1. Write a program to perform the following operations:

a) Insert an element into a binary search tree.

b) Delete an element from a binary search tree.

c) Search for a key element in a binary search tree.

```
import java.util.Scanner;
class BinarySearchTree {
 class Node {
  int key;
  Node left, right;
  public Node(int item) {
   key = item;
   left = right = null;}}
 Node root;
 BinarySearchTree() {
  root = null;}
 void insert(int key) {
  root = insertKey(root, key);}
 int search(int key){
  root = searchkey(root,key);
  if (root == null){
   return 0; }
  else{
   return 1; }}
 Node searchkey(Node root,int key) {
    // Base Cases: root is null or key is present at
    // root
    if (root == null || root.key == key)
      return root;
    // Key is greater than root's key
    if (root.key < key)
      return searchkey(root.right, key);
    // Key is smaller than root's key
```

```
    return searchkey(root.left, key);}
 // Insert key in the tree
 Node insertKey(Node root, int key) {
  // Return a new node if the tree is empty
  if (root == null) {
   root = new Node(key);
   return root; }
  // Traverse to the right place and insert the node
  if (key < root.key)
   root.left = insertKey(root.left, key);
  else if (key > root.key)
   root.right = insertKey(root.right, key);
  return root; }
 void inorder() {
  inorderRec(root); }
 // Inorder Traversal
 void inorderRec(Node root) {
  if (root != null) {
   inorderRec(root.left);
   System.out.print(root.key + " -> ");
   inorderRec(root.right); }}
void deleteKey(int key) {
  root = deleteRec(root, key); }
 Node deleteRec(Node root, int key) {
  // Return if the tree is empty
  if (root == null)
   return root;
// Find the node to be deleted
  if (key < root.key)
   root.left = deleteRec(root.left, key);
  else if (key > root.key)
   root.right = deleteRec(root.right, key);
```

```
else {
  // If the node is with only one child or no child
  if (root.left == null)
    return root.right;
  else if (root.right == null)
    return root.left;
   // If the node has two children
  // Place the inorder successor in position of the node to be deleted
  root.key = minValue(root.right);
  // Delete the inorder successor
  root.right = deleteRec(root.right, root.key); }
return root;}
// Find the inorder successor
int minValue(Node root) {
 int minv = root.key;
 while (root.left != null) {
   minv = root.left.key;
   root = root.left;}
 return minv; }
public static void main(String[] args) {
 Scanner sc=new Scanner(System.in);
 BinarySearchTree tree = new BinarySearchTree();
 int n=8;
 int p,search_ele;
 int ins_ele,del_ele;
 int arr[]=new int[n];
 System.out.println("Enter the elements :");
 for(int i=0;i<n;i++){
   arr[i]=sc.nextInt();}
 //creating the tree
 for(int i=0;i<n;i++){
   tree.insert(arr[i]); }
```

```
System.out.print("Inorder traversal: ");
tree.inorder();
System.out.println("Enter the element for insertion");
ins_ele=sc.nextInt();
tree.insert(ins_ele);
System.out.println("After insertion");
tree.inorder();
System.out.println("Enter the element for deletion");
del_ele=sc.nextInt();
System.out.println("After deleting");
tree.deleteKey(del_ele);
System.out.print("Inorder traversal: ");
tree.inorder();
System.out.println("Enter the element for search");
search_ele=sc.nextInt();
p=tree.search(search_ele);
if (p == 1){
  System.out.println("Search Element found"}
else{
  System.out.println("Search Element Not found");}}}
```

Output:

```
Enter the elements :
2

4

1

5

6

7

8

3

Inorder traversal: 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> Enter the element for insertion
9

After insertion
1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> Enter the element for deletion
4

After deleting
Inorder traversal: 1 -> 2 -> 3 -> 5 -> 6 -> 7 -> 8 -> 9 -> Enter the element for search
```

2. Write a program for implementing the following sorting methods: a) Merge sort b) Heap sort c) Quick sort

MERGE SORT:

```
import java.util.*;
public class MergeSort{
  public void merge(int arr[],int p,int q,int r){
    int n1=q-p+1;
    int n2=r-q;
    int left[]=new int[n1];
    int right[]=new int[n2];
    for(int i=0;i<n1;i++){
      left[i]=arr[p+i];}
    for(int i=0;i<n2;i++){
      right[i]=arr[q+i+1];}
    int i=0;
    int j=0;
    int k=p;
    while(i<n1 && j<n2){
      if(left[i]<=right[j]){
        arr[k]=left[i];
        i++;
        k++; }
      else{
        arr[k]=right[j];
        j++;
        k++;}}
    while(i<n1){
      arr[k]=left[i];
      i++;
      k++; }
    while(j<n2){
      arr[k]=right[j];
      j++;
```

```
        k++;} }
    public void mergesort(int arr[],int left,int right){
      if(left<right){
        int mid=(left+right)/2;
        mergesort(arr,left,mid);
        mergesort(arr, mid+1, right);
        merge(arr,left,mid,right);} }
    public static void main(String[] args) {
      Scanner sc=new Scanner(System.in);
      System.out.println("Enter size of array");
      int arr_size=sc.nextInt();
      int arr[]=new int[arr_size];
      System.out.println("Enter array elements");
      for(int i=0;i<arr_size;i++){
        arr[i]=sc.nextInt();}
      MergeSort ms=new MergeSort();
      System.out.println("Before sorting the array is "+Arrays.toString(arr));
      ms.mergesort(arr, 0, arr_size-1);
      System.out.println("After sorting array is"+Arrays.toString(arr));}}
```

OUTPUT:

```
Enter size of array
5

Enter array elements
3

6

1

2

4

Before sorting the array is [3, 6, 1, 2, 4]
After sorting array is[1, 2, 3, 4, 6]
```

HEAP SORT

```
import java.util.*;
public class HeapSort{
    public void sort(int arr[]){
```

```
        int n=arr.length;
        for(int i=n/2-1;i>=0;i--){
            heapify(arr,n,i);}
        for(int i=n-1;i>=0;i--){
            int temp=arr[i];
            arr[i]=arr[0];
            arr[0]=temp;
            heapify(arr,i,0);}}
    public void heapify(int arr[],int n,int i){
        int largest=i;
        int left=2*i+1;
        int right=2*i+2;
        if(left<n && arr[left]>arr[largest]){
            largest=left; }
        if(right<n && arr[right]>arr[largest]){
            largest=right;}
        if(largest!=i){
            int temp=arr[i];
            arr[i]=arr[largest];
            arr[largest]=temp;
            heapify(arr, n, largest);}}
    public static void main(String[] args) {
        int arr[]={5,1,2,6,4};
        HeapSort hp=new HeapSort();
        System.out.println("Before Sorting the array is "+Arrays.toString(arr));
        hp.sort(arr);
        System.out.println("After sorting the array is "+Arrays.toString(arr));}}
```

OUTPUT:

```
Before Sorting the array is [5, 1, 2, 6, 4]
After sorting the array is [1, 2, 4, 5, 6]
```

QUICK SORT:

```
import java.util.*;
public class QuickSort{
  public int partition(int arr[],int low,int high){
    int pivot=arr[high];
    int i=low-1;
    for(int j=low;j<high;j++){
      if(arr[j]<pivot){
        i++;
        int temp=arr[i];
        arr[i]=arr[j];
        arr[j]=temp;}}
    int temp=arr[i+1];
    arr[i+1]=arr[high];
    arr[high]=temp;
    return i+1;}
  public void quicksort(int arr[],int low,int high){
    if(low<high){
      int pi=partition(arr, low, high);
      quicksort(arr, low, pi-1);
      quicksort(arr, pi+1, high);}}
  public static void main(String[] args) {
    int arr[]={4,3,7,1,2};
    int arr_size=arr.length;
    QuickSort qs=new QuickSort();
    System.out.println("Before sorting the array is"+Arrays.toString(arr));
    qs.quicksort(arr, 0, arr_size-1);
    System.out.println("After sorting the array is"+Arrays.toString(arr));}}
```

OUTPUT:

```
Before sorting the array is[4, 3, 7, 1, 2]
After sorting the array is[1, 2, 3, 4, 7]
```

3. Write a program to perform the following operations:

a) Insert an element into a Min & Max heap

b) Delete an element from a Min &Max heap

c) Search for a key element in a Min & Max heap

MIN-HEAP

```
import java.util.*;
public class MinHeap{
  ArrayList<Integer> heap;
  public MinHeap(){
    heap=new ArrayList<>();}
  public int parent(int i){
    return (i-1)/2;}
  public int getleftchild(int i){
    return 2*i+1;}
  public int getrightchild(int i){
    return 2*i+2;}
  public void insert(int value){
    heap.add(value);
    int current=heap.size()-1;
    while(current>0 && heap.get(current)<heap.get(parent(current))){
      swap(current,parent(current));
      current=parent(current);}}
  public void delete(){
    if (heap.isEmpty()){
      System.out.println("heap is empty");}
    int min=heap.get(0);
    heap.set(0,heap.get(heap.size()-1));
    heap.remove(heap.size()-1);
    heapify_down(0);}
  public void heapify_down(int i){
    int smallest=i;
```

```
    int left=getleftchild(i);
    int right=getrightchild(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);
        heapify_down(smallest);} }
public boolean search(int value){
    return heap.contains(value);}
public void swap(int i,int j){
    int temp=heap.get(i);
    heap.set(i,heap.get(j));
    heap.set(j, temp);}
public void printheap(){
    System.out.println(heap);}
public static void main(String[] args) {
    MinHeap heap=new MinHeap();
    heap.insert(3);
    heap.insert(5);
    heap.insert(1);
    heap.insert(2);
    heap.insert(4);
    System.out.println("After insertion the heap is ");
    heap.printheap();
    heap.delete();
    System.out.println("After deletion the heap is ");
    heap.printheap();
    System.out.println("element found : "+heap.search(4));}}
```

Output:

```
After insertion the heap is
[1, 2, 3, 5, 4]
After deletion the heap is
[2, 4, 3, 5]
element found : true
```

MAX-HEAP

```
import java.util.*;
public class MaxHeap{
  ArrayList<Integer> heap;
  public MaxHeap(){
    heap=new ArrayList<>();}
  public int parent(int i){return (i-1)/2;}
  public int leftchild(int i){return 2*i+1;}
  public int rightchild(int i){return 2*i+2;}
  public void insert(int value){
    heap.add(value);
    int current=heap.size()-1;
    while(current>0 && heap.get(current)>heap.get(parent(current))){
      swap(current,parent(current));
      current=parent(current);} }
  public void delete(){
    if(heap.isEmpty()){
      System.out.println("Heap is empty");}
    int max=heap.get(0);
    heap.set(0,heap.get(heap.size()-1));
    heap.remove(heap.size()-1);
    heapify_down(0);}
  public boolean search(int value){
    return heap.contains(value);}
  public void heapify_down(int i){
    int largest=i;
```

```
    int left=leftchild(i);
    int right=rightchild(i);
    if(left<heap.size() && heap.get(left)>heap.get(largest)){
      largest=left;}
    if(right<heap.size() && heap.get(right)>heap.get(largest)){
      largest=right;}
    if(largest!=i){
      swap(i,largest);
      heapify_down(largest); }}
  public void swap(int i,int j){
    int temp=heap.get(i);
    heap.set(i, heap.get(j));
    heap.set(j, temp);}
  public void printheap(){
    System.out.println(heap);}
  public static void main(String[] args) {
    MaxHeap heap=new MaxHeap();
    heap.insert(4);
    heap.insert(10);
    heap.insert(3);
    heap.insert(1);
    heap.insert(5);
    System.out.println("After insertion the heap is");
    heap.printheap();
    heap.delete();
    System.out.println("After deletion the heap is");
    heap.printheap();
    System.out.println("Element found: "+heap.search(5));    }}
```

Output:

```
After insertion the heap is
[10, 5, 3, 1, 4]
After deletion the heap is
[5, 4, 3, 1]
Element found: true
```

4. Write a program to perform the following operations:

a) Insert an element into a AVL tree.

b) Delete an element from a AVL search tree.

c) Search for a key element in a AVL search tree.

```
class Node {
   int key;
   Node left, right;
   int height;
   Node(int k) {
      key = k;
      left = null;
      right = null;
      height = 1;}}
class AVLtree {
   // A utility function to get the height of the tree
   static int height(Node N) {
      if (N == null)
         return 0;
      return N.height; }
   // A utility function to right rotate subtree rooted with y
   static 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;}
```

```
// A utility function to left rotate subtree rooted with x
static 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; }
// Get balance factor of node N
static int getBalance(Node N) {
   if (N == null)
      return 0;
   return height(N.left) - height(N.right);}
// A utility function to balance the node and perform the appropriate rotations
static Node balance(Node node) {
   // Get the balance factor
   int balance = getBalance(node);
   // If this node becomes unbalanced, then there are 4 cases
   // Left Left Case
   if (balance > 1 && getBalance(node.left) >= 0)
      return rightRotate(node);
   // Left Right Case
   if (balance > 1 && getBalance(node.left) < 0) {
      node.left = leftRotate(node.left);
      return rightRotate(node);}
   // Right Right Case
   if (balance < -1 && getBalance(node.right) <= 0)
      return leftRotate(node);
```

```
    // Right Left Case
    if (balance < -1 && getBalance(node.right) > 0) {
        node.right = rightRotate(node.right);
        return leftRotate(node);}
    // Return the node (unchanged if balanced)
    return node;}
// Recursive function to insert a key in the subtree rooted with node
static 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 // Equal keys are not allowed in BST
        return node;
    // Update height of this ancestor node
    node.height = 1 + Math.max(height(node.left), height(node.right));
    // Balance the node
    return balance(node); }
// Recursive function to delete a node with a given key from the subtree rooted with a given node
static Node deleteNode(Node root, int key) {
    // Perform standard BST deletion
    if (root == null)
        return root;
    // If the key to be deleted is smaller than the root's key, it lies in the left subtree
    if (key < root.key)
        root.left = deleteNode(root.left, key);
    // If the key to be deleted is greater than the root's key, it lies in the right subtree
    else if (key > root.key)
        root.right = deleteNode(root.right, key);
```

```
    // If key is the same as root's key, this is the node to be deleted
    else {
        // Node with only one child or no child
        if ((root.left == null) || (root.right == null)) {
            Node temp = (root.left != null) ? root.left : root.right;
            // No child case
            if (temp == null) {
                temp = root;
                root = null;
            } else // One child case
                root = temp;
        } 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 = deleteNode(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;
    // Balance the node
    return balance(root);}
// A utility function to find the node with minimum key value in the subtree rooted at a given node
static Node minValueNode(Node node) {
    Node current = node;
    // Loop to find the leftmost leaf
    while (current.left != null)
        current = current.left;
```

```
    return current; }
  // A function to search for a key in the AVL tree
  static Node search(Node root, int key) {
    // Base case: root is null or key is present at root
    if (root == null || root.key == key)
      return root;
    // Key is greater than root's key
    if (key > root.key)
      return search(root.right, key);
    // Key is smaller than root's key
    return search(root.left, key);}
  // A utility function to print preorder traversal of the tree
  static void preOrder(Node root) {
    if (root != null) {
      System.out.print(root.key + " ");
      preOrder(root.left);
      preOrder(root.right);
    }
  }
  // Driver code
  public static void main(String[] args) {
    Node root = null;
    // Constructing the tree
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);
    System.out.println("Preorder traversal before deletion: ");
    preOrder(root);
// Deleting node 40
```

```
root = deleteNode(root, 40);
        System.out.println("\nPreorder traversal after deletion: ");
        preOrder(root);
        // Searching for node 25
        Node result = search(root, 25);
        if (result != null) {
            System.out.println("\nNode 25 found in the tree.");
        } else {
            System.out.println("\nNode 25 not found in the tree.");}}}
```

Output:

```
Preorder traversal before deletion:
30 20 10 25 40 50
Preorder traversal after deletion:
30 20 10 25 50
Node 25 found in the tree.
```

5. Write a program to implement all the functions of a dictionary using hashing.

```
import java.util.*;
public class Hashing{
   public int get_hash_key(int key,int tables_size){
      int hash_key=key%tables_size;
      return hash_key; }
   public void insert(Dictionary<Integer,Integer> dict,int value,int tables_size){
      int hash_key=get_hash_key(value, tables_size);
      int f=0;
      while(f==0){
         if(dict.get(hash_key)==0){
            dict.put(hash_key,value);
            f=1;}
         else{
            hash_key=(hash_key+1)%tables_size;}}}
   public int search(Dictionary<Integer,Integer> dict,int value,int tables_size){
      int hash_key=get_hash_key(value, tables_size);
      int f=0;
      while(f==0){
         if(dict.get(hash_key)==value){
            System.out.print("ELement found at ");
            f=1;
            return hash_key;}
         else{
            hash_key=(hash_key+1)%tables_size;}}
      System.out.println("Element not found");
      return -1;}
   public void delete(Dictionary<Integer,Integer> dict,int value,int tables_size){
      int hash_key=get_hash_key(value, tables_size);
      int f=0;
      while(f==0){
         if(dict.get(hash_key)==value){
```

```
            dict.put(hash_key,0);
            f=1;}
         else{
            hash_key=(hash_key+1)%tables_size;}}}
   public static void main(String[] args) {
      Dictionary<Integer,Integer> dict=new Hashtable<>();
      int tables_size=10;
      for(int i=0;i<tables_size;i++){
         dict.put(i,0); }
      Hashing h=new Hashing();
      h.insert(dict,55, tables_size);
      h.insert(dict, 63, tables_size);
      h.insert(dict, 75, tables_size);
      h.insert(dict, 32, tables_size);
      h.insert(dict, 16, tables_size);
      System.err.println(dict);
      int index=h.search(dict, 32, tables_size);
      if(index!=-1){
         System.err.println(index); }
      h.delete(dict, 16, tables_size);
      System.out.println(dict);  }}
```

Output:

```
{9=0, 8=0, 7=16, 6=75, 5=55, 4=0, 3=63, 2=32, 1=0, 0=0}
ELement found at 2
{9=0, 8=0, 7=0, 6=75, 5=55, 4=0, 3=63, 2=32, 1=0, 0=0}
```

6. Write a program for implementing Knuth-Morris-Pratt pattern matching algorithm.

```
public class KMP{
   public int[] generate_lps(char[] pat_arr){
      int n=pat_arr.length;
      int i=0;
      int j=1;
      int lps[]=new int[n];
      lps[0]=0;
      while(j<n){
         if(pat_arr[i]==pat_arr[j]){
            i++;
            lps[j]=i;
            j++;}
         else{
            if(i!=0){
               i=lps[i-1];}
            else{
               lps[j]=0;
               j++;}}}
   return lps;}
   public void pattern_match(String text,String pattern){
      int n=text.length();
      int m=pattern.length();
      char[] text_arr=text.toCharArray();
      char[] pat_arr=pattern.toCharArray();
      int i=0;
      int j=0;
      int f=0;
      int[] lps=generate_lps(pat_arr);
      while(i<n){
         if(text_arr[i]==pat_arr[j]){
            i++;
```

```
            j++;}
         if(j==m){
            System.out.println("pattern found at"+(i-j));
            f=1;
            j=lps[j-1];}
         else if(i<n && text_arr[i]!=pat_arr[j]){
            if(j!=0){
               j=lps[j-1];}
            else{
               i++;}}}
      if(f==0){
         System.out.println("Pattern not found");}}
   public static void main(String[] args) {
      KMP kmp=new KMP();
      String Text="AAABAAABCCCCCDDEF";
      String pattern="AABC";
      kmp.pattern_match(Text, pattern);}}
```

Output:

```
pattern found at5
```

7. Write a program for implementing Brute Force pattern matching algorithm.

```
public class BruteForce{
   public void patternmatch(String text,String pattern){
      int n=text.length();
      int m=pattern.length();
      char[] text_arr=text.toCharArray();
      char[] pat_arr=pattern.toCharArray();
      int f=0;
      for(int i=0;i<n-m;i++){
         int j=0;
         while(j<m && text_arr[i+j]==pat_arr[j]){
            j++;
            if(j==m){
               System.out.println("Pattern found at"+(i));
               f=1;}}}
      if(f==0){
         System.out.println("Pattern not found");}}
   public static void main(String[] args) {
      BruteForce bf=new BruteForce();
      String text="AABAAACBAAB";
      String pattern="BAA";
      bf.patternmatch(text, pattern);}}
```

Output:

```
Pattern found at2
Pattern found at7
```

8. Write a program for implementing Boyer pattern matching algorithm

```
import java.util.Arrays;
class BoyerMoore {
   // Method to create the bad character heuristic table
   public int[] preprocessBadCharacterTable(char[] pat) {
      int[] badCharTable = new int[256];  // Assuming ASCII characters (256)
      Arrays.fill(badCharTable, -1);      // Initialize all values to -1
      // Fill the actual values in the bad character table
      for (int i = 0; i < pat.length; i++) {
         badCharTable[pat[i]] = i;  // Store the last occurrence of each character in the pattern}
      return badCharTable;}
   public void searchPattern(String txt, String pat) {
      char[] textArr = txt.toCharArray();
      char[] patArr = pat.toCharArray();
      int n1 = textArr.length;
      int n2 = patArr.length;
      int[] badCharTable = preprocessBadCharacterTable(patArr);  // Create bad character table
      int shift = 0;  // Shift of the pattern with respect to the text
      // Pattern matching loop
      while (shift <= (n1 - n2)) {
         int j = n2 - 1;
         // Traverse the pattern from right to left
         while (j >= 0 && patArr[j] == textArr[shift + j]) {
            j--;}
         // If the pattern is found
         if (j < 0) {
            System.out.println("Pattern found at index " + shift);
            // Shift the pattern to align with the next possible match in text
            shift += (shift + n2 < n1) ? n2 - badCharTable[textArr[shift + n2]] : 1;
         } else {
            // Calculate shift based on the bad character rule
            shift += Math.max(1, j - badCharTable[textArr[shift + j]]);}}}
```

```
public static void main(String[] args) {
    String txt = "AABAACAADAABAABA";
    String pat = "AABA";
    new BoyerMoore().searchPattern(txt, pat);}}
```

Output:

```
Pattern found at index 0
Pattern found at index 9
Pattern found at index 12
```

9. Write a program for implementing Shortest path algorithm.

```
import java.util.*;
// Class to represent a graph
class GraphDij {
   private int vertices; // Number of vertices
   private LinkedList<Edge>[] adjacencyList; // Adjacency list to store edges
   // Constructor to initialize the graph
   public GraphDij(int vertices) {
      this.vertices = vertices;
      adjacencyList = new LinkedList[vertices];
      for (int i = 0; i < vertices; i++) {
         adjacencyList[i] = new LinkedList<>();} }
   // Class to represent an edge between two vertices
   static class Edge {
      int targetVertex;
      int weight;
      Edge(int targetVertex, int weight) {
         this.targetVertex = targetVertex;
         this.weight = weight;}}
   // Add an edge to the graph
   public void addEdge(int source, int destination, int weight) {
      adjacencyList[source].add(new Edge(destination, weight));
      adjacencyList[destination].add(new Edge(source, weight)); // For undirected graph
   // Dijkstra's algorithm for Single Source Shortest Path
   public void dijkstra(int source) {
      // Array to store the shortest distance from source to each vertex
      int[] distance = new int[vertices];
      // Set all distances to infinity (or a very large value)
      Arrays.fill(distance, Integer.MAX_VALUE);
      distance[source] = 0;
      // Priority queue to select the minimum distance vertex (min-heap)
      PriorityQueue<Edge> pq = new PriorityQueue<>(vertices, Comparator.comparingInt(edge ->
edge.weight));
```

```java
    // Add the source vertex to the priority queue with distance 0
    pq.add(new Edge(source, 0));
    // Boolean array to keep track of visited vertices
    boolean[] visited = new boolean[vertices];
    // While there are vertices to process
    while (!pq.isEmpty()) {
      // Extract the vertex with the smallest distance
      Edge currentEdge = pq.poll();
      int currentVertex = currentEdge.targetVertex;
      // Skip processing if the vertex has already been visited
      if (visited[currentVertex]) continue;
      // Mark the vertex as visited
      visited[currentVertex] = true;
      // Explore all the adjacent vertices of the current vertex
      for (Edge edge : adjacencyList[currentVertex]) {
        int neighbor = edge.targetVertex;
        int newDist = distance[currentVertex] + edge.weight;
        // If a shorter path to the neighbor is found, update its distance
        if (!visited[neighbor] && newDist < distance[neighbor]) {
          distance[neighbor] = newDist;
          pq.add(new Edge(neighbor, newDist));}}}
    // Print the shortest distances from the source to all other vertices
    printShortestDistances(distance, source);}
  // Function to print the shortest distances from the source
  private void printShortestDistances(int[] distance, int source) {
    System.out.println("Shortest distances from vertex " + source + ":");
    for (int i = 0; i < distance.length; i++) {
      System.out.println("To vertex " + i + " is " + distance[i]);}}
  // Main function to test the Dijkstra algorithm
  public static void main(String[] args) {
    GraphDij graph = new GraphDij(6);
    // Adding edges to the graph (undirected)
```

```
graph.addEdge(0, 1, 4);
graph.addEdge(0, 2, 3);
graph.addEdge(1, 2, 1);
graph.addEdge(1, 3, 2);
graph.addEdge(2, 3, 4);
graph.addEdge(3, 4, 2);
graph.addEdge(4, 5, 6);
// Apply Dijkstra's algorithm from vertex 0
graph.dijkstra(0);}}
```

Output:

```
Shortest distances from vertex 0:
To vertex 0 is 0
To vertex 1 is 4
To vertex 2 is 3
To vertex 3 is 6
To vertex 4 is 8
To vertex 5 is 14
```

10. Write a program for implementing graph traversal DFS and BFS.

```
import java.util.*;
// Graph class for implementing BFS and DFS
class Graph {
  private int vertices; // Number of vertices
  private LinkedList<Integer>[] adjacencyList; // Adjacency List
  // Constructor to initialize the graph
  public Graph(int vertices) {
    this.vertices = vertices;
    adjacencyList = new LinkedList[vertices];
    for (int i = 0; i < vertices; i++) {
      adjacencyList[i] = new LinkedList<>();} }
  // Add edge to the graph
  public void addEdge(int v, int w) {
    adjacencyList[v].add(w); // Add w to v's adjacency list}
  // BFS algorithm
  public void bfs(int startVertex) {
    boolean[] visited = new boolean[vertices];
    LinkedList<Integer> queue = new LinkedList<>();
   visited[startVertex] = true;
    queue.add(startVertex);
    while (!queue.isEmpty()) {
      int vertex = queue.poll();
      System.out.print(vertex + " ");
      for (int neighbor : adjacencyList[vertex]) {
        if (!visited[neighbor]) {
          visited[neighbor] = true;
          queue.add(neighbor);}}}}

  // DFS algorithm using recursion
  public void dfs(int startVertex) {
    boolean[] visited = new boolean[vertices];
```

```
    dfsUtil(startVertex, visited);}
  // Utility function for DFS
  private void dfsUtil(int vertex, boolean[] visited) {
    visited[vertex] = true;
    System.out.print(vertex + " ");
    for (int neighbor : adjacencyList[vertex]) {
      if (!visited[neighbor]) {
        dfsUtil(neighbor, visited);}}}
  // Main function to test the BFS and DFS algorithms
  public static void main(String[] args) {
    Graph graph = new Graph(6);
    // Adding edges to the graph
    graph.addEdge(0, 1);
    graph.addEdge(0, 2);
    graph.addEdge(1, 3);
    graph.addEdge(1, 4);
    graph.addEdge(2, 4);
    graph.addEdge(3, 5);
    graph.addEdge(4, 5);
    System.out.println("Breadth-First Search (starting from vertex 0):");
    graph.bfs(0);
    System.out.println("\nDepth-First Search (starting from vertex 0):");
    graph.dfs(0);}}
```

Output:

```
Breadth-First Search (starting from vertex 0):
0 1 2 3 4 5
Depth-First Search (starting from vertex 0):
0 1 3 5 4 2
```

11. Write a program for implementing geometric algorithms.

```
import java.util.*;
public class GeometricAlg{
   public static class Point{
      double x;
      double y;
      Point(double x,double y){
         this.x=x;
         this.y=y;}}
   public double calculate_distance(double A,double B,double C,double x,double y){
      return Math.abs(A*x+B*y+C)/Math.sqrt(A*A+B*B);}
   public void calculate_mean(String label,List<Double> Distances){
      if(Distances.isEmpty()){
         System.out.println("No point on the "+label); }
      else{
         double sum=0;
         for(double dist:Distances){
            sum+=dist;}
         double mean=sum/Distances.size();
         System.out.println("Mean of "+label+" is "+mean);} }
   public void classify_calculate_mean(double A,double B,double C,List<Point> points){
      List<Double> positive_dist=new ArrayList<>();
      List<Double> negative_dist=new ArrayList<>();
      for(Point point:points){
         double distance=calculate_distance(A, B, C, point.x, point.y);
         if(distance>0){
            System.out.println("Point("+point.x+","+point.y+") lies on positive side of the line at a
distance of "+distance);
            positive_dist.add(distance);}
         else if(distance<0){
            System.out.println("Point("+point.x+","+point.y+") lies on negative side of the line at a
distance of "+distance);
            negative_dist.add(distance);}
```

```
        else{
          System.out.println("Point("+point.x+","+point.y+") lies on the line");
        }}
      calculate_mean("positive_side",positive_dist);
      calculate_mean("negative_side",negative_dist);}
    public static void main(String[] args) {
      GeometricAlg ga=new GeometricAlg();
      double A=3;
      double B=2;
      double C=5;
      List<Point> points=List.of(
        new Point(3,-1),
        new Point(2,1),
        new Point(-2,-3),
        new Point(0,5),
        new Point(-3,4));
      ga.classify_calculate_mean(A,B,C,points); }}
```

Output:

```
Point(3.0,-1.0) lies on positive side of the line at a distance of 3.328201177351375
Point(2.0,1.0) lies on positive side of the line at a distance of 3.6055512754639896
Point(-2.0,-3.0) lies on positive side of the line at a distance of 1.9414506867883021
Point(0.0,5.0) lies on positive side of the line at a distance of 4.160251471689219
Point(-3.0,4.0) lies on positive side of the line at a distance of 1.1094003924504583
Mean of positive_side is 2.828971000748669
No point on the negative_side
```