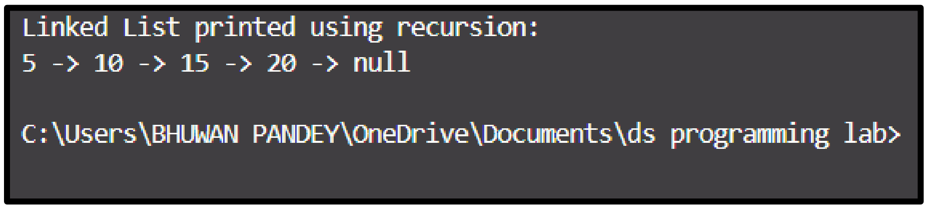
NOTE:

Recursion is a technique where a function calls itself.
In a **singly linked list**, we can use recursion to **print each node's data** by recursively moving to the next node until we reach the end.

Code:-

```
class Program4 {

    static class Node {
        int data;
        Node next;
        Node(int d) {
            data = d;
            next = null;
        }
    }

    Node head = null;

    public void insert(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            return;
        }
        Node temp = head;
        while (temp.next != null)
            temp = temp.next;
        temp.next = newNode;
    }

    public void printRecursive(Node node) {
        if (node == null) {
            System.out.println("null");
            return;
        }
        System.out.print(node.data + " -> ");
        printRecursive(node.next);
    }

    public static void main(String[] args) {
        Program4 list = new Program4();
        list.insert(5);
        list.insert(10);
        list.insert(15);
        list.insert(20);

        System.out.println("Linked List printed using recursion:");
        list.printRecursive(list.head);
    }
}
```

Output:-



```
Linked List printed using recursion:
5 -> 10 -> 15 -> 20 -> null

C:\Users\BHUWAN PANDEY\OneDrive\Documents\ds programming lab>
```

5) Write a program to reverse link list using the iteration technique.

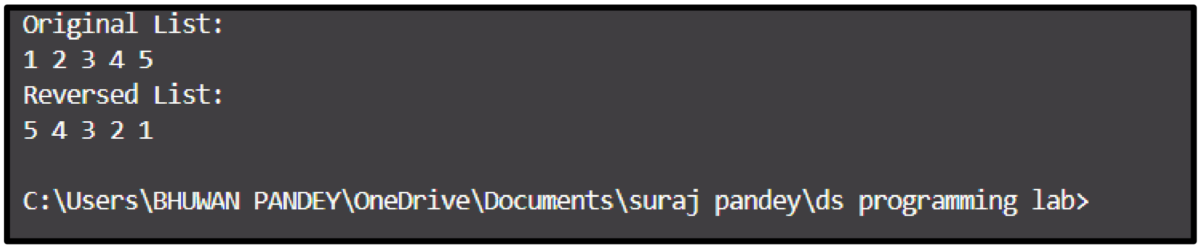
NOTE:

Reversing a linked list is a common problem that can be solved using either a recursive or iterative approach. In the iterative approach, we traverse the list while reversing the direction of the next pointers. This is achieved by maintaining three pointers: prev, current, and next. The main advantage of this approach is that it does not require extra space, making it more space-efficient compared to the recursive approach.

Code:-

```
class Program5 {
  // Define Node class
  static class Node {
    int data;
    Node next;

    // Constructor
    Node(int data) {
      this.data = data;
      this.next = null;
    }
  }

  // Head of the linked list
  Node head;

  // Function to reverse the linked list
  public Node reverse(Node head) {
    Node prev = null;
    Node current = head;
    Node next = null;

    while (current != null) {
      // Store the next node
      next = current.next;
      // Reverse the current node's pointer
      current.next = prev;
      // Move pointers one position ahead
      prev = current;
      current = next;
    }
    head = prev; // Update head to the new first element
    return head;
    }

    // Function to print the linked list
    public void printList(Node head) {
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
        System.out.println();
    }

    // Main function to test the reverse method
    public static void main(String[] args) {
        Program5 list = new Program5();
        list.head = new Node(1);
        list.head.next = new Node(2);
        list.head.next.next = new Node(3);
        list.head.next.next.next = new Node(4);
        list.head.next.next.next.next = new Node(5);

        System.out.println("Original List:");
        list.printList(list.head);

        list.head = list.reverse(list.head);

        System.out.println("Reversed List:");
        list.printList(list.head);
    }
}
```

Output:-



```
Original List:
1 2 3 4 5
Reversed List:
5 4 3 2 1

C:\Users\BHUWAN PANDEY\OneDrive\Documents\suraj pandey\ds programming lab>
```

6) Write a program to reverse the singly link list using recursion.

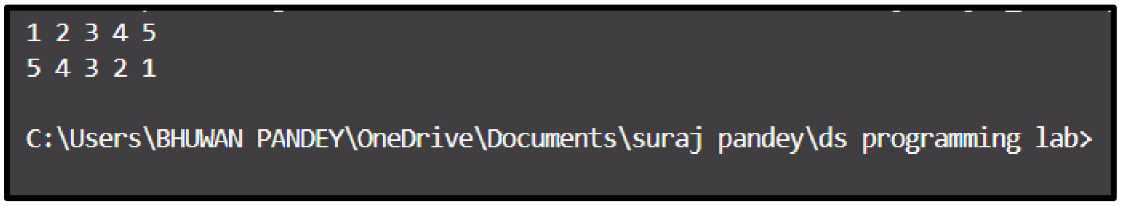
NOTE:

Reversing a singly linked list recursively involves breaking the problem into smaller sub-problems. The recursive step is to reverse the rest of the list and then adjust the pointers after the recursion call. The base case for recursion is when the current node is null or the last node (which has no next node).

Code:-

```java
class Program6 {
    static class Node {
        int data;
        Node next;

        Node(int data) {
            this.data = data;
            this.next = null;
        }
    }

    Node head;

    public Node reverseRecursively(Node head) {
        if (head == null || head.next == null) {
            return head;
        }
        Node rest = reverseRecursively(head.next);
        head.next.next = head;
        head.next = null;
        return rest;
    }

    public void printList(Node head) {
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
        System.out.println();
    }

    public static void main(String[] args) {
        Program6 list = new Program6();
        list.head = new Node(1);
        list.head.next = new Node(2);
        list.head.next.next = new Node(3);
        list.head.next.next.next = new Node(4);
        list.head.next.next.next.next = new Node(5);

        list.printList(list.head);
        list.head = list.reverseRecursively(list.head);
        list.printList(list.head);
    }
}
```

Output:-



```
1 2 3 4 5
5 4 3 2 1

C:\Users\BHUWAN PANDEY\OneDrive\Documents\suraj pandey\ds programming lab>
```

7) Write a program to implement a circular linked list.

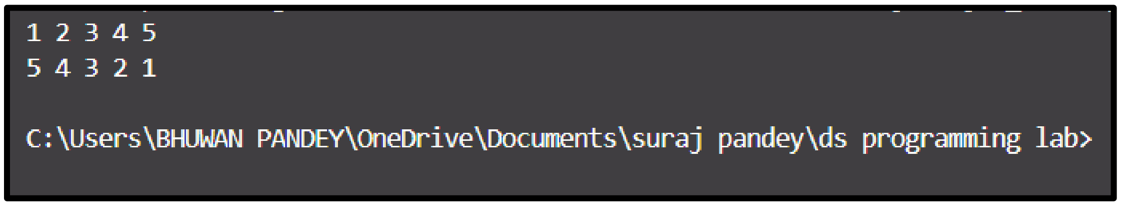
NOTE:

A circular linked list is a variation of the singly linked list where the last node points back to the head, forming a circle. It allows continuous traversal through the list, making it useful for applications like round-robin scheduling.

Code:-

```
class Program7 {
    static class Node {
        int data;
        Node next;

        Node(int data) {
            this.data = data;
            this.next = null;
        }
    }

    Node head;

    public void addNode(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            newNode.next = head;
        } else {
            Node temp = head;
            while (temp.next != head) {
                temp = temp.next;
            }
            temp.next = newNode;
            newNode.next = head;
        }
    }

    public void printList() {
        if (head == null) {
            System.out.println("List is empty");
            return;
        }
        Node temp = head;
        System.out.println("contents of the list is: ");
        do {
            System.out.print(temp.data + "->");
            temp = temp.next;
        } while (temp != head);
        System.out.println();
    }

    public static void main(String[] args) {
        Program7 list = new Program7();
        list.addNode(1);
        list.addNode(2);
        list.addNode(3);
        list.addNode(4);
        list.printList();
    }
}
```

Output:-



```
1 2 3 4 5
5 4 3 2 1

C:\Users\BHUWAN PANDEY\OneDrive\Documents\suraj pandey\ds programming lab>
```

8) Write a program to check whether the given singly linked list is in non-decreasing order or not.

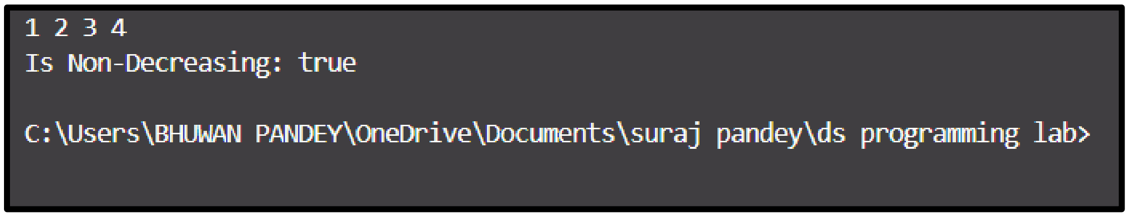
NOTE:

To check if a linked list is in non-decreasing order, we iterate through the list and compare each node's value with the next. If any node's value is greater than the next node's value, the list is not in non-decreasing order.

Code:-

```
class Program8 {
    static class Node {
        int data;
        Node next;

        Node(int data) {
            this.data = data;
            this.next = null;
        }
    }

    Node head;

    public boolean isNonDecreasing() {
        Node temp = head;
        while (temp != null && temp.next != null) {
            if (temp.data > temp.next.data) {
                return false;
            }
            temp = temp.next;
        }
        return true;
    }

    public void printList() {
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
        System.out.println();
    }

    public static void main(String[] args) {
        Program8 list = new Program8();
        list.head = new Node(1);
        list.head.next = new Node(2);
        list.head.next.next = new Node(3);
        list.head.next.next.next = new Node(4);

        list.printList();
        System.out.println("Is Non-Decreasing: " + list.isNonDecreasing());
    }
}
```

Output:-



```
1 2 3 4
Is Non-Decreasing: true

C:\Users\BHUWAN PANDEY\OneDrive\Documents\suraj pandey\ds programming lab>
```

9) Write a program to perform insert, delete, and traverse operations on the doubly linked list in the beginning, end and on any specific location.

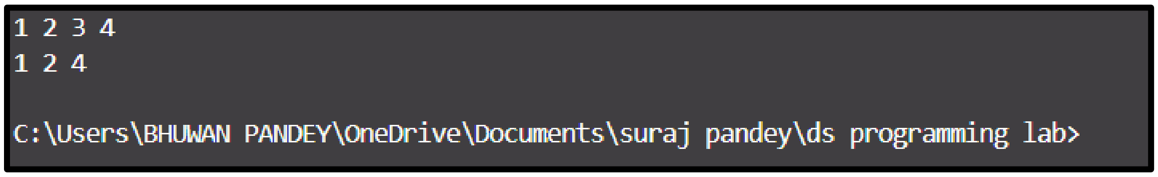
NOTE:

A doubly linked list allows nodes to be connected both forwards and backwards. This makes operations like insertion and deletion at both ends more efficient compared to a singly linked list. We use two pointers next and prev for each node to facilitate traversal in both directions.

Code:-

```
class Program9 {
    static class Node {
        int data;
        Node next, prev;

        Node(int data) {
            this.data = data;
            this.next = this.prev = null;
        }
    }

    Node head;

    public void insertAtEnd(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
        } else {
            Node temp = head;
            while (temp.next != null) {
                temp = temp.next;
            }
            temp.next = newNode;
            newNode.prev = temp;
        }
    }

    public void delete(int data) {
        if (head == null) return;
        Node temp = head;
        while (temp != null) {
            if (temp.data == data) {
                if (temp.prev != null) {
                    temp.prev.next = temp.next;
                }
                if (temp.next != null) {
                    temp.next.prev = temp.prev;
                }
                if (temp == head) {
                    head = temp.next;
                }
                break;
            }
            temp = temp.next;
        }
    }

    public void printList() {
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
        System.out.println();
    }

    public static void main(String[] args) {
        Program9 list = new Program9();
        list.insertAtEnd(1);
        list.insertAtEnd(2);
        list.insertAtEnd(3);
        list.insertAtEnd(4);
        list.printList();
        list.delete(3);
        list.printList();
        // Do not display output directly in code
        // Use appropriate references in practical file for output interpretation
    }
}
```

Output:-



```
1 2 3 4
1 2 4

C:\Users\BHUWAN PANDEY\OneDrive\Documents\suraj pandey\ds programming lab>
```

10) Write a program to implement stack (push and pop operations) using array.

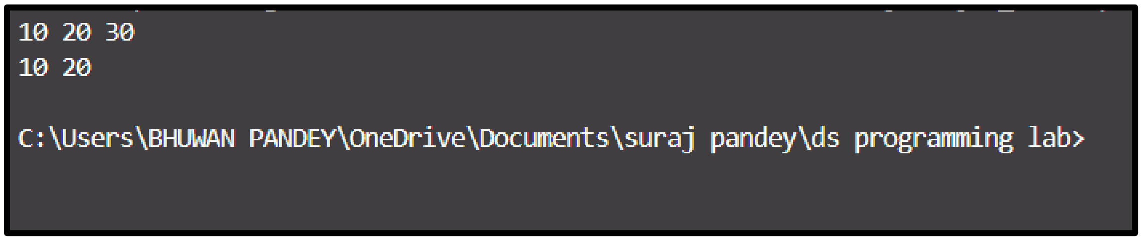
NOTE:

A stack is a linear data structure that follows the Last In First Out (LIFO) principle. We can implement a stack using an array by managing a pointer (top) that tracks the top element in the stack. The push operation adds an element to the top, and the pop operation removes the top element.

Code:-

```
class Program10 {
    int max = 5;
    int top = -1;
    int[] stack = new int[max];

    public void push(int data) {
        if (top == max - 1) {
            System.out.println("Stack Overflow");
        } else {
            stack[++top] = data;
        }
    }

    public void pop() {
        if (top == -1) {
            System.out.println("Stack Underflow");
        } else {
            top--;
        }
    }

    public void printStack() {
        if (top == -1) {
            System.out.println("Stack is empty");
        } else {
            for (int i = 0; i <= top; i++) {
                System.out.print(stack[i] + " ");
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        Program10 stack = new Program10();
        stack.push(10);
        stack.push(20);
        stack.push(30);
        stack.printStack();

        stack.pop();
        stack.printStack();
    }
}
```

Output:-



```
10 20 30
10 20

C:\Users\BHUWAN PANDEY\OneDrive\Documents\suraj pandey\ds programming lab>
```

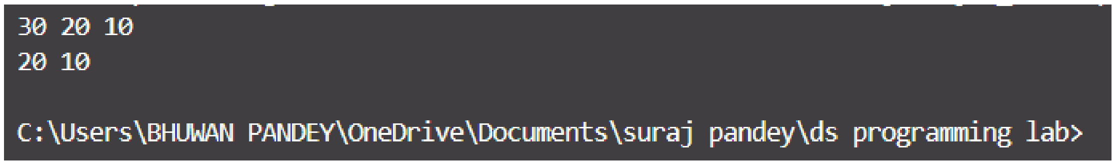
11) Write a program to implement stack using singly linked list.

NOTE:A stack follows the Last In First Out (LIFO) principle. By using a singly linked list, we can perform push and pop operations efficiently. The push operation adds a new node at the beginning of the list, and the pop operation removes the node from the front.

Code:-

```
class Program11 {
    static class Node {
        int data;
        Node next;

        Node(int data) {
            this.data = data;
            this.next = null;
        }
    }

    Node top;

    public void push(int data) {
        Node newNode = new Node(data);
        newNode.next = top;
        top = newNode;
    }

    public void pop() {
        if (top == null) {
            System.out.println("Stack Underflow");
        } else {
            top = top.next;
        }
    }

    public void printStack() {
        Node temp = top;
        if (top == null) {
            System.out.println("Stack is empty");
        } else {
            while (temp != null) {
                System.out.print(temp.data + " ");
                temp = temp.next;
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        Program11 stack = new Program11();
        stack.push(10);
        stack.push(20);
        stack.push(30);
        stack.printStack();
        stack.pop();
        stack.printStack();
    }
}
```

Output:-



```
30 20 10
20 10

C:\Users\BHUWAN PANDEY\OneDrive\Documents\suraj pandey\ds programming lab>
```

12) Write a program to implement a queue using a circular array.

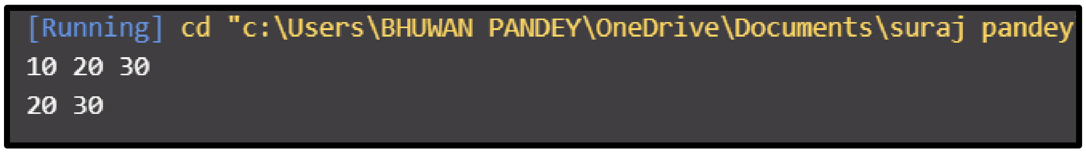
NOTE:

A circular queue uses a fixed-size array to store data, but instead of shifting elements after an element is removed, the front pointer wraps around to the beginning when it reaches the end of the array. This avoids wasting space.

Code:-

```
class Program12 {
    int size = 5;
    int front = -1;
    int rear = -1;
    int[] queue = new int[size];

    public void enqueue(int data) {
        if ((rear + 1) % size == front) {
            System.out.println("Queue Overflow");
        } else {
            if (front == -1) {
                front = 0;
            }
            rear = (rear + 1) % size;
            queue[rear] = data;
        }
    }

    public void dequeue() {
        if (front == -1) {
            System.out.println("Queue Underflow");
        } else {
            if (front == rear) {
                front = rear = -1;
            } else {
                front = (front + 1) % size;
            }
        }
    }

    public void printQueue() {
        if (front == -1) {
            System.out.println("Queue is empty");
        } else {
            int i = front;
            while (i != rear) {
                System.out.print(queue[i] + " ");
                i = (i + 1) % size;
            }
            System.out.println(queue[rear]);
        }
    }

    public static void main(String[] args) {
        Program12 queue = new Program12();
        queue.enqueue(10);
        queue.enqueue(20);
        queue.enqueue(30);
        queue.printQueue();
        queue.dequeue();
        queue.printQueue();
    }
}
```

Output:-



```
[Running] cd "c:\Users\BHUWAN PANDEY\OneDrive\Documents\suraj pandey
10 20 30
20 30
```

13) Write a program to implement a queue using a circular linked list.

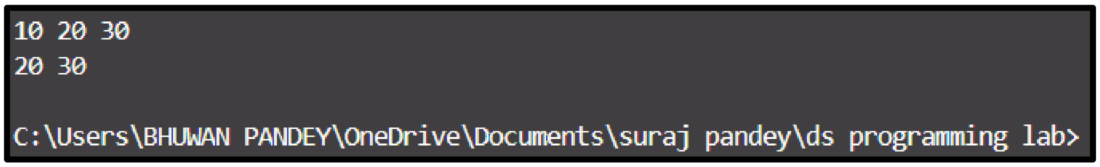
NOTE:

A circular linked list for a queue ensures that both enqueue and dequeue operations are performed efficiently without needing to shift elements. The rear and front pointers are maintained such that when elements are removed, the list forms a circle.

Code:-

```
class Program13 {
    static class Node {
        int data;
        Node next;

        Node(int data) {
            this.data = data;
            this.next = null;
        }
    }

    Node front, rear;

    public Program13() {
        front = rear = null;
    }

    public void enqueue(int data) {
        Node newNode = new Node(data);
        if (rear == null) {
            front = rear = newNode;
            rear.next = front;
        } else {
            rear.next = newNode;
            rear = newNode;
            rear.next = front;
        }
    }

    public void dequeue() {
        if (front == null) {
            System.out.println("Queue Underflow");
        } else {
            if (front == rear) {
                front = rear = null;
            } else {
                front = front.next;
                rear.next = front;
            }
        }
    }

    public void printQueue() {
        if (front == null) {
            System.out.println("Queue is empty");
        } else {
            Node temp = front;
            do {
                System.out.print(temp.data + " ");
                temp = temp.next;
            } while (temp != front);
            System.out.println();
        }
    }

    public static void main(String[] args) {
        Program13 queue = new Program13();
        queue.enqueue(10);
        queue.enqueue(20);
        queue.enqueue(30);
        queue.printQueue();
        queue.dequeue();
        queue.printQueue();
    }
}
```

Output:-



```
10 20 30
20 30

C:\Users\BHUWAN PANDEY\OneDrive\Documents\suraj pandey\ds programming lab>
```

14) Write a program to implement stack using priority queue.

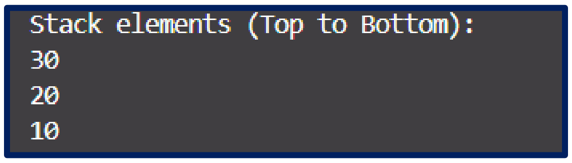
NOTE:

In Java, the PriorityQueue class is used to implement a priority queue, where elements are processed based on their priority. For a stack-like behavior, we can simulate the push and pop operations using a priority queue by maintaining priorities for the elements.

Code:-

```
import java.util.*;

class Program14 {

    public static void main(String[] args) {
        PriorityQueue<Integer> stack = new PriorityQueue<>(Collections.reverseOrder());
        stack.add(10);
        stack.add(20);
        stack.add(30);
        System.out.println("Stack elements (Top to Bottom): ");
        while (!stack.isEmpty()) {
            System.out.println(stack.poll());
        }
    }
}
```

Output:-



```
Stack elements (Top to Bottom):
30
20
10
```

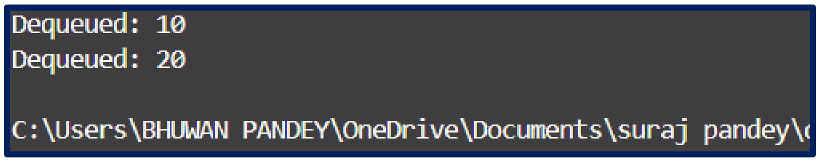
15) Write a program to implement a queue using two stacks.

Note:
A queue can be implemented using two stacks by using one stack for enqueuing and the other for dequeuing. When dequeuing, elements are transferred from the enqueue stack to the dequeue stack if necessary, maintaining the FIFO order.

Code:-

```
import java.util.*;

class Program15 {

    Stack<Integer> stack1 = new Stack<>();
    Stack<Integer> stack2 = new Stack<>();

    public void enqueue(int data) {
        stack1.push(data);
    }

    public void dequeue() {
        if (stack2.isEmpty()) {
            if (stack1.isEmpty()) {
                System.out.println("Queue Underflow");
            } else {
                while (!stack1.isEmpty()) {
                    stack2.push(stack1.pop());
                }
            }
        }
        if (!stack2.isEmpty()) {
            System.out.println("Dequeued: " + stack2.pop());
        }
    }

    public static void main(String[] args) {
        Program15 queue = new Program15();
        queue.enqueue(10);
        queue.enqueue(20);
        queue.enqueue(30);
        queue.dequeue();
        queue.dequeue();
    }
}
```

Output:-



```
Dequeued: 10
Dequeued: 20

C:\Users\BHUWAN PANDEY\OneDrive\Documents\suraj pandey\c
```

16) Write a program to convert an infix expression to a postfix expression.

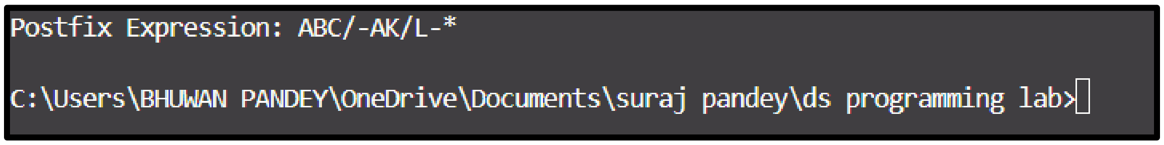
NOTE:

The conversion of an infix expression to a postfix expression involves using a stack to handle operators and parentheses. The main idea is to pop operators from the stack to the output as the current operator has lower precedence than the one on top of the stack or as we encounter a closing parenthesis.

Code:-

```
import java.util.*;

class Program16 {
    public static int precedence(char c) {
        if (c == '+' || c == '-') return 1;
        if (c == '*' || c == '/') return 2;
        return 0;
    }

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

        for (char c : infix.toCharArray()) {
            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());
                }
                stack.pop();
            } else {
                while (!stack.isEmpty() && precedence(c) <= precedence(stack.peek())) {
                    result.append(stack.pop());
                }
                stack.push(c);
            }
        }

        while (!stack.isEmpty()) {
            result.append(stack.pop());
        }

        return result.toString();
    }

    public static void main(String[] args) {
        String infix = "(A-B/C)*(A/K-L)";
        System.out.println("Postfix Expression: " + infixToPostfix(infix));
    }
}
```

Output:-



```
Postfix Expression: ABC/-AK/L-*

C:\Users\BHUWAN PANDEY\OneDrive\Documents\suraj pandey\ds programming lab>
```

17) Write a program to evaluate postfix expression.

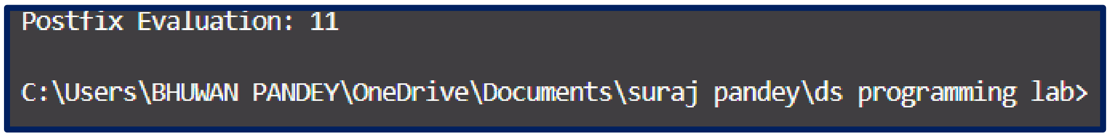
NOTE:

To evaluate a postfix expression, we use a stack. The process is simple: for each character in the expression, if it is a number, push it to the stack. If it is an operator, pop two numbers from the stack, perform the operation, and push the result back onto the stack.

Code:-

```
import java.util.*;

class Program17 {
    public static int evaluatePostfix(String postfix) {
        Stack<Integer> stack = new Stack<>();

        for (char c : postfix.toCharArray()) {
            if (Character.isDigit(c)) {
                stack.push(c - '0');
            } else {
                int b = stack.pop();
                int a = stack.pop();

                switch (c) {
                    case '+': stack.push(a + b); break;
                    case '-': stack.push(a - b); break;
                    case '*': stack.push(a * b); break;
                    case '/': stack.push(a / b); break;
                }
            }
        }

        return stack.pop();
    }

    public static void main(String[] args) {
        String postfix = "23*5+";
        System.out.println("Postfix Evaluation: " + evaluatePostfix(postfix));
    }
}
```

Output:-



```
Postfix Evaluation: 11

C:\Users\BHUWAN PANDEY\OneDrive\Documents\suraj pandey\ds programming lab>
```

18) Write a program to find out the preorder, inorder and postorder traversal of the tree.

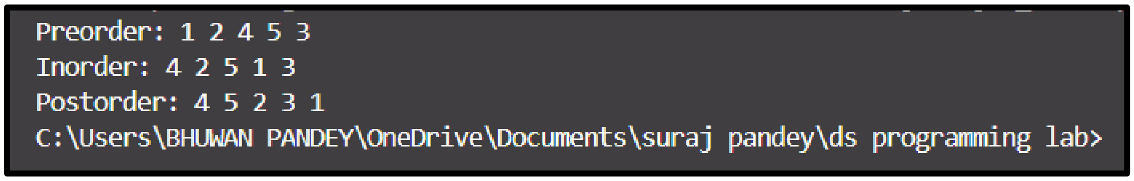
NOTE:

Preorder, inorder, and postorder are tree traversal methods. Preorder visits the root first, then left subtree, then right subtree. Inorder visits the left subtree, root, and right subtree. Postorder visits the left subtree, right subtree, and root last.

Code:-

```
class Program18 {
    static class Node {
        int data;
        Node left, right;

        Node(int data) {
            this.data = data;
            left = right = null;
        }
    }

    Node root;

    public void preorder(Node node) {
        if (node != null) {
            System.out.print(node.data + " ");
            preorder(node.left);
            preorder(node.right);
        }
    }

    public void inorder(Node node) {
        if (node != null) {
            inorder(node.left);
            System.out.print(node.data + " ");
            inorder(node.right);
        }
    }

    public void postorder(Node node) {
        if (node != null) {
            postorder(node.left);
            postorder(node.right);
            System.out.print(node.data + " ");
        }
    }

    public static void main(String[] args) {
        Program18 tree = new Program18();
        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(5);

        System.out.print("Preorder: ");
        tree.preorder(tree.root);
        System.out.print("\nInorder: ");
        tree.inorder(tree.root);
        System.out.print("\nPostorder: ");
        tree.postorder(tree.root);
    }
}
```

Output:-



```
Preorder: 1 2 4 5 3
Inorder: 4 2 5 1 3
Postorder: 4 5 2 3 1
C:\Users\BHUWAN PANDEY\OneDrive\Documents\suraj pandey\ds programming lab>
```

19) Write a program to perform double-order traversal and triple-order traversal on the tree.

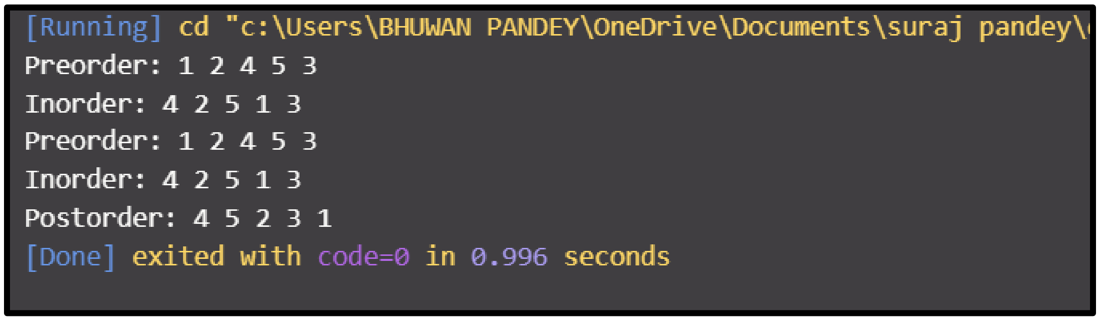
NOTE:

Double-order traversal is essentially a combination of preorder and inorder traversal, while triple-order traversal includes preorder, inorder, and postorder traversals in sequence. These are used to explore the tree in different orders.

Code:-

```
class Program19 {
    static class Node {
        int data;
        Node left, right;

        Node(int data) {
            this.data = data;
            left = right = null;
        }
    }

    Node root;

    public void preorder(Node node) {
        if (node != null) {
            System.out.print(node.data + " ");
            preorder(node.left);
            preorder(node.right);
        }
    }

    public void inorder(Node node) {
        if (node != null) {
            inorder(node.left);
            System.out.print(node.data + " ");
            inorder(node.right);
        }
    }

    public void postorder(Node node) {
        if (node != null) {
            postorder(node.left);
            postorder(node.right);
            System.out.print(node.data + " ");
        }
    }
    public void doubleOrderTraversal() {
        System.out.print("Preorder: ");
        preorder(root);
        System.out.print("\nInorder: ");
        inorder(root);
    }

    public void tripleOrderTraversal() {
        System.out.print("Preorder: ");
        preorder(root);
        System.out.print("\nInorder: ");
        inorder(root);
        System.out.print("\nPostorder: ");
        postorder(root);
    }

    public static void main(String[] args) {
        Program19 tree = new Program19();
        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(5);

        tree.doubleOrderTraversal();
        System.out.println();
        tree.tripleOrderTraversal();
    }
}
```

Output:-



```
[Running] cd "c:\Users\BHUWAN PANDEY\OneDrive\Documents\suraj pandey\
Preorder: 1 2 4 5 3
Inorder: 4 2 5 1 3
Preorder: 1 2 4 5 3
Inorder: 4 2 5 1 3
Postorder: 4 5 2 3 1
[Done] exited with code=0 in 0.996 seconds
```

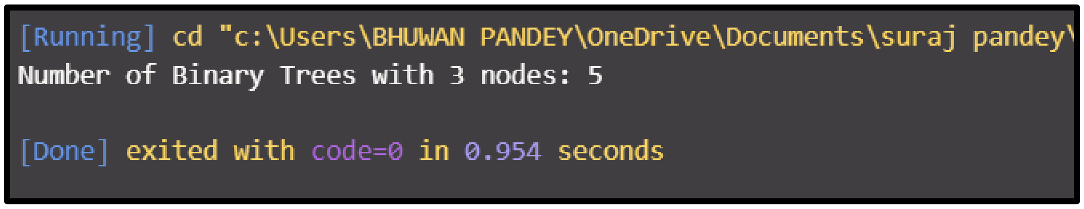
20) Write a program to find the number of binary trees possible with given number of nodes.

NOTE: The number of distinct binary trees that can be formed with n nodes is given by the nth Catalan number. The formula is C(n) = (2n)! / (n! * (n+1)!), which can be computed using dynamic programming.

Code:-

```
class Program20 {

    public static int catalan(int n) {
        int[] dp = new int[n + 1];
        dp[0] = dp[1] = 1;

        for (int i = 2; i <= n; i++) {
            dp[i] = 0;
            for (int j = 0; j < i; j++) {
                dp[i] += dp[j] * dp[i - j - 1];
            }
        }
        return dp[n];
    }

    public static void main(String[] args) {
        int n = 3;
        System.out.println("Number of Binary Trees with " + n + " nodes: " + catalan(n));
    }
}
```

Output:-



```
[Running] cd "c:\Users\BHUWAN PANDEY\OneDrive\Documents\suraj pandey\
Number of Binary Trees with 3 nodes: 5

[Done] exited with code=0 in 0.954 seconds
```

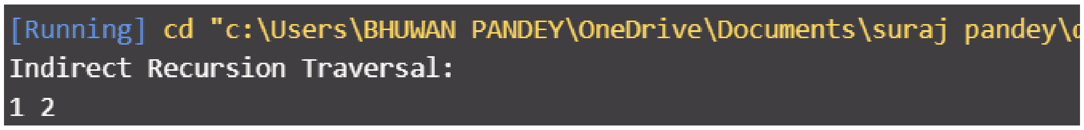
21) Write a program to perform indirect recursion on the tree.

NOTE: In indirect recursion, two or more functions call each other in a cyclic manner. Here, we demonstrate indirect recursion on a binary tree.

Code:-

```
class Program21 {
    static class Node {
        int data;
        Node left, right;

        Node(int item) {
            data = item;
            left = right = null;
        }
    }

    static Node root;

    public static void funcA(Node node) {
        if (node != null) {
            System.out.print(node.data + " ");
            funcB(node.left);
        }
    }

    public static void funcB(Node node) {
        if (node != null) {
            System.out.print(node.data + " ");
            funcA(node.right);
        }
    }

    public static void main(String[] args) {
        root = new Node(1);
        root.left = new Node(2);
        root.right = new Node(3);
        root.left.left = new Node(4);

        System.out.println("Indirect Recursion Traversal:");
        funcA(root);
    }
}
```

Output:-



```
[Running] cd "c:\Users\BHUWAN PANDEY\OneDrive\Documents\suraj pandey\
Indirect Recursion Traversal:
1 2
```

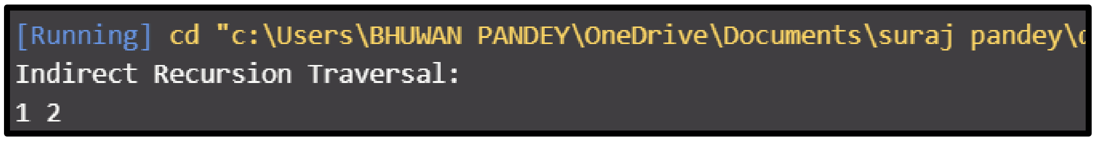
22) Write a program to find out possible labelled and unlabeled binary trees with the given number of nodes.

NOTE: The number of **labelled binary trees** is given by n! × Catalan(n).
The number of **unlabelled binary trees** is given by Catalan number alone.

Code:-

```
class Program22 {
   public static int catalan(int n) {
      int[] dp = new int[n + 1];
      dp[0] = dp[1] = 1;

      for (int i = 2; i <= n; i++) {
         dp[i] = 0;
         for (int j = 0; j < i; j++) {
            dp[i] += dp[j] * dp[i - j - 1];
         }
      }
      return dp[n];
   }

   public static int factorial(int n) {
      if (n <= 1) return 1;
      return n * factorial(n - 1);
   }

   public static void main(String[] args) {
      int n = 3;
      int unlabelled = catalan(n);
      int labelled = factorial(n) * unlabelled;

      System.out.println("Unlabelled Binary Trees: " + unlabelled);
      System.out.println("Labelled Binary Trees: " + labelled);
   }
}
```

Output:-



```
[Running] cd "c:\Users\BHUWAN PANDEY\OneDrive\Documents\suraj pandey\c
Indirect Recursion Traversal:
1 2
```

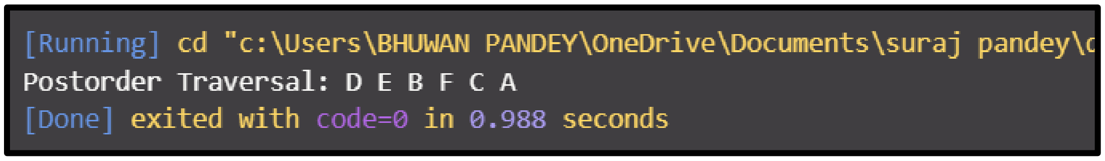
23) Write a program to construct the unique binary tree using inorder and preorder traversal and hence find postorder.

NOTE: Given preorder and inorder traversals of a binary tree, we can reconstruct the tree uniquely. Postorder can be found by traversing the constructed tree.

Code:-

```
import java.util.*;

class Program23 {
    static class Node {
        char data;
        Node left, right;

        Node(char item) {
            data = item;
            left = right = null;
        }
    }

    static int preIndex = 0;

    public static Node buildTree(char[] inorder, char[] preorder, int inStart, int inEnd) {
        if (inStart > inEnd) return null;

        Node node = new Node(preorder[preIndex++]);

        if (inStart == inEnd) return node;

        int inIndex = search(inorder, node.data, inStart, inEnd);

        node.left = buildTree(inorder, preorder, inStart, inIndex - 1);
        node.right = buildTree(inorder, preorder, inIndex + 1, inEnd);

        return node;
    }
    public static int search(char[] inorder, char data, int start, int end) {
        for (int i = start; i <= end; i++) {
            if (inorder[i] == data) return i;
        }
        return -1;
    }

    public static void printPostorder(Node node) {
        if (node != null) {
            printPostorder(node.left);
            printPostorder(node.right);
            System.out.print(node.data + " ");
        }
    }

    public static void main(String[] args) {
        char[] inorder = {'D', 'B', 'E', 'A', 'F', 'C'};
        char[] preorder = {'A', 'B', 'D', 'E', 'C', 'F'};

        Node root = buildTree(inorder, preorder, 0, inorder.length - 1);

        System.out.print("Postorder Traversal: ");
        printPostorder(root);
    }
}
```

Output:-



```
[Running] cd "c:\Users\BHUWAN PANDEY\OneDrive\Documents\suraj pandey\c
Postorder Traversal: D E B F C A
[Done] exited with code=0 in 0.988 seconds
```

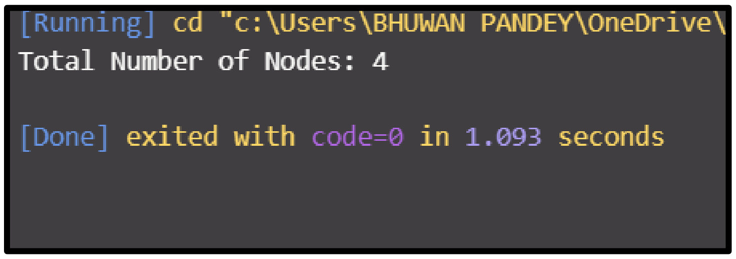
24) Write a recursive program to count the total number of nodes in the tree.

NOTE: To count total nodes, recursively count nodes in left and right subtree and add 1 (for the current node).

Code:-

```
class Program24 {
    static class Node {
        int data;
        Node left, right;

        Node(int item) {
            data = item;
            left = right = null;
        }
    }

    static Node root;

    public static int countNodes(Node node) {
        if (node == null) return 0;
        return 1 + countNodes(node.left) + countNodes(node.right);
    }

    public static void main(String[] args) {
        root = new Node(1);
        root.left = new Node(2);
        root.right = new Node(3);
        root.left.left = new Node(4);

        System.out.println("Total Number of Nodes: " + countNodes(root));
    }
}
```

Output:-



```
[Running] cd "c:\Users\BHUWAN PANDEY\OneDrive\
Total Number of Nodes: 4

[Done] exited with code=0 in 1.093 seconds
```

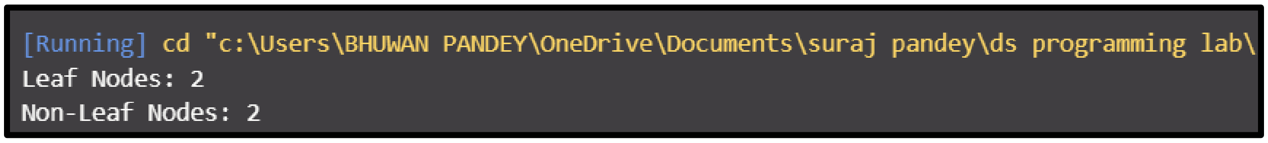
25) Write a recursive program to count the number of the leaf or non-leaf nodes of the tree.

NOTE: Leaf nodes have no children. Non-leaf nodes have at least one child. We can recursively differentiate and count them.

Code:-

```
class Program25 {
    static class Node {
        int data;
        Node left, right;

        Node(int item) {
            data = item;
            left = right = null;
        }
    }

    static Node root;

    public static int countLeafNodes(Node node) {
        if (node == null) return 0;
        if (node.left == null && node.right == null) return 1;
        return countLeafNodes(node.left) + countLeafNodes(node.right);
    }

    public static int countNonLeafNodes(Node node) {
        if (node == null || (node.left == null && node.right == null)) return 0;
        return 1 + countNonLeafNodes(node.left) + countNonLeafNodes(node.right);
    }

    public static void main(String[] args) {
        root = new Node(1);
        root.left = new Node(2);
        root.right = new Node(3);
        root.left.left = new Node(4);

        System.out.println("Leaf Nodes: " + countLeafNodes(root));
        System.out.println("Non-Leaf Nodes: " + countNonLeafNodes(root));
    }
}
```

Output:-



```
[Running] cd "c:\Users\BHUWAN PANDEY\OneDrive\Documents\suraj pandey\ds programming lab\
Leaf Nodes: 2
Non-Leaf Nodes: 2
```

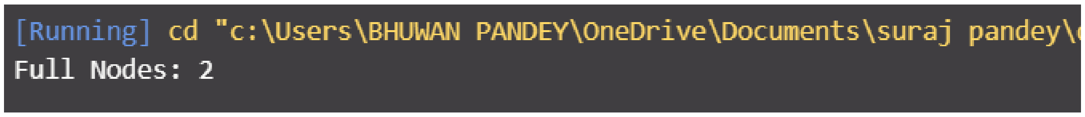
26) Write a recursive program to count the number of full nodes of the tree (Full Nodes are nodes which has both left and right children as non-empty).

NOTE: Full nodes are nodes with **both** left and right children. We count such nodes recursively.

Code:-

```
class Program26 {
    static class Node {
        int data;
        Node left, right;

        Node(int item) {
            data = item;
            left = right = null;
        }
    }

    static Node root;

    public static int countFullNodes(Node node) {
        if (node == null) return 0;
        int count = 0;
        if (node.left != null && node.right != null) count = 1;
        return count + countFullNodes(node.left) + countFullNodes(node.right);
    }

    public static void main(String[] args) {
        root = new Node(1);
        root.left = new Node(2);
        root.right = new Node(3);
        root.left.left = new Node(4);
        root.left.right = new Node(5);

        System.out.println("Full Nodes: " + countFullNodes(root));
    }
}
```

Output:-



```
[Running] cd "c:\Users\BHUWAN PANDEY\OneDrive\Documents\suraj pandey\
Full Nodes: 2
```

27) Write a recursive program to find the height of the tree.

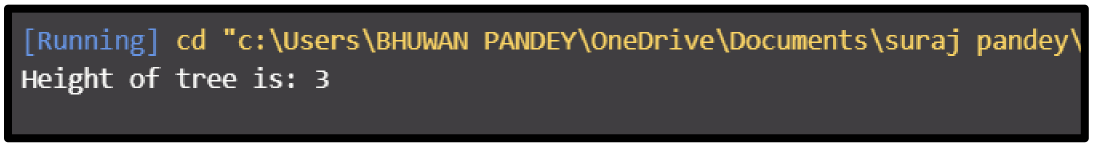
NOTE: The **height** of a binary tree is defined as the number of edges in the longest path from the **root node** to a **leaf node**. It can be computed recursively using:

Code:-

```
import java.util.*;

class Node {
   int data;
   Node left, right;

   Node(int item) {
      data = item;
      left = right = null;
   }
}

class Program27 {
   Node root;

   int height(Node node) {
      if (node == null)
         return 0;
      int left = height(node.left);
      int right = height(node.right);
      return 1 + Math.max(left, right);
   }

   public static void main(String[] args) {
      Program27 tree = new Program27();
      tree.root = new Node(1);
      tree.root.left = new Node(2);
      tree.root.right = new Node(3);
      tree.root.left.left = new Node(4);
      tree.root.left.right = new Node(5);

      System.out.println("Height of tree is: " + tree.height(tree.root));
```

```
    }
}
```

Output:-



```
[Running] cd "c:\Users\BHUWAN PANDEY\OneDrive\Documents\suraj pandey\
Height of tree is: 3
```

28) Write a program to construct BST using inorder and postorder traversal and hence find preorder.

NOTE:

To **construct a BST from Inorder and Postorder**:

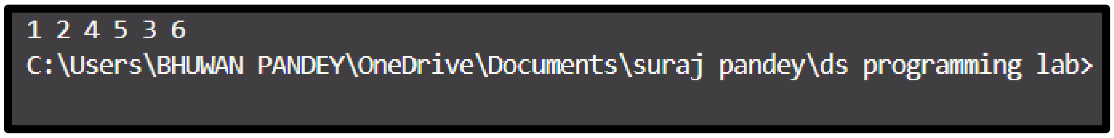
- The last element in postorder is the **root**.
- Find the index of that root in inorder array.
- Recurse on left and right parts of the inorder and postorder arrays.

After construction, use **preorder traversal** to display the tree.

Code:-

```
import java.util.*;

class Node {
    int data;
    Node left, right;

    Node(int item) {
        data = item;
        left = right = null;
    }
}

class Program28 {
    int postIndex;
    Node buildTree(int inorder[], int postorder[], int inStart, int inEnd, Map<Integer, Integer>
inMap) {
        if (inStart > inEnd)
            return null;

        int rootVal = postorder[postIndex--];
        Node root = new Node(rootVal);

        int inIndex = inMap.get(rootVal);
        root.right = buildTree(inorder, postorder, inIndex + 1, inEnd, inMap);
        root.left = buildTree(inorder, postorder, inStart, inIndex - 1, inMap);

        return root;
    }

    void preorder(Node node) {
        if (node == null)
            return;
        System.out.print(node.data + " ");
        preorder(node.left);
        preorder(node.right);
    }

    public static void main(String[] args) {
        Program28 tree = new Program28();
        int inorder[] = {4, 2, 5, 1, 6, 3};
        int postorder[] = {4, 5, 2, 6, 3, 1};
        int n = inorder.length;

        tree.postIndex = n - 1;
        Map<Integer, Integer> inMap = new HashMap<>();
        for (int i = 0; i < n; i++)
            inMap.put(inorder[i], i);

        Node root = tree.buildTree(inorder, postorder, 0, n - 1, inMap);
        tree.preorder(root);
    }
}
```

Output:-



```
1 2 4 5 3 6
C:\Users\BHUWAN PANDEY\OneDrive\Documents\suraj pandey\ds programming lab>
```

29) Write a program to find how many BSTs are possible with given distinct keys.

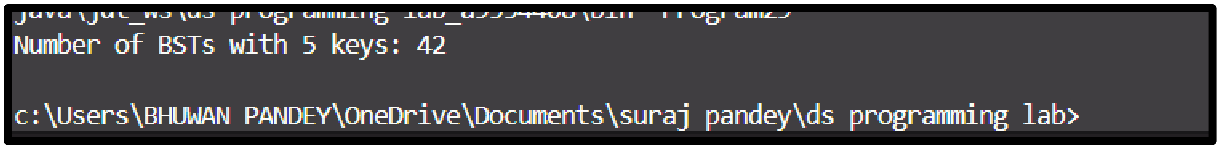
NOTE:

The number of structurally unique BSTs with n distinct keys is the **Catalan number**:
C(n) = (2n)! / ((n + 1)! * n!)

Code:-

```
import java.math.BigInteger;

class Program29 {
    static BigInteger factorial(int n) {
        BigInteger result = BigInteger.ONE;
        for (int i = 2; i <= n; i++)
            result = result.multiply(BigInteger.valueOf(i));
        return result;
    }

    static BigInteger countBST(int n) {
        BigInteger num = factorial(2 * n);
        BigInteger den = factorial(n + 1).multiply(factorial(n));
        return num.divide(den);
    }

    public static void main(String[] args) {
        int n = 5;
        System.out.println("Number of BSTs with " + n + " keys: " + countBST(n));
    }
}
```

Output:-



```
Number of BSTs with 5 keys: 42

c:\Users\BHUWAN PANDEY\OneDrive\Documents\suraj pandey\ds programming lab>
```

30) Write a program to find out the postorder, preorder and inorder traversal on constructed bst and then perform delete operation on the tree and again perform inorder traversal.

NOTE:

Perform all three traversals, then delete a key using BST rules and do inorder again.
Delete cases: node has no child, one child, or two children (replace with inorder successor).

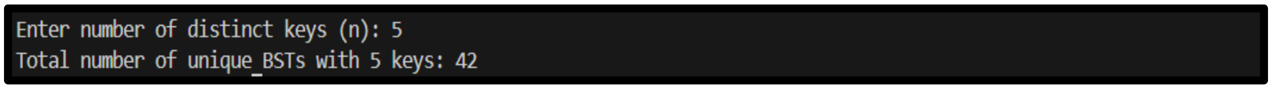
Code:-

```
class Node {
    int data;
    Node left, right;

    public Node(int value) {
        data = value;
        left = right = null;
    }
}

public class BSTOperations {
    Node root;

    // Insert a node into BST
    Node insert(Node root, int key) {
        if (root == null) return new Node(key);

        if (key < root.data)
            root.left = insert(root.left, key);
        else if (key > root.data)
            root.right = insert(root.right, key);
        return root;
    }

    // Inorder Traversal (Left, Root, Right)
    void inorder(Node root) {
        if (root == null) return;
        inorder(root.left);
        System.out.print(root.data + " ");
        inorder(root.right);
    }

    // Preorder Traversal (Root, Left, Right)
    void preorder(Node root) {
        if (root == null) return;
        System.out.print(root.data + " ");
        preorder(root.left);
        preorder(root.right);
    }

    // Postorder Traversal (Left, Right, Root)
    void postorder(Node root) {
        if (root == null) return;
        postorder(root.left);
        postorder(root.right);
        System.out.print(root.data + " ");
    }
```

```
// Find minimum value node (used in delete)
Node minValueNode(Node node) {
    Node current = node;
    while (current.left != null)
        current = current.left;
    return current;
}

// Delete a node from BST
Node delete(Node root, int key) {
    if (root == null) return null;

    if (key < root.data)
        root.left = delete(root.left, key);
    else if (key > root.data)
        root.right = delete(root.right, key);
    else {
        // Node with only one child or no child
        if (root.left == null)
            return root.right;
        else if (root.right == null)
            return root.left;

        // Node with two children: get the inorder successor (smallest in right subtree)
        Node temp = minValueNode(root.right);
        root.data = temp.data; // Copy inorder successor data
```

```
            root.right = delete(root.right, temp.data); // Delete the inorder successor
        }

        return root;
    }

    public static void main(String[] args) {
        BSTOperations bst = new BSTOperations();

        // Construct BST by inserting nodes
        int[] keys = {50, 30, 20, 40, 70, 60, 80};
        for (int key : keys) {
            bst.root = bst.insert(bst.root, key);
        }

        // Perform traversals
        System.out.print("Inorder traversal: ");
        bst.inorder(bst.root);
        System.out.println();

        System.out.print("Preorder traversal: ");
        bst.preorder(bst.root);
        System.out.println();

        System.out.print("Postorder traversal: ");
        bst.postorder(bst.root);
        System.out.println();
        // Perform delete operation
        System.out.println("\nDeleting 50 from the BST...");
        bst.root = bst.delete(bst.root, 50);

        // Inorder after deletion
        System.out.print("Inorder traversal after deletion: ");
        bst.inorder(bst.root);
        System.out.println();
    }
}
```

Output:-



```
Enter number of distinct keys (n): 5
Total number of unique_BSTs with 5 keys: 42
```

31) Write a program to find the minimum and maximum key values from BSTs.

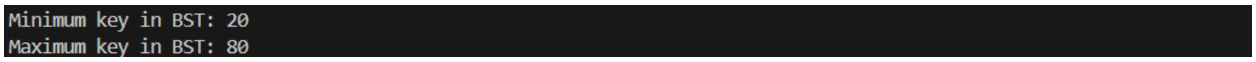
NOTE:

- **Min**: Traverse to the **leftmost node**
- **Max**: Traverse to the **rightmost node**

Code:-

```
class Node {
    int data;
    Node left, right;

    public Node(int value) {
        data = value;
        left = right = null;
    }
}

public class MinMaxInBST {
    Node root;

    // Insert a node in BST
    Node insert(Node root, int key) {
        if (root == null) return new Node(key);

        if (key < root.data)
            root.left = insert(root.left, key);
        else if (key > root.data)
            root.right = insert(root.right, key);

        return root;
    }

    // Find the node with minimum key value (leftmost node)
    int findMin(Node root) {
        if (root == null) {
            System.out.println("Tree is empty");
            return -1;
        }

        Node current = root;
        while (current.left != null) {
            current = current.left;
        }
        return current.data;
    }

    // Find the node with maximum key value (rightmost node)
    int findMax(Node root) {
        if (root == null) {
            System.out.println("Tree is empty");
            return -1;
        }

        Node current = root;
        while (current.right != null) {
            current = current.right;
        }
        return current.data;
    }

    public static void main(String[] args) {
        MinMaxInBST bst = new MinMaxInBST();

        // Sample input
        int[] keys = {50, 30, 20, 40, 70, 60, 80};
        for (int key : keys) {
            bst.root = bst.insert(bst.root, key);
        }


        // Finding and displaying min and max

        System.out.println("Minimum key in BST: " + bst.findMin(bst.root));

        System.out.println("Maximum key in BST: " + bst.findMax(bst.root));

    }

}
```

Output:-



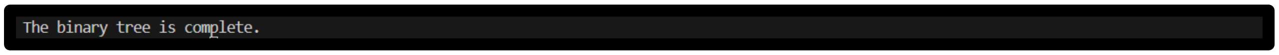
```
Minimum key in BST: 20
Maximum key in BST: 80
```

32) Write a recursive program to check whether given tree is complete tree or not.

Code:-

```
import java.util.LinkedList;

import java.util.Queue;


class Node {

    int data;

    Node left, right;


    public Node(int value) {

        data = value;

        left = right = null;

    }

}


public class EasyCompleteBinaryTreeCheck {


    // Function to check if tree is complete

    boolean isComplete(Node root) {

        if (root == null) return true;


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

        queue.add(root);

        boolean seenNull = false;


        while (!queue.isEmpty()) {
            Node current = queue.poll();

            // If we see a null, set the flag
            if (current == null) {
                seenNull = true;
            } else {
                // If we already saw null and now a non-null, it's not complete
                if (seenNull)
                    return false;

                // Add children to queue
                queue.add(current.left);
                queue.add(current.right);
            }
        }

        return true;
    }

    public static void main(String[] args) {
        EasyCompleteBinaryTreeCheck tree = new EasyCompleteBinaryTreeCheck();

        // Sample tree
        Node root = new Node(1);
        root.left = new Node(2);
        root.right = new Node(3);
        root.left.left = new Node(4);
        root.left.right = new Node(5);
        root.right.left = new Node(6);
        // root.right.right = new Node(7); // Uncomment to make it full

        if (tree.isComplete(root))
            System.out.println("The binary tree is complete.");
        else
            System.out.println("The binary tree is NOT complete.");
    }
}
```

Output:-



```
The binary tree is complete.
```

33) Write a program to construct an AVL tree and perform postorder traversal.

Code:-

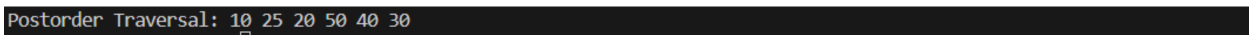
```
class Node {
    int data, height;
    Node left, right;

    Node(int value) {
        data = value;
        height = 1;
    }
}

public class SimpleAVLTree {
```

```
// Get height of node
int height(Node node) {
    return node == null ? 0 : node.height;
}

// Get balance factor
int getBalance(Node node) {
    return node == null ? 0 : height(node.left) - height(node.right);
}

// Right rotation
Node rightRotate(Node y) {
    Node x = y.left;
    Node T2 = x.right;

    // Perform rotation
    x.right = y;
    y.left = T2;

    // Update heights
    y.height = Math.max(height(y.left), height(y.right)) + 1;
    x.height = Math.max(height(x.left), height(x.right)) + 1;

    return x;
}

// Left rotation
```

```
Node leftRotate(Node x) {
    Node y = x.right;
    Node T2 = y.left;

    // Perform rotation
    y.left = x;
    x.right = T2;

    // Update heights
    x.height = Math.max(height(x.left), height(x.right)) + 1;
    y.height = Math.max(height(y.left), height(y.right)) + 1;

    return y;
}

// Insert into AVL Tree
Node insert(Node node, int key) {
    if (node == null) return new Node(key);

    if (key < node.data)
        node.left = insert(node.left, key);
    else if (key > node.data)
        node.right = insert(node.right, key);
    else
        return node; // Duplicates not allowed

    // Update height
```

```
        node.height = 1 + Math.max(height(node.left), height(node.right));

        // Balance the node
        int balance = getBalance(node);

        // Left Left
        if (balance > 1 && key < node.left.data)
            return rightRotate(node);

        // Right Right
        if (balance < -1 && key > node.right.data)
            return leftRotate(node);

        // Left Right
        if (balance > 1 && key > node.left.data) {
            node.left = leftRotate(node.left);
            return rightRotate(node);
        }

        // Right Left
        if (balance < -1 && key < node.right.data) {
            node.right = rightRotate(node.right);
            return leftRotate(node);
        }

        return node;
    }
    // Postorder traversal
    void postorder(Node node) {
        if (node == null) return;
        postorder(node.left);
        postorder(node.right);
        System.out.print(node.data + " ");
    }

    public static void main(String[] args) {
        SimpleAVLTree tree = new SimpleAVLTree();
        Node root = null;

        // Insert values into AVL Tree
        int[] keys = {10, 20, 30, 40, 50, 25};
        for (int key : keys) {
            root = tree.insert(root, key);
        }

        // Perform postorder traversal
        System.out.print("Postorder Traversal: ");
        tree.postorder(root);
        System.out.println();
    }
}
```

Output:-



```
Postorder Traversal: 10 25 20 50 40 30
```

34) Write a program to find the minimum and maximum nodes in an AVL tree of given height.

NOTE:

Base Case: N(0) = 1

N(1) = 2

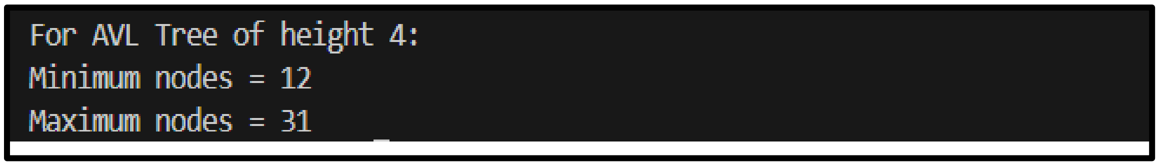
Min number of nodes: N(h) = 1 + N(h - 1) + N(h - 2)

Max number of nodes: M(h) = 2^(h + 1) - 1

Code:-

```
public class AVLTreeMinMax {

    // Function to find minimum nodes in AVL Tree of height h
    static int minNodes(int h) {
        if (h == 0) return 1;
        if (h == 1) return 2;

        int a = 1; // N(0)
        int b = 2; // N(1)
        int c = 0;

        for (int i = 2; i <= h; i++) {
            c = 1 + a + b;
            a = b;
            b = c;
        }
        return c;
    }

    // Function to find maximum nodes in AVL Tree of height h
    static int maxNodes(int h) {
        return (int)Math.pow(2, h + 1) - 1;
    }

    public static void main(String[] args) {
        int height = 4; // Change height as needed

        System.out.println("For AVL Tree of height " + height + ":");
        System.out.println("Minimum nodes = " + minNodes(height));
        System.out.println("Maximum nodes = " + maxNodes(height));
    }
}
```

Output:-



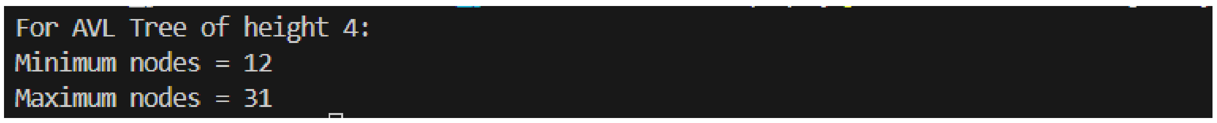
```
For AVL Tree of height 4:
Minimum nodes = 12
Maximum nodes = 31
```

35) Write a program to find the minimum number of nodes on a size-balanced tree.

Code:-

```
// Program to find the minimum number of nodes in a size-balanced binary tree

class BinaryTree {
    // Node structure for the binary tree
    class Node {
        int data;
        Node left, right;

        // Constructor to create a new node
        public Node(int data) {
            this.data = data;
            left = right = null;
        }
    }

    // Function to find the minimum number of nodes in a size-balanced binary tree
    public int minSizeBalancedNodes(int h) {
        // Base case: If the height is 0 or 1, the minimum number of nodes is the height itself.
        if (h <= 0) {
            return 0;
        }
        if (h == 1) {
            return 1;
        }

        // Minimum nodes for height h is given by the formula: 2^h - 1 for a full binary tree
        // For a size-balanced tree, it will be the combination of the left and right subtrees.
        return 1 + minSizeBalancedNodes(h - 1) + minSizeBalancedNodes(h - 2);
    }

    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree();

        // Test: Find minimum number of nodes for size-balanced tree of height 4
        int height = 4;
        int minNodes = tree.minSizeBalancedNodes(height);

        System.out.println("Minimum number of nodes for a size-balanced tree of height " + height
+ " is: " + minNodes);
    }
}
```

Output:-



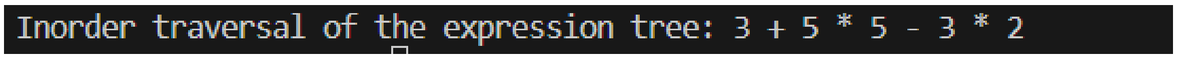
```
For AVL Tree of height 4:
Minimum nodes = 12
Maximum nodes = 31
```

36) Write a program to implement a tree for a given infix expression.

Code:-

```
import java.util.Stack;
class Node {
    String value;
    Node left, right;

    // Constructor for creating a new node
    public Node(String value) {
        this.value = value;
        this.left = this.right = null;
    }
}

public class InfixExpressionTree {

    // Function to check if a string is an operator
    public static boolean isOperator(String c) {
        return c.equals("+") || c.equals("-") || c.equals("*") || c.equals("/");
    }

    // Function to check if the character is a parenthesis
    public static boolean isParenthesis(String c) {
        return c.equals("(") || c.equals(")");
    }

    // Function to handle precedence of operators
    public static boolean hasHigherPrecedence(String op1, String op2) {
        if (op1.equals("*") || op1.equals("/")) {
            return true;
        }
        return op2.equals("+") || op2.equals("-");
    }

    // Function to build the expression tree from infix expression
    public static Node buildTree(String infix) {
        Stack<Node> operands = new Stack<>();  // Stack for operands (numbers/variables)
        Stack<String> operators = new Stack<>(); // Stack for operators (+, -, *, /)

        String[] tokens = infix.split(" "); // Split the infix expression by spaces

        for (String token : tokens) {
            if (token.equals("(")) {
                // Push open parenthesis onto operators stack
                operators.push(token);
            } else if (token.equals(")")) {
                // Pop until matching open parenthesis
                while (!operators.isEmpty() && !operators.peek().equals("(")) {
                    String operator = operators.pop();
                    Node rightNode = operands.pop();
                    Node leftNode = operands.pop();
                    Node operatorNode = new Node(operator);
                    operatorNode.left = leftNode;
                    operatorNode.right = rightNode;
                    operands.push(operatorNode);
                }
                // Pop the '('
                if (!operators.isEmpty() && operators.peek().equals("(")) {
                    operators.pop();
                }
            } else if (!isOperator(token)) {
                // If token is an operand (number/variable), push it onto the operands stack
                operands.push(new Node(token));
            } else {
                // If token is an operator
                while (!operators.isEmpty() && isOperator(operators.peek()) &&
hasHigherPrecedence(operators.peek(), token)) {
                    String operator = operators.pop();
                    Node rightNode = operands.pop();
                    Node leftNode = operands.pop();
                    Node operatorNode = new Node(operator);
                    operatorNode.left = leftNode;
                    operatorNode.right = rightNode;
                    operands.push(operatorNode);
                }
                operators.push(token); // Push the current operator onto the stack
            }
        }

        // Process remaining operators in the stack
        while (!operators.isEmpty()) {
            String operator = operators.pop();
            Node rightNode = operands.pop();
            Node leftNode = operands.pop();
            Node operatorNode = new Node(operator);
            operatorNode.left = leftNode;
            operatorNode.right = rightNode;
            operands.push(operatorNode);
        }

        // The final node in the stack is the root of the expression tree
        return operands.pop();
    }

    // Function to print the tree using inorder traversal
    public static void inorderTraversal(Node root) {
        if (root != null) {
            inorderTraversal(root.left);  // Visit left subtree
            System.out.print(root.value + " ");  // Print node value
            inorderTraversal(root.right);  // Visit right subtree
        }
    }

    public static void main(String[] args) {
        // Example infix expression with parentheses
        String infix = "3 + 5 * ( 5 - 3 ) * 2";

        // Build the tree for the given infix expression
        Node root = buildTree(infix);

        // Print the inorder traversal of the tree
        System.out.print("Inorder traversal of the expression tree: ");
        inorderTraversal(root);
    }
}
```

Output:-



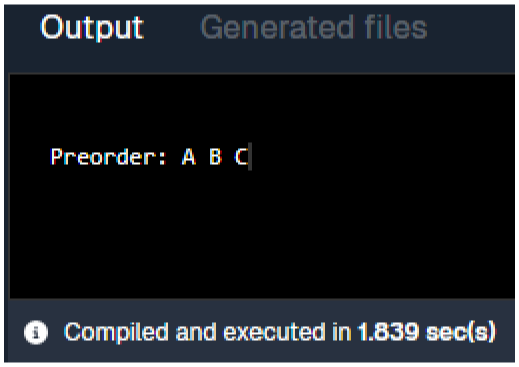
```
Inorder traversal of the expression tree: 3 + 5 * 5 - 3 * 2
```

37) Write a program to draw a tree for a given nested tree representation expression.

Code:-

```
public class NestedTree {
    static class Node {
        char data;
        Node left, right;
        Node(char d) { data = d; }
    }
    static Node buildTree(String expr, int[] index) {
        if (index[0] >= expr.length()) return null;
        char ch = expr.charAt(index[0]);
        if (ch == '(') {
            index[0]++;
            char nodeData = expr.charAt(index[0]++);
            Node root = new Node(nodeData);
            root.left = buildTree(expr, index);
            index[0]++;
            root.right = buildTree(expr, index);
            index[0]++;
            return root;
        }
        return null;
    }
    static void preorder(Node root) {
        if (root != null) {
            System.out.print(root.data + " ");
            preorder(root.left);
            preorder(root.right);
        }
    }
    public static void main(String[] args) {
        String expr = "(A(B,C))";
        int[] index = {0};
        Node root = buildTree(expr, index);
        System.out.print("Preorder: ");
        preorder(root);
    }
}
```

Output:-



```
Preorder: A B C
```

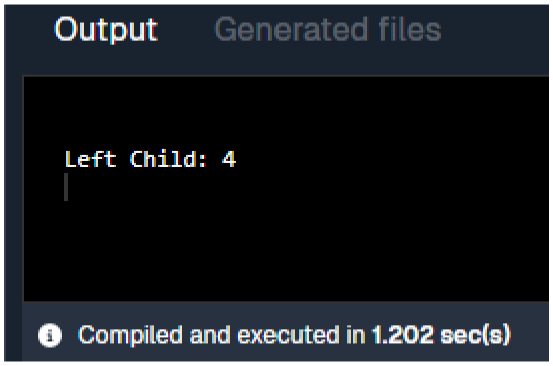
Compiled and executed in 1.839 sec(s)

38) Write a program to find the left child of kth element from the given array representation tree.

Code:-

```
public class LeftChildArrayTree {
    public static void main(String[] args) {
        int[] tree = {1, 2, 3, 4, 5, 6, 7};
        int k = 1;
        int leftChildIdx = 2 * k + 1;
        if (leftChildIdx < tree.length)
            System.out.println("Left Child: " + tree[leftChildIdx]);
        else
            System.out.println("No Left Child");
    }
}
```

Output:-



```
Left Child: 4
```

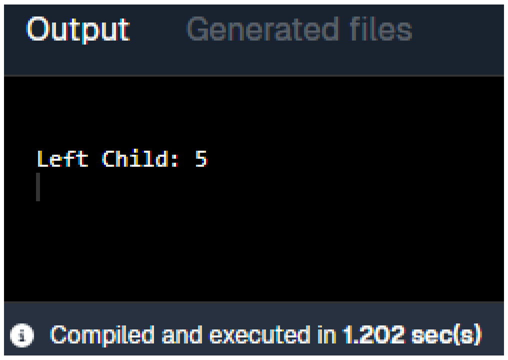
Compiled and executed in 1.202 sec(s)

39) Write a program to find the left child of kth element from the given leftmost child right sibling representation tree.

Code:-

```
public class LeftmostChildTree {
    static class Node {
        int data;
        Node leftChild, rightSibling;
        Node(int d) { data = d; }

    static Node findKth(Node root, int k, int[] count) {
        if (root == null) return null;
        if (count[0] == k) return root;
        count[0]++;
        Node res = findKth(root.leftChild, k, count);
        if (res == null) res = findKth(root.rightSibling, k, count);
        return res;
    }

    public static void main(String[] args) {
        Node root = new Node(1);
        root.leftChild = new Node(2);
        root.leftChild.rightSibling = new Node(3);
        root.leftChild.rightSibling.rightSibling = new Node(4);
        root.leftChild.leftChild = new Node(5);
        int k = 1;

        Node kth = findKth(root, k, new int[]{0});
        if (kth != null && kth.leftChild != null)
            System.out.println("Left Child: " + kth.leftChild.data);
        else
            System.out.println("No Left Child");
    }
}
```

Output:-



```
Left Child: 5
```

Compiled and executed in 1.202 sec(s)

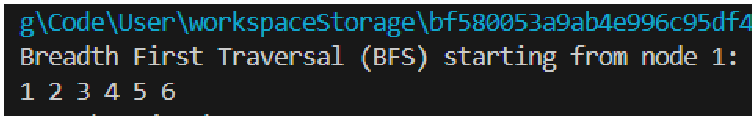
40) Write a program for breadth first traversal on graph.

Topic: BFS on Graph

Code:-

```
import java.util.*;

public class Graph {
    private Map<Integer, List<Integer>> adjList; // adjacency list to represent the graph

    // Constructor to initialize the graph
    public Graph() {
        adjList = new HashMap<>();
    }

    // Method to add an edge between two nodes
    public void addEdge(int u, int v) {
        adjList.putIfAbsent(u, new ArrayList<>());
        adjList.putIfAbsent(v, new ArrayList<>());
        adjList.get(u).add(v);
        adjList.get(v).add(u);  // For an undirected graph, add edge in both directions
    }

    // BFS traversal of the graph
    public void bfs(int start) {
        // Create a visited set to keep track of visited nodes
        Set<Integer> visited = new HashSet<>();

        // Create a queue for BFS
        Queue<Integer> queue = new LinkedList<>();

        // Start by visiting the start node
        visited.add(start);
        queue.offer(start); // Enqueue the start node

        System.out.println("Breadth First Traversal (BFS) starting from node " + start + ":");

        // Loop while there are nodes to visit
        while (!queue.isEmpty()) {
            int node = queue.poll(); // Dequeue the front node
            System.out.print(node + " "); // Process the node (here, we're just printing it)

            // Enqueue all unvisited neighbors of the current node
            for (int neighbor : adjList.get(node)) {
                if (!visited.contains(neighbor)) {
                    visited.add(neighbor); // Mark as visited
                    queue.offer(neighbor); // Enqueue the neighbor
                }
            }
        }
        System.out.println(); // Print a newline after the traversal
    }

    public static void main(String[] args) {
        Graph graph = new Graph();

        // Add edges to the graph
        graph.addEdge(1, 2);
        graph.addEdge(1, 3);
        graph.addEdge(2, 4);
        graph.addEdge(2, 5);
        graph.addEdge(3, 6);

        // Perform BFS starting from node 1
        graph.bfs(1);
    }
}
```

Output:-



```
g\Code\User\workspaceStorage\bf580053a9ab4e996c95df4
Breadth First Traversal (BFS) starting from node 1:
1 2 3 4 5 6
```

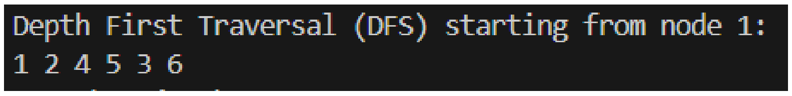
41) Write a program for depth first traversal on graph.

Topic: DFS on Graph

Code:-

```
import java.util.*;

public class Graph {
    // A map (hash map) to store the graph as an adjacency list
    private Map<Integer, List<Integer>> adjList;

    // Constructor to create the graph
    public Graph() {
        adjList = new HashMap<>(); // Initializes the map
    }

    // Method to add an edge between two nodes (connections between nodes)
    public void addEdge(int u, int v) {
        adjList.putIfAbsent(u, new ArrayList<>()); // Ensure node u exists
        adjList.putIfAbsent(v, new ArrayList<>()); // Ensure node v exists

        // Add v to the adjacency list of u and vice versa (undirected graph)
        adjList.get(u).add(v);
        adjList.get(v).add(u);
    }

    // Depth First Search (DFS) - Recursive method
    public void dfs(int node, Set<Integer> visited) {
        // Visit the current node (print it)
        System.out.print(node + " ");
        visited.add(node); // Mark this node as visited

        // Visit all the neighbors of the current node that haven't been visited
        for (int neighbor : adjList.get(node)) {
            if (!visited.contains(neighbor)) {
                dfs(neighbor, visited); // Recur for each unvisited neighbor
            }
        }
    }

    // Main method where the graph is created and DFS is run
    public static void main(String[] args) {
        Graph graph = new Graph(); // Create a new graph

        // Add edges (connections) between nodes
        graph.addEdge(1, 2);
        graph.addEdge(1, 3);
        graph.addEdge(2, 4);
        graph.addEdge(2, 5);
        graph.addEdge(3, 6);

        // Create a set to keep track of visited nodes
        Set<Integer> visited = new HashSet<>();

        // Start DFS from node 1
        System.out.println("Depth First Traversal (DFS) starting from node 1:");
        graph.dfs(1, visited); // Perform DFS
    }
}
```

Output:-



```
Depth First Traversal (DFS) starting from node 1:
1 2 4 5 3 6
```

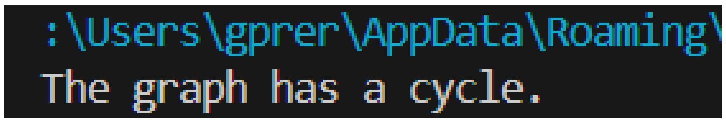
42) Write a program to check whether there is a cycle in a given directed graph or not.

Topic : Directed Graph

Code:-

```
import java.util.*;

public class Graph {
    private Map<Integer, List<Integer>> adjList; // Adjacency list to store the graph

    // Constructor to initialize the graph
    public Graph() {
        adjList = new HashMap<>();
    }

    // Method to add an edge between two nodes (directed graph)
    public void addEdge(int u, int v) {
        adjList.putIfAbsent(u, new ArrayList<>());
        adjList.get(u).add(v); // Add v to the adjacency list of u
    }

    // Helper method to perform DFS and check for a cycle
    private boolean dfs(int node, Set<Integer> visited, Set<Integer> recursionStack) {
        // Mark the current node as visited and add it to the recursion stack
        visited.add(node);
        recursionStack.add(node);

        // Recur for all the neighbors of the current node
        for (int neighbor : adjList.getOrDefault(node, new ArrayList<>())) {
            // If the neighbor is not visited yet, do a DFS call
            if (!visited.contains(neighbor) && dfs(neighbor, visited, recursionStack)) {
                return true;
            }
            // If the neighbor is in the recursion stack, a cycle is found
            else if (recursionStack.contains(neighbor)) {
                return true; // Cycle detected
            }
        }

        // Backtrack: Remove the node from recursion stack after processing
        recursionStack.remove(node);
        return false; // No cycle found from this node
    }

    // Method to check if the graph has a cycle
    public boolean hasCycle() {
        Set<Integer> visited = new HashSet<>(); // Set to track visited nodes
        Set<Integer> recursionStack = new HashSet<>(); // Set to track the current path

        // Check for cycles in all nodes (since the graph could be disconnected)
        for (Integer node : adjList.keySet()) {
            if (!visited.contains(node)) {
                if (dfs(node, visited, recursionStack)) {
                    return true; // Cycle found
                }
            }
        }
        return false; // No cycle found in the graph
    }

    public static void main(String[] args) {
        Graph graph = new Graph();

        // Add edges to the graph (directed)
        graph.addEdge(1, 2);
        graph.addEdge(2, 3);
        graph.addEdge(3, 4);
        graph.addEdge(4, 2); // This creates a cycle (2 -> 3 -> 4 -> 2)

        // Check if the graph has a cycle
        if (graph.hasCycle()) {
            System.out.println("The graph has a cycle.");
        } else {
            System.out.println("The graph does not have a cycle.");
        }
    }
}
```

Output:-



```
:\Users\gprer\AppData\Roaming\
The graph has a cycle.
```

Searching and Sorting

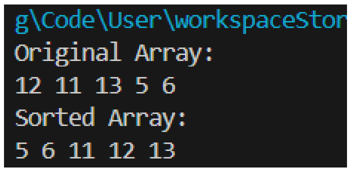
Use this data for all following programs: 12, 56, 3, 7, 9, 35, 11, 19, 25, 75

43) Write a program to sort given elements using insertion sort method.

Code:-

```
public class InsertionSort {

    // Method to perform Insertion Sort on an array
    public static void insertionSort(int[] arr) {
        int n = arr.length;

        // Start from the second element (index 1)
        for (int i = 1; i < n; i++) {
            int key = arr[i];  // The element to be inserted into the sorted portion
            int j = i - 1;     // The index of the last element in the sorted portion

            // Move elements of arr[0..i-1] that are greater than 'key' to one position ahead
            // of their current position
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];  // Shift element to the right
                j = j - 1;            // Move to the previous element
            }

            // Insert the key into its correct position
            arr[j + 1] = key;
        }
    }

    // Method to print the array (for output)
    public static void printArray(int[] arr) {
        for (int i : arr) {
            System.out.print(i + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        // Example array
        int[] arr = {12, 11, 13, 5, 6};

        // Print the original array
        System.out.println("Original Array:");
        printArray(arr);
        // Call insertionSort to sort the array
        insertionSort(arr);

        // Print the sorted array
        System.out.println("Sorted Array:");
        printArray(arr);
    }
}
```

Output:-



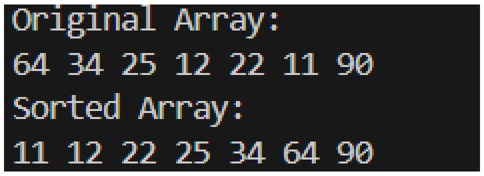
```
g\Code\User\workspaceStor
Original Array:
12 11 13 5 6
Sorted Array:
5 6 11 12 13
```

44) Write a program to sort given elements using bubble sort method.

Code:-

```
public class BubbleSort {

    // Method to perform Bubble Sort on an array
    public static void bubbleSort(int[] arr) {
        int n = arr.length;

        // Traverse through all elements in the array
        for (int i = 0; i < n - 1; i++) {
            // Last i elements are already sorted, so no need to check them
            for (int j = 0; j < n - i - 1; j++) {
                // Compare adjacent elements
                if (arr[j] > arr[j + 1]) {
                    // Swap if the element is greater than the next element
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }
    }

    // Method to print the array (for output)
    public static void printArray(int[] arr) {
        for (int i : arr) {
            System.out.print(i + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        // Example array
        int[] arr = {64, 34, 25, 12, 22, 11, 90};

        // Print the original array
        System.out.println("Original Array:");
        printArray(arr);

        // Call bubbleSort to sort the array
        bubbleSort(arr);

        // Print the sorted array
        System.out.println("Sorted Array:");
        printArray(arr);
    }
}
```

Output:-



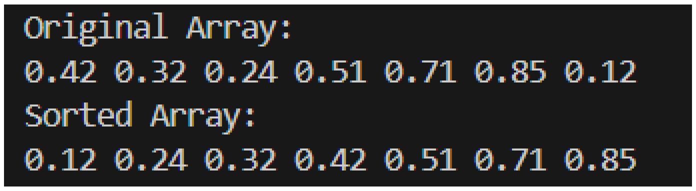
```
Original Array:
64 34 25 12 22 11 90
Sorted Array:
11 12 22 25 34 64 90
```

45) Write a program to sort given elements using bucket sort method.

Code:-

```
import java.util.ArrayList;
import java.util.Collections;

public class BucketSort {

    // Method to perform Bucket Sort
    public static void bucketSort(float[] arr) {
        int n = arr.length;

        // 1. Create empty buckets
        ArrayList<Float>[] buckets = new ArrayList[n];

        // Create empty buckets
        for (int i = 0; i < n; i++) {
            buckets[i] = new ArrayList<>();
        }

        // 2. Distribute input array elements into buckets
        for (float num : arr) {
            int index = (int) (num * n);  // Calculate bucket index
            buckets[index].add(num);      // Add element to the respective bucket
        }

        // 3. Sort each bucket (using Collections.sort for simplicity)
        for (int i = 0; i < n; i++) {
            Collections.sort(buckets[i]);  // Sort individual buckets
        }

        // 4. Concatenate all buckets into the original array
        int index = 0;
        for (int i = 0; i < n; i++) {
            for (float num : buckets[i]) {
                arr[index++] = num;  // Place sorted elements back into the original array
            }
        }
    }

    // Method to print the array (for output)
    public static void printArray(float[] arr) {
        for (float i : arr) {
            System.out.print(i + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        // Example array (floating-point numbers between 0 and 1)
        float[] arr = {0.42f, 0.32f, 0.24f, 0.51f, 0.71f, 0.85f, 0.12f};

        // Print the original array
        System.out.println("Original Array:");
        printArray(arr);

        // Call bucketSort to sort the array
        bucketSort(arr);

        // Print the sorted array
        System.out.println("Sorted Array:");
        printArray(arr);
    }
}
```

Output:-



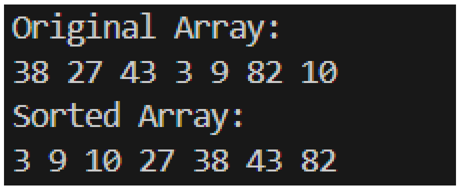
```
Original Array:
0.42 0.32 0.24 0.51 0.71 0.85 0.12
Sorted Array:
0.12 0.24 0.32 0.42 0.51 0.71 0.85
```

46) Write a program to sort given elements using merge sort method.

Code:-

```
public class MergeSort {

    // Method to perform Merge Sort on the array
    public static void mergeSort(int[] arr) {
        if (arr.length < 2) {
            return;  // Base case: an array with 1 or 0 elements is already sorted
        }
        int mid = arr.length / 2;

        // Divide the array into two halves
        int[] left = new int[mid];
        int[] right = new int[arr.length - mid];

        // Copy data to left and right arrays
        System.arraycopy(arr, 0, left, 0, mid);
        System.arraycopy(arr, mid, right, 0, arr.length - mid);

        // Recursively sort both halves
        mergeSort(left);
        mergeSort(right);

        // Merge the sorted halves back into the original array
        merge(arr, left, right);
    }

    // Method to merge two sorted arrays into the original array
    public static void merge(int[] arr, int[] left, int[] right) {
        int i = 0, j = 0, k = 0;

        // Merge the left and right arrays into arr[]
        while (i < left.length && j < right.length) {
            if (left[i] <= right[j]) {
                arr[k++] = left[i++];
            } else {
                arr[k++] = right[j++];
            }
        }

        // Copy any remaining elements from left[]
        while (i < left.length) {
            arr[k++] = left[i++];
        }

        // Copy any remaining elements from right[]
        while (j < right.length) {
            arr[k++] = right[j++];
        }
    }

    // Method to print the array (for output)
    public static void printArray(int[] arr) {
        for (int i : arr) {
            System.out.print(i + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        // Example array
        int[] arr = {38, 27, 43, 3, 9, 82, 10};

        // Print the original array
        System.out.println("Original Array:");
        printArray(arr);

        // Call mergeSort to sort the array
        mergeSort(arr);

        // Print the sorted array
        System.out.println("Sorted Array:");
        printArray(arr);
    }
}
```

Output:-



```
Original Array:
38 27 43 3 9 82 10
Sorted Array:
3 9 10 27 38 43 82
```

47) Write a program to sort given elements using quick sort method.

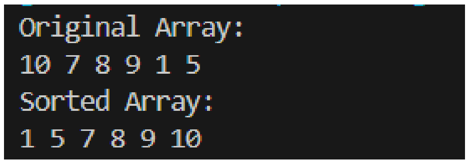
Code:-

```
public class QuickSort {

    // Method to perform Quick Sort
    public static void quickSort(int[] arr, int low, int high) {
        if (low < high) {
            // Find pivot element such that elements smaller than the pivot are on the left,
            // and elements greater than the pivot are on the right
            int pi = partition(arr, low, high);

            // Recursively sort the left and right subarrays
            quickSort(arr, low, pi - 1);
            quickSort(arr, pi + 1, high);
        }
    }
```

```
// Method to partition the array and return the pivot index
public static int partition(int[] arr, int low, int high) {
    int pivot = arr[high];  // Select the last element as the pivot
    int i = (low - 1);  // Index of smaller element

    // Rearranging the array such that elements less than pivot are on the left
    // and elements greater than pivot are on the right
    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;  // Increment index of smaller element
            // Swap arr[i] and arr[j]
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }

    // Swap the pivot element with the element at index i + 1
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    return i + 1;  // Return the pivot index
}

// Method to print the array (for output)
public static void printArray(int[] arr) {
    for (int i : arr) {
        System.out.print(i + " ");
    }
    System.out.println();
}

public static void main(String[] args) {
    // Example array
    int[] arr = {10, 7, 8, 9, 1, 5};

    // Print the original array
    System.out.println("Original Array:");
    printArray(arr);

    // Call quickSort to sort the array
    quickSort(arr, 0, arr.length - 1);

    // Print the sorted array
```

```
        System.out.println("Sorted Array:");
        printArray(arr);
    }
}
```

Output:-



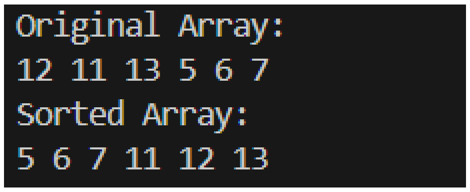
```
Original Array:
10 7 8 9 1 5
Sorted Array:
1 5 7 8 9 10
```

48) Write a program to sort given elements using heap sort method.

Code:-

```
public class HeapSort {

    // Method to perform heap sort
    public static void heapSort(int[] arr) {
        int n = arr.length;

        // Step 1: Build a max heap (rearrange the array)
        for (int i = n / 2 - 1; i >= 0; i--) {
            heapify(arr, n, i);
        }

        // Step 2: Extract elements from the heap one by one
        for (int i = n - 1; i > 0; i--) {
            // Move the current root (max element) to the end of the array
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;

            // Call heapify on the reduced heap
            heapify(arr, i, 0);
        }
    }

    // Method to maintain the heap property
    public static void heapify(int[] arr, int n, int i) {
        int largest = i;  // Initialize largest as root
        int left = 2 * i + 1;  // left child index
        int right = 2 * i + 2;  // right child index

        // If left child is larger than root
        if (left < n && arr[left] > arr[largest]) {
            largest = left;
        }

        // If right child is larger than largest so far
        if (right < n && arr[right] > arr[largest]) {
            largest = right;
        }

        // If largest is not root, swap and recursively heapify the affected sub-tree
        if (largest != i) {
            int temp = arr[i];
            arr[i] = arr[largest];
            arr[largest] = temp;

            // Recursively heapify the affected sub-tree
            heapify(arr, n, largest);
        }
    }

    // Method to print the array (for output)
    public static void printArray(int[] arr) {
        for (int i : arr) {
            System.out.print(i + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        // Example array
        int[] arr = {12, 11, 13, 5, 6, 7};

        // Print the original array
        System.out.println("Original Array:");
        printArray(arr);

        // Call heapSort to sort the array
        heapSort(arr);

        // Print the sorted array
        System.out.println("Sorted Array:");
        printArray(arr);
    }
}
```

Output:-



```
Original Array:
12 11 13 5 6 7
Sorted Array:
5 6 7 11 12 13
```

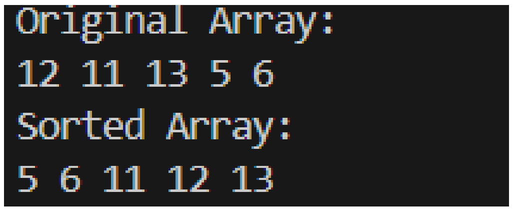
49) Write a program to sort given elements using insertion sort method.

Topic: Insertion Sort

Code:-

```
public class InsertionSort {

    // Method to perform Insertion Sort
    public static void insertionSort(int[] arr) {
        int n = arr.length;

        // Traverse through the array from the second element
        for (int i = 1; i < n; i++) {
            int key = arr[i];  // The element to be inserted into the sorted part
            int j = i - 1;

            // Move elements of arr[0..i-1] that are greater than the key
            // to one position ahead of their current position
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j = j - 1;
            }

            // Insert the key into its correct position
            arr[j + 1] = key;
        }
    }

    // Method to print the array (for output)
    public static void printArray(int[] arr) {
        for (int i : arr) {
            System.out.print(i + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        // Example array
        int[] arr = {12, 11, 13, 5, 6};

        // Print the original array
        System.out.println("Original Array:");
        printArray(arr);

        // Call insertionSort to sort the array
        insertionSort(arr);

        // Print the sorted array
        System.out.println("Sorted Array:");
        printArray(arr);
    }
}
```

Output:-



```
Original Array:
12 11 13 5 6
Sorted Array:
5 6 11 12 13
```

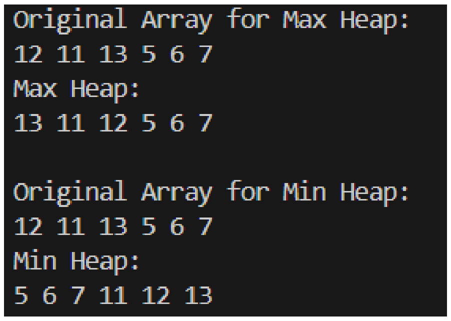
50) Write a program to construct min heap and max heap.

Topic: Max Heap and Min Heap

Code:-

```
public class HeapExample {

    // Method to build a Max Heap
    public static void buildMaxHeap(int[] arr) {
        int n = arr.length;

        // Start from the last non-leaf node and move towards the root
        for (int i = n / 2 - 1; i >= 0; i--) {
            maxHeapify(arr, n, i);
        }
    }

    // Method to maintain Max Heap property
    public static void maxHeapify(int[] arr, int n, int i) {
        int largest = i;  // Initialize largest as root
        int left = 2 * i + 1;  // Left child index
        int right = 2 * i + 2;  // Right child index
        // If left child is larger than root
        if (left < n && arr[left] > arr[largest]) {
            largest = left;
        }

        // If right child is larger than largest so far
        if (right < n && arr[right] > arr[largest]) {
            largest = right;
        }

        // If largest is not root, swap and heapify the affected subtree
        if (largest != i) {
            int temp = arr[i];
            arr[i] = arr[largest];
            arr[largest] = temp;

            maxHeapify(arr, n, largest);
        }
    }

    // Method to build a Min Heap
    public static void buildMinHeap(int[] arr) {
        int n = arr.length;

        // Start from the last non-leaf node and move towards the root
        for (int i = n / 2 - 1; i >= 0; i--) {
            minHeapify(arr, n, i);
        }
    }

    // Method to maintain Min Heap property
    public static void minHeapify(int[] arr, int n, int i) {
        int smallest = i;  // Initialize smallest as root
        int left = 2 * i + 1;  // Left child index
        int right = 2 * i + 2;  // Right child index

        // If left child is smaller than root
        if (left < n && arr[left] < arr[smallest]) {
            smallest = left;
        }

        // If right child is smaller than smallest so far
        if (right < n && arr[right] < arr[smallest]) {
            smallest = right;
        }
        // If smallest is not root, swap and heapify the affected subtree
        if (smallest != i) {
            int temp = arr[i];
            arr[i] = arr[smallest];
            arr[smallest] = temp;

            minHeapify(arr, n, smallest);
        }
    }

    // Method to print the array (for output)
    public static void printArray(int[] arr) {
        for (int i : arr) {
            System.out.print(i + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        // Example array for Max Heap
        int[] maxHeapArr = {12, 11, 13, 5, 6, 7};

        System.out.println("Original Array for Max Heap:");
        printArray(maxHeapArr);

        // Build Max Heap
        buildMaxHeap(maxHeapArr);

        System.out.println("Max Heap:");
        printArray(maxHeapArr);

        // Example array for Min Heap
        int[] minHeapArr = {12, 11, 13, 5, 6, 7};

        System.out.println("\nOriginal Array for Min Heap:");
        printArray(minHeapArr);

        // Build Min Heap
        buildMinHeap(minHeapArr);

        System.out.println("Min Heap:");
        printArray(minHeapArr);
    }
}
```

Output:-



```
Original Array for Max Heap:
12 11 13 5 6 7
Max Heap:
13 11 12 5 6 7

Original Array for Min Heap:
12 11 13 5 6 7
Min Heap:
5 6 7 11 12 13
```

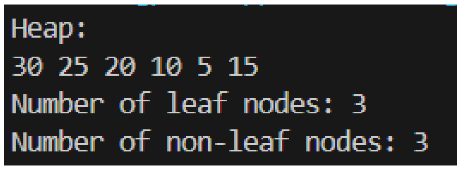
51) Write a program to find the number of leaf and non-leaf nodes of a max heap.

Topic: Max Heap

Code:-

```
public class MaxHeap {
    private int[] heap;  // Array to store the heap
    private int size;    // Number of elements in the heap
    private int capacity; // Maximum capacity of the heap

    // Constructor to initialize the heap
    public MaxHeap(int capacity) {
        this.capacity = capacity;
        this.size = 0;
        heap = new int[capacity];  // Create an array to store heap elements
    }

    // Method to insert a new value into the heap
    public void insert(int value) {
        if (size == capacity) {
            System.out.println("Heap is full, cannot insert");
            return;
        }

        // Insert the new value at the end of the heap
        heap[size] = value;
        size++;

        // Heapify the value up to maintain the max-heap property
        heapifyUp(size - 1);
    }

    // Method to heapify the value up to maintain the max-heap property
    private void heapifyUp(int index) {
        // While the current node is greater than its parent, swap them
        while (index > 0 && heap[parent(index)] < heap[index]) {
            swap(index, parent(index));
            index = parent(index); // Move to the parent's index
        }
    }

    // Method to get the index of the parent of a node
    private int parent(int index) {
        return (index - 1) / 2;
    }

    // Method to swap two elements in the heap
    private void swap(int i, int j) {
        int temp = heap[i];
        heap[i] = heap[j];
        heap[j] = temp;
    }

    // Method to find the number of leaf nodes in the heap
    public int countLeafNodes() {
        // Leaf nodes start from index n/2 to n-1
        int leafCount = 0;
        for (int i = size / 2; i < size; i++) {
            leafCount++;
        }
        return leafCount;
    }

    // Method to find the number of non-leaf nodes in the heap
    public int countNonLeafNodes() {
        // Non-leaf nodes are from index 0 to n/2 - 1
        int nonLeafCount = 0;
        for (int i = 0; i < size / 2; i++) {
            nonLeafCount++;
        }
        return nonLeafCount;
    }

    // Method to display the heap elements
    public void display() {
        System.out.println("Heap: ");
        for (int i = 0; i < size; i++) {
            System.out.print(heap[i] + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        MaxHeap maxHeap = new MaxHeap(10);  // Create a Max Heap with capacity 10

        // Insert elements into the Max Heap
        maxHeap.insert(30);
        maxHeap.insert(25);
        maxHeap.insert(20);
        maxHeap.insert(10);
        maxHeap.insert(5);
        maxHeap.insert(15);

        // Display the heap
        maxHeap.display();

        // Find and display the number of leaf and non-leaf nodes
        int leafNodes = maxHeap.countLeafNodes();
        int nonLeafNodes = maxHeap.countNonLeafNodes();

        System.out.println("Number of leaf nodes: " + leafNodes);
        System.out.println("Number of non-leaf nodes: " + nonLeafNodes);
    }
}
```

Output:-



```
Heap:
30 25 20 10 5 15
Number of leaf nodes: 3
Number of non-leaf nodes: 3
```

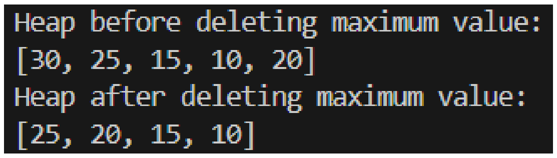
52) Write a program to delete maximum value from a max heap and then reheapify.

Topic: Max Heap

Code:-

```
import java.util.Arrays;

public class MaxHeap {
    private int[] heap;  // Array to store the heap
    private int size;    // Number of elements in the heap
    private int capacity; // Maximum capacity of the heap

    // Constructor to initialize the heap
    public MaxHeap(int capacity) {
        this.capacity = capacity;
        this.size = 0;
        heap = new int[capacity];  // Create an array to store heap elements
    }

    // Method to get the index of the parent of a node
    private int parent(int index) {
        return (index - 1) / 2;
    }

    // Method to get the index of the left child of a node
    private int leftChild(int index) {
        return 2 * index + 1;
    }

    // Method to get the index of the right child of a node
    private int rightChild(int index) {
        return 2 * index + 2;
    }

    // Method to swap two elements in the heap
    private void swap(int i, int j) {
        int temp = heap[i];
        heap[i] = heap[j];
        heap[j] = temp;
    }

    // Method to heapify (rearrange) the heap to maintain max-heap property
    private void heapifyDown(int index) {
        int largest = index;  // Assume the current index is the largest
        int left = leftChild(index);  // Get the left child index
        int right = rightChild(index); // Get the right child index
        // If the left child exists and is greater than the current largest, update largest
        if (left < size && heap[left] > heap[largest]) {
            largest = left;
        }

        // If the right child exists and is greater than the current largest, update largest
        if (right < size && heap[right] > heap[largest]) {
            largest = right;
        }

        // If largest is not the current index, swap and recursively heapify down
        if (largest != index) {
            swap(index, largest);
            heapifyDown(largest);  // Reheapify the affected subtree
        }
    }

    // Method to remove the maximum element (root of the heap) and reheapify
    public void deleteMax() {
        if (size == 0) {
            System.out.println("Heap is empty, cannot delete.");
            return;
        }

        // Replace the root (maximum value) with the last element
        heap[0] = heap[size - 1];
        size--;  // Decrease the size of the heap

        // Reheapify the heap from the root to restore the max-heap property
        heapifyDown(0);
    }

    // Method to insert a new value into the heap
    public void insert(int value) {
        if (size == capacity) {
            System.out.println("Heap is full, cannot insert");
            return;
        }

        // Insert the new value at the end of the heap
        heap[size] = value;
        size++;

        // Heapify the value up to maintain the max-heap property
        heapifyUp(size - 1);
    }

    // Method to heapify the value up to maintain the max-heap property
    private void heapifyUp(int index) {
        // While the current node is greater than its parent, swap them
        while (index > 0 && heap[parent(index)] < heap[index]) {
            swap(index, parent(index));
            index = parent(index); // Move to the parent's index
        }
    }

    // Method to display the heap elements
    public void display() {
        System.out.println(Arrays.toString(Arrays.copyOfRange(heap, 0, size)));
    }

    public static void main(String[] args) {
        MaxHeap maxHeap = new MaxHeap(10);  // Create a Max Heap with capacity 10

        // Insert elements into the Max Heap
        maxHeap.insert(10);
        maxHeap.insert(20);
        maxHeap.insert(15);
        maxHeap.insert(30);
        maxHeap.insert(25);

        // Display the heap before deleting the maximum element
        System.out.println("Heap before deleting maximum value:");
        maxHeap.display();

        // Now delete the maximum element (root) and reheapify
        maxHeap.deleteMax();

        // Display the heap after deleting the maximum element
        System.out.println("Heap after deleting maximum value:");
        maxHeap.display();
    }
}
```

Output:-



```
Heap before deleting maximum value:
[30, 25, 15, 10, 20]
Heap after deleting maximum value:
[25, 20, 15, 10]
```

53) Write a program to insert 20 in max heap.

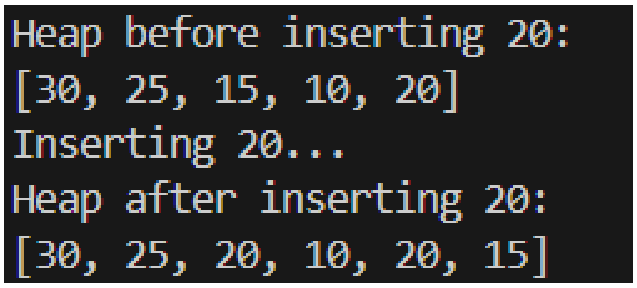
Topic: Max Heap

Code:-

```
import java.util.Arrays;

public class MaxHeap {
    private int[] heap;  // Array to store the heap
    private int size;    // Number of elements in the heap
    private int capacity; // Maximum capacity of the heap

    // Constructor to initialize the heap
    public MaxHeap(int capacity) {
        this.capacity = capacity;
        this.size = 0;
        heap = new int[capacity];  // Create an array to store heap elements
    }

    // Method to get the index of the parent of a node
    private int parent(int index) {
        return (index - 1) / 2;
    }

    // Method to get the index of the left child of a node
    private int leftChild(int index) {
        return 2 * index + 1;
    }

    // Method to get the index of the right child of a node
    private int rightChild(int index) {
        return 2 * index + 2;
    }

    // Method to swap two elements in the heap
    private void swap(int i, int j) {
        int temp = heap[i];
        heap[i] = heap[j];
        heap[j] = temp;
    }

    // Method to heapify (rearrange) the heap to maintain max-heap property
    private void heapifyUp(int index) {
        // While the current node is greater than its parent, swap them
        while (index > 0 && heap[parent(index)] < heap[index]) {
            swap(index, parent(index));
            index = parent(index); // Move to the parent's index
        }
    }

    // Method to insert a new value into the heap
    public void insert(int value) {
        // If heap is full, print a message and return
        if (size == capacity) {
            System.out.println("Heap is full, cannot insert");
            return;
        }

        // Insert the new value at the end of the heap
        heap[size] = value;
        size++;

        // Heapify the value up to maintain the max-heap property
        heapifyUp(size - 1);
    }

    // Method to display the heap elements
    public void display() {
        System.out.println(Arrays.toString(Arrays.copyOfRange(heap, 0, size)));
    }

    public static void main(String[] args) {
        MaxHeap maxHeap = new MaxHeap(10);  // Create a Max Heap with capacity 10

        // Insert elements into the Max Heap
        maxHeap.insert(10);
        maxHeap.insert(20);
        maxHeap.insert(15);
        maxHeap.insert(30);
        maxHeap.insert(25);

        // Display the heap after insertion
        System.out.println("Heap before inserting 20:");
        maxHeap.display();

        // Now insert 20 into the Max Heap
        System.out.println("Inserting 20...");
        maxHeap.insert(20);

        // Display the heap after inserting 20
        System.out.println("Heap after inserting 20:");
        maxHeap.display();
    }
```

}

Output:-



```
Heap before inserting 20:
[30, 25, 15, 10, 20]
Inserting 20...
Heap after inserting 20:
[30, 25, 20, 10, 20, 15]
```

54) Insert following keys 5, 28, 19, 15, 20, 33, 12, 17 and 10 in hash table using chaining hashing method and find minimum, maximum and average chain length in hash table.

Topic: Hashing using Chaining method

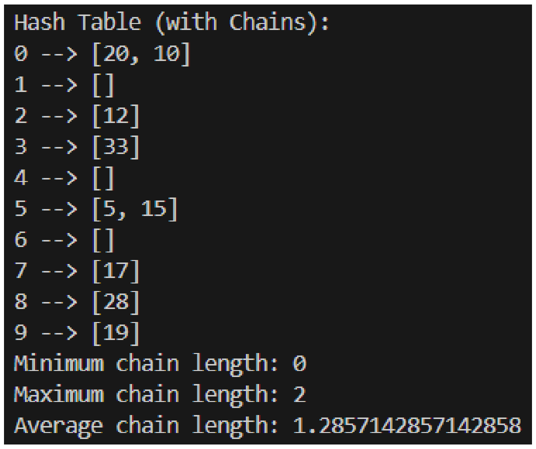
Code:-

```
import java.util.LinkedList;


public class ChainingHashTable {
    // Array to store the chains (linked lists)
    private LinkedList<Integer>[] table;
    // Size of the hash table
    private int size;

    // Constructor to initialize the hash table with a given size
    public ChainingHashTable(int size) {
        this.size = size;
        table = new LinkedList[size];  // Initialize the table with the given size
        for (int i = 0; i < size; i++) {
            table[i] = new LinkedList<>();  // Each index will hold a linked list
        }
    }
```

```
// Hash function to compute the index for a given key
private int hash(int key) {
    return key % size;  // Simple modulo operation to ensure the index fits within the table's size
}

// Method to insert a key into the hash table using chaining
public void insert(int key) {
    int hashIndex = hash(key);  // Calculate the index using the hash function
    table[hashIndex].add(key);  // Add the key to the corresponding chain (linked list)
}

// Method to calculate minimum, maximum, and average chain lengths
public void calculateChainStats() {
    int minLength = Integer.MAX_VALUE;
    int maxLength = Integer.MIN_VALUE;
    int totalLength = 0;
    int nonEmptyChains = 0;

    // Iterate through the table to calculate the lengths of the chains
    for (LinkedList<Integer> chain : table) {
        int chainLength = chain.size();  // Get the length of the current chain
        if (chainLength > 0) {
            nonEmptyChains++;
        }
        totalLength += chainLength;  // Add the chain length to the total length
        minLength = Math.min(minLength, chainLength);  // Find the minimum chain length
```

```
            maxLength = Math.max(maxLength, chainLength);  // Find the maximum chain length
        }

        // Calculate the average chain length
        double avgLength = (nonEmptyChains > 0) ? (double) totalLength / nonEmptyChains : 0;

        // Display the results
        System.out.println("Minimum chain length: " + minLength);
        System.out.println("Maximum chain length: " + maxLength);
        System.out.println("Average chain length: " + avgLength);
    }

    // Method to display the contents of the hash table
    public void display() {
        System.out.println("Hash Table (with Chains):");
        for (int i = 0; i < size; i++) {
            System.out.println(i + " --> " + table[i]);  // Print each chain (linked list) at the index
        }
    }

    // Main method to test the hash table implementation
    public static void main(String[] args) {
        // Create a hash table of size 10
        ChainingHashTable hashTable = new ChainingHashTable(10);

        // Array of elements to insert into the hash table
        int[] elements = {5, 28, 19, 15, 20, 33, 12, 17, 10};
        // Insert each element into the hash table
        for (int element : elements) {
            hashTable.insert(element);
        }

        // Display the contents of the hash table
        hashTable.display();

        // Calculate and display the minimum, maximum, and average chain lengths
        hashTable.calculateChainStats();
    }
}
```

Output:-



```
Hash Table (with Chains):
0 --> [20, 10]
1 --> []
2 --> [12]
3 --> [33]
4 --> []
5 --> [5, 15]
6 --> []
7 --> [17]
8 --> [28]
9 --> [19]
Minimum chain length: 0
Maximum chain length: 2
Average chain length: 1.2857142857142858
```

55) Write a program to display hash table after inserting elements 17, 16, 22, 36, 33, 46, 26, 144 into a hash table of size 10, using Double hashing, where h(x) = x mod 10, h_2(x) = x mod 6 + 1.

NOTE:
Double Hashing is an open addressing collision resolution technique in hash tables.
When a collision occurs, instead of probing linearly or quadratically, it uses a **second hash function** to calculate the interval between probes.

- Primary hash function: h(x) = x % table_size
- Secondary hash function: h_2(x) = (x % secondary_mod) + 1

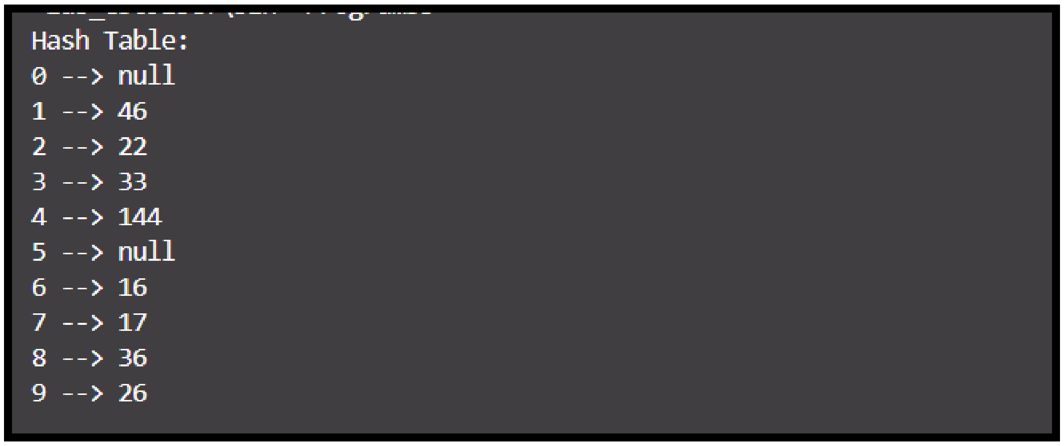
New index on collision = (h(x) + i * h_2(x)) % table_size, where i = 0, 1, 2, ...
This reduces clustering and gives better performance compared to linear and quadratic probing.

Code:-

```
public class Program55 {
   private Integer[] table;
   private int size;

   public Program55(int size) {
      this.size = size;
      table = new Integer[size];
   }

   private int hash1(int key) {
      return key % size;
   }

   private int hash2(int key) {
      return (key % 6) + 1; // secondary hash function
   }

   public void insert(int key) {
      int hashIndex = hash1(key);
      int stepSize = hash2(key);
      int i = 0;

      // Double hashing probing
      while (table[(hashIndex + i * stepSize) % size] != null) {
         i++;
         if (i == size) {
            System.out.println("Hash table is full, cannot insert key: " + key);
            return;
         }
      }
      table[(hashIndex + i * stepSize) % size] = key;
```

```
    }

    public void display() {
        System.out.println("Hash Table:");
        for (int i = 0; i < size; i++) {
            if (table[i] != null)
                System.out.println(i + " --> " + table[i]);
            else
                System.out.println(i + " --> " + "null");
        }
    }

    public static void main(String[] args) {
        Program55 hashTable = new Program55(10);

        int[] elements = {17, 16, 22, 36, 33, 46, 26, 144};
        for (int element : elements) {
            hashTable.insert(element);
        }

        hashTable.display();
    }
}
```

Output:-



```
Hash Table:
0 --> null
1 --> 46
2 --> 22
3 --> 33
4 --> 144
5 --> null
6 --> 16
7 --> 17
8 --> 36
9 --> 26
```

56) Write a program to display hash table after inserting elements 17, 16, 22, 36, 33, 46, 26, 144 into a hash table of size 10, using linear probing, where h(x) = x mod 10.

Linear probing:

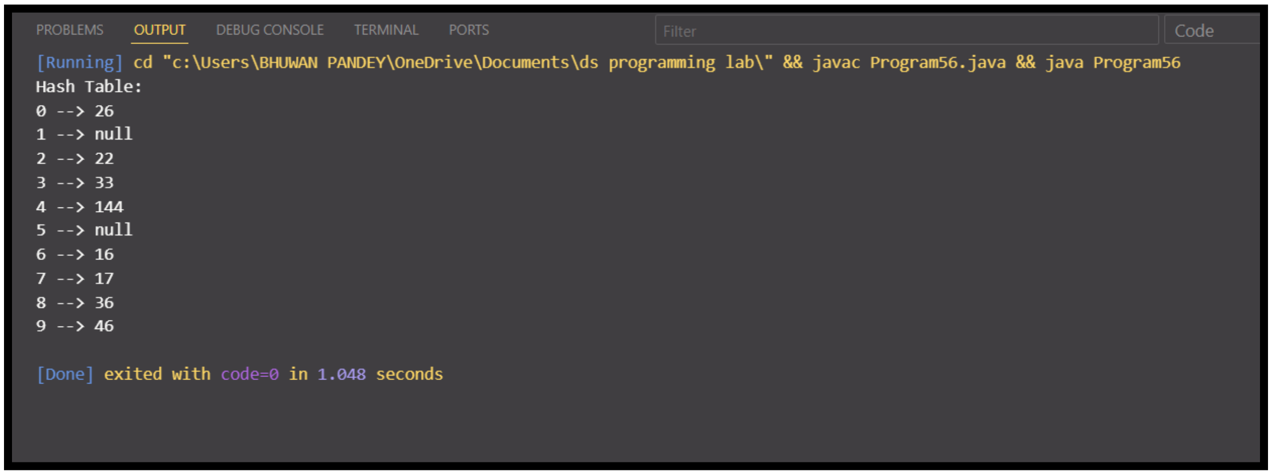
Linear probing is a **collision resolution technique** used in **hash tables**.
When two keys hash to the **same index**, linear probing **searches sequentially** (index +1, +2, etc.) for the **next empty slot** and inserts the element there.

- If collision happens at index i, it checks (i + 1) % table_size, (i + 2) % table_size, and so on.
- It wraps around the table if the end is reached.

Code:-

```
public class Program56 {
    private Integer[] table;
    private int size;

    public Program56(int size) {
        this.size = size;
        table = new Integer[size];
    }

    private int hash(int key) {
        return key % size;
    }

    public void insert(int key) {
        int hashIndex = hash(key);

        // Linear probing: move sequentially
        while (table[hashIndex] != null) {
            hashIndex = (hashIndex + 1) % size;
        }
        table[hashIndex] = key;
    }

    public void display() {
        System.out.println("Hash Table:");
        for (int i = 0; i < size; i++) {
            if (table[i] != null)
                System.out.println(i + " --> " + table[i]);
            else
                System.out.println(i + " --> " + "null");
        }
    }

    public static void main(String[] args) {
        Program56 hashTable = new Program56(10);

        int[] elements = {17, 16, 22, 36, 33, 46, 26, 144};
        for (int element : elements) {
            hashTable.insert(element);
        }

        hashTable.display();
    }
}
```

Output:-



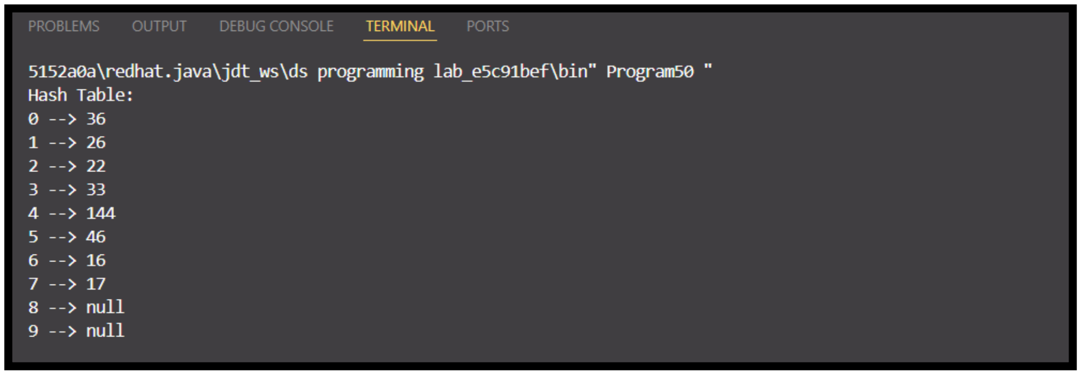
```
[Running] cd "c:\Users\BHUWAN PANDEY\OneDrive\Documents\ds programming lab\" && javac Program56.java && java Program56
Hash Table:
0 --> 26
1 --> null
2 --> 22
3 --> 33
4 --> 144
5 --> null
6 --> 16
7 --> 17
8 --> 36
9 --> 46

[Done] exited with code=0 in 1.048 seconds
```

57) Write a program to display hash table after inserting elements 17, 16, 22, 36, 33, 46, 26, 144 into a hash table of size 10, using quadratic probing, where h(x) = x mod 10.

Code:-

```
public class QuadraticProbingHashTable {
    private Integer[] table;
    private int size;

    public QuadraticProbingHashTable(int size) {
        this.size = size;
        table = new Integer[size];
    }

    private int hash(int key) {
        return key % size;
    }

    public void insert(int key) {
        int hashIndex = hash(key);
        int i = 0;

        while (table[(hashIndex + i * i) % size] != null) {
            i++;
            if (i == size) {
                System.out.println("Hash table is full, cannot insert key: " + key);
                return;
            }
        }
        table[(hashIndex + i * i) % size] = key;
    }

    public void display() {
        System.out.println("Hash Table:");
        for (int i = 0; i < size; i++) {
            if (table[i] != null)
                System.out.println(i + " --> " + table[i]);
            else
                System.out.println(i + " --> " + "null");
        }
    }

    public static void main(String[] args) {
        QuadraticProbingHashTable hashTable = new QuadraticProbingHashTable(10);
        int[] elements = {17, 16, 22, 36, 33, 46, 26, 144};
        for (int element : elements) {
            hashTable.insert(element);
        }

        hashTable.display();
    }
}
```

Output:-



```
5152a0a\redhat.java\jdt_ws\ds programming lab_e5c91bef\bin" Program50 "
Hash Table:
0 --> 36
1 --> 26
2 --> 22
3 --> 33
4 --> 144
5 --> 46
6 --> 16
7 --> 17
8 --> null
9 --> null
```