Project 2: Graph Algorithms
Single-source shortest path and Minimum Spanning Tree (MST)

Single Source Shortest Path(Dijkstra's Algorithm):

Description: This algorithm computes the shortest distance to all the vertices from a given source vertex. There will be no negative edges in the graph. The graph can be directed or undirected.

The graph considers a single vertex as source.

From the source, it calculates the distance to its neighbours and keeps the neighbouring vertices' distance in the min heap.

The min heap picks the vertex with shorter distance. Now, this vertex with minimum distance checks it's neighbours and performs the above steps.

The given source vertex is marked as visited.

During the process of checking it's neighbours, if already visited node is encountered as a neighbour, it won't be added to min heap again.

The termination condition is when there are no nodes in the min heap (this occurs when all the vertices are visited).

Data Structure : We have used the min heap and adjacency(Linked) list to implement this algorithm.

Time Complexity: The run time of our code is (V+E)log V.

- O(V) to intialize.
- O(V) to construct the min Heap.
- VlogV to perform the extract min operation
- ElogV to perform Decrease Key

Sample Input/Output:

```
6 10 U
A B 1
A C 2
B C 1
B D 3
B E 2
C D 1
C E 2
D E 4
D F 3
E F 3
A
```

Dijkstra Algorithm: (Adjacency List + Min Heap)

Vertex A to vertex A distance is 0

Vertex A to vertex B distance is 1

Path:A-->B

Vertex A to vertex C distance is 2

Path:A-->C

Vertex A to vertex D distance is 3

Path:A-->C-->D

Vertex A to vertex E distance is 3

Path:A-->B-->E

Vertex A to vertex F distance is 6

Path:A-->B-->E-->F

Minimum Spanning Tree (Prim's Algorithm):

Description: The algorithm finds a spanning tree using edges that minimize the total weight. This is a greedy algorithm similar to the Dijkstra's.

This algorithm maintains two sets of vertices.

At every step, it considers all the edges and picks minimum weight edge. We do the relaxing operation to the edges to identify the shortest edge. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

So, the first set contains the vertices that are part of the minimum spanning tree and the other one contains the vertices that are not.

The initial shortest distance between two vertices will get updated with time.

Data Structure: We have used the min heap and adjacency(Linked) list to implement this algorithm.

Time Complexity: The run time of our code is O((V+E)logV)

Sample Input/Output:

```
6 10 U
A B 1
A C 2
B C 1
B D 3
B E 2
C D 1
C E 2
D E 4
D F 3
E F 3
A
```

Minimum Spanning Tree Using Prims Algorithm:

Edge: B - A weight: 1

Edge: C - B weight: 1

Edge: D - C weight: 1

Edge: E - B weight: 2

Edge: F - D weight: 3

Total minimum distance: 8

Instruction to Run the code:
The code contains two files DijkstraandPrim.java and MinHeap.java.
DijkstraandPrim.java contains the main class. Change the path of the input file to test. We have included input1.txt, input2.txt and input3.txt in the zip file.
If the source is not mentioned in the input file, the code randomly picks a vertex as source. The vertices should be English Alphabet. Our code won't accept numbers. Place the input .txt files within the same working directory as that of your code.

Undirected Graph 3 Test Cases:

1)
```
9 14 U
A B 4
A H 8
B C 8
B H 11
C D 7
C I 2
C F 4
D E 9
D F 14
E F 10
F G 2
G H 1
G I 6
H I 7
```

Source Vertex taken is : B

Dijkstra Algorithm: (Adjacency List + Min Heap)

Source Vertex B to vertex A distance: 4 Previous Vertex B

Path:B-->A

Source Vertex: B to vertex: B distance: 0

Source Vertex B to vertex C distance: 8 Previous Vertex B

Path:B-->C

Source Vertex B to vertex D distance: 15 Previous Vertex C

Path:B-->C-->D

Source Vertex B to vertex E distance: 22 Previous Vertex F

Path:B-->C-->F-->E

Source Vertex B to vertex F distance: 12 Previous Vertex C

Path:B-->C-->F

Source Vertex B to vertex G distance: 12 Previous Vertex H

Path:B-->H-->G

Source Vertex B to vertex H distance: 11 Previous Vertex B

Path:B-->H

Source Vertex B to vertex I distance: 10 Previous Vertex C

Path:B-->C-->I

Minimum Spanning Tree Using Prims Algorithm:

Edge: B - A weight: 4

Edge: C - F weight: 4

Edge: D - C weight: 7

Edge: E - D weight: 9

Edge: F - G weight: 2

Edge: G - H weight: 1

Edge: H - A weight: 8

Edge: I - C weight: 2

Total minimum distance: 37

```
2)  9 15 U
    A B 10
    A C 7
    B D 15
    B E 10
    B F 12
    C E 15
    C F 7
    D G 9
    D H 13
    E G 12
    E H 8
    F G 22
    F H 15
    G I 9
    H I 5
```

Source Vertex taken is : G

Dijkstra Algorithm: (Adjacency List + Min Heap)

Source Vertex G to vertex A distance: 32 Previous Vertex B

Path:G-->E-->B-->A

Source Vertex G to vertex B distance: 22 Previous Vertex E

```
Path:G-->E-->B

Source Vertex G to vertex C distance: 27 Previous Vertex E

Path:G-->E-->C

Source Vertex G to vertex D distance: 9 Previous Vertex G

Path:G-->D

Source Vertex G to vertex E distance: 12 Previous Vertex G

Path:G-->E

Source Vertex G to vertex F distance: 22 Previous Vertex G

Path:G-->F

Source Vertex: G to vertex: G distance: 0

Source Vertex G to vertex H distance: 14 Previous Vertex I

Path:G-->I-->H

Source Vertex G to vertex I distance: 9 Previous Vertex G

Path:G-->I



Minimum Spanning Tree Using Prims Algorithm:

Edge: B - A weight: 10

Edge: C - A weight: 7

Edge: D - G weight: 9

Edge: E - B weight: 10

Edge: F - C weight: 7

Edge: G - I weight: 9

Edge: H - E weight: 8

Edge: I - H weight: 5

Total minimum distance: 65
```

```
3)  9 15 U
    A B  9
    A C  12
    B C  8
    B D  4
    B E  7
    C E  5
    C F  2
```

```
D E  2
D G  10
E G  2
E F  11
F H  4
G H  4
G I  3
H I  13
```

Dijkstra Algorithm: (Adjacency List + Min Heap)

```
Source Vertex: A to vertex: A distance: 0

Source Vertex A to vertex B distance: 9 Previous Vertex A

Path:A-->B

Source Vertex A to vertex C distance: 12 Previous Vertex A

Path:A-->C

Source Vertex A to vertex D distance: 13 Previous Vertex B

Path:A-->B-->D

Source Vertex A to vertex E distance: 15 Previous Vertex D

Path:A-->B-->D-->E

Source Vertex A to vertex F distance: 14 Previous Vertex C

Path:A-->C-->F

Source Vertex A to vertex G distance: 17 Previous Vertex E

Path:A-->B-->D-->E-->G

Source Vertex A to vertex H distance: 18 Previous Vertex F

Path:A-->C-->F-->H

Source Vertex A to vertex I distance: 20 Previous Vertex G

Path:A-->B-->D-->E-->G-->I
```

Minimum Spanning Tree Using Prims Algorithm:

```
Edge: B - A weight: 9

Edge: C - F weight: 2

Edge: D - B weight: 4

Edge: E - D weight: 2

Edge: F - H weight: 4
```

Edge: G - E weight: 2

Edge: H - G weight: 4

Edge: I - G weight: 3

Total minimum distance: 30

Directed Graph 3 Test Cases:

1)
```
8 15 D
A B 9
A C 15
A D 14
B H 42
B E 23
C H 38
C G 37
D E 17
D F 30
D C 5
E F 3
E G 20
F G 16
G H 12
H E 21
G
```

Dijkstra Algorithm: (Adjacency List + Min Heap)

Source Vertex G to vertex A distance: INFINITY Previous Vertex A

Path: Can not be reached

Source Vertex G to vertex B distance: INFINITY Previous Vertex A

Path: Can not be reached

Source Vertex G to vertex C distance: INFINITY Previous Vertex D

Path: Can not be reached

Source Vertex G to vertex D distance: INFINITY Previous Vertex A

Path: Can not be reached

Source Vertex G to vertex E distance: 33 Previous Vertex H

Path:G-->H-->E

Source Vertex G to vertex F distance: 36 Previous Vertex E

Path:G-->H-->E-->F

Source Vertex: G to vertex: G distance: 0

Source Vertex G to vertex H distance: 12 Previous Vertex G

Path:G-->H

```
Minimum Spanning Tree Using Prims Algorithm:

Edge: B - A weight: 9

Edge: C - D weight: 5

Edge: D - A weight: 14

Edge: E - D weight: 17

Edge: F - E weight: 3

Edge: G - F weight: 16

Edge: H - G weight: 12

Total minimum distance: 76
```

```
2)  9 15 D
    A B 2
    A C 4
    B C 1
    B D 4
    B E 2
    C E 3
    E D 3
    D F 2
    E F 2
    F G 7
    G E 9
    H F 8
    I H 6
    I F 11
    I G 2
    A
```

```
Dijkstra Algorithm: (Adjacency List + Min Heap)

Source Vertex: A to vertex: A distance: 0

Source Vertex A to vertex B distance: 2 Previous Vertex A

Path:A-->B

Source Vertex A to vertex C distance: 3 Previous Vertex B

Path:A-->B-->C

Source Vertex A to vertex D distance: 6 Previous Vertex B

Path:A-->B-->D

Source Vertex A to vertex E distance: 4 Previous Vertex B

Path:A-->B-->E

Source Vertex A to vertex F distance: 6 Previous Vertex E
```

Path:A-->B-->E-->F

Source Vertex A to vertex G distance: 13 Previous Vertex F

Path:A-->B-->E-->F-->G

Source Vertex A to vertex H distance: INFINITY Previous Vertex I

Path: Can not be reached

Source Vertex A to vertex I distance: INFINITY Previous Vertex A

Path: Can not be reached

Minimum Spanning Tree Using Prims Algorithm:

Edge: B - A weight: 2

Edge: C - B weight: 1

Edge: D - E weight: 3

Edge: E - B weight: 2

Edge: F - E weight: 2

Edge: G - F weight: 7

Edge: H - I weight: 6

Edge: I - null weight: 0

Total minimum distance: 23

```
3)  15 25 D
    A A 0
    A C 9
    B A 4
    C B 4
    C E 1
    D A 6
    D B 12
    D L 5
    D M 4
    E F 7
    E H 8
    F G 8
    F J 10
    G J 11
    H J 2
    I E 2
    I K 8
    J I 3
    J N 4
    K L 10
    K N 6
    L M 9
    M O 3
```

```
N M 1
O N 4
A
```

```
Dijkstra Algorithm: (Adjacency List + Min Heap)
Source Vertex: A to vertex: A distance: 0
Source Vertex A to vertex B distance: 13 Previous Vertex C
Path:A-->C-->B
Source Vertex A to vertex C distance: 9 Previous Vertex A
Path:A-->C
Source Vertex A to vertex D distance: INFINITY Previous Vertex A
Path: Can not be reached
Source Vertex A to vertex E distance: 10 Previous Vertex C
Path:A-->C-->E
Source Vertex A to vertex F distance: 17 Previous Vertex E
Path:A-->C-->E-->F
Source Vertex A to vertex G distance: 25 Previous Vertex F
Path:A-->C-->E-->F-->G
Source Vertex A to vertex H distance: 18 Previous Vertex E
Path:A-->C-->E-->H
Source Vertex A to vertex I distance: 23 Previous Vertex J
Path:A-->C-->E-->H-->J-->I
Source Vertex A to vertex J distance: 20 Previous Vertex H
Path:A-->C-->E-->H-->J
Source Vertex A to vertex K distance: 31 Previous Vertex I
Path:A-->C-->E-->H-->J-->I-->K
Source Vertex A to vertex L distance: 41 Previous Vertex K
Path:A-->C-->E-->H-->J-->I-->K-->L
Source Vertex A to vertex M distance: 25 Previous Vertex N
Path:A-->C-->E-->H-->J-->N-->M
Source Vertex A to vertex N distance: 24 Previous Vertex J
Path:A-->C-->E-->H-->J-->N
Source Vertex A to vertex O distance: 28 Previous Vertex M
Path:A-->C-->E-->H-->J-->N-->M-->O


Minimum Spanning Tree Using Prims Algorithm:
Edge: B - C weight: 4
Edge: C - A weight: 9
Edge: D - null weight: 0
Edge: E - C weight: 1
Edge: F - E weight: 7
Edge: G - F weight: 8
Edge: H - E weight: 8
Edge: I - J weight: 3
Edge: J - H weight: 2
Edge: K - I weight: 8
Edge: L - K weight: 10
Edge: M - N weight: 1
Edge: N - J weight: 4
Edge: O - M weight: 3
Total minimum distance: 68
```

Code:

DijkstraandPrim.java:

```java
import java.util.HashSet;
import java.util.LinkedList;
import java.util.*;
import java.io.*;

public class DijkstraandPrim{
    // taking hashmap to take unique vertices
    private static HashMap<Character, Integer> mapVertices = new HashMap<>();
    private static final int BUFFER_LENGTH = 16;
    private static final int MARK_LENGTH = 24 * BUFFER_LENGTH;
    // Each edge has source, destinaton and weight of the edge
    static class Edge{
        int source;
        int destination;
        int weight;
        public Edge(int source, int destination, int weight) {
            this.source = source;
            this.destination = destination;
            this.weight = weight;
        }
    }
    // This HeapNode is used for getting the minimum weight edge
    static class HeapNode{
        int vertex;
        int distance;
        // previous Vertex is used to backtrack for printing the path
        int previousVertex;
    }
    // Used to store parent and weight of the edge
    static class Answer{
        int parent;
        int weight;
    }
    // Graph DataStructure Consruction
    static class Graph {
        int vertices;
        LinkedList<Edge>[] adjacencylist;
        Graph(int vertices) {
            this.vertices = vertices;
            adjacencylist = new LinkedList[vertices];
            for (int i = 0; i <vertices ; i++) {
                adjacencylist[i] = new LinkedList<>();
            }
        }
    // add the edge to the Graph (Undirected)
        public void addEdge(int source, int destination, int weight) {
            Edge edge = new Edge(source, destination, weight);
            adjacencylist[source].addFirst(edge);
            edge = new Edge(destination, source, weight);
            adjacencylist[destination].addFirst(edge); //for undirected graph
        }
    // add the edge to the Graph(Directed)
        public void addEdgeDirected(int source, int destination, int weight) {
            Edge edge = new Edge(source, destination, weight);
            adjacencylist[source].addFirst(edge);
        }

    public void dijkstra_GetMinDistances(int sourceVertex){
        int INFINITY = Integer.MAX_VALUE;
        boolean[] SPT = new boolean[vertices];

        HeapNode [] heapNodes = new HeapNode[vertices];
        for (int i = 0; i <vertices ; i++) {
            heapNodes[i] = new HeapNode();
            heapNodes[i].vertex = i;
            heapNodes[i].distance = INFINITY;
            heapNodes[i].previousVertex = 0;
        }

        heapNodes[sourceVertex].distance = 0;
        MinHeap minHeap = new MinHeap(vertices);
        for (int i = 0; i <vertices ; i++) {
            minHeap.insert(heapNodes[i]);
        }
        while(!minHeap.isEmpty()){

            HeapNode extractedNode = minHeap.extractMin();

            int extractedVertex = extractedNode.vertex;
            SPT[extractedVertex] = true;
            LinkedList<Edge> list = adjacencylist[extractedVertex];
            for (int i = 0; i <list.size() ; i++) {
                Edge edge = list.get(i);
                int destination = edge.destination;

                if(SPT[destination]==false ) {

                    int newKey =  heapNodes[extractedVertex].distance + edge.weight
;

                    int currentKey = heapNodes[destination].distance;
                    if(currentKey>newKey){
                        relax(minHeap, newKey, destination);
                        heapNodes[destination].distance = newKey;
                        heapNodes[destination].previousVertex = extractedVertex;
                    }
                }
            }
        }
        printDijkstra(heapNodes, sourceVertex);
    }

        //method contains implementation for Prim's algorithm
        public void primMST(){
            boolean[] Visited = new boolean[vertices];
            Answer[] resultSet = new Answer[vertices];
            int [] key = new int[vertices];
            HeapNode [] heapNodes = new HeapNode[vertices];
            for (int i = 0; i <vertices ; i++) {
                heapNodes[i] = new HeapNode();
                heapNodes[i].vertex = i;
                heapNodes[i].distance = Integer.MAX_VALUE;
                resultSet[i] = new Answer();
                resultSet[i].parent = -1;
                Visited[i] = true;
                key[i] = Integer.MAX_VALUE;
            }

            heapNodes[0].distance = 0;


            MinHeap minHeap = new MinHeap(vertices);
            // Add vertices to the minimum heap
            for (int i = 0; i <vertices ; i++) {
                minHeap.insert(heapNodes[i]);
            }

            //Loop until minheap is not empty
            while(!minHeap.isEmpty()){
                //extract the min vertex
                HeapNode extractedNode = minHeap.extractMin();
                int extractedVertex = extractedNode.vertex;
                Visited[extractedVertex] = false;
                //iterate through all the adjacent vertices
                LinkedList<Edge> list = adjacencylist[extractedVertex];
                for (int i = 0; i <list.size() ; i++) {
                    Edge edge = list.get(i);
                    if(Visited[edge.destination]) {
                        int destination = edge.destination;
                        int newKey = edge.weight;
                        if(key[destination]>newKey) {
                            relax(minHeap, newKey, destination);
                            resultSet[destination].parent = extractedVertex;
                            resultSet[destination].weight = newKey;
                            key[destination] = newKey;
                        }
                    }
                }
            }
            //Print the minimum spanning tree
            printMST(resultSet);
        }

        //relaxing the edges
        public void relax(MinHeap minHeap, int newKey, int vertex){
            //get the index which distance's needs a decrease;
            int index = minHeap.indices[vertex];
            //get the node and update its value
            HeapNode node = minHeap.minHeap[index];
            node.distance = newKey;
            minHeap.upHeap(index);
                }
                public String getVertexKey(Integer getVertexValue)
                {
                    String sourceVertexChar;
                    for (Map.Entry entry : mapVertices.entrySet()) {
                        if(getVertexValue.equals(entry.getValue())) {
                    sourceVertexChar = entry.getKey().toString();
                    return sourceVertexChar;
                }
            }
            return null;
        }

        public void printDijkstra(HeapNode[] resultSet, int sourceVertex) {
            System.out.println("Dijkstra Algorithm: (Adjacency List + Min Heap)");
            Integer value;
            Integer sourceVertexValue;
            String sourceVertexChar=null;
            sourceVertexValue =sourceVertex;
            sourceVertexChar = getVertexKey(sourceVertexValue);

            int counter=0;
            for (int i = 0; i < vertices; i++) {

                value =i;
                for (Map.Entry entry : mapVertices.entrySet()) {

                    if(sourceVertexValue == entry.getValue()  && counter<=0)
                    {
                        System.out.println("Source Vertex: " + sourceVertexChar + "
to vertex: " + entry.getKey() +
                                " distance: " + resultSet[i].distance);
                        counter++;
                    }
                    else if(value.equals(entry.getValue()) && (sourceVertexValue !=
entry.getValue())) {
                        int printCounter=0;
                        if(resultSet[i].distance >=250000000 ||
resultSet[i].distance <= -250000000)
                        {
                            System.out.println("Source Vertex " + sourceVertexChar
+ " to vertex " + entry.getKey() +
                                    " distance: " + "INFINITY" + " Previous Vertex
" + getVertexKey(resultSet[i].previousVertex));
                        }
                        else
                        {
                            System.out.println("Source Vertex " + sourceVertexChar
+ " to vertex " + entry.getKey() +
                                    " distance: " + resultSet[i].distance + "
Previous Vertex " + getVertexKey(resultSet[i].previousVertex));
                        }
                        int pathVertex =
Integer.parseInt(entry.getValue().toString());
                        ArrayList<Integer> PathList = new ArrayList<Integer>();
                        if(resultSet[i].distance >=250000000 ||
resultSet[i].distance <= -250000000) {
                            System.out.println("Path: Can not be reached");
                        }
                        else {
                            while (pathVertex != sourceVertex) {
                                PathList.add(pathVertex);
                                pathVertex = resultSet[pathVertex].previousVertex;
                            }
                            Collections.reverse(PathList);
                            Iterator itr = PathList.iterator();
                            System.out.print("Path:");
                            System.out.print(getVertexKey(sourceVertex));
                            while (itr.hasNext()) {
                                int t = Integer.parseInt(itr.next().toString());
                                System.out.print("-->");
                                System.out.print(getVertexKey(t));
                            }
                            System.out.println();
                        }
                        break;
                    }
                }
            }
        }

        public void printMST(Answer[] resultSet){
            int total_min_weight = 0;
            System.out.println();
            System.out.println();
            System.out.println("Minimum Spanning Tree Using Prims Algorithm: ");
            for (int i = 1; i <vertices ; i++) {
                System.out.println("Edge: " + getVertexKey(i) + " - " +
getVertexKey(resultSet[i].parent) +
                        " weight: " + resultSet[i].weight);
                total_min_weight += resultSet[i].weight;
            }
            System.out.println("Total minimum distance: " + total_min_weight);
        }
    }


    public static void main(String[] args) {

        int vertices;
        int edges;
        char source;
        char destination;
        int distance;
        char sourceVertex;
        HashSet<Character> verticesSet = new HashSet<Character>();
        String Type;
        String presentWorkingDirectory = System.getProperty("user.dir");
        File file = new File(presentWorkingDirectory + "//input2.txt");
        try {
            BufferedReader br = new BufferedReader(new FileReader(file));
            String st;
            String st1;
            st = br.readLine();
            String[] A = new String[3];
            A = st.split("\\s+");
            vertices = Integer.parseInt(A[0]);
            edges = Integer.parseInt(A[1]);
            Type = A[2];
            br.mark(MARK_LENGTH);

            int numberVertices =0;
            Graph graph = new Graph(vertices);
            for(int i=0;i<edges;i++)
            {
                st = br.readLine();
                A = st.split("\\s+");
                source = A[0].charAt(0);
                destination = A[1].charAt(0);
                distance = Integer.parseInt(A[2]);
                verticesSet.add(source);
                verticesSet.add(destination);
            }
            br.reset();
            Iterator<Character> itr=verticesSet.iterator();
            while(itr.hasNext()){
                mapVertices.put(itr.next(),numberVertices);
                numberVertices++;
            }
            for(int i=0;i<edges;i++)
            {
                st = br.readLine();
                A = st.split("\\s+");
                source = A[0].charAt(0);
                destination = A[1].charAt(0);
                distance = Integer.parseInt(A[2]);
                if(Type.equals("U"))

graph.addEdge(mapVertices.get(source),mapVertices.get(destination),distance);
                else

graph.addEdgeDirected(mapVertices.get(source),mapVertices.get(destination),distance
);
            }
            st = br.readLine();
            if(st!=null)
            {
                sourceVertex = st.charAt(0);
                graph.dijkstra_GetMinDistances(mapVertices.get(sourceVertex));
                graph.primMST();
            }
            else
            {
                Object[] hashMapKeys = mapVertices.keySet().toArray();
                Object randomKey = hashMapKeys[new
Random().nextInt(hashMapKeys.length)];
                sourceVertex = randomKey.toString().charAt(0);
                System.out.println("Source Vertex taken is : " + sourceVertex);
                graph.dijkstra_GetMinDistances(mapVertices.get(sourceVertex));
                graph.primMST();
            }
        }
        catch (Exception e)
        {
            System.out.println(e);
        }
    }
}
```

MinHeap.java:

```
// MinHeap DataStructure
public class MinHeap{
    int capacity;
    int currentSize;
    DijkstraandPrim.HeapNode[] minHeap;
    int [] indices;


    public MinHeap(int capacity) {
        this.capacity = capacity;
        minHeap = new DijkstraandPrim.HeapNode[capacity + 1];
        indices = new int[capacity];
        minHeap[0] = new DijkstraandPrim.HeapNode();
        minHeap[0].distance = Integer.MIN_VALUE;
        minHeap[0].vertex=-1;
        currentSize = 0;
    }

    public void insert(DijkstraandPrim.HeapNode x) {
        currentSize++;
        int index = currentSize;
        minHeap[index] = x;
        indices[x.vertex] = index;
        upHeap(index);
    }

    public void upHeap(int pos) {
        int parentIndex = pos/2;
        int currentIndex = pos;
        while (currentIndex > 0 && minHeap[parentIndex].distance >
minHeap[currentIndex].distance) {
            DijkstraandPrim.HeapNode currentNode = minHeap[currentIndex];
            DijkstraandPrim.HeapNode parentNode = minHeap[parentIndex];
            indices[currentNode.vertex] = parentIndex;
            indices[parentNode.vertex] = currentIndex;
            swap(currentIndex,parentIndex);
            currentIndex = parentIndex;
            parentIndex = parentIndex/2;
        }
    }

    public DijkstraandPrim.HeapNode extractMin() {
        DijkstraandPrim.HeapNode min = minHeap[1];
        DijkstraandPrim.HeapNode lastNode = minHeap[currentSize];

        indices[lastNode.vertex] = 1;
        minHeap[1] = lastNode;
        minHeap[currentSize] = null;
        downHeap(1);
        currentSize--;
        return min;
    }

    public void downHeap(int k) {
        int smallest = k;
        int leftChildIndex = 2 * k;
        int rightChildIndex = 2 * k+1;
        if (leftChildIndex < heapSize() && minHeap[smallest].distance >
minHeap[leftChildIndex].distance) {
            smallest = leftChildIndex;
        }
        if (rightChildIndex < heapSize() && minHeap[smallest].distance >
minHeap[rightChildIndex].distance) {
            smallest = rightChildIndex;
        }
        if (smallest != k) {
            DijkstraandPrim.HeapNode smallestNode = minHeap[smallest];
            DijkstraandPrim.HeapNode kNode = minHeap[k];
            indices[smallestNode.vertex] = k;
            indices[kNode.vertex] = smallest;
            swap(k, smallest);
            downHeap(smallest);
        }
    }

    public void swap(int a, int b) {
        DijkstraandPrim.HeapNode temp = minHeap[a];
        minHeap[a] = minHeap[b];
        minHeap[b] = temp;
    }

    public boolean isEmpty() {
        return currentSize == 0;
    }

    public int heapSize(){
        return currentSize;
    }
}
```