1. Implement the Merge Sort algorithm. Create a method to measure the time it takes to sort an array using Merge Sort. Generate or read different sets of elements to be sorted with varying values of n Run the sorting algorithm for each set of elements and record the time taken. Plot a graph of time taken versus the number of elements.

Java program

```
public class MergeSortTimeAnalysis {
    public static void main(String[] args) {
        // Define an array of different values of n
        int[] nValues = {1000, 5000, 10000, 20000, 30000};

        // Create a Random object for generating random numbers
        Random random = new Random();

        // Iterate through different values of n
        for (int n : nValues) {
            int[] arr = new int[n];

            // Fill the array with random integers
            for (int i = 0; i < n; i++) {
                arr[i] = random.nextInt(100000); // Adjust the range as needed
            }

            // Record the start time
            long startTime = System.nanoTime();

            // Call the merge sort function
            mergeSort(arr);

            // Record the end time
            long endTime = System.nanoTime();

            // Calculate the time taken in milliseconds
            long timeTaken = (endTime - startTime) / 1000000;

            // Print the result
            System.out.println("Time taken to sort " + n + " elements: " +
timeTaken + " ms");
        }
    }

    public static void mergeSort(int[] arr) {
        if (arr.length <= 1) {
            return;
        }

        // Split the array into two halves
        int mid = arr.length / 2;
        int[] left = Arrays.copyOfRange(arr, 0, mid);
        int[] right = Arrays.copyOfRange(arr, mid, arr.length);

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

        // Merge the sorted halves
        merge(arr, left, right);
    }

    public static void merge(int[] arr, int[] left, int[] right) {
        int i = 0, j = 0, k = 0;

        while (i < left.length && j < right.length) {
            if (left[i] < right[j]) {
                arr[k++] = left[i++];
            } else {
                arr[k++] = right[j++];
            }
        }

        while (i < left.length) {
            arr[k++] = left[i++];
        }

        while (j < right.length) {
            arr[k++] = right[j++];
        }
    }
}
```

2) To solve the problem of implementing Quick Sort in Java, determining worst and average case time complexity, and plotting a graph of time taken versus the number of elements, follow these steps:

->Implement Quick Sort in Java:

```
public class QuickSort {
    public static void quickSort(int[] arr, int low, int high) {
        if (low < high) {
            int pivotIndex = partition(arr, low, high);
            quickSort(arr, low, pivotIndex - 1);
            quickSort(arr, pivotIndex + 1, high);
        }
    }

    private static 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;
    }
}
```

-> Write code to measure time taken for sorting:

```
java
import java.util.Random;

public class Main {
    public static void main(String[] args) {
        int[] nValues = {10, 100, 1000, 10000}; // Different values of n
        for (int n : nValues) {
            int[] arr = generateRandomArray(n);
            long startTime = System.nanoTime();
            QuickSort.quickSort(arr, 0, n - 1);
            long endTime = System.nanoTime();
            long elapsedTime = endTime - startTime;
            System.out.println("Time taken to sort " + n + " elements: " +
elapsedTime + " ns");
        }
    }

    private static int[] generateRandomArray(int n) {
        int[] arr = new int[n];
        Random random = new Random();
        for (int i = 0; i < n; i++) {
            arr[i] = random.nextInt(10000); // Adjust the range as needed
        }
        return arr;
    }
}
```

3) insert and delete operations in a Binary Search Tree (BST)

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

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

class BST {
    Node root;

    BST() {
        root = null;
    }

    void insert(int key) {
        root = insertRec(root, key);
    }

    Node insertRec(Node root, int key) {
        if (root == null) {
            root = new Node(key);
            return root;
        }

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

        return root;
    }

    void delete(int key) {
        root = deleteRec(root, key);
    }

    Node deleteRec(Node root, int key) {
        if (root == null) {
            return root;
        }

        if (key < root.key) {
            root.left = deleteRec(root.left, key);
        } else if (key > root.key) {
            root.right = deleteRec(root.right, key);
        } else {
            if (root.left == null) {
                return root.right;
            } else if (root.right == null) {
                return root.left;
            }

            root.key = minValue(root.right);

            root.right = deleteRec(root.right, root.key);
        }

        return root;
    }

    int minValue(Node root) {
        int minValue = root.key;
        while (root.left != null) {
            minValue = root.left.key;
            root = root.left;
        }
        return minValue;
    }
}
```

4) Breadth-First Search (BFS) method:

java
```
import java.util.*;

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

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

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

    void BFS(int s) {
        boolean visited[] = new boolean[V];
        LinkedList<Integer> queue = new LinkedList<>();

        visited[s] = true;
        queue.add(s);
        while (queue.size() != 0) {
            s = queue.poll();
            System.out.print(s + " ");

            Iterator<Integer> i = adjList[s].listIterator();
            while (i.hasNext()) {
                int n = i.next();
                if (!visited[n]) {
                    visited[n] = true;
                    queue.add(n);
                }
            }
        }
    }
}

public class Main {
    public static void main(String args[]) {
        Graph g = new Graph(7); // Change the number of vertices as needed
        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 3);
        g.addEdge(1, 4);
        g.addEdge(2, 5);
        g.addEdge(2, 6);

        int startNode = 0; // Change the starting node as needed
        System.out.println("Nodes reachable from node " + startNode + " using
BFS:");
        g.BFS(startNode);
    }
}
```

5.a) obtaining the topological ordering of vertices in a directed graph.

```
java
import java.util.*;

class Graph {
    private int V; // Number of vertices
    private LinkedList<Integer>[] adj; // Adjacency list

    Graph(int v) {
        V = v;
        adj = new LinkedList[v];
        for (int i = 0; i < v; i++) {
            adj[i] = new LinkedList<>();
        }
    }

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

    void topologicalSortUtil(int v, boolean[] visited, Stack<Integer> stack) {
        visited[v] = true;
        for (Integer neighbor : adj[v]) {
            if (!visited[neighbor]) {
                topologicalSortUtil(neighbor, visited, stack);
            }
        }
        stack.push(v);
    }

    void topologicalSort() {
        Stack<Integer> stack = new Stack<>();
        boolean[] visited = new boolean[V];

        for (int i = 0; i < V; i++) {
            if (!visited[i]) {
                topologicalSortUtil(i, visited, stack);
            }
        }

        while (!stack.isEmpty()) {
            System.out.print(stack.pop() + " ");
        }
    }
}

public class TopologicalOrderingExample {
    public static void main(String[] args) {
        Graph graph = new Graph(6);
        graph.addEdge(5, 2);
        graph.addEdge(5, 0);
        graph.addEdge(4, 0);
        graph.addEdge(4, 1);
        graph.addEdge(2, 3);
        graph.addEdge(3, 1);

        System.out.println("Topological Sort: ");
        graph.topologicalSort();
    }
}
```

6)compute the transitive closure of a given directed graph using Warshall's algorithm.

```
java
import java.util.*;

class WarshallTransitiveClosure {
    private int V;
    private boolean[][] tc;

    WarshallTransitiveClosure(int vertices) {
        V = vertices;
        tc = new boolean[vertices][vertices];
    }

    void addEdge(int from, int to) {
        tc[from][to] = true;
    }

    void computeTransitiveClosure() {
        for (int k = 0; k < V; k++) {
            for (int i = 0; i < V; i++) {
                for (int j = 0; j < V; j++) {
                    tc[i][j] = tc[i][j] || (tc[i][k] && tc[k][j]);
                }
            }
        }
    }

    void printTransitiveClosure() {
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                System.out.print(tc[i][j] ? "1 " : "0 ");
            }
            System.out.println();
        }
    }
}

public class WarshallTransitiveClosureExample {
    public static void main(String[] args) {
        int vertices = 4;
        WarshallTransitiveClosure g = new WarshallTransitiveClosure(vertices);

        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);

        System.out.println("Transitive Closure:");
        g.computeTransitiveClosure();
        g.printTransitiveClosure();
    }
}
```

7)Heap Sort

```
java
import org.knowm.xchart.*;

import java.util.Random;

class HeapSort {
    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 swap = arr[i];
            arr[i] = arr[largest];
            arr[largest] = swap;

            heapify(arr, n, largest);
        }
    }

    void heapSort(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[0];
            arr[0] = arr[i];
            arr[i] = temp;

            heapify(arr, i, 0);
        }
    }
}
```

8)search for a pattern string in a given text using the Horspool String Matching algorithm.

```
java
import java.util.HashMap;
import java.util.Map;

public class HorspoolStringMatching {
    public static Map<Character, Integer> preprocessShiftTable(String pattern) {
        Map<Character, Integer> shiftTable = new HashMap<>();
        int m = pattern.length();

        for (int i = 0; i < m - 1; i++) {
            char c = pattern.charAt(i);
            shiftTable.put(c, m - i - 1);
        }

        return shiftTable;
    }

    public static int searchPattern(String text, String pattern) {
        int n = text.length();
        int m = pattern.length();

        Map<Character, Integer> shiftTable = preprocessShiftTable(pattern);

        int i = m - 1; // Start comparing from the end of the pattern
        while (i < n) {
            int j = 0;

            while (j < m && text.charAt(i - j) == pattern.charAt(m - 1 - j)) {
                j++;
            }

            if (j == m) {
                return i - m + 1; // Pattern found
            }

            char lastChar = text.charAt(i);
            if (shiftTable.containsKey(lastChar)) {
                i += shiftTable.get(lastChar);
            } else {
                i += m; // Shift by pattern length if character not in pattern
            }
        }

        return -1; // Pattern not found
    }
    public static void main(String[] args) {
        String text = "This is an example of Horspool String Matching algorithm";
        String pattern = "algorithm";

        int index = searchPattern(text, pattern);

        if (index != -1) {
            System.out.println("Pattern found at index " + index);
        } else {
            System.out.println("Pattern not found in the text");
        }
    }
}
```