ASM-Part1-Data Structures & Algorithms

Full Name: NGUYEN NHU PHUC
MSSV: BH01072

Table of content

I. A stack ADT, a concrete data structure for a First In First out (FIFO) queue.

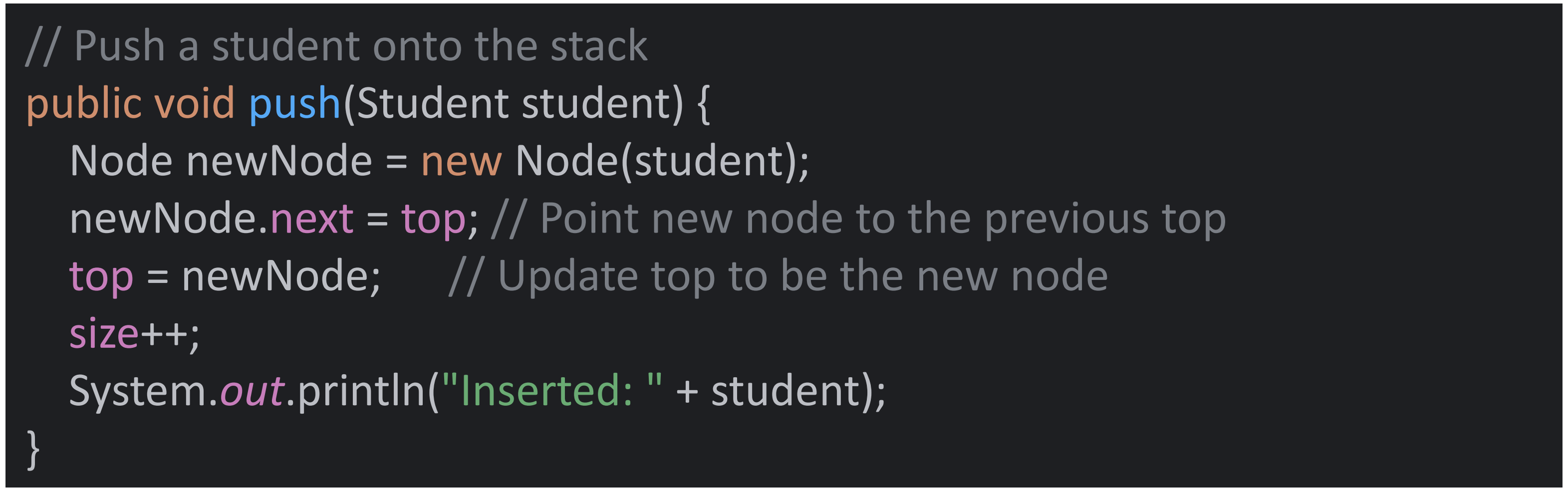
A Stack ADT and a Queue ADT are two different abstract data types, and they operate based on different principles:

- Stack: Follows Last In, First Out (LIFO).
- Queue: Follows First In, First Out (FIFO).

It seems there might be confusion in your query as a stack cannot inherently behave like a FIFO queue because their operational rules differ. Below, I'll explain Stack ADT and the differences from Queue ADT clearly.

I. A stack ADT, a concrete data structure for a First In First out (FIFO) queue.

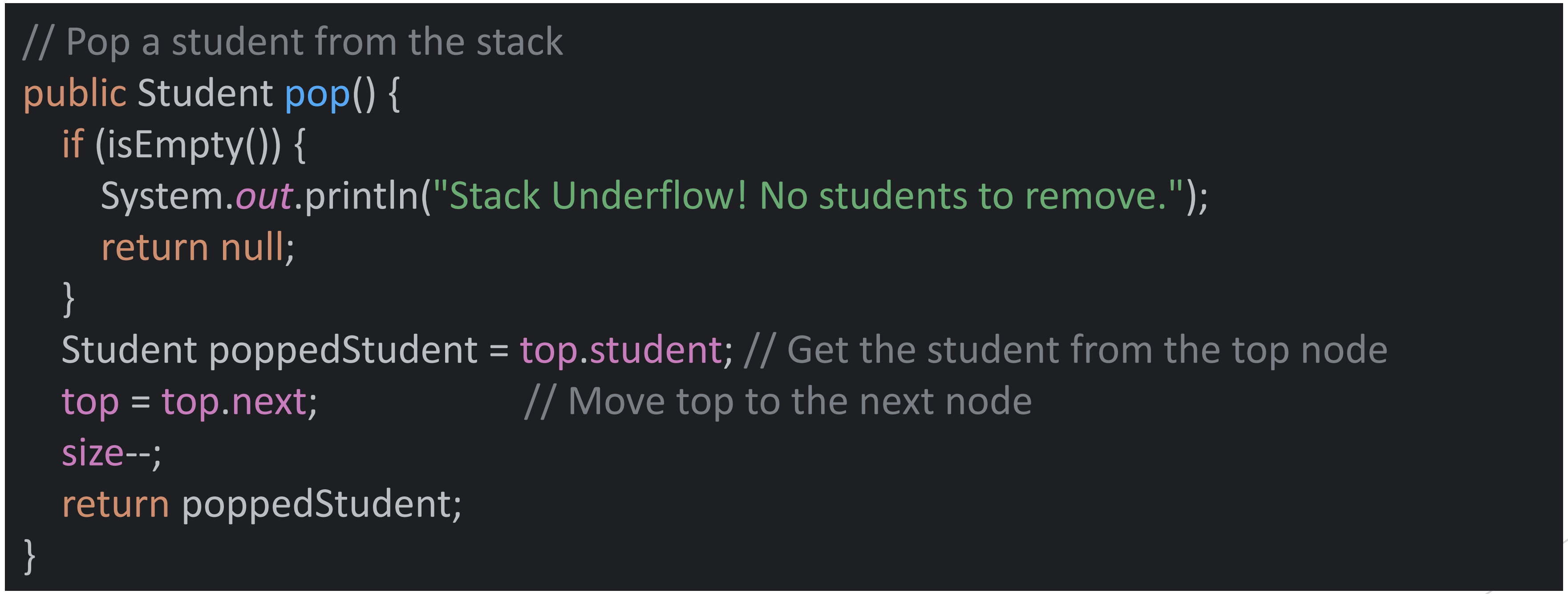
Stack ADTA Stack ADT is an abstract data type that provides the following operations:
Push: Add an element to the top of the stack.



```
// Push a student onto the stack
public void push(Student student) {
   Node newNode = new Node(student);
   newNode.next = top; // Point new node to the previous top
   top = newNode;      // Update top to be the new node
   size++;
   System.out.println("Inserted: " + student);
}
```

I. A stack ADT, a concrete data structure for a First In First out (FIFO) queue.

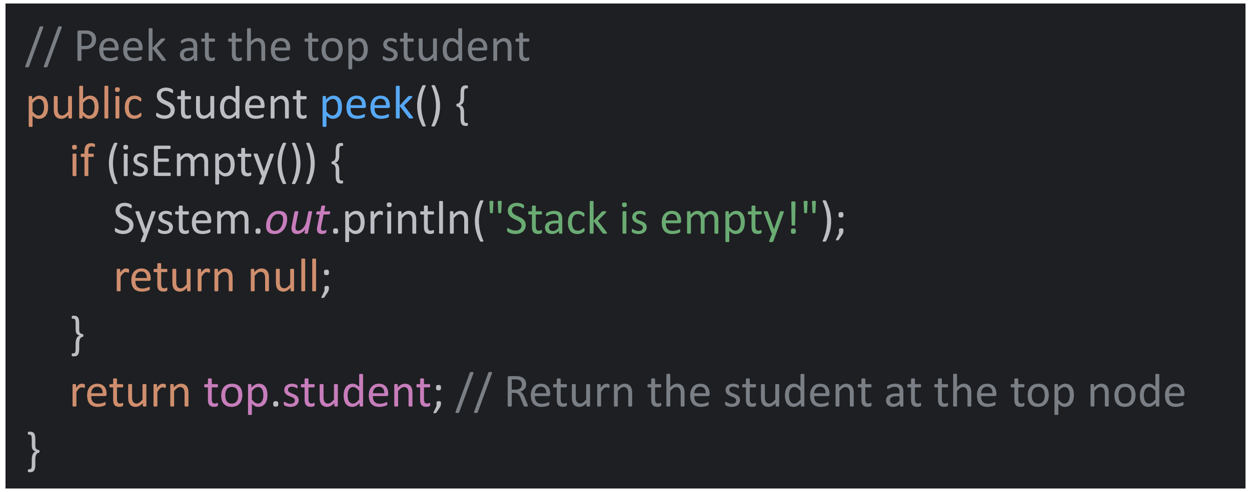
Stack ADTA Stack ADT is an abstract data type that provides the following operations:
Pop: Remove the top element from the stack.



```java
// Pop a student from the stack
public Student pop() {
    if (isEmpty()) {
        System.out.println("Stack Underflow! No students to remove.");
        return null;
    }
    Student poppedStudent = top.student; // Get the student from the top node
    top = top.next;                      // Move top to the next node
    size--;
    return poppedStudent;
}
```

I. A stack ADT, a concrete data structure for a First In First out (FIFO) queue.

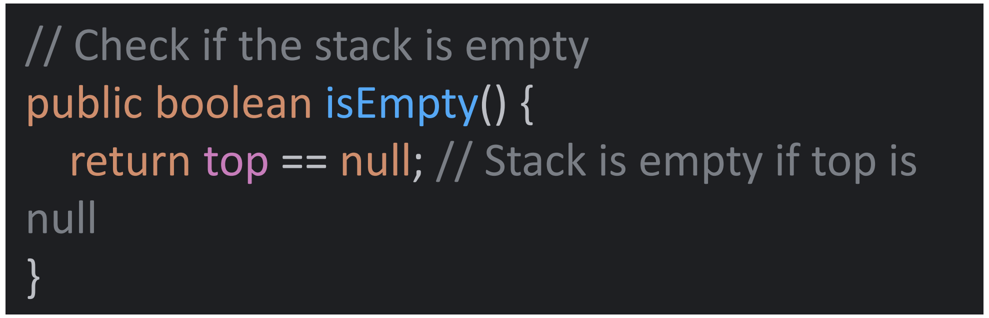
Stack ADTA Stack ADT is an abstract data type that provides the following operations:
Peek/Top: View the top element without removing it.



```
// Peek at the top student
public Student peek() {
    if (isEmpty()) {
        System.out.println("Stack is empty!");
        return null;
    }
    return top.student; // Return the student at the top node
}
```

I. A stack ADT, a concrete data structure for a First In First out (FIFO) queue.

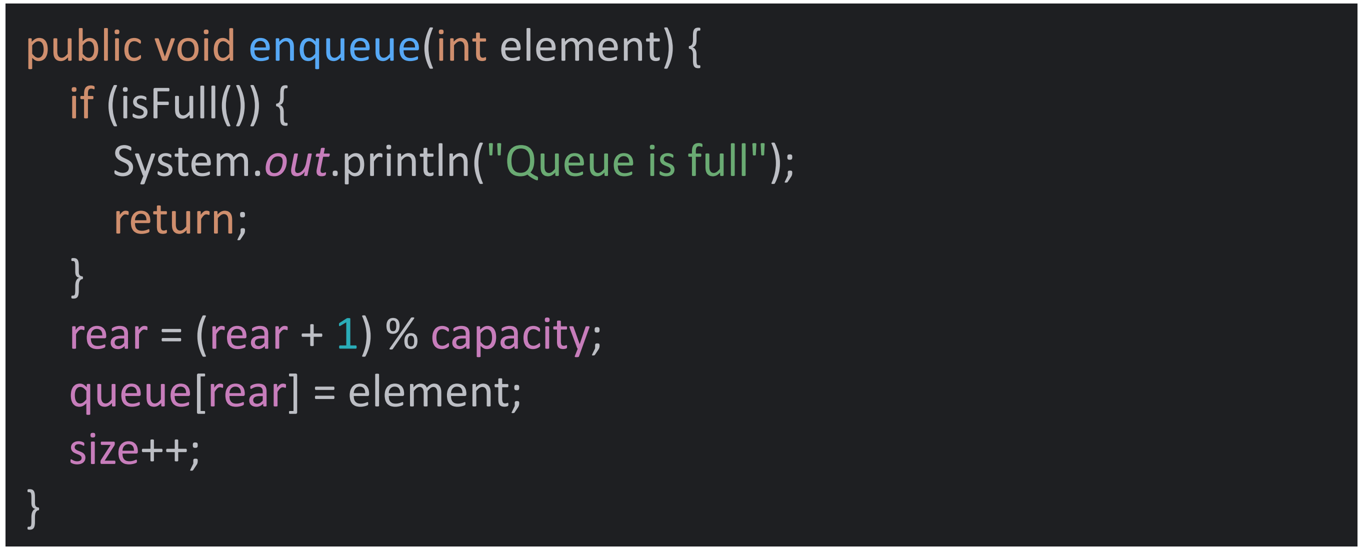
Stack ADTA Stack ADT is an abstract data type that provides the following operations:
IsEmpty: Check if the stack is empty.



```
// Check if the stack is empty
public boolean isEmpty() {
   return top == null; // Stack is empty if top is
null
}
```

I. A stack ADT, a concrete data structure for a First In First out (FIFO) queue.

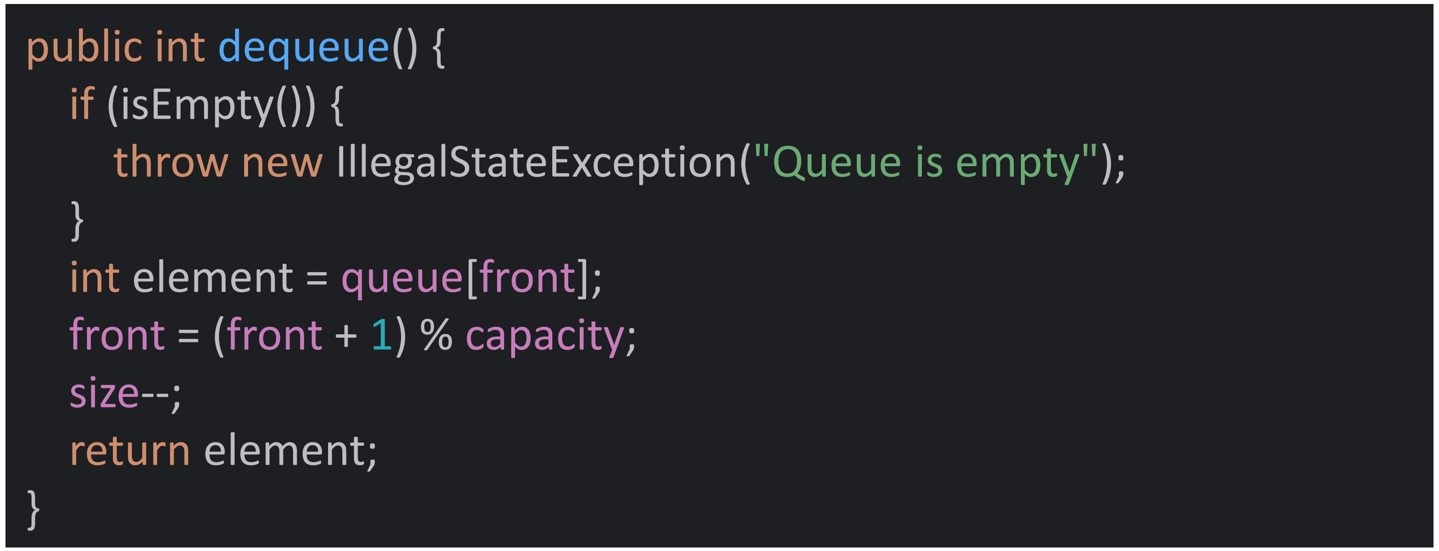
Queue ADT
Enqueue: Add an element to the rear of the queue.



```
public void enqueue(int element) {
    if (isFull()) {
        System.out.println("Queue is full");
        return;
    }
    rear = (rear + 1) % capacity;
    queue[rear] = element;
    size++;
}
```

I. A stack ADT, a concrete data structure for a First In First out (FIFO) queue.

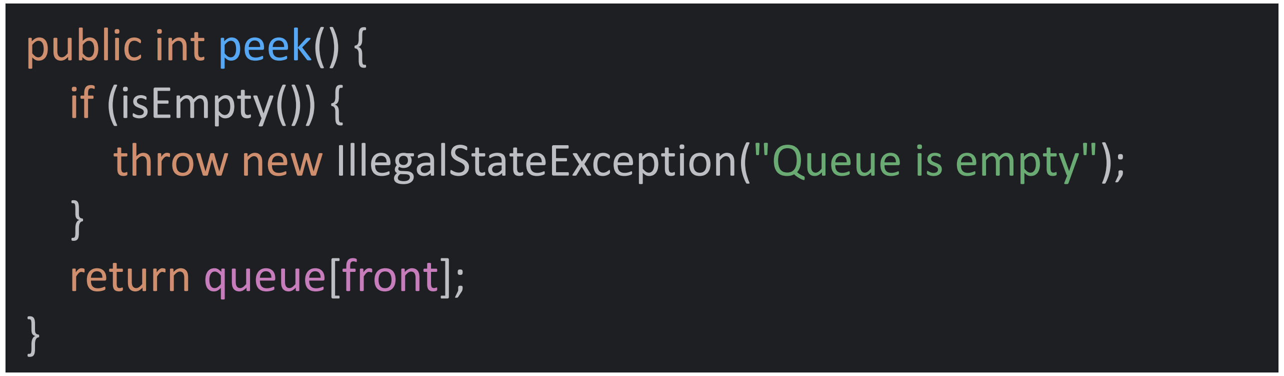
Queue ADT
Dequeue: Remove the element at the front of the queue.



```java
public int dequeue() {
   if (isEmpty()) {
      throw new IllegalStateException("Queue is empty");
   }
   int element = queue[front];
   front = (front + 1) % capacity;
   size--;
   return element;
}
```

I. A stack ADT, a concrete data structure for a First In First out (FIFO) queue.

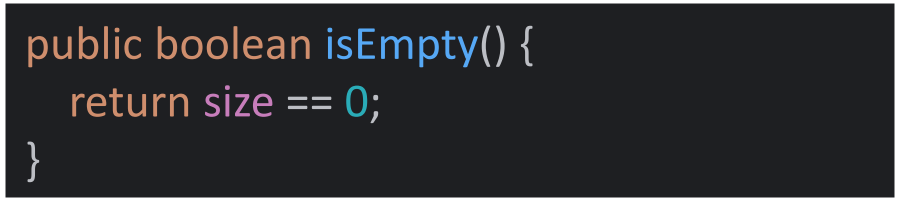
Queue ADT
Peek/Front: View the front element without removing it.



```
public int peek() {
    if (isEmpty()) {
        throw new IllegalStateException("Queue is empty");
    }
    return queue[front];
}
```

I. A stack ADT, a concrete data structure for a First In First out (FIFO) queue.

Queue ADT
IsEmpty: Check if the queue is empty.



```
public boolean isEmpty() {
    return size == 0;
}
```

I. A stack ADT, a concrete data structure for a First In First out (FIFO) queue.

1. DSA (Data Structures and Algorithms):

- Data Structures: Organized ways to store and manipulate data (e.g., arrays, linked lists, stacks, queues, trees, graphs).
- Algorithms: Step-by-step methods or instructions to solve problems or perform operations on data structures (e.g., searching, sorting).

2. What is ADT?

An ADT is a mathematical model for data types that specifies the behavior and operations that can be performed on data without specifying their implementation.

Examples include Stack, Queue, List, Set, etc.

ADTs focus on what the data structure does rather than how it does it.

3. Compare different between Stack and Queue

Aspect	Stack	Queue
Definition	Follows Last In First Out (LIFO). The last element added is the first to be removed.	Follows First In First Out (FIFO). The first element added is the first to be removed.
Operations	Push (insert), Pop (remove), Peek (view top).	Enqueue (insert), Dequeue (remove), Peek (view front).
Usage Examples	Function calls, backtracking, undo operations.	Scheduling, buffering (e.g., printer queue, network packets).
Insertion/Removal	Operates on the same end (top).	Operates on opposite ends: insertion at the rear, removal from the front.
Visualization	Stack of plates.	Waiting line (queue) at a ticket counter.

4. How many ways are there to implement Stack and Queue?

1, Array-Based Implementation:

- Use a static array to hold elements.
- Advantages: Simple to implement, efficient for small, fixed sizes.
- Disadvantages: Fixed size may lead to overflow or underutilization.

2, Linked List-Based Implementation:

- Use nodes dynamically allocated to store elements.
- Advantages: Dynamic size, no overflow if memory is available.
- Disadvantages: Extra memory overhead for pointers.

4. How many ways are there to implement Stack and Queue?

3, Using Python's Built-in Data Structures:

- Stack: Use list with append() for push and pop() for removal.
- Queue: Use collections.deque for efficient FIFO operations or queue.Queue for thread-safe queues.

4, Circular Queue for Queues Only:

- Use an array with wrap-around logic to efficiently utilize space.
- Useful for fixed-size buffering.

5,Stack Using Two Queues / Queue Using Two Stacks:

- Simulate one structure using the other for specific requirements.

II. Two sorting algorithms.

1. Merge Sort

Merge Sort is another divide-and-conquer algorithm that recursively splits the array into two halves, sorts them, and then merges the sorted halves back together.

Time Complexity

•**Best Case**: O(nlogn)

•**Average Case**: O(nlogn)

•**Worst Case**: O(nlogn)

Space Complexity

•Requires O(n) auxiliary space for temporary arrays used during the merging process.

Characteristics

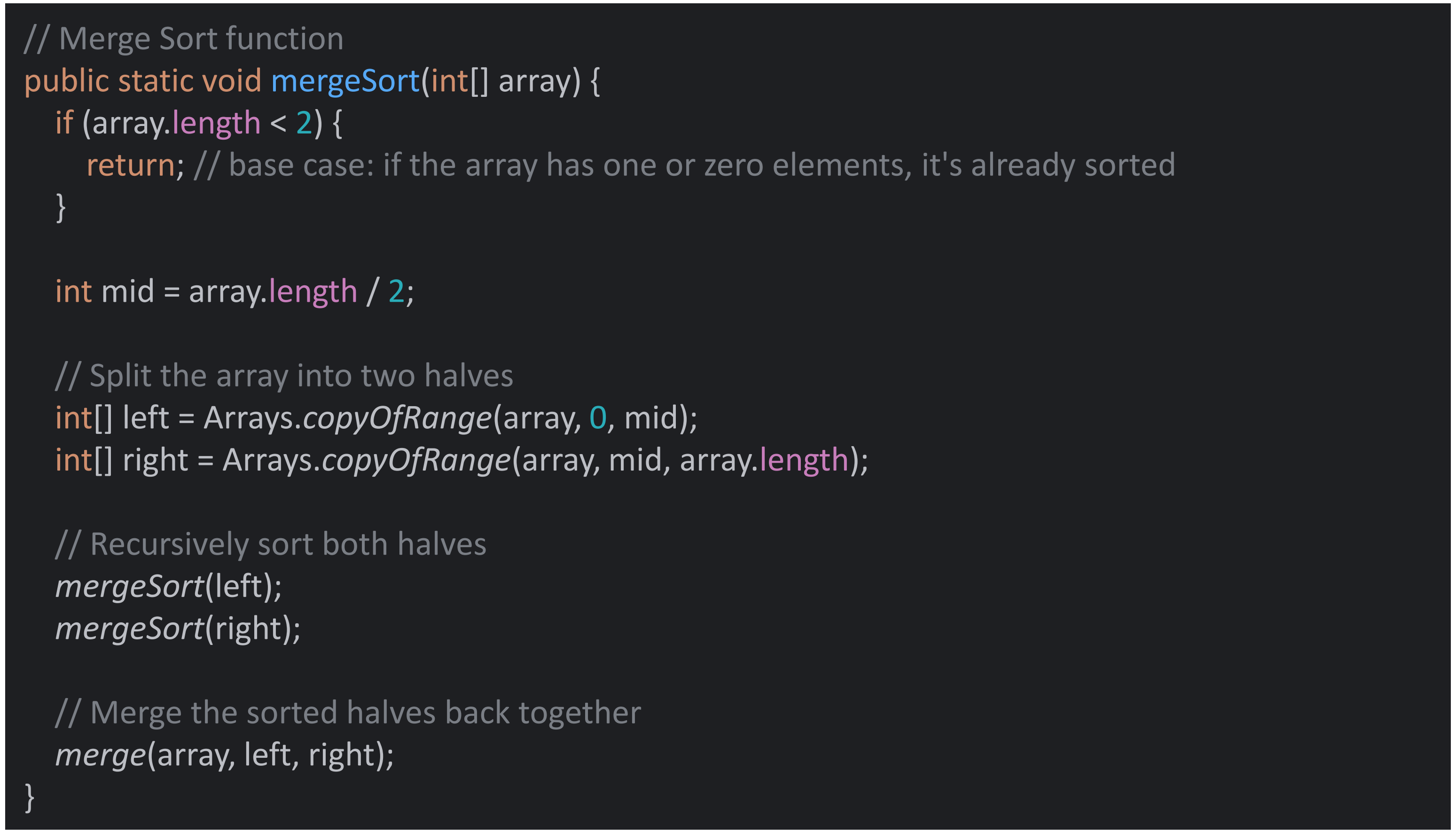
•Stable (preserves the relative order of equal elements).

•Performs well on large datasets but has higher space requirements than Quick Sort.

II. Two sorting algorithms.

1. Merge Sort

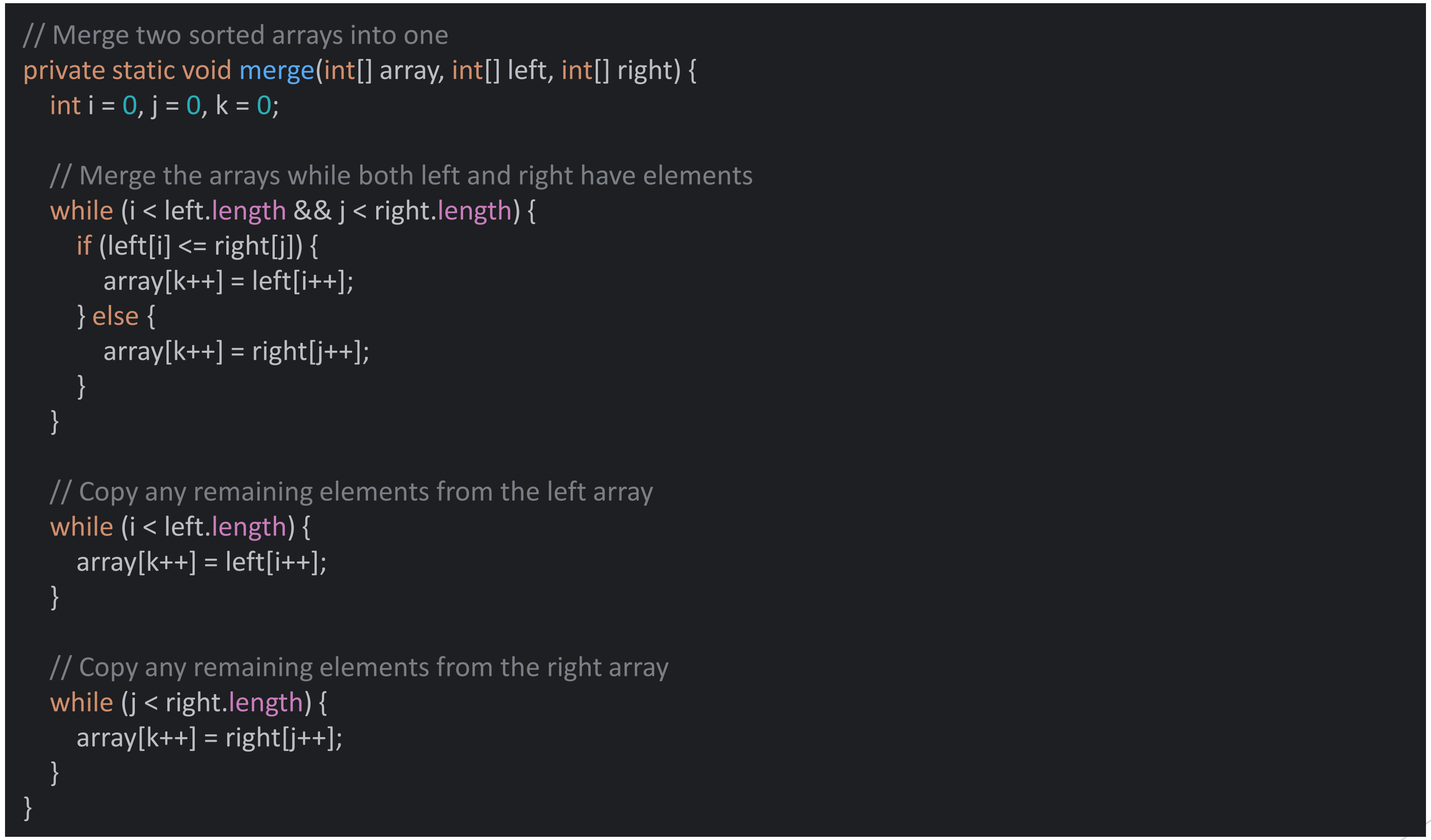
Coding



```
// Merge Sort function
public static void mergeSort(int[] array) {
    if (array.length < 2) {
        return; // base case: if the array has one or zero elements, it's already sorted
    }

    int mid = array.length / 2;

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

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

    // Merge the sorted halves back together
    merge(array, left, right);
}
```

II. Two sorting algorithms.

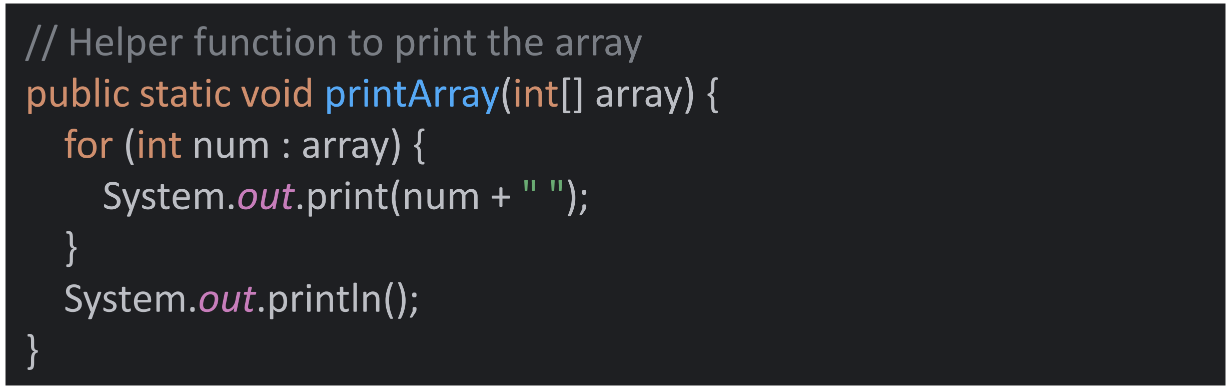
1. Merge Sort



```
// Merge two sorted arrays into one
private static void merge(int[] array, int[] left, int[] right) {
  int i = 0, j = 0, k = 0;

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

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

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

II. Two sorting algorithms.

1. Merge Sort



```
// Helper function to print the array
public static void printArray(int[] array) {
   for (int num : array) {
      System.out.print(num + " ");
   }
   System.out.println();
}
```

II. Two sorting algorithms.

2. Quick Sort

Quick Sort is a divide-and-conquer algorithm that selects a "pivot" element and partitions the array around it, placing elements smaller than the pivot to its left and larger elements to its right. It then recursively sorts the subarrays.

Time Complexity

•**Best Case**: O(nlogn) (When the pivot divides the array into two nearly equal halves)

•**Average Case**: O(nlogn)

•**Worst Case**: O(n2) (When the pivot is consistently the smallest or largest element)

Space Complexity

•**In-place Sorting**: Requires O(logn) auxiliary space for the recursion stack in the average case.

•**Worst Case**: O(n) (in case of skewed partitioning).

Characteristics

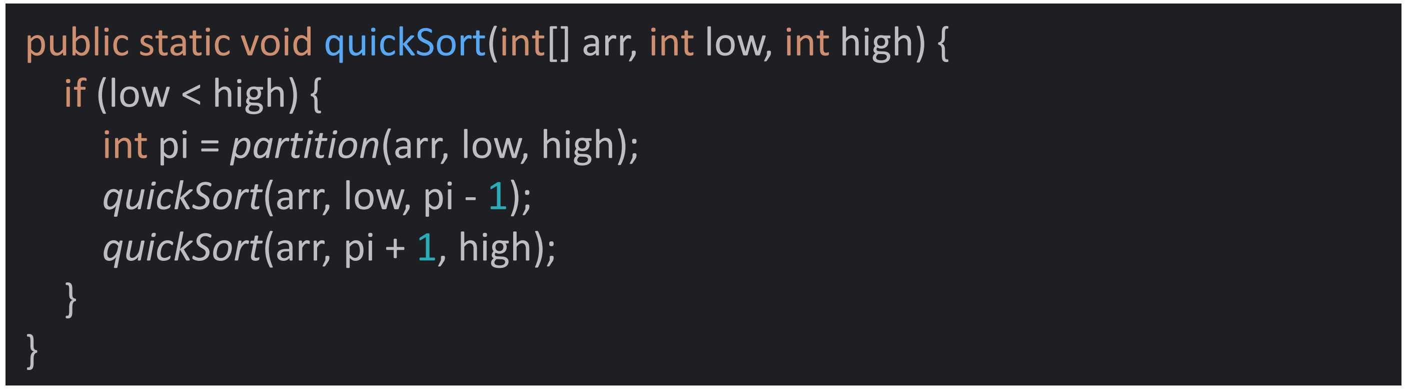
•Not stable (does not preserve the relative order of equal elements).

•Efficient for small to medium-sized datasets.

•Can be improved with randomized pivot selection to reduce the likelihood of worst-case scenarios.

II. Two sorting algorithms.

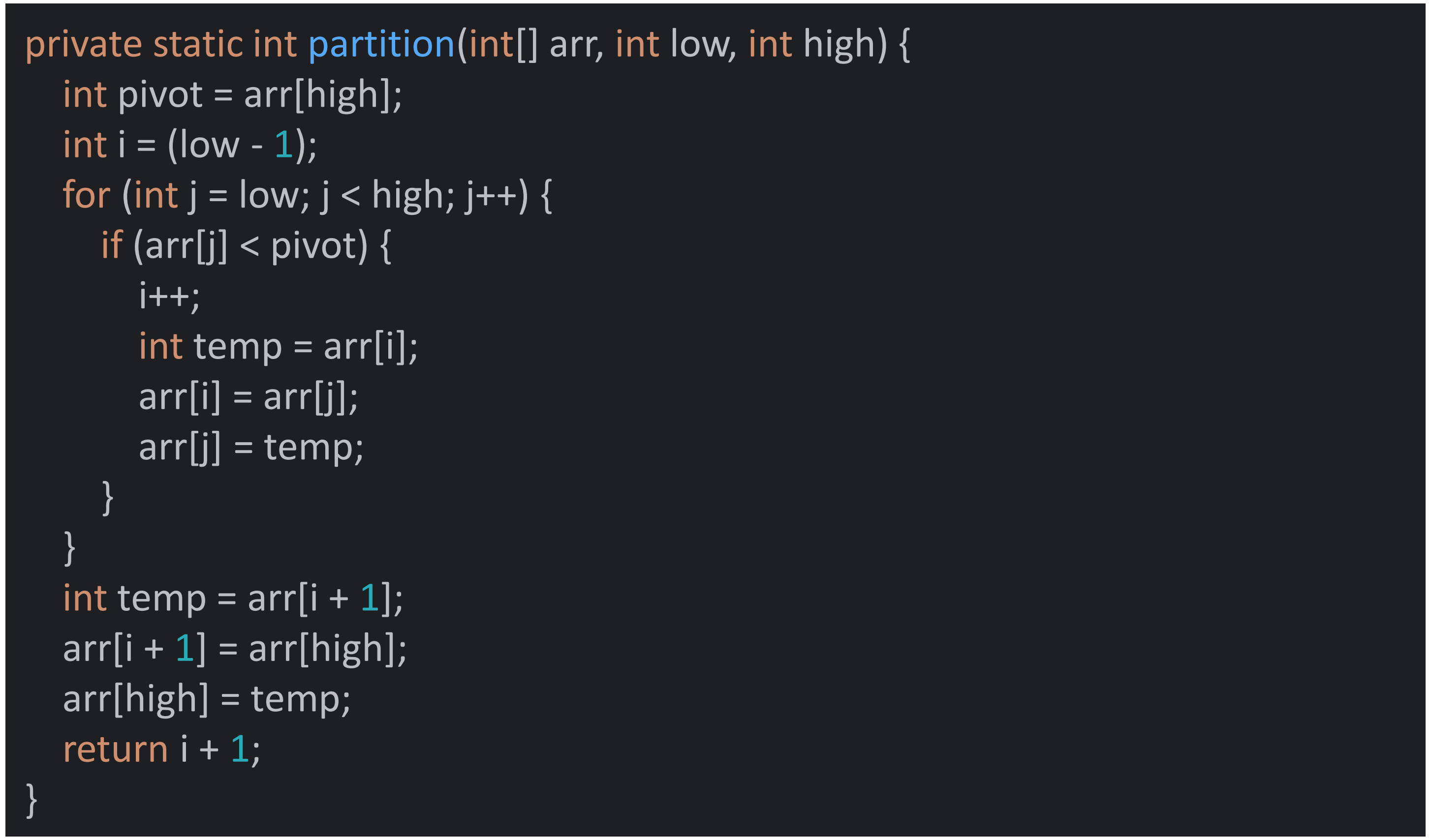
2. Quick Sort
Coding



```java
public static 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);
    }
}
```

II. Two sorting algorithms.

2. Quick Sort



```java
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;
}
```

II. Two sorting algorithms.

Comparison Table

Aspect	Merge Sort	Quick Sort
Best Case Time	O(nlogn)	O(nlogn)
Average Case Time	O(nlogn)	O(nlogn)
Worst Case Time	O(nlogn)	O(n 2)
Space Complexity	O(n) (extra arrays)	O(logn) (in-place)
Stability	Stable	Not stable
Use Cases	Large datasets; stability needed	Small/medium datasets

II. Two sorting algorithms.

When to Choose Each?

- Quick Sort is preferred for in-place sorting when memory usage is critical and the dataset is not already sorted or close to sorted.

- Merge Sort is better for larger datasets or when stability is important, such as sorting linked lists or ensuring consistent ordering of duplicate elements.

III. Analyse the operation, using illustrations, of two network shortest path algorithms, providing an example of each.

1. Introducing the concept of network shortest path algorithms

A network shortest path algorithm is a computational procedure used to determine the shortest path between nodes in a graph. This graph can represent various structures, including roads in a city, network nodes in telecommunications, or connections in social networks. The primary goal is to find the least costly path based on weights assigned to the edges (e.g., distance, time, cost).

III. Analyse the operation, using illustrations, of two network shortest path algorithms, providing an example of each.

2. Algorithm 1: Dijkstra's Algorithm

a, Overview

Dijkstra's Algorithm finds the shortest path from a single source node to all other nodes in a weighted graph with non-negative edge weights. It does this by iteratively selecting the node with the smallest tentative distance, updating the distances of its neighbors, and marking it as visited.

b, Steps

1, Initialize the distance to the source node as 0 and all other nodes as infinity.

2, Create a priority queue and add the source node.

3, While the priority queue is not empty:

- Extract the node with the smallest distance (let's call this node u).
- For each neighbor v of u, calculate the distance through u. If this distance is less than the currently known distance, update it and add v to the priority queue.

4, Repeat until all nodes have been processed.

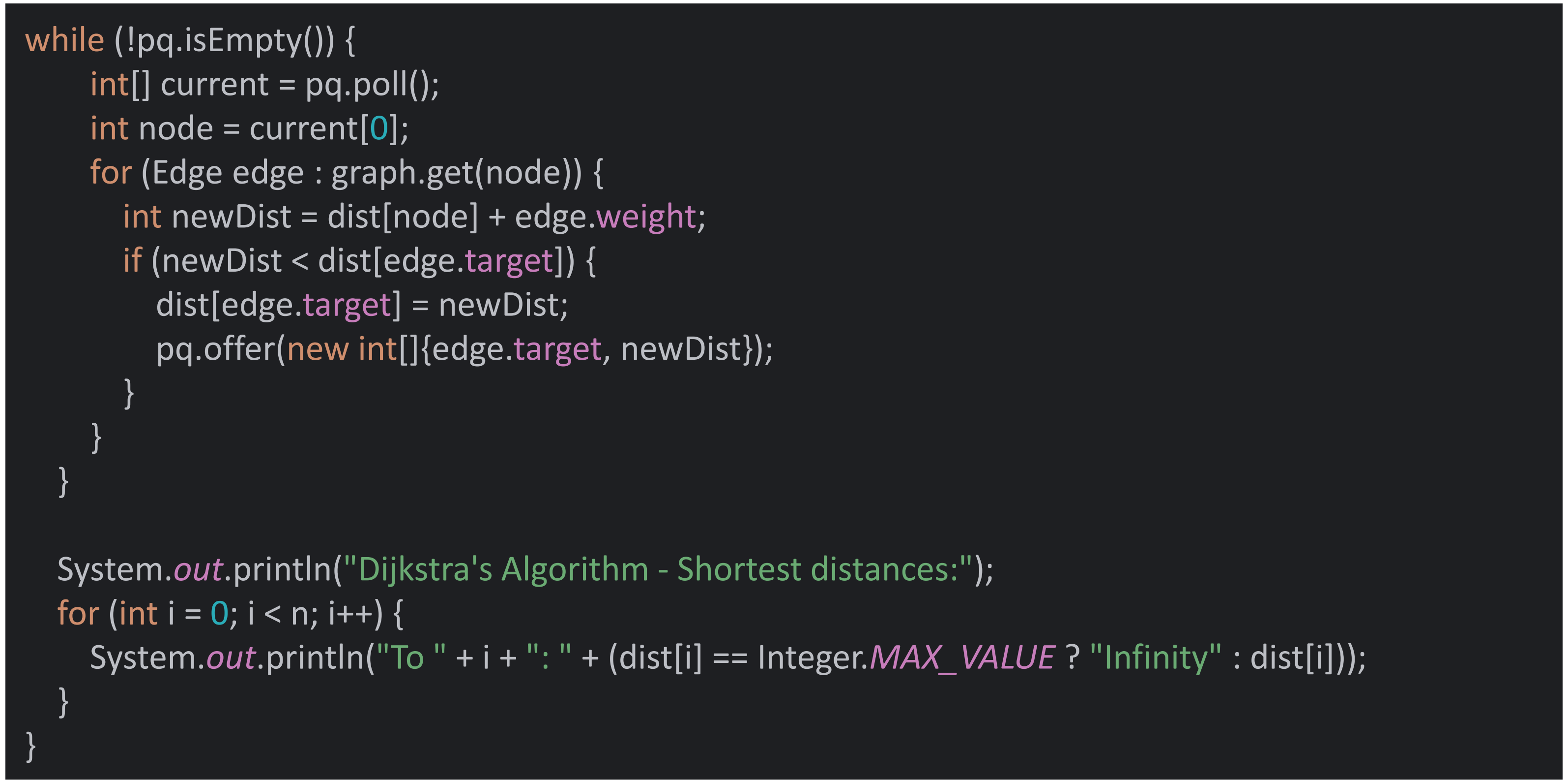
III. Analyse the operation, using illustrations, of two network shortest path algorithms, providing an example of each.

2. Algorithm 1: Dijkstra's Algorithm

```
public static void dijkstra(List<List<Edge>> graph, int source) {
    int n = graph.size();
    int[] dist = new int[n];
    Arrays.fill(dist, Integer.MAX_VALUE);
    dist[source] = 0;

    PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(a ->
a[1]));
    pq.offer(new int[]{source, 0});
```

III. Analyse the operation, using illustrations, of two network shortest path algorithms, providing an example of each.

2. Algorithm 1: Dijkstra's Algorithm



```
while (!pq.isEmpty()) {
    int[] current = pq.poll();
    int node = current[0];
    for (Edge edge : graph.get(node)) {
      int newDist = dist[node] + edge.weight;
      if (newDist < dist[edge.target]) {
        dist[edge.target] = newDist;
        pq.offer(new int[]{edge.target, newDist});
      }
    }
  }

  System.out.println("Dijkstra's Algorithm - Shortest distances:");
  for (int i = 0; i < n; i++) {
    System.out.println("To " + i + ": " + (dist[i] == Integer.MAX_VALUE ? "Infinity" : dist[i]));
  }
}
```

III. Analyse the operation, using illustrations, of two network shortest path algorithms, providing an example of each.

3. Algorithm 2: Prim-Jarnik Algorithm

a, Overview

The Prim-Jarnik Algorithm is used to find a Minimum Spanning Tree (MST) of a weighted, undirected graph. While it is not a shortest path algorithm in the traditional sense (finding a single path), it ensures that the total weight of the edges is minimized while connecting all vertices.

b, Steps

1, Start with a node and mark it as part of the MST.

2, Add the smallest edge that connects a vertex in the MST to a vertex outside it.

3, Repeat the process until all vertices are included in the MST.

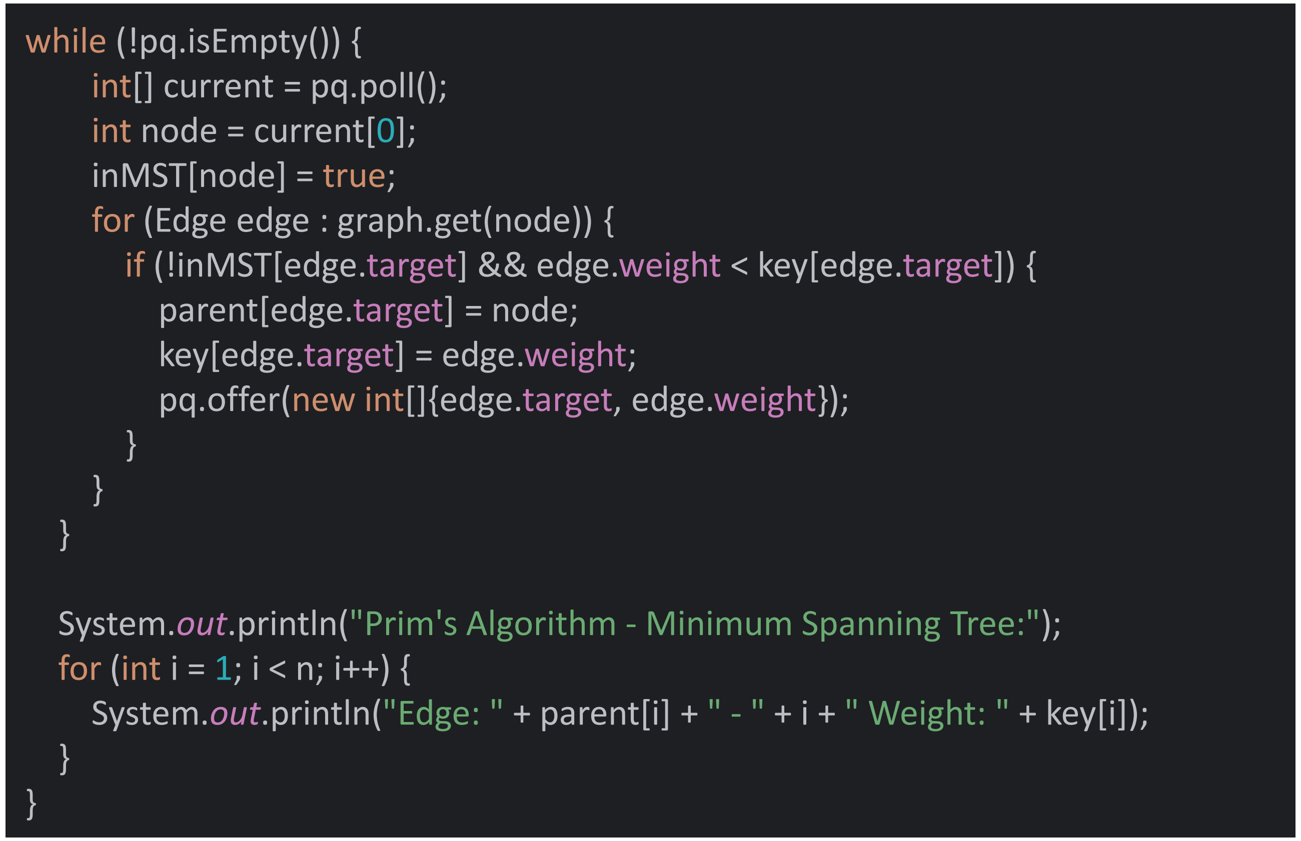
III. Analyse the operation, using illustrations, of two network shortest path algorithms, providing an example of each.

3. Algorithm 2: Prim-Jarnik Algorithm

```
// Prim's Algorithm for Minimum Spanning Tree
public static void prim(List<List<Edge>> graph) {
    int n = graph.size();
    boolean[] inMST = new boolean[n];
    int[] key = new int[n];
    int[] parent = new int[n];
    Arrays.fill(key, Integer.MAX_VALUE);
    key[0] = 0;

    PriorityQueue<int[]> pq = new
PriorityQueue<>(Comparator.comparingInt(a -> a[1]));
    pq.offer(new int[]{0, 0});
```

III. Analyse the operation, using illustrations, of two network shortest path algorithms, providing an example of each.

3. Algorithm 2: Prim-Jarnik Algorithm



```
while (!pq.isEmpty()) {
    int[] current = pq.poll();
    int node = current[0];
    inMST[node] = true;
    for (Edge edge : graph.get(node)) {
      if (!inMST[edge.target] && edge.weight < key[edge.target]) {
        parent[edge.target] = node;
        key[edge.target] = edge.weight;
        pq.offer(new int[]{edge.target, edge.weight});
      }
    }
  }

  System.out.println("Prim's Algorithm - Minimum Spanning Tree:");
  for (int i = 1; i < n; i++) {
    System.out.println("Edge: " + parent[i] + " - " + i + " Weight: " + key[i]);
  }
}
```

III. Analyse the operation, using illustrations, of two network shortest path algorithms, providing an example of each.

4. Performance Analysis

Time Complexity

Algorithm	Time Complexity
Dijkstra's Algorithm	O((V + E) log V) (using a priority queue)
Prim-Jarnik Algorithm	O(E log V) (using a priority queue)

V: Number of vertices
E: Number of edges

III. Analyse the operation, using illustrations, of two network shortest path algorithms, providing an example of each.

4. Performance Analysis

Space Complexity

Algorithm	Space Complexity
Dijkstra's Algorithm	O(V)
Prim-Jarnik Algorithm	O(V)

Conclusion

- Dijkstra's Algorithm is optimal for finding the shortest path from a single source to all other nodes in a weighted graph with non-negative weights. It is efficient and widely used in network routing protocols.
- Prim-Jarnik Algorithm, while not a shortest path algorithm, is essential for finding the Minimum Spanning Tree of a graph, minimizing the total weight of edges needed to connect all vertices.

THANK FOR LISTENING