1.

/*DATE :
AUTHOR : Y.Ganesh***/**
AIM : Implementation of singly linked list

PROGRAM :

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

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

public class LinkedList {
    Node head;

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

    public void prepend(int data) {
        Node newNode = new Node(data);
        newNode.next = head;
        head = newNode;
    }

    public void delete(int data) {
        if (head == null) {
            return;
        }
```

```
        if (head.data == data) {
            head = head.next;
            return;
        }

        Node current = head;
        while (current.next != null) {
            if (current.next.data == data) {
                current.next = current.next.next;
                return;
            }
            current = current.next;
        }
    }

    public void display() {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " -> ");
            current = current.next;
        }
        System.out.println("null");
    }

    public static void main(String[] args) {
        LinkedList myList = new LinkedList();
        myList.append(1);
        myList.append(2);
        myList.append(3);
        myList.prepend(0);
        myList.display();

        myList.delete(2);
        myList.display();     }
}
```

OUTPUT :

0 -> 1 -> 2 -> 3 -> null
0 -> 1 -> 2 -> 3 -> null

2.
/*DATE :
AUTHOR : Y.Ganesh***/**
AIM : Implementation of doubly linked list.

PROGRAM :
```
class Node {
    int data;
    Node next;
    Node prev;

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

public class DoublyLinkedList {
    private Node head;
    private Node tail;

    // Append an element to the end of the list
    public void append(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            tail = newNode;
        } else {
            newNode.prev = tail;
            tail.next = newNode;
            tail = newNode;
        }
    }

    // Prepend an element to the beginning of the list
    public void prepend(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            tail = newNode;
        } else {
            newNode.next = head;
            head.prev = newNode;
```

```
        head = newNode;
    }
}

// Delete an element from the list
public void delete(int data) {
    Node current = head;
    while (current != null) {
        if (current.data == data) {
            if (current == head) {
                head = current.next;
                if (head != null) {
                    head.prev = null;
                }
            } else if (current == tail) {
                tail = current.prev;
                tail.next = null;
            } else {
                current.prev.next = current.next;
                current.next.prev = current.prev;
            }
            return;
        }
        current = current.next;
    }
}

// Display the elements of the list from head to tail
public void displayForward() {
    Node current = head;
    while (current != null) {
        System.out.print(current.data + " <-> ");
        current = current.next;
    }
    System.out.println("null");
}

// Display the elements of the list from tail to head
public void displayBackward() {
    Node current = tail;
    while (current != null) {
        System.out.print(current.data + " <-> ");
        current = current.prev;
    }
```

```
        System.out.println("null");
    }

    public static void main(String[] args) {
        DoublyLinkedList myList = new DoublyLinkedList();

        // Append elements to the list
        myList.append(1);
        myList.append(2);
        myList.append(3);

        // Prepend an element to the list
        myList.prepend(0);

        // Display the list from head to tail
        myList.displayForward();

        // Display the list from tail to head
        myList.displayBackward();

        // Delete an element from the list
        myList.delete(2);

        // Display the updated list
        myList.displayForward();
    }
}
```

OUTPUT :

```
0 <-> 1 <-> 2 <-> 3 <-> null
3 <-> 2 <-> 1 <-> 0 <-> null
0 <-> 1 <-> 3 <-> null
```

3.
/*DATE :
AUTHOR : Y.Ganesh***/**
AIM : Program to reverse the nodes in a circular linked list.

PROGRAM :

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

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

public class CircularLinkedList {
    private Node head;

    // Add a node to the end of the circular linked list
    public void append(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            newNode.next = head; // Point back to itself to create a circular list
        } else {
            Node current = head;
            while (current.next != head) {
                current = current.next;
            }
            current.next = newNode;
            newNode.next = head;
        }
    }

    // Display the circular linked list
    public void display() {
        Node current = head;
        do {
            System.out.print(current.data + " -> ");
            current = current.next;
        } while (current != head);
        System.out.println();
    }
```

```
    // Reverse the circular linked list
    public void reverse() {
        if (head == null || head.next == head) {
            return; // List is empty or has only one element, no need to reverse
        }

        Node current = head;
        Node prev = null;
        Node nextNode = null;

        do {
            nextNode = current.next;
            current.next = prev;
            prev = current;
            current = nextNode;
        } while (current != head);

        // Update circular connections to maintain the circular structure
        head.next = prev;
        head = prev; // Update the head to the new last element
    }


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

        // Append elements to the circular linked list
        list.append(1);
        list.append(2);
        list.append(3);
        list.append(4);

        System.out.println("Original Circular Linked List:");
        list.display();

        list.reverse();

        System.out.println("Reversed Circular Linked List:");
        list.display();
    }
}
```
OUTPUT :

Original Circular Linked List:

1 -> 2 -> 3 -> 4 ->
Reversed Circular Linked List:
4 -> 3 -> 2 -> 1 ->

4.
/*DATE :
AUTHOR : Y.Ganesh***/**
AIM : Program to perform operations on two polynomials using linked list.

PROGRAM :

```
import java.util.*;

// 1st Number: 5x^2+4x^1+2x^0
// 2nd Number: -5x^1-5x^0
public class Polynomial {

        // Driver code
        public static void main(String args[])
        {
                // 1st Number: 5x^2+4x^1+2x^0
                Node poly1 = new Node(5, 2);
                append(poly1, 4, 1);
                append(poly1, 2, 0);

                // 2nd Number: -5x^1-5x^0
                Node poly2 = new Node(-5, 1);
                append(poly2, -5, 0);

                Node sum = addPolynomial(poly1, poly2);
                for (Node ptr = sum; ptr != null; ptr = ptr.next) {
                        // printing polynomial
                        System.out.print(ptr.coeff + "x^"
                                                        + ptr.pow);
                        if (ptr.next != null)
                                System.out.print(" + ");
                }
                System.out.println();
        }

        // insert in linked list
        public static void append(Node head, int coeff,
                                                        int power)
        {
```

```
        Node new_node = new Node(coeff, power);
        while (head.next != null) {
            head = head.next;
        }
        head.next = new_node;
    }

    /* The below method print the required sum of polynomial
    p1 and p2 as specified in output */
    public static Node addPolynomial(Node p1, Node p2)
    {
        Node res = new Node(
            0, 0); // dummy node ...head of resultant list
        Node prev
            = res; // pointer to last node of resultant list
        while (p1 != null && p2 != null) {
            if (p1.pow < p2.pow) {
                prev.next = p2;
                prev = p2;
                p2 = p2.next;
            }
            else if (p1.pow > p2.pow) {
                prev.next = p1;
                prev = p1;
                p1 = p1.next;
            }
            else {
                p1.coeff = p1.coeff + p2.coeff;
                prev.next = p1;
                prev = p1;
                p1 = p1.next;
                p2 = p2.next;
            }
        }
        if (p1 != null) {
            prev.next = p1;
        }
        else if (p2 != null) {
            prev.next = p2;
        }
        return res.next;
    }
}
```

```
/* Link list Node */
class Node {
		public int coeff;
		public int pow;
		public Node next;

		public Node(int c, int p)
		{
			this.coeff = c;
			this.pow = p;
			this.next = null;
		}
}
```

OUTPUT :

5x^2 + -1x^1 + -3x^0

5.

/*DATE :
AUTHOR : Y.Ganesh***/**
AIM : Implement traversal techniques in binary tree

PROGRAM :

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

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

public class BinaryTree {
    Node root;

    public BinaryTree() {
        root = null;
    }

    // Inorder Traversal (Left - Root - Right)
```

```
public void inorderTraversal(Node node) {
    if (node == null)
        return;

    inorderTraversal(node.left);
    System.out.print(node.data + " ");
    inorderTraversal(node.right);
}

// Preorder Traversal (Root - Left - Right)
public void preorderTraversal(Node node) {
    if (node == null)
        return;

    System.out.print(node.data + " ");
    preorderTraversal(node.left);
    preorderTraversal(node.right);
}

// Postorder Traversal (Left - Right - Root)
public void postorderTraversal(Node node) {
    if (node == null)
        return;

    postorderTraversal(node.left);
    postorderTraversal(node.right);
    System.out.print(node.data + " ");
}

public static void main(String[] args) {
    BinaryTree tree = new BinaryTree();
    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("Inorder Traversal:");
    tree.inorderTraversal(tree.root);

    System.out.println("\nPreorder Traversal:");
    tree.preorderTraversal(tree.root);

    System.out.println("\n Postorder Traversal:");
```

```
        tree.postorderTraversal(tree.root);
    }
}
```

OUTPUT :
Inorder Traversal:
4 2 5 1 3
Preorder Traversal:
1 2 4 5 3
Postorder Traversal:
4 5 2 3 1

6.
/*DATE :
AUTHOR : Y.Ganesh***/**
AIM :
Beginning with an empty binary search tree, construct binary search tree by inserting the values in the order given. After constructing a binary tree-

- ➢ Insert new node
- ➢ Find number of nodes in longest
- ➢ Minimum data value found in the tree
- ➢ Change a tree so that the roles of the left and right pointers are wrapped at every node.
- ➢ Search a value

PROGRAM :

```
class TreeNode {
    int data;
    TreeNode left;
    TreeNode right;

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

public class BinarySearchTree {
    private TreeNode root;

    public BinarySearchTree() {
        root = null;
    }
```

```
public void insert(int data) {
    root = insertRec(root, data);
}

private TreeNode insertRec(TreeNode root, int data) {
    if (root == null) {
        root = new TreeNode(data);
        return root;
    }

    if (data < root.data) {
        root.left = insertRec(root.left, data);
    } else if (data > root.data) {
        root.right = insertRec(root.right, data);
    }

    return root;
}

public int findLongestPath() {
    return findLongestPathRec(root);
}

private int findLongestPathRec(TreeNode root) {
    if (root == null) {
        return 0;
    }

    int leftPath = findLongestPathRec(root.left);
    int rightPath = findLongestPathRec(root.right);

    return Math.max(leftPath, rightPath) + 1;
}

public int findMinValue() {
    return findMinValueRec(root);
}

private int findMinValueRec(TreeNode root) {
    if (root == null) {
        throw new IllegalStateException("Tree is empty");
    }

    if (root.left == null) {
```

```
        return root.data;
    }

    return findMinValueRec(root.left);
}

public void invertTree() {
    root = invertTreeRec(root);
}

private TreeNode invertTreeRec(TreeNode root) {
    if (root == null) {
        return null;
    }

    TreeNode left = invertTreeRec(root.left);
    TreeNode right = invertTreeRec(root.right);

    root.left = right;
    root.right = left;

    return root;
}

public boolean search(int data) {
    return searchRec(root, data);
}

private boolean searchRec(TreeNode root, int data) {
    if (root == null) {
        return false;
    }

    if (root.data == data) {
        return true;
    }

    if (data < root.data) {
        return searchRec(root.left, data);
    } else {
        return searchRec(root.right, data);
    }
}
```

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

        // Insert values
        bst.insert(50);
        bst.insert(30);
        bst.insert(70);
        bst.insert(20);
        bst.insert(40);

        // Find the number of nodes in the longest path
        int longestPath = bst.findLongestPath();
        System.out.println("Number of nodes in the longest path: " + longestPath);

        // Find the minimum value in the tree
        int minValue = bst.findMinValue();
        System.out.println("Minimum value in the tree: " + minValue);

        // Invert the tree
        bst.invertTree();

        // Search for a value
        boolean found = bst.search(50);
        System.out.println("Value 40 found in the tree: " + found);
    }
}
```

OUTPUT :

Number of nodes in the longest path: 3
Minimum value in the tree: 20
Value 40 found in the tree: true

7.

/*DATE :
AUTHOR : Y.Ganesh***/**
AIM : Write a program to perform the following operations

- ➢ Insertion into an AVL-tree
- ➢ Deletion from an AVL-tree

PROGRAM :

```
class Node {
    int key, height;
```

```
    Node left, right;

    Node(int d) {
        key = d;
        height = 1;
    }
}

class AVLTree {
    Node root;

    // Get height of a node
    int height(Node N) {
        if (N == null)
            return 0;
        return N.height;
    }

    // Get balance factor of a node
    int getBalance(Node N) {
        if (N == null)
            return 0;
        return height(N.left) - height(N.right);
    }

    // Right rotate subtree rooted with y
    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 new root
        return x;
    }

    // Left rotate subtree rooted with x
    Node leftRotate(Node x) {
```

```
    Node y = x.right;
    Node T2 = y.left;

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

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

    // Return new root
    return y;
}

// Insert a key into the AVL tree
Node insert(Node node, int key) {
    // Perform the normal BST insertion
    if (node == null)
        return (new Node(key));

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

    // Update height of the current node
    node.height = 1 + Math.max(height(node.left), height(node.right));

    // Get the balance factor of this node to check whether it became unbalanced
    int balance = getBalance(node);

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

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

    // Left Right Case
    if (balance > 1 && key > node.left.key) {
```

```
        node.left = leftRotate(node.left);
        return rightRotate(node);
    }

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

    // No rotation needed
    return node;
}

// Delete a key from the AVL tree
Node delete(Node root, int key) {
    // Perform standard BST delete
    if (root == null)
        return root;

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

            // No child case
            if (temp == null) {
                temp = root;
                root = null;
            } else // One child case
                root = temp; // Copy the contents of the non-empty child

        } else {
            // Node with two children: Get the inorder successor (smallest
            // in the right subtree)
            Node temp = minValueNode(root.right);
```

```
            // Copy the inorder successor's data to this node
            root.key = temp.key;

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

    // If the tree had only one node then return
    if (root == null)
        return root;

    // Update height of the current node
    root.height = Math.max(height(root.left), height(root.right)) + 1;

    // Get the balance factor of this node to check whether it became unbalanced
    int balance = getBalance(root);

    // Left Left Case
    if (balance > 1 && getBalance(root.left) >= 0)
        return rightRotate(root);

    // Left Right Case
    if (balance > 1 && getBalance(root.left) < 0) {
        root.left = leftRotate(root.left);
        return rightRotate(root);
    }

    // Right Right Case
    if (balance < -1 && getBalance(root.right) <= 0)
        return leftRotate(root);

    // Right Left Case
    if (balance < -1 && getBalance(root.right) > 0) {
        root.right = rightRotate(root.right);
        return leftRotate(root);
    }

    return root;
}

// Get the node with the smallest key value in the tree
Node minValueNode(Node node) {
```

```
        Node current = node;

        // Find the leftmost leaf
        while (current.left != null)
            current = current.left;

        return current;
    }

    // A utility function to print preorder traversal of the tree
    void preOrder(Node node) {
        if (node != null) {
            System.out.print(node.key + " ");
            preOrder(node.left);
            preOrder(node.right);
        }
    }

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

        /* Example usage */
        tree.root = tree.insert(tree.root, 10);
        tree.root = tree.insert(tree.root, 20);
        tree.root = tree.insert(tree.root, 30);
        tree.root = tree.insert(tree.root, 40);
        tree.root = tree.insert(tree.root, 50);
        tree.root = tree.insert(tree.root, 25);

        System.out.println("Preorder traversal after insertion:");
        tree.preOrder(tree.root);

        tree.root = tree.delete(tree.root, 30);

        System.out.println("\nPreorder traversal after deletion:");
        tree.preOrder(tree.root);
    }
}
```

OUTPUT :

Preorder traversal after insertion:
30 20 10 25 40 50
Preorder traversal after deletion:
30 20 10 25 50

8.
/*DATE :
AUTHOR : Y.Ganesh*/
AIM : Program to implement priority queue using Heap
➢ Inserting new element
➢ Deletion of minimum element

PROGRAM :
```
import java.util.ArrayList;

public class MinHeap {
    private ArrayList<Integer> heap;

    public MinHeap() {
        heap = new ArrayList<>();
    }

    private int parent(int i) {
        return (i - 1) / 2;
    }

    private int leftChild(int i) {
        return 2 * i + 1;
    }

    private int rightChild(int i) {
        return 2 * i + 2;
    }

    public void insert(int key) {
        heap.add(key);
        heapifyUp(heap.size() - 1);
    }

    private void heapifyUp(int i) {
        while (i > 0 && heap.get(i) < heap.get(parent(i))) {
            swap(i, parent(i));
            i = parent(i);
        }
    }

    public int deleteMin() {
        if (heap.isEmpty()) {
            return -1; // Priority queue is empty
```

```java
    }

    if (heap.size() == 1) {
        return heap.remove(0);
    }

    int root = heap.get(0);
    heap.set(0, heap.remove(heap.size() - 1));
    heapifyDown(0);
    return root;
}

private void heapifyDown(int i) {
    int left = leftChild(i);
    int right = rightChild(i);
    int smallest = i;

    if (left < heap.size() && heap.get(left) < heap.get(smallest)) {
        smallest = left;
    }

    if (right < heap.size() && heap.get(right) < heap.get(smallest)) {
        smallest = right;
    }

    if (smallest != i) {
        swap(i, smallest);
        heapifyDown(smallest);
    }
}

private void swap(int i, int j) {
    int temp = heap.get(i);
    heap.set(i, heap.get(j));
    heap.set(j, temp);
}

public static void main(String[] args) {
    MinHeap minHeap = new MinHeap();

    minHeap.insert(5);
    minHeap.insert(3);
    minHeap.insert(7);
    minHeap.insert(2);
```

```
        minHeap.insert(8);

        System.out.println("Minimum element: " + minHeap.deleteMin());  // Output: 2
        System.out.println("Minimum element: " + minHeap.deleteMin());  // Output: 3
        System.out.println("Minimum element: " + minHeap.deleteMin());  // Output: 5
    }
}
```

OUTPUT :
Minimum element: 2
Minimum element: 3
Minimum element: 5

9.
/*DATE :
AUTHOR : Y.Ganesh***/**
AIM : Write a program to implement DFS and BFS traversals.
PROGRAM :

```
import java.util.*;

class Graph {
    private int vertices;
    private LinkedList<Integer>[] adjacencyList;

    public Graph(int vertices) {
        this.vertices = vertices;
        this.adjacencyList = new LinkedList[vertices];
        for (int i = 0; i < vertices; i++) {
            this.adjacencyList[i] = new LinkedList<>();
        }
    }

    public void addEdge(int v, int w) {
        adjacencyList[v].add(w);
    }

    // Depth-First Search (DFS) traversal
    public void dfs(int startVertex) {
        boolean[] visited = new boolean[vertices];
        dfsUtil(startVertex, visited);
    }

    private void dfsUtil(int vertex, boolean[] visited) {
```

```
        visited[vertex] = true;
        System.out.print(vertex + " ");

        for (Integer neighbor : adjacencyList[vertex]) {
            if (!visited[neighbor]) {
                dfsUtil(neighbor, visited);
            }
        }
    }

    // Breadth-First Search (BFS) traversal
    public void bfs(int startVertex) {
        boolean[] visited = new boolean[vertices];
        Queue<Integer> queue = new LinkedList<>();

        visited[startVertex] = true;
        queue.add(startVertex);

        while (!queue.isEmpty()) {
            int vertex = queue.poll();
            System.out.print(vertex + " ");

            for (Integer neighbor : adjacencyList[vertex]) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    queue.add(neighbor);
                }
            }
        }
    }

    public static void main(String[] args) {
        Graph graph = new Graph(5);

        // Adding edges to the graph
        graph.addEdge(0, 1);
        graph.addEdge(0, 2);
        graph.addEdge(1, 3);
        graph.addEdge(1, 4);

        System.out.println("Depth-First Search (DFS) traversal starting from vertex 0:");
        graph.dfs(0);

        System.out.println("\n\nBreadth-First Search (BFS) traversal starting from vertex 0:");
```

```
        graph.bfs(0);
    }
}
```

OUTPUT :

Depth-First Search (DFS) traversal starting from vertex 0:
0 1 3 4 2

Breadth-First Search (BFS) traversal starting from vertex 0:
0 1 2 3 4

10.

/*DATE :

AUTHOR : Y.Ganesh***/**

AIM : Write a program to find the minimum spanning tree using Prim's Algorithm.

PROGRAM :

```
import java.util.InputMismatchException;

import java.util.Scanner;


public class Prims

{

  private boolean unsettled[];

  private boolean settled[];

  private int numberofvertices;

  private int adjacencyMatrix[][];

  private int key[];

  public static final int INFINITE = 999;

  private int parent[];


  public Prims(int numberofvertices)

  {

     this.numberofvertices = numberofvertices;

     unsettled = new boolean[numberofvertices + 1];

     settled = new boolean[numberofvertices + 1];
```

```
    adjacencyMatrix = new int[numberofvertices + 1][numberofvertices + 1];
    key = new int[numberofvertices + 1];
    parent = new int[numberofvertices + 1];
}

public int getUnsettledCount(boolean unsettled[])
{
    int count = 0;
    for (int index = 0; index < unsettled.length; index++)
    {
        if (unsettled[index])
        {
            count++;
        }
    }
    return count;
}

public void primsAlgorithm(int adjacencyMatrix[][])
{
    int evaluationVertex;
    for (int source = 1; source <= numberofvertices; source++)
    {
        for (int destination = 1; destination <= numberofvertices; destination++)
        {
            this.adjacencyMatrix[source][destination] = adjacencyMatrix[source][destination];
        }
    }
```

```
    for (int index = 1; index <= numberofvertices; index++)
    {
        key[index] = INFINITE;
    }
    key[1] = 0;
    unsettled[1] = true;
    parent[1] = 1;

    while (getUnsettledCount(unsettled) != 0)
    {
        evaluationVertex = getMimumKeyVertexFromUnsettled(unsettled);
        unsettled[evaluationVertex] = false;
        settled[evaluationVertex] = true;
        evaluateNeighbours(evaluationVertex);
    }
}

private int getMimumKeyVertexFromUnsettled(boolean[] unsettled2)
{
    int min = Integer.MAX_VALUE;
    int node = 0;
    for (int vertex = 1; vertex <= numberofvertices; vertex++)
    {
        if (unsettled[vertex] == true && key[vertex] < min)
        {
            node = vertex;
            min = key[vertex];
        }
    }
```

```
    return node;
}

public void evaluateNeighbours(int evaluationVertex)
{

    for (int destinationvertex = 1; destinationvertex <= numberofvertices; destinationvertex++)
    {
        if (settled[destinationvertex] == false)
        {
            if (adjacencyMatrix[evaluationVertex][destinationvertex] != INFINITE)
            {
                if (adjacencyMatrix[evaluationVertex][destinationvertex] < key[destinationvertex])
                {
                    key[destinationvertex] = adjacencyMatrix[evaluationVertex][destinationvertex];
                    parent[destinationvertex] = evaluationVertex;
                }
                unsettled[destinationvertex] = true;
            }
        }
    }
}

public void printMST()
{
    System.out.println("SOURCE  : DESTINATION = WEIGHT");
    for (int vertex = 2; vertex <= numberofvertices; vertex++)
    {
```

```
            System.out.println(parent[vertex] + "\t:\t" + vertex +"\t=\t"+
adjacencyMatrix[parent[vertex]][vertex]);
        }
    }

    public static void main(String... arg)
    {
        int adjacency_matrix[][];
        int number_of_vertices;
        Scanner scan = new Scanner(System.in);

        try
        {
            System.out.println("Enter the number of vertices");
            number_of_vertices = scan.nextInt();
            adjacency_matrix = new int[number_of_vertices + 1][number_of_vertices + 1];

            System.out.println("Enter the Weighted Matrix for the graph");
            for (int i = 1; i <= number_of_vertices; i++)
            {
                for (int j = 1; j <= number_of_vertices; j++)
                {
                    adjacency_matrix[i][j] = scan.nextInt();
                    if (i == j)
                    {
                        adjacency_matrix[i][j] = 0;
                        continue;
                    }
                    if (adjacency_matrix[i][j] == 0)
```

```
                {
                    adjacency_matrix[i][j] = INFINITE;
                }
            }
        }

        Prims prims = new Prims(number_of_vertices);
        prims.primsAlgorithm(adjacency_matrix);
        prims.printMST();

    } catch (InputMismatchException inputMismatch)
    {
        System.out.println("Wrong Input Format");
    }
    scan.close();
  }
}
```

OUTPUT :

```
Enter the number of vertices
5
Enter the Weighted Matrix for the graph
0 4 0 0 5
4 0 3 6 1
0 3 0 6 2
0 6 6 0 7
5 1 2 7 0
SOURCE  : DESTINATION   =   WEIGHT
1      :     2       =      4
5      :     3       =      2
```

2 : 4 = 6

2 : 5 = 1

11.
/*DATE :
AUTHOR : Y.Ganesh***/**
AIM : Write a program to find the minimum spanning tree using Kruskal's Algorithm.

PROGRAM :
```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;

public class KruskalsMST {

    // Defines edge structure
    static class Edge {
        int src, dest, weight;

        public Edge(int src, int dest, int weight)
        {
            this.src = src;
            this.dest = dest;
            this.weight = weight;
        }
    }

    // Defines subset element structure
    static class Subset {
        int parent, rank;

        public Subset(int parent, int rank)
        {
            this.parent = parent;
            this.rank = rank;
        }
    }

    // Starting point of program execution
    public static void main(String[] args)
    {
        int V = 4;
        List<Edge> graphEdges = new ArrayList<Edge>(
```

```java
        List.of(new Edge(0, 1, 10), new Edge(0, 2, 6),
                new Edge(0, 3, 5), new Edge(1, 3, 15),
                new Edge(2, 3, 4)));

    // Sort the edges in non-decreasing order
    // (increasing with repetition allowed)
    graphEdges.sort(new Comparator<Edge>() {
        @Override public int compare(Edge o1, Edge o2)
        {
            return o1.weight - o2.weight;
        }
    });

    kruskals(V, graphEdges);
}

// Function to find the MST
private static void kruskals(int V, List<Edge> edges)
{
    int j = 0;
    int noOfEdges = 0;

    // Allocate memory for creating V subsets
    Subset subsets[] = new Subset[V];

    // Allocate memory for results
    Edge results[] = new Edge[V];

    // Create V subsets with single elements
    for (int i = 0; i < V; i++) {
        subsets[i] = new Subset(i, 0);
    }

    // Number of edges to be taken is equal to V-1
    while (noOfEdges < V - 1) {

        // Pick the smallest edge. And increment
        // the index for next iteration
        Edge nextEdge = edges.get(j);
        int x = findRoot(subsets, nextEdge.src);
        int y = findRoot(subsets, nextEdge.dest);

        // If including this edge doesn't cause cycle,
        // include it in result and increment the index
```

```
        // of result for next edge
        if (x != y) {
            results[noOfEdges] = nextEdge;
            union(subsets, x, y);
            noOfEdges++;
        }

        j++;
    }

    // Print the contents of result[] to display the
    // built MST
    System.out.println(
        "Following are the edges of the constructed MST:");
    int minCost = 0;
    for (int i = 0; i < noOfEdges; i++) {
        System.out.println(results[i].src + " -- "
                           + results[i].dest + " == "
                           + results[i].weight);
        minCost += results[i].weight;
    }
    System.out.println("Total cost of MST: " + minCost);
}

// Function to unite two disjoint sets
private static void union(Subset[] subsets, int x,
                          int y)
{
    int rootX = findRoot(subsets, x);
    int rootY = findRoot(subsets, y);

    if (subsets[rootY].rank < subsets[rootX].rank) {
        subsets[rootY].parent = rootX;
    }
    else if (subsets[rootX].rank
             < subsets[rootY].rank) {
        subsets[rootX].parent = rootY;
    }
    else {
        subsets[rootY].parent = rootX;
        subsets[rootX].rank++;
    }
}
```

```
    // Function to find parent of a set
    private static int findRoot(Subset[] subsets, int i)
    {
        if (subsets[i].parent == i)
            return subsets[i].parent;

        subsets[i].parent
            = findRoot(subsets, subsets[i].parent);
        return subsets[i].parent;
    }
}
```

OUTPUT :

Following are the edges of the constructed MST:
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Total cost of MST: 19

12.

/*DATE :
AUTHOR : Y.Ganesh***/**
AIM : Write a program to implement topological sort.

PROGRAM :

```
import java.util.*;

public class TopologicalSort {
    private int V; // Number of vertices
    private LinkedList<Integer> adjList[];

    public TopologicalSort(int v) {
        V = v;
        adjList = new LinkedList[v];
        for (int i = 0; i < v; ++i)
            adjList[i] = new LinkedList();
    }

    // Function to add an edge to the graph
    public void addEdge(int v, int w) {
        adjList[v].add(w);
    }

    // A recursive function used by topologicalSort
    private void topologicalSortUtil(int v, boolean visited[], Stack<Integer> stack) {
```

```
    // Mark the current node as visited
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    for (Integer neighbor : adjList[v]) {
        if (!visited[neighbor]) {
            topologicalSortUtil(neighbor, visited, stack);
        }
    }

    // Push current vertex to stack which stores the result
    stack.push(v);
}

// The function to do Topological Sort. It uses recursive
// topologicalSortUtil()
public void topologicalSort() {
    Stack<Integer> stack = new Stack<>();

    // Mark all the vertices as not visited
    boolean visited[] = new boolean[V];
    Arrays.fill(visited, false);

    // Call the recursive helper function to store Topological
    // Sort starting from all vertices one by one
    for (int i = 0; i < V; i++) {
        if (!visited[i]) {
            topologicalSortUtil(i, visited, stack);
        }
    }

    // Print contents of the stack
    System.out.println("Topological Sort:");
    while (!stack.isEmpty()) {
        System.out.print(stack.pop() + " ");
    }
}

public static void main(String args[]) {
    TopologicalSort g = new TopologicalSort(6);
    g.addEdge(5, 2);
    g.addEdge(5, 0);
    g.addEdge(4, 0);
    g.addEdge(4, 1);
```

```
        g.addEdge(2, 3);
        g.addEdge(3, 1);

        g.topologicalSort();
    }
}
```

OUTPUT :

Topological Sort:
5 4 2 3 1 0

13.
/*DATE :
AUTHOR : Y.Ganesh***/**
AIM : Write a program for creating an Open Addressing Hash Table with linear probing and quadratic probing .

PROGRAM :

Implementing own Hash Table with Open Addressing Linear Probing

```
// Our own HashNode class
class HashNode {
	int key;
	int value;

	public HashNode(int key, int value) {
		this.key = key;
		this.value = value;
	}
}

// Our own Hashmap class
class HashMap {
	// hash element array
	int capacity;
	int size;
	HashNode[] arr;
	// dummy node
	HashNode dummy;

	public HashMap() {
		this.capacity = 20;
		this.size = 0;
		this.arr = new HashNode[this.capacity];
		// initialize with dummy node
```

```
		this.dummy = new HashNode(-1, -1);
	}

	// This implements hash function to find index for a key
	public int hashCode(int key) {
		return key % this.capacity;
	}

	// Function to add key value pair
	public void insertNode(int key, int value) {
		HashNode temp = new HashNode(key, value);
		// Apply hash function to find index for given key
		int hashIndex = hashCode(key);
		// find next free space
		while (this.arr[hashIndex] != null && this.arr[hashIndex].key != key &&
this.arr[hashIndex].key != -1) {
			hashIndex++;
			hashIndex %= this.capacity;
		}
		// if new node to be inserted, increase the current size
		if (this.arr[hashIndex] == null || this.arr[hashIndex].key == -1) {
			this.size++;
		}
		this.arr[hashIndex] = temp;
	}

	// Function to delete a key value pair
	public int deleteNode(int key) {
		// Apply hash function to find index for given key
		int hashIndex = hashCode(key);
		// finding the node with given key
		while (this.arr[hashIndex] != null) {
			// if node found
			if (this.arr[hashIndex].key == key) {
				HashNode temp = this.arr[hashIndex];
				// Insert dummy node here for further use
				this.arr[hashIndex] = this.dummy;
				// Reduce size
				this.size--;
				return temp.value;
			}
			hashIndex++;
			hashIndex %= this.capacity;
		}
```

```
			// If not found return -1
			return -1;
		}

		// Function to search the value for a given key
		public int get(int key) {
			// Apply hash function to find index for given key
			int hashIndex = hashCode(key);
			int counter = 0;
			// finding the node with given key
			while (this.arr[hashIndex] != null) {
				// If counter is greater than capacity to avoid infinite loop
				if (counter > this.capacity) {
					return -1;
				}
				// if node found return its value
				if (this.arr[hashIndex].key == key) {
					return this.arr[hashIndex].value;
				}
				hashIndex++;
				hashIndex %= this.capacity;
				counter++;
			}
			// If not found return 0
			return 0;
		}

		// Return current size
		public int sizeofMap() {
			return this.size;
		}

		// Return true if size is 0
		public boolean isEmpty() {
			return this.size == 0;
		}

		// Function to display the stored key value pairs
		public void display() {
			for (int i = 0; i < this.capacity; i++) {
				if (this.arr[i] != null && this.arr[i].key != -1) {
					System.out.println("key = " + this.arr[i].key + " value = " +
this.arr[i].value);
				}
```

```
        }
    }
}

public class Main {
    public static void main(String[] args) {
        HashMap h = new HashMap();
        h.insertNode(1, 1);
        h.insertNode(2, 2);
        h.insertNode(2, 3);
        h.display();
        System.out.println(h.sizeofMap());
        System.out.println(h.deleteNode(2));
        System.out.println(h.sizeofMap());
        System.out.println(h.isEmpty());
        System.out.println(h.get(2));
    }
}
```

OUTPUT :

```
key = 1  value = 1
key = 2  value = 3
2
3
1
0
0
```

Quadratic Probing in Hashing

```
// Java implementation of the Quadratic Probing

class GFG {

    // Function to print an array
    static void printArray(int arr[])
    {

        // Iterating and printing the array
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }
    }

    // Function to implement the
```

```
// quadratic probing
static void hashing(int table[], int tsize, int arr[],
                                        int N)
{

        // Iterating through the array
        for (int i = 0; i < N; i++) {

                // Computing the hash value
                int hv = arr[i] % tsize;

                // Insert in the table if there
                // is no collision
                if (table[hv] == -1)
                        table[hv] = arr[i];
                else {

                        // If there is a collision
                        // iterating through all
                        // possible quadratic values
                        for (int j = 0; j < tsize; j++) {

                                // Computing the new hash value
                                int t = (hv + j * j) % tsize;
                                if (table[t] == -1) {

                                        // Break the loop after
                                        // inserting the value
                                        // in the table
                                        table[t] = arr[i];
                                        break;
                                }
                        }
                }
        }

        printArray(table);
}

// Driver code
public static void main(String args[])
{
        int arr[] = { 50, 700, 76, 85, 92, 73, 101 };
        int N = 7;
```

```
        // Size of the hash table
        int L = 7;
        int hash_table[] = new int[L];

        // Initializing the hash table
        for (int i = 0; i < L; i++) {
            hash_table[i] = -1;
        }

        // Function call
        hashing(hash_table, L, arr, N);
    }
}
```

OUTPUT :

700 50 85 73 101 92 76

14.

/*DATE :

AUTHOR : Y.Ganesh***/**

AIM : Write a program to implement Naive Pattern Matching algorithm

.

PROGRAM :

```
public class NaivePatternMatching {
    // Function to perform naive pattern matching
    static void naivePatternMatch(String text, String pattern) {
        int textLength = text.length();
        int patternLength = pattern.length();

        // Iterate through the text
        for (int i = 0; i <= textLength - patternLength; i++) {
            int j;

            // Check for pattern match
            for (j = 0; j < patternLength; j++) {
                if (text.charAt(i + j) != pattern.charAt(j))
                    break;
            }

            // If the inner loop didn't break, it means a match was found
            if (j == patternLength) {
                System.out.println("Pattern found at index " + i);
            }
        }
```

```
    }

    public static void main(String[] args) {
        String text = "ABABDABACDABABCABAB";
        String pattern = "AB";

        System.out.println("Text: " + text);
        System.out.println("Pattern: " + pattern);
        System.out.println("Pattern Matching Results:");

        naivePatternMatch(text, pattern);
    }
}
```

OUTPUT :

Text: ABABDABACDABABCABAB
Pattern: AB
Pattern Matching Results:
Pattern found at index 0
Pattern found at index 2
Pattern found at index 5
Pattern found at index 10
Pattern found at index 12
Pattern found at index 15
Pattern found at index 17

15.

/*DATE :
AUTHOR : Y.Ganesh***/**
AIM : Write a program to identify the desired patterns with Knuth-Morris-Pratt (KMP) algorithm .

PROGRAM :

```
public class KMPAlgorithm {
    // Function to compute the LPS (Longest Prefix Suffix) array
    private static int[] computeLPSArray(String pattern) {
        int m = pattern.length();
        int[] lps = new int[m];
        int len = 0; // Length of the previous longest prefix suffix

        for (int i = 1; i < m; ) {
            if (pattern.charAt(i) == pattern.charAt(len)) {
                len++;
                lps[i] = len;
```

```
            i++;
        } else {
            if (len != 0) {
                len = lps[len - 1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
    return lps;
}

// Function to perform pattern matching using KMP algorithm
public static void KMPPatternMatch(String text, String pattern) {
    int n = text.length();
    int m = pattern.length();

    int[] lps = computeLPSArray(pattern);
    int i = 0; // Index for text[]
    int j = 0; // Index for pattern[]

    while (i < n) {
        if (pattern.charAt(j) == text.charAt(i)) {
            i++;
            j++;
        }

        if (j == m) {
            System.out.println("Pattern found at index " + (i - j));
            j = lps[j - 1];
        } else if (i < n && pattern.charAt(j) != text.charAt(i)) {
            if (j != 0) {
                j = lps[j - 1];
            } else {
                i++;
            }
        }
    }
}

public static void main(String[] args) {
    String text = "ABABDABACDABABCABAB";
    String pattern = "AB";
```

```
        System.out.println("Text: " + text);
        System.out.println("Pattern: " + pattern);
        System.out.println("Pattern Matching Results:");

        KMPPatternMatch(text, pattern);
    }
}
```

OUTPUT :
Text: ABABDABACDABABCABAB
Pattern: AB
Pattern Matching Results:
Pattern found at index 0
Pattern found at index 2
Pattern found at index 5
Pattern found at index 10
Pattern found at index 12
Pattern found at index 15
Pattern found at index 17

16.
/*DATE :
AUTHOR : Y.Ganesh***/**
AIM : Write a program to implement the Rabin Karp pattern matching algorithm.
.
PROGRAM :

```
public class RabinKarpAlgorithm {
    private static final int PRIME = 101; // A prime number to use for hashing

    // Function to perform Rabin-Karp pattern matching
    private static void rabinKarpPatternMatch(String text, String pattern) {
        int m = pattern.length();
        int n = text.length();
        int patternHash = hash(pattern, m);
        int textHash = hash(text.substring(0, m), m);

        for (int i = 0; i <= n - m; i++) {
            if (patternHash == textHash && checkEqual(text, pattern, i, i + m)) {
                System.out.println("Pattern found at index " + i);
            }

            if (i < n - m) {
```

```java
            textHash = recalculateHash(text, i, i + m, textHash, m);
        }
    }
}

// Function to calculate hash value for a substring
private static int hash(String str, int length) {
    int hashValue = 0;
    for (int i = 0; i < length; i++) {
        hashValue += str.charAt(i) * Math.pow(PRIME, i);
    }
    return hashValue;
}

// Function to recalculate hash value for the next substring
private static int recalculateHash(String str, int oldIndex, int newIndex, int oldHash, int
patternLength) {
    int newHash = oldHash - str.charAt(oldIndex);
    newHash /= PRIME;
    newHash += str.charAt(newIndex) * Math.pow(PRIME, patternLength - 1);
    return newHash;
}

// Function to check if substrings are equal
private static boolean checkEqual(String text, String pattern, int start, int end) {
    int patternIndex = 0;
    for (int i = start; i < end; i++) {
        if (text.charAt(i) != pattern.charAt(patternIndex)) {
            return false;
        }
        patternIndex++;
    }
    return true;
}

public static void main(String[] args) {
    String text = "ABABDABACDABABCABAB";
    String pattern = "AB";

    System.out.println("Text: " + text);
    System.out.println("Pattern: " + pattern);
    System.out.println("Pattern Matching Results:");

    rabinKarpPatternMatch(text, pattern);
```

```
    }
}
```

OUTPUT :

Text: ABABDABACDABABCABAB
Pattern: AB
Pattern Matching Results:
Pattern found at index 0
Pattern found at index 2
Pattern found at index 5
Pattern found at index 10
Pattern found at index 12
Pattern found at index 15
Pattern found at index 17