Phạm Quang Huy

BH01568

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

1.1 What is DSA?

What is DSA

- **Data Structures** is about how data can be stored in different structures.
- **Algorithms** is about how to solve different problems, often by searching through and manipulating data structures.

1.1 What is DSA?

What is Data Structure

A data structure is a way to store data. We structure data in different ways depending on what data we have, and what we want to do with it.

There are 4 types of data structure: Linear, Non-Linear, Static, Dynamic.

1.1 What is DSA?

What is Data Structure

Linear Data Structure: Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure.
Example: Array, Stack, Queue, Linked List, etc.

1.1 What is DSA?

What is Data Structure

Static Data Structure: Static data structure has a fixed memory size. It is easier to access the elements in a static data structure.
Example: array.

1.1 What is DSA?

What is Data Structure

Dynamic Data Structure: In dynamic data structure, the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning the memory (space) complexity of the code.
Example: Queue, Stack, etc.

1.1 What is DSA?

What is Data Structure

1. **Non-Linear Data Structure:** Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. In a non-linear data structure, we can’t traverse all the elements in a single run only.
2. **Examples:** Trees and Graphs.

1.1 What is DSA?

What is Data Structure

In addition, in computer science, data structures can be divided into two types: **Primitive and Abstract.**

1.1 What is DSA?

Primitive data structure

are basic data structures provided by programming languages to represent single values, such as integers, floating-point numbers, characters, and booleans.

Abstact data structure

are higher-level data structures that are built using primitive data types and provide more complex and specialized operations. Some common examples of abstract data structures include arrays, linked lists, stacks, queues, trees, and graphs.

1.1 What is DSA?

What is Algoritm

An algorithm is a set of step-by-step instructions to solve a given problem or achieve a specific goal.

1.2 What is ADT

What is ADT

- Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of values and a set of operations.
- The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called “abstract” because it gives an implementation-independent view.

1.3 Compare different between Stack and Queue

Among the various data structures, stacks and queues are two of the most basic yet essential structures used in programming and algorithm design

Features	Stack	Queue
Definition	A linear data structure that follows the Last In First Out (LIFO) principle.	A linear data structure that follows the First In First Out (FIFO) principle.
Primary Operations	Push (add an item), Pop (remove an item), Peek (view the top item)	Enqueue (add an item), Dequeue (remove an item), Front (view the first item), Rear (view the last item)
Insertion / Removal	Elements are added and removed from the same end (the top).	Elements are added at the rear and removed from the front.
Complexity	Push: O(1), Pop: O(1), Peek: O(1)	Enqueue: O(1), Dequeue: O(1), Front: O(1), Rear: O(1)
Implementation	Can be implemented using arrays or linked lists.	Can be implemented using arrays, linked lists, or circular buffers.

1.4 How many ways are there to implement Stack and Queue?

Stack

- **Array-based Stack**: Uses a fixed-size array to store elements. Simple but can lead to overflow if capacity is reached.
- **Linked List-based Stack**: Uses nodes where each node points to the next, providing a dynamic size that adjusts as elements are added or removed.

Queue

- **Array-based Queue**: Uses a fixed-size array, but can have issues with space wastage if implemented without shifting elements (solvable by using circular queues).
- **Linked List-based Queue**: Dynamic structure where each node has a pointer to the next, avoiding fixed capacity issues.
- **Circular Queue**: An improvement on array-based queues where the end wraps around to the beginning, optimizing space and avoiding data shifts.

1.4.1.

Stack:

- Array-Based
- Linked List-Based

Array-Based Stack

Array-based Stack_ Example Code

```
public class ArrayStack {
    4 usages
    private int[] stack;  // Array to store
    9 usages
    private int top;      // Top pointer
    3 usages
    private int capacity; // Maximum capaci

    // Constructor to initialize stack
    1 usage
    public ArrayStack(int capacity) {
        this.capacity = capacity;
        stack = new int[capacity];  // Allo
        top = -1;  // Initialize top to -1
    }
```

- **Class: ArrayStack**
  - **Attributes:**
    - **stack:** An array that stores the stack's elements.
    - **top:** An integer pointer representing the index of the last added element.
    - **capacity:** An integer representing the maximum number of elements the stack can hold**.**
  - **Constructor:**
    - capacity to the value provided when creating the stack.
    - stack as an integer array of the specified capacity.
    - top to -1, which indicates that the stack is empty.

Array-based Stack_ Example Code

```
public void push(int value) {
    if (top == capacity - 1) {  // Stack overf
        System.out.println("Stack Overflow");
    } else {
        stack[++top] = value;  // Increment to
    }
}
```

- **Methods:**
  - **push(int value) Method:**
    - **Purpose:** Adds an element to the top of the stack.
    - **Steps:**
      - Checks if the stack is full by verifying if top equals capacity - 1.
      - If full, it outputs "Stack Overflow" since no more elements can be added.
      - Otherwise, it increments top by 1 and assigns value to stack[top], effectively adding the element to the top of the stack.

Array-based Stack_ Example Code

```
1 usage
public int pop() {
    if (top == -1) {  // Stack underflow condi
        System.out.println("Stack Underflow");
        return -1;
    } else {
        return stack[top--];  // Return top el
    }
}
```

- **Methods:**
  - **pop() Method:**
    - **Purpose:** Removes and returns the element at the top of the stack.
    - **Steps:**
      - Checks if the stack is empty by verifying if top is -1.
      - If empty, it outputs "Stack Underflow" and returns -1 (error code for empty stack).
      - Otherwise, it retrieves stack[top], then decrements top by 1 to remove the element from the stack.

Array-based Stack_ Example Code

```
// Peek operation
2 usages
public int peek() {
    if (top == -1) {
        System.out.println("Stack is Empty");
        return -1;
    } else {
        return stack[top];  // Return the top
    }
}
```

- **Methods:**
  - **peek() Method:**
    - **Purpose:** Returns the element at the top of the stack without removing it.
    - **Steps:**
      - Checks if the stack is empty by verifying if top is -1.
      - If empty, it outputs "Stack is Empty" and returns -1.
      - Otherwise, it returns the element at stack[top], representing the top of the stack.

Array-based Stack_ Example Code

```
// Check if the stack is empty
no usages
public boolean isEmpty() {
    return top == -1;
}

// Check if the stack is full
no usages
public boolean isFull() {
    return top == capacity - 1;
}
```

- **Methods:**
  - **isEmpty() Method:**
    - **Purpose:** Checks if the stack has no elements.
    - **Steps:**
      - Returns true if top is -1 (indicating an empty stack).
      - Returns false otherwise.
  - **isFull() Method:**
    - **Purpose**: Checks if the stack has reached its maximum capacity.
    - **Steps**:
      - Returns true if top is equal to capacity - 1.
      - Returns false otherwise.

Linked List-Based Stack

Linked List-based Stack_ Example Code

```
public class LinkedListStack {
    10 usages
    private Node top;  // Top nod

    // Constructor to initialize
    1 usage
    public LinkedListStack() {
        this.top = null;  // Stac
    }
```

- **Class: LinkedListStack**
  - **Attributes:**
    - **top:** A reference to the Node at the top of the stack. The top node represents the last added element in the stack.
  - **Constructor:**
    - top to null, indicating that the stack is empty initially.

Linked List-based Stack_ Example Code

```
2 usages
public void push(int value) {
    Node newNode = new Node(value);
    newNode.next = top;  // New nod
    top = newNode;  // Update top t
}
```

- **Methods:**
  - **push(int value) Method:**
    - **Purpose:** Adds a new element to the top of the stack.
    - **Steps:**
      - Creates a new Node with the given value.
      - Sets the next pointer of the new node to the current top, making the new node point to the previous top of the stack.
      - Updates top to the new node, effectively making this new node the top of the stack.

Linked List-based Stack_ Example Code

```
1 usage
public int pop() {
    if (top == null) {  // Stack underflow con
        System.out.println("Stack Underflow");
        return -1;
    } else {
        int value = top.value;  // Get the top
        top = top.next;  // Update top to the
        return value;
    }
}
```

- **Methods:**
  - **pop() Method**:
    - **Purpose**: Removes and returns the element at the top of the stack.
    - **Steps**:
      - Checks if the stack is empty by verifying if top is null.
      - If the stack is empty, it outputs "Stack Underflow" and returns -1 (error code for empty stack).
      - Otherwise, retrieves the value of the current top node, then updates top to the next node in the stack (top.next). This effectively removes the top node from the stack.
      - Returns the value of the removed node.

Linked List-based Stack_ Example Code

```
2 usages
public int peek() {
    if (top == null) {
        System.out.println("Stack is Empty")
        return -1;
    } else {
        return top.value;  // Return the top
    }
}
```

- **Methods:**
  - **peek() Method**:
    - **Purpose**: Returns the element at the top of the stack without removing it.
    - **Steps**:
      - Checks if the stack is empty by verifying if top is null.
      - If empty, it outputs "Stack is Empty" and returns -1.
      - Otherwise, it returns the value of the top node, which is the element at the top of the stack.

Linked List-based Stack_ Example Code

```
no usages
public boolean isEmpty() {
    return top == null;
}
```

- **Methods:**
  - **isEmpty() Method:**
    - **Purpose:** Checks if the stack has no elements.
    - **Steps:**
      - Returns true if top is null (indicating an empty stack).
      - Returns false otherwise.

Linked List-based Stack_ Example Code

```
4
    4 usages
5   class Node {
        3 usages
6       int value;
        3 usages
7       Node next;
8
9       // Constructor to initialize a node
        1 usage
0       public Node(int value) {
1           this.value = value;
2           this.next = null;
3       }
4   }
5
6
```

- **Class: Node Class:**
  - **value:** An integer that holds the data of the node.
  - **next:** A reference to the next node in the stack. This allows linking nodes in sequence.

1.4.2

Queue:

- Linked List-Based
- Array-Based
- Circular-Based

Array-Based Queue

Array-based Queue _ Example Code

```
no usages
public class ArrayQueue {
    4 usages
    private int[] arr;
    5 usages
    private int front;
    4 usages
    private int rear;
    4 usages
    private int capacity;
    6 usages
    private int size;

    no usages
    public ArrayQueue(int capacity) {
        this.capacity = capacity;
        arr = new int[capacity];
        front = 0;
        rear = -1;
        size = 0;
    }
```

- **Attributes:**
  - **arr:** An array that holds the queue elements.
  - **front:** The index of the front element (where the next element will be dequeued).
  - **rear:** The index of the last element added to the queue.
  - **capacity:** The maximum number of elements the queue can hold.
  - **size:** The current number of elements in the queue.
- **Constructor:** Initializes the queue's attributes, including creating an array of the specified capacity, setting the front and rear indexes, and initializing the size to zero.

Array-based Queue _ Example Code

```
no usages
public void enqueue(int element) {
    if (size == capacity) {
        throw new IllegalStateException("Queue is full");
    }
    rear = (rear + 1) % capacity;
    arr[rear] = element;
    size++;
}

no usages
public int dequeue() {
    if (isEmpty()) {
        throw new IllegalStateException("Queue is empty");
    }
    int element = arr[front];
    front = (front + 1) % capacity;
    size--;
    return element;
}
```

- **Methods:**
  - **Enqueue Method:** Adds an element to the rear of the queue. It checks if the queue is full and throws an exception if it is. The rear index is incremented in a circular manner (using modulo), and the new element is stored at that index.
  - **Dequeue Method:** Removes and returns the front element. It checks if the queue is empty and throws an exception if it is. The front index is also incremented in a circular manner.

Array-based Queue _ Example Code

```
    no usages
    public int peek() {
        if (isEmpty()) {
            throw new IllegalStateException("Queue is empty");
        }
        return arr[front];
    }

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

    no usages
    public int size() {
        return size;
    }
}
```

- **Methods:**
  - **Peek Method:** Returns the front element without removing it. It throws an exception if the queue is empty.
  - **isEmpty Method:** Checks if the queue is empty by comparing the size to zero.
  - **Size Method:** Returns the current size of the queue.

Linked List-Based Queue

Linked List-based Queue _ Example Code

```
5 usages
class Node {
    3 usages
    int data;
    3 usages
    Node next;

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

- **Node Class:**
  - **Attributes:**
    - **data:** The value stored in the node.
    - **next:** A reference to the next node in the linked list.
  - **Constructor:** Initializes the data and sets the next reference to null.

Linked List-based Queue _ Example Code

```
no usages
public class LinkedListQueue {
    7 usages
    private Node front;
    6 usages
    private Node rear;
    5 usages
    private int size;

    no usages
    public LinkedListQueue() {
        front = null;
        rear = null;
        size = 0;
    }
```

- **Class: LinkedListQueue:**
  - **Attributes:**
    - **front:** A reference to the first node in the queue (where elements are dequeued from).
    - **rear:** A reference to the last node in the queue (where elements are enqueued).
    - **size:** The current number of elements in the queue.
  - **Constructor:** Initializes the front and rear pointers to null and sets the size to 0.

Linked List-based Queue _ Example Code

```java
// Enqueue
no usages
public void enqueue(int element) {
    Node newNode = new Node(element);
    if (rear == null) { // If queue is empty
        front = rear = newNode;
    } else {
        rear.next = newNode; // Link new node at the end
        rear = newNode;
    }
    size++;
}

// Dequeue
no usages
public int dequeue() {
    if (isEmpty()) {
        throw new IllegalStateException("Queue is empty");
    }
    int element = front.data;
    front = front.next; // Move front to the next node
    if (front == null) { // If queue becomes empty
        rear = null;
    }
    size--;
    return element;
}
```

- **Methods:**
  - **Enqueue Method:**
    - Creates a new node with the given element.
    - If the queue is empty (i.e., both front and rear are null), both pointers are set to point to the new node.
    - If the queue is not empty, the new node is added at the end of the queue by linking the current rear node's next reference to the new node and updating the rear pointer.
  - **Dequeue Method:**
    - Removes the front node and returns its data. It checks if the queue is empty and throws an exception if it is.
    - The front pointer is moved to the next node in the queue. If the queue becomes empty (i.e., the front becomes null), the rear pointer is also set to null.
    - The size of the queue is decremented.

Linked List-based Queue _ Example Code

```
    // Peek
    no usages
    public int peek() {
        if (isEmpty()) {
            throw new IllegalStateException("Queue is empty");
        }
        return front.data;
    }

    // isEmpty
    2 usages
    public boolean isEmpty() {
        return size == 0;
    }

    // Size
    no usages
    public int size() {
        return size;
    }
}
```

- **Methods:**
  - **Peek Method:** Returns the data of the front node without removing it, throwing an exception if the queue is empty.
  - **isEmpty Method:** Checks if the queue is empty by comparing the size to 0.
  - **Size Method:** Returns the current size of the queue.

Circular-Based Queue

Circular-based Queue _ Example Code

```java
public class CircularQueue {
    5 usages
    private int[] queue;
    11 usages
    private int front, rear, size;

    1 usage
    public CircularQueue(int size) {
        this.size = size;
        queue = new int[size];
        front = -1;
        rear = -1;
    }
```

- **Class:** CircularQueue
  - **Attributes:**
    - **queue:** An array used to store elements of the queue.
    - **front:** An integer pointing to the index of the first element in the queue
    - **rear:** An integer pointing to the index of the last element in the queue
    - **size:** The maximum number of elements the queue can hold, or the queue's capacity.
  - **Constructor:**
    - Initializes the array queue with the given size.
    - Sets front and rear pointers to -1 to indicate that the queue is empty.

Circular-based Queue _ Example Code

```
public boolean isFull() {
    return (rear + 1) % size == front;
}

// Check if the queue is empty
4 usages
public boolean isEmpty() {
    return front == -1;
}
```

- **Methods:**
  - **isFull Method:**
    - Checks if the queue has reached its maximum capacity by verifying if the rear + 1 position wraps around to the front. If so, the queue is considered full.
  - **isEmpty Method:**
    - Checks if the queue is empty by seeing if front is -1. If true, the queue has no elements.

Circular-based Queue _ Example Code

```java
public void enqueue(int value) {
    if (isFull()) {
        System.out.println("Queue is full. Cannot enqueue " + value);
        return;
    }
    if (isEmpty()) {
        front = 0;
    }
    rear = (rear + 1) % size;
    queue[rear] = value;
    System.out.println("Enqueued " + value);
}
```

- **Methods:**
  - **enqueue Method:**
    - **Purpose:** Adds a new element to the end (rear) of the queue.
    - **Steps:**
      - If the queue is full (isFull() returns true), it outputs a message and exits.
      - If the queue is empty (isEmpty() returns true), it sets front to 0 to mark the start of the queue.
      - It moves the rear pointer to the next position, handling wrap-around with (rear + 1) % size.
      - Stores the new element at queue[rear].

Circular-based Queue _ Example Code

```
1 usage
public int dequeue() {
    if (isEmpty()) {
        System.out.println("Queue is empty. Cannot dequeue");
        return -1;
    }
    int value = queue[front];
    if (front == rear) { // Queue becomes empty
        front = -1;
        rear = -1;
    } else {
        front = (front + 1) % size;
    }
    System.out.println("Dequeued " + value);
    return value;
}
```

- **Methods:**
  - **dequeue Method:**
    - **Purpose**: Removes and returns the element at the front of the queue.
    - **Steps**:
      - If the queue is empty (isEmpty() returns true), it outputs a message and returns -1 as an error code.
      - Retrieves the value at queue[front] to return later.
      - Moves the front pointer to the next position using (front + 1) % size.
      - If removing the last element (where front == rear), it resets both front and rear to -1, marking the queue as empty.

Circular-based Queue _ Example Code

```
1 usage
public int peek() {
    if (isEmpty()) {
        System.out.println("Queue is empty. Nothing to peek");
        return -1;
    }
    return queue[front];
}

// Display the queue elements
3 usages
public void display() {
    if (isEmpty()) {
        System.out.println("Queue is empty.");
        return;
    }
    System.out.print("Queue: ");
    int i = front;
    while (true) {
        System.out.print(queue[i] + " ");
        if (i == rear) break;
        i = (i + 1) % size;
    }
    System.out.println();
}
```

- **Methods:**
  - **peek Method**:
    - **Purpose**: Returns the element at the front without removing it.
    - **Steps**:
      - Checks if the queue is empty; if so, it returns -1.
      - Returns the element at queue[front] if the queue is not empty.
  - **display Method**:
    - **Purpose**: Shows all elements in the queue in order.
    - **Steps**:
      - If the queue is empty, it displays a message.
      - Starts from front and prints each element until it reaches rear, using (i + 1) % size for circular traversal.

2 Compare the performance of two sorting algorithms.

2.1 Introducing the two sorting algorithms you will be comparing

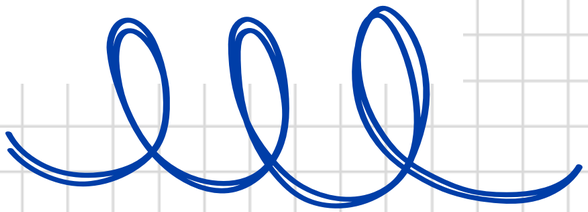
Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order.
This algorithm is not suitable for large data sets as its average and worst-case time complexity are quite high.

Merge Sort

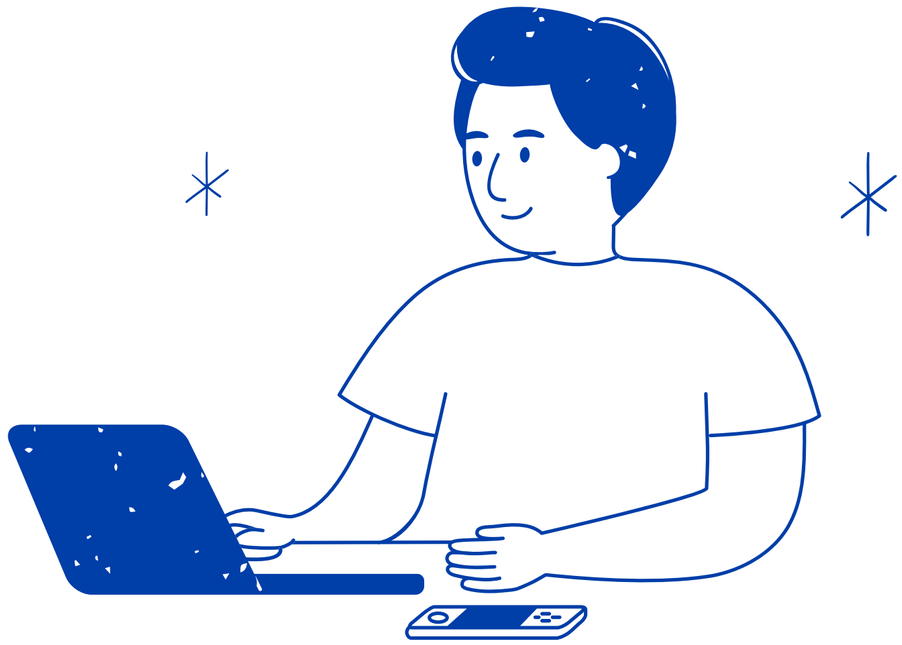
Merge sort is a sorting algorithm that follows the divide-and-conquer approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.

2.2 Time and Space Complexity Analysis



2.2.1

Bubble Sort



2.2.1. Bubble Sort

Time Complexity

Best Case Time Complexity: O(N)

- The best case occurs when the array is already sorted. Here, the number of comparisons is N−1N-1N−1, and no swaps are needed, resulting in a best case complexity of O(N).

2.2.1. Bubble Sort

Time Complexity

Worst Case Time Complexity: $O(N^2)$

The worst case occurs when the array is sorted in decreasing order. The number of iterations needed is N–1N-1N–1.

- **Pass 1:**
  - **Comparisons =** N–1N-1N–1
  - **Swaps =** N–1N-1N–1
- **Pass 2:**
  - **Comparisons =** N–2N-2N–2
  - **Swaps =** N–2N-2N–2

...

- **Pass N–1N-1N–1:**
  - **Comparisons =** 1
  - **Swaps =** 1

2.2.1. Bubble Sort

Time Complexity

- ***Now, calculating total number of comparison required to sort the array***
  - *= (N-1) + (N-2) + (N-3) + . . . 2 + 1*
  - *= (N-1)*(N-1+1)/2* { by using sum of N natural Number formula }
  - *= (N * (N-1)) / 2*
- **In worst case, Total number of swaps = Total number of comparison**
  - **Total number of comparison (Worst case) =** N(N-1)/2
  - **Total number of swaps (Worst case) =** N(N-1)/2

2.2.1. Bubble Sort

Time Complexity

Average Case Time Complexity Analysis of Bubble Sort: O(N^2)

- ***For the number of swaps, consider the following points**:*
  - *If an element is in index I1 but it should be in index I2, then it will take a minimum of (I2-I1) swaps to bring the element to the correct position.*
  - *Consider an element E is at a distance of I3 from its position in sorted array. Then the maximum value of I3 will be (N-1) for the edge elements and N/2 for the elements at the middle.*
- ***The sum of maximum difference in position across all elements will be**:*
  - *(N – 1) + (N – 3) + (N – 5) . . . + 0 + . . . + (N-3) + (N-1)*
  - *= N x (N – 2) x (1 + 3 + 5 + ... + N/2)*
  - *= N^2 – (2 x N^2 / 4)*
  - *= N^2 – N^2 / 2*
  - *= N^2 / 2*
- ***Therefore, in average case the number of comparisons is O(N^2)***

2.2.1. Bubble Sort

Space Complexity

- **The space complexity of Bubble Sort is O(1),** meaning it requires a constant amount of extra space regardless of the input size. It only needs space for temporary variables or indices, making it very efficient as it does not depend on the array size.

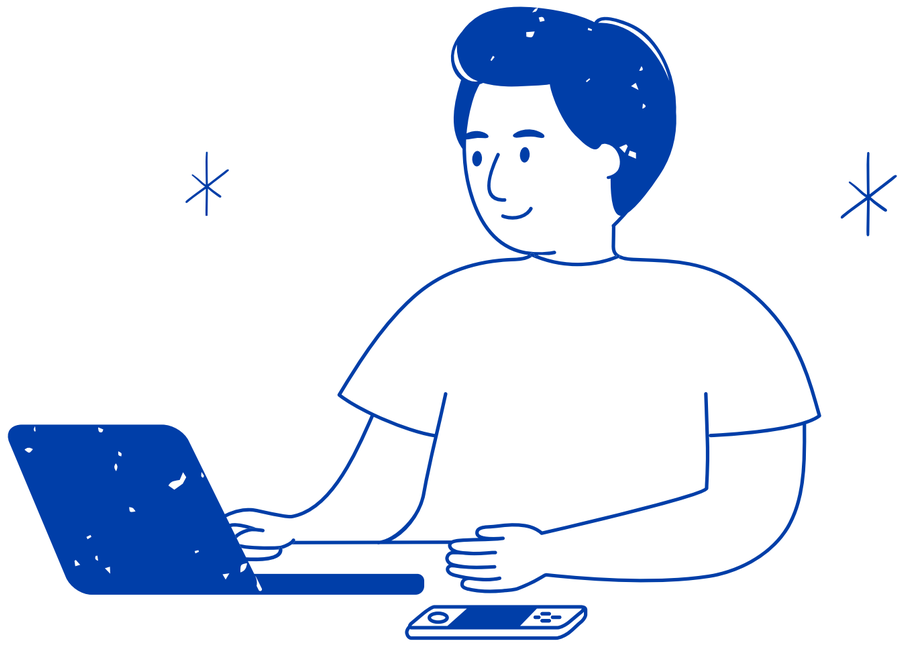
2.2.1. Bubble Sort

Stability

- Bubble Sort maintains the relative order of equal elements because adjacent swaps do not change their relative positions.

2.2.2

Merge Sort



2.2.2. Merge Sort

Time Complexity

Best Case Time Complexity: O(N log N)

- In the best case, Merge Sort still divides the array into halves and merges them. The number of comparisons and merges is proportional to NNN at each level of recursion, with the depth of the recursion being logNlog NlogN. Therefore, the best case complexity is O(N log N).

2.2.2. Merge Sort

Time Complexity

Worst Case Time Complexity: O(N log N)

- The worst case for Merge Sort occurs with any arrangement of elements, as the algorithm consistently divides and merges the array. The process involves NNN comparisons and merges at each of the logNlog NlogN levels of recursion. Thus, the worst case complexity is also O(N log N).

2.2.2. Merge Sort

Time Complexity

Average Case Time Complexity: O(N log N)

- Similar to the worst case, the average case involves dividing the array and merging it back together. Regardless of the initial arrangement of elements, Merge Sort performs NNN operations at each recursive level, leading to an average case complexity of O(N log N).

2.2.2. Merge Sort

Space Complexity

- **The space complexity of Merge Sort is O(N).** This is because the algorithm requires additional space to store the temporary arrays used during the merging process. Specifically, it creates copies of the subarrays being merged, which necessitates space proportional to the size of the input array.

2.2.2. Merge Sort

Stability

- Merge Sort preserves the relative order of equal elements during the merge process. If two elements are equal, the element from the left half is placed before the one from the right half, maintaining their original order.

2.3. Comparison between two sorting

Characteristic	Bubble Sort	Merge Sort
Time Complexity (Best)	O(n)	O(n log n)
Time Complexity (Worst)	$O(n^2)$	O(n log n)
Time Complexity (Average)	$O(n^2)$	O(n log n)
Space Complexity	O(1)	O(n)
Stability	Stable	Stable
In-Place Sorting	Yes	No
Practical Use	Small datasets	Large datasets, stable sorts

2.4 differences in performance between the two algorithms

2.4.1 Bubble Sort

```
1 usage
public static void bubbleSort(int[] arr) {
    int n = arr.length;
    boolean swapped;
    for (int i = 0; i < n - 1; i++) {
        swapped = false;
        // Traverse the array from 0 to n-i-1
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap arr[j] and arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = true;
            }
        }
        // If no two elements were swapped in the inner loop, then the array is
        if (!swapped) break;
    }
}

public static void main(String[] args) {
    int[] arr = {5, 2, 9, 1, 5, 6};
    System.out.println("Original Array: " + Arrays.toString(arr));
    bubbleSort(arr);
    System.out.println("Sorted using Bubble Sort: " + Arrays.toString(arr));
}
```

- **Step 1: Initialization:**
  - **n = arr.length:** This gets the size of the input array.
  - **swapped:** This is a flag used to check if a swap was made during the current pass through the array.
- **Step 2: Outer loop:**
  - The outer loop runs n-1 times. Each iteration of the outer loop makes one full pass over the unsorted part of the array. At the end of each iteration, the largest element "bubbles up" to the correct position.

2.4.1 Bubble Sort

```
1 usage
public static void bubbleSort(int[] arr) {
    int n = arr.length;
    boolean swapped;
    for (int i = 0; i < n - 1; i++) {
        swapped = false;
        // Traverse the array from 0 to n-i-1
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap arr[j] and arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = true;
            }
        }
        // If no two elements were swapped in the inner loop, then the array is
        if (!swapped) break;
    }
}

public static void main(String[] args) {
    int[] arr = {5, 2, 9, 1, 5, 6};
    System.out.println("Original Array: " + Arrays.toString(arr));
    bubbleSort(arr);
    System.out.println("Sorted using Bubble Sort: " + Arrays.toString(arr));
}
```

- **Step 3: Inner loop:**
  - The inner loop compares each adjacent pair of elements (arr[j] and arr[j + 1]). If the current element is greater than the next, they are swapped. This ensures that the largest unsorted element moves towards the end of the array.
- **Step 4: Early exit:**
  - If no swaps occur during a pass (i.e., the array is already sorted), the algorithm breaks out of the loop early. This makes Bubble Sort slightly faster when the array is nearly sorted.

2.4.1 Bubble Sort

How Bubble Sort Work

- **Pass 1:** [2, 5, 1, 5, 6, 9] → largest element 9 "bubbles up" to the last position.
- **Pass 2:** [2, 1, 5, 5, 6, 9] → next largest element 6 "bubbles up".
- **Pass 3:** [1, 2, 5, 5, 6, 9].
- **Result:** After multiple passes, the array is fully sorted. This took several comparisons and swaps, with a time complexity of $O(n^2)$.

2.4.2 Merge Sort

```
3 usages
public static void mergeSort(int[] arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        // Recursively sort first and second halves
        mergeSort(arr, left, mid);
        mergeSort(arr, left: mid + 1, right);

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

public static void main(String[] args) {
    int[] arr = {5, 2, 9, 1, 5, 6};
    System.out.println("Original Array: " + Arrays.toString(arr));
    mergeSort(arr, left: 0, right: arr.length - 1);
    System.out.println("Sorted using Merge Sort: " + Arrays.toString(arr));
}
```

- **mergeSort (Recursive Function):**
  - **Step 1: Base Case:**
    - if (left < right): The recursion continues as long as the left index is less than the right index. When left == right, the array has only one element and is already sorted.
  - **Step 2: Find the Middle:**
    - int mid = left + (right - left) / 2: The array is divided into two halves using the middle index.

2.4.2 Merge Sort

```
3 usages
public static void mergeSort(int[] arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        // Recursively sort first and second halves
        mergeSort(arr, left, mid);
        mergeSort(arr,  left: mid + 1, right);

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

public static void main(String[] args) {
    int[] arr = {5, 2, 9, 1, 5, 6};
    System.out.println("Original Array: " + Arrays.toString(arr));
    mergeSort(arr,  left: 0,  right: arr.length - 1);
    System.out.println("Sorted using Merge Sort: " + Arrays.toString(arr));
}
```

- **mergeSort (Recursive Function):**
  - **Step 3: Recursive Sorting:**
    - mergeSort(arr, left, mid): Sort the left half of the array recursively.
    - mergeSort(arr, mid + 1, right): Sort the right half of the array recursively.
  - **Step 4: Merge:**
    - After sorting both halves, the merge function is called to merge the two sorted halves back together.

2.4.2 Merge Sort

```java
public class MergeSort {
    // Function to merge two subarrays
    1 usage
    public static void merge(int[] arr, int left, int mid, int right) {
        int n1 = mid - left + 1;
        int n2 = right - mid;

        // Create temporary arrays
        int[] leftArray = new int[n1];
        int[] rightArray = new int[n2];

        // Copy data into the temporary arrays
        for (int i = 0; i < n1; i++) leftArray[i] = arr[left + i];
        for (int i = 0; i < n2; i++) rightArray[i] = arr[mid + 1 + i];

        // Merge the two arrays
```

- **merge (Merging Function):**
  - **Step 1: Create Temporary Arrays:**
    - leftArray and rightArray are temporary arrays used to hold the elements of the left and right halves of the array being merged.
  - **Step 2: Copy Data:**
    - The data from the original array is copied into leftArray and rightArray.

2.4.2 Merge Sort

```
// Merge the two arrays
int i = 0, j = 0, k = left;
while (i < n1 && j < n2) {
    if (leftArray[i] <= rightArray[j]) {
        arr[k] = leftArray[i];
        i++;
    } else {
        arr[k] = rightArray[j];
        j++;
    }
    k++;
}

// Copy the remaining elements of leftArray, if any
while (i < n1) {
    arr[k] = leftArray[i];
    i++;
    k++;
}

// Copy the remaining elements of rightArray, if any
while (j < n2) {
    arr[k] = rightArray[j];
    j++;
    k++;
}
```

- **merge (Merging Function):**
  - **Step 3: Merge:**
    - Two pointers i and j are used to traverse the leftArray and rightArray. The elements from these arrays are compared and the smaller element is placed back into the original array.
    - This process continues until one of the arrays is exhausted.
  - **Step 4: Copy Remaining Elements:**
    - If any elements remain in leftArray or rightArray, they are copied into the original array.

2.4.2 Merge Sort

How Bubble Sort Work

- **Step 1:** Split the array into two halves: [5, 2, 9] and [1, 5, 6].
- **Step 2:** Recursively sort each half:
  - **Left half:** [2, 5, 9].
  - **Right half:** [1, 5, 6].
- **Step 3:** Merge the sorted halves into one sorted array: [1, 2, 5, 5, 6, 9].
- **Result:** The array is sorted after O(n log n) operations, which is more efficient for large datasets.

3 Dijkstra’s Algorithm

3.1. What is Dijkstra’s Algorithm

3.1 What is Dijkstra’s algorithm

Definition

Dijkstra’s algorithm is a popular algorithms for solving many single-source shortest path problems having non-negative edge weight in the graphs i.e., it is to find the shortest distance between two vertices on a graph. It was conceived by Dutch computer scientist Edsger W. Dijkstra in 1956.

3.1 What is Dijkstra’s algorithm

How it works

The algorithm maintains a set of visited vertices and a set of unvisited vertices. It starts at the source vertex and iteratively selects the unvisited vertex with the smallest tentative distance from the source. It then visits the neighbors of this vertex and updates their tentative distances if a shorter path is found. This process continues until the destination vertex is reached, or all reachable vertices have been visited.

3.2. Algorithm for Dijkstra’s algorithm

3.2. Algorithm for Dijkstra’s algorithm

- Mark the source node with a current distance of 0 and the rest with infinity.
- Set the non-visited node with the smallest current distance as the current node.
- For each neighbor, N of the current node adds the current distance of the adjacent node with the weight of the edge connecting 0->1. If it is smaller than the current distance of Node, set it as the new current distance of N.
- Mark the current node 1 as visited.
- Go to step 2 if there are any nodes are unvisited.

3.2. Algorithm for Diskstra’s algorithm

```java
package DijkstraAlgorithm;
import java.util.*;

2 usages
class Graph {
    4 usages
    private int V;
    5 usages
    private List<List<iPair>> adj;

    1 usage
    Graph(int V) {
        this.V = V;
        adj = new ArrayList<>();
        for (int i = 0; i < V; i++) {
            adj.add(new ArrayList<>());
        }
    }
```

- **Class: Graph.**
  - **Attributes:**
    - **V:** An integer representing the number of vertices (nodes) in the graph.
    - **adj:** A list of lists (List<List<iPair>>) where each element at index i contains a list of edges connected to vertex i. Each edge is represented by an iPair object containing the destination vertex and the edge weight.
  - **Constructor:**
    - **V** to the given number of vertices.
    - **adj** as an array list of array lists. Each sublist represents the edges from a particular vertex and is initially empty.

3.2. Algorithm for Diskstra’s algorithm

```
12 usages
void addEdge(int u, int v, int w) {
    adj.get(u).add(new iPair(v, w));
    adj.get(v).add(new iPair(u, w));
}
```

- **Methods:**
  - **addEdge(int u, int v, int w) Method:**
    - **Purpose:** Adds an undirected edge between vertices u and v with a weight of w.
    - **Steps:**
      - Creates an iPair for each endpoint with the other endpoint and the weight w.
      - Adds each iPair to the adjacency list for both vertices, effectively adding an undirected edge.

3.2. Algorithm for Diskstra’s algorithm

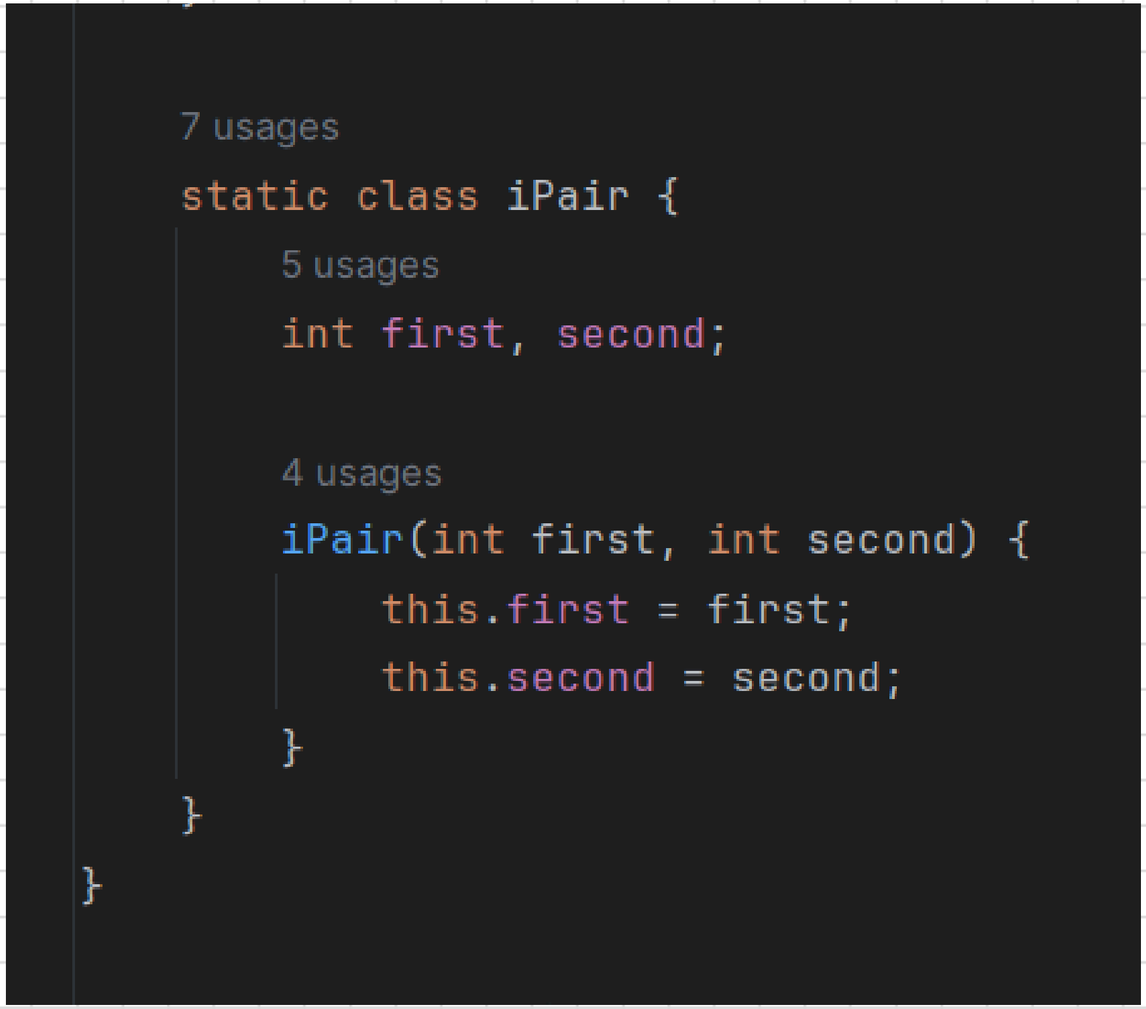
```
1 usage
void shortestPath(int src) {
    PriorityQueue<iPair> pq =
            new PriorityQueue<>(V, Comparator.comparingInt(o -> o.second));
    int[] dist = new int[V];
    Arrays.fill(dist, Integer.MAX_VALUE);

    pq.add(new iPair(0, src));
    dist[src] = 0;

    while (!pq.isEmpty()) {
        int u = pq.poll().second;

        for (iPair v : adj.get(u)) {
            if (dist[v.first] > dist[u] + v.second) {
                dist[v.first] = dist[u] + v.second;
                pq.add(new iPair(dist[v.first], v.first));
            }
        }
    }

    System.out.println("Vertex Distance from Source");
    for (int i = 0; i < V; i++) {
        System.out.println(i + "\t\t" + dist[i]);
    }
}
```

- **Methods:**
  - **shortestPath(int src) Method:**
    - **Purpose:** Uses Dijkstra’s algorithm to compute the shortest path from the source vertex src to all other vertices in the graph.

3.2. Algorithm for Diskstra’s algorithm

- **Methods:**
  - **shortestPath(int src) Method:**
    - Steps:
      - Initializes a priority queue pq that orders elements based on their distance value (second field of iPair).
      - Initializes an integer array dist of size V, setting all distances to Integer.MAX_VALUE to represent initially unreachable nodes.
      - Sets dist[src] to 0 for the source node and adds an iPair with distance 0 and vertex src to the priority queue.
      - Enters a while loop that runs as long as the priority queue is not empty:
        - Removes the node u with the smallest distance from the queue.
        - For each neighboring vertex v of u, checks if going through u gives a shorter path to v than the current recorded distance.
        - If so, updates dist[v] to the new shorter distance and adds v with the updated distance to the priority queue.
      - After the loop, prints each vertex's distance from the source.

3.2. Algorithm for Diskstra’s algorithm



```
    7 usages
    static class iPair {
        5 usages
        int first, second;

        4 usages
        iPair(int first, int second) {
            this.first = first;
            this.second = second;
        }
    }
}
```

- **Class: iPair:** A nested class representing a pair of integers, where:
  - **first:** Represents the vertex number.
  - **second:** Represents the weight or distance to this vertex.

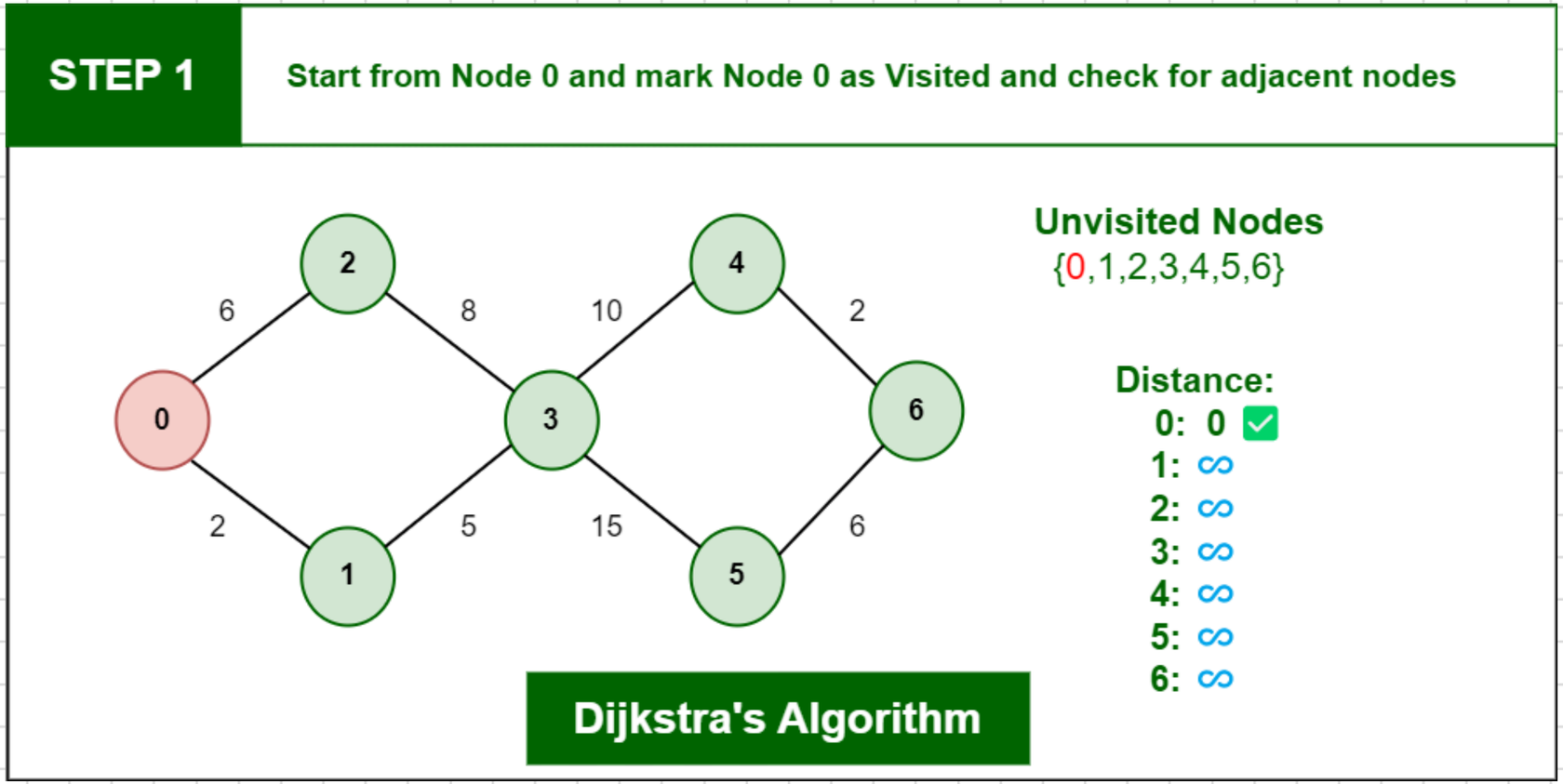
3.2. Algorithm for Diskstra’s algorithm

```
public class Main {
    public static void main(String[] args) {
        int V =7;
        Graph g = new Graph(V);

        g.addEdge( u: 0,  v: 1,  w: 2);
        g.addEdge( u: 0,  v: 2,  w: 6);
        g.addEdge( u: 1,  v: 3,  w: 5);
        g.addEdge( u: 2,  v: 3,  w: 8);
        g.addEdge( u: 3,  v: 4,  w: 10);
        g.addEdge( u: 3,  v: 5,  w: 15);
        g.addEdge( u: 4,  v: 6,  w: 2);
        g.addEdge( u: 5,  v: 6,  w: 6);



        g.shortestPath( src: 0);
    }
}
```

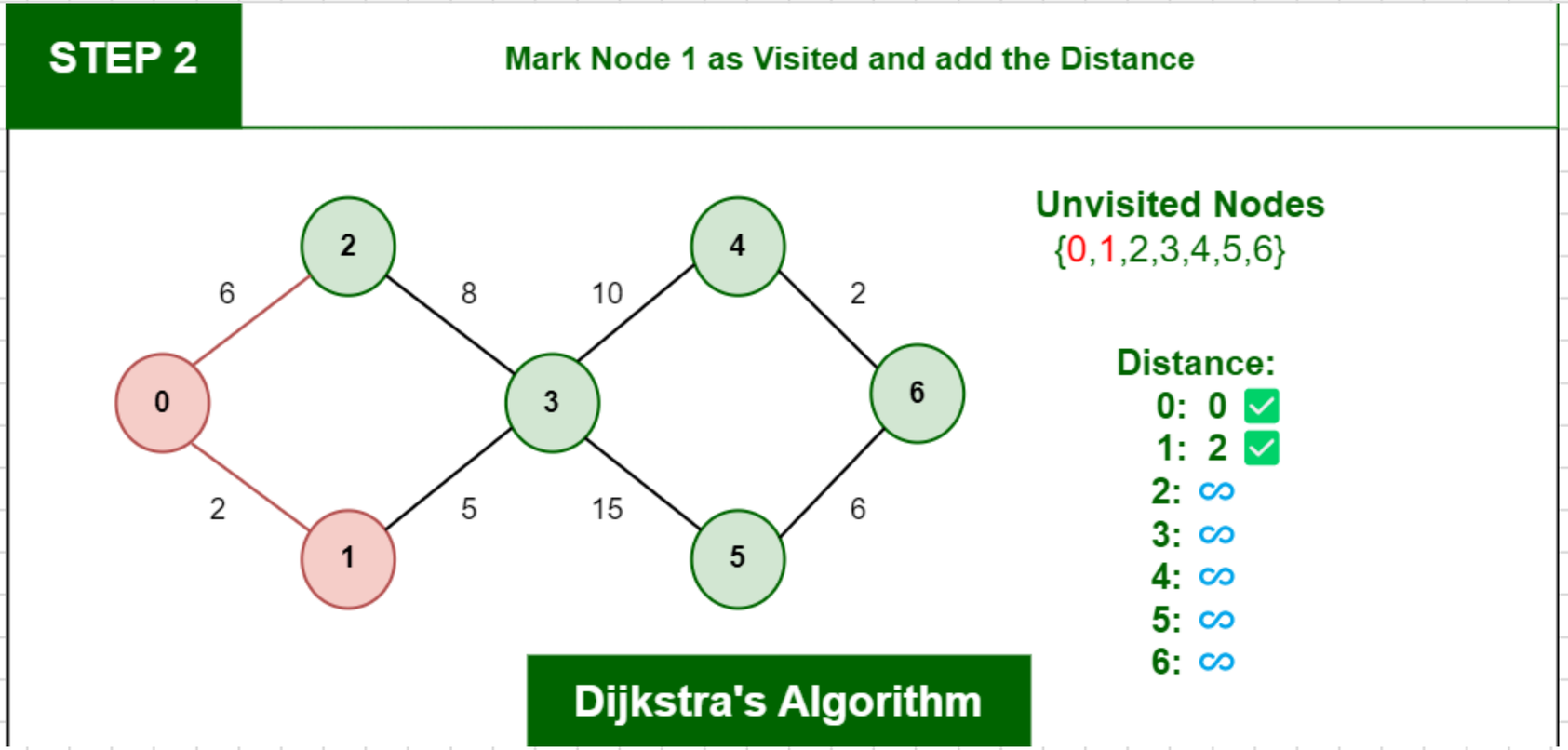
- **Main Class**:
  - Creates a Graph object g with 7 vertices.
  - Adds edges between pairs of vertices with their respective weights using addEdge.
  - Calls shortestPath(0) to compute and print the shortest distances from vertex 0 to all other vertices.

3.3. How does Dijkstra’s Algorithm works

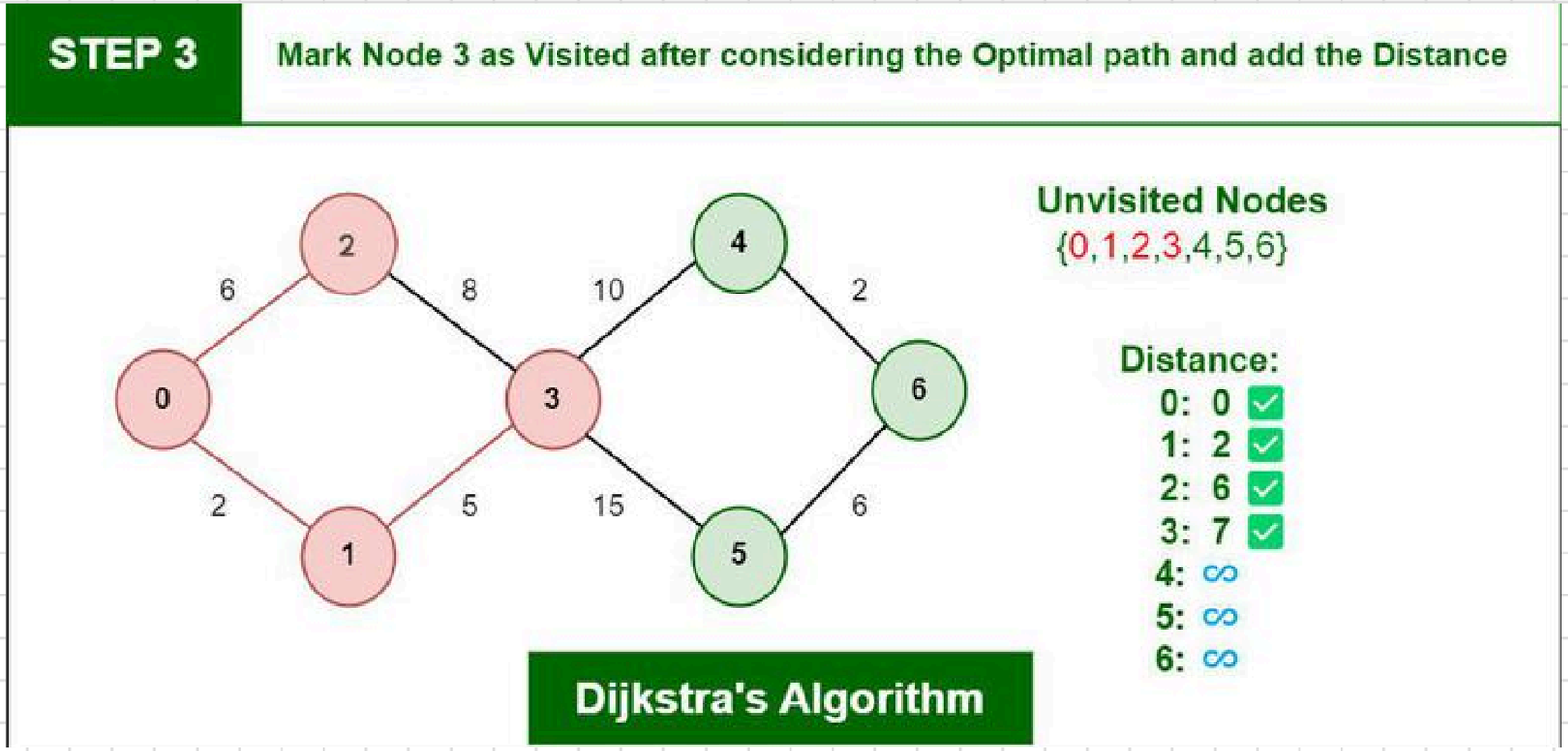
Step 1: Start from Node 0 and mark Node as visited as you can check in below image visited Node is marked red.



Step 2: Check for adjacent Nodes, Now we have to choices (Either choose Node1 with distance 2 or either choose Node 2 with distance 6) and choose Node with minimum distance. In this step Node 1 is Minimum distance adjacent Node, so marked it as visited and add up the distance.

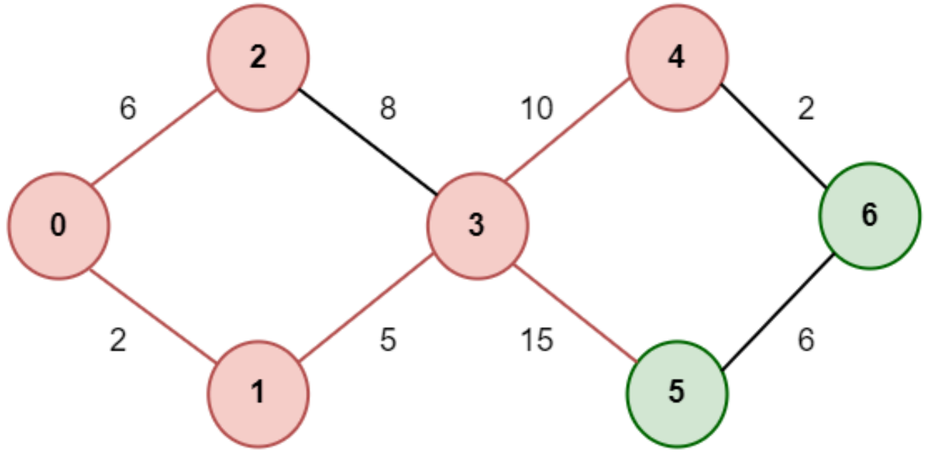


Step 3: Then Move Forward and check for adjacent Node which is Node 3, so marked it as visited and add up the distance, Now the distance will be:



Step 4: Again we have two choices for adjacent Nodes (Either we can choose Node 4 with distance 10 or either we can choose Node 5 with distance 15) so choose Node with minimum distance. In this step Node 4 is Minimum distance adjacent Node, so marked it as visited and add up the distance.

STEP 4 Mark Node 4 as Visited after considering the Optimal path and add the Distance

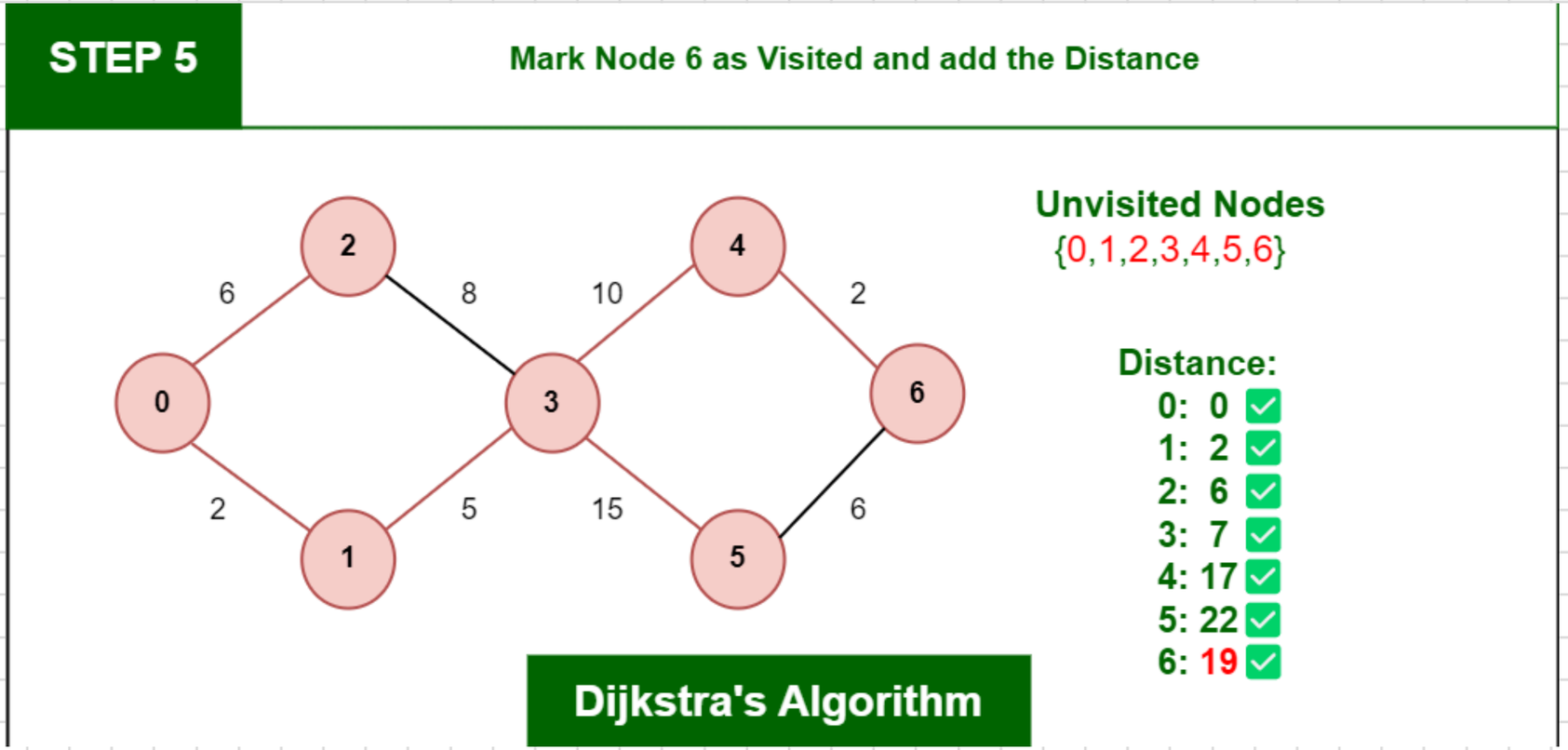


Unvisited Nodes
{0,1,2,3,4,5,6}

Distance:
0: 0 ✅
1: 2 ✅
2: 6 ✅
3: 7 ✅
4: 17 ✅
5: ∞
6: ∞

Dijkstra's Algorithm

Step 5: Again, Move Forward and check for adjacent Node which is Node 6, so marked it as visited and add up the distance, Now the distance will be:



End