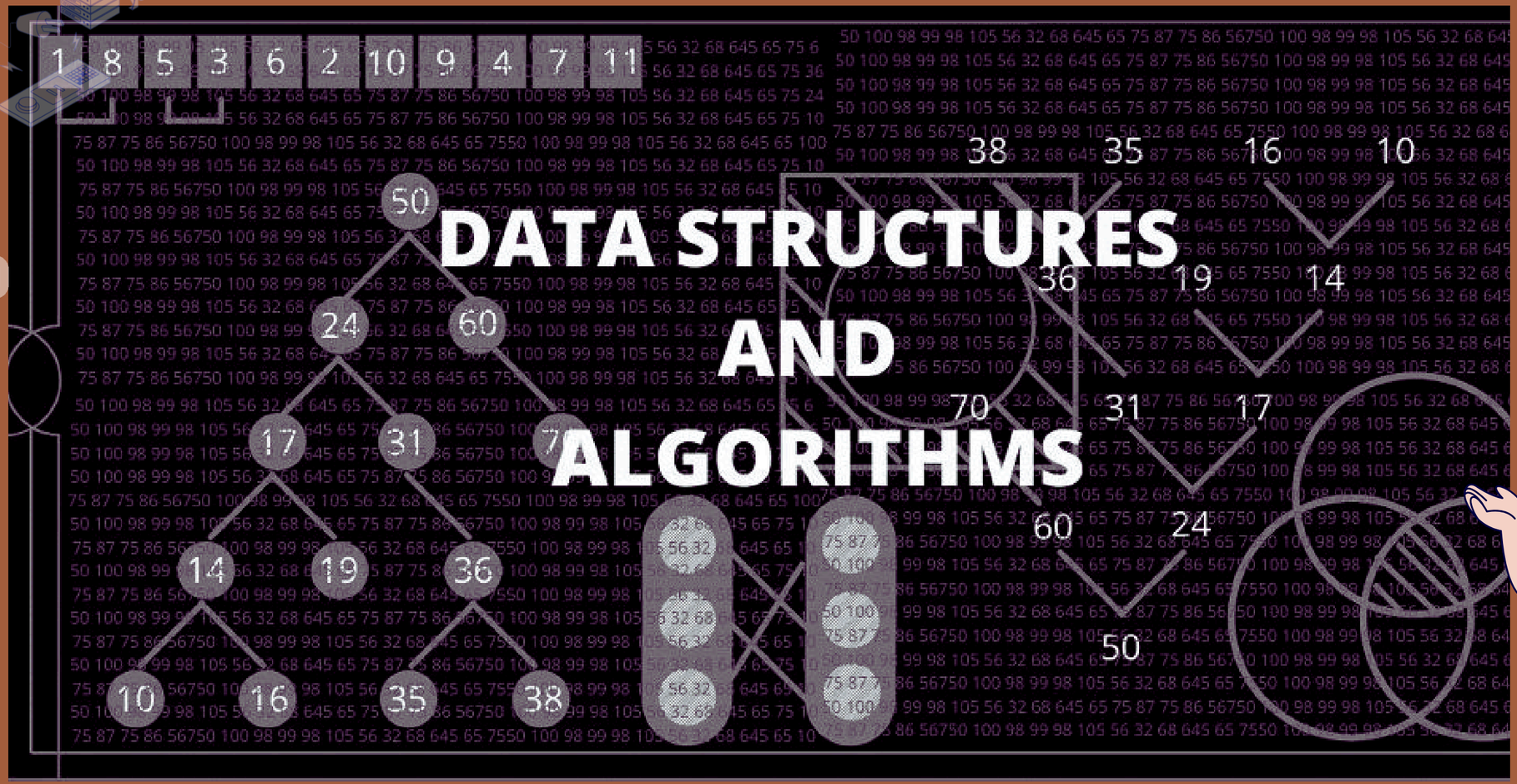
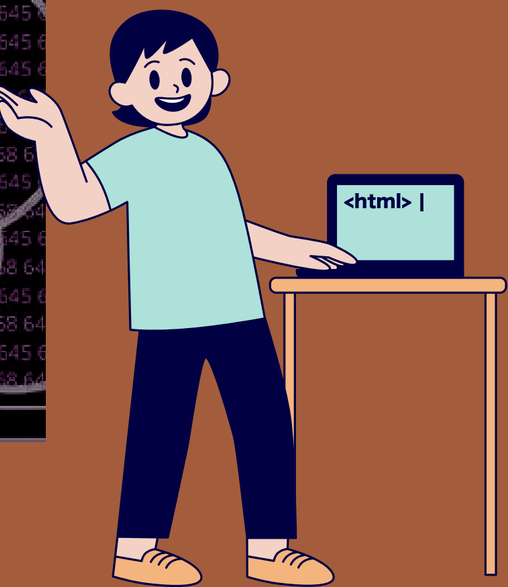
Examine abstract data types, concrete data structures and algorithms

(Chu Viet Chien - BH00915.)





The content in this process is:

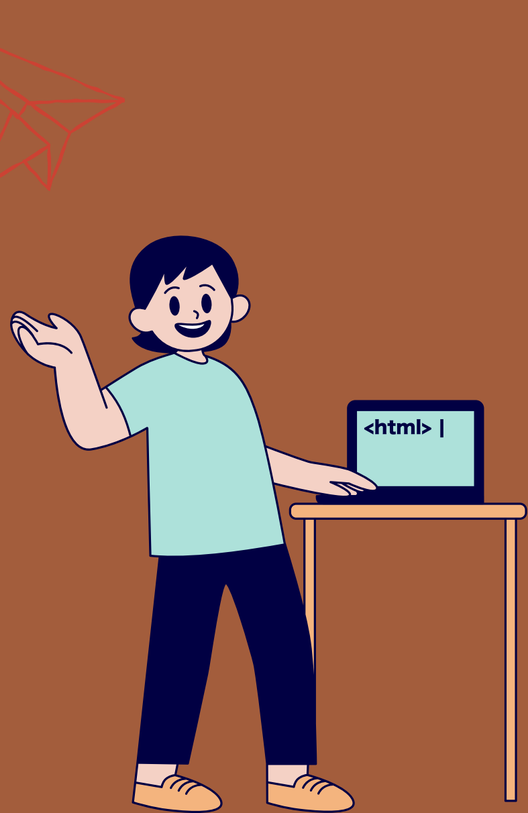
1, Data Structures and Complexity.

2, Memory Stack.

3, Queue.

4, Compare the performance of two sorting algorithms.

5, Network shortest path algorithms.



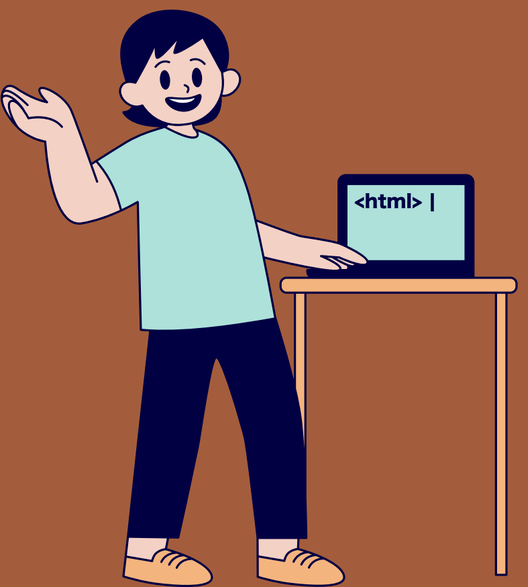
1, Data Structures and Complexity..

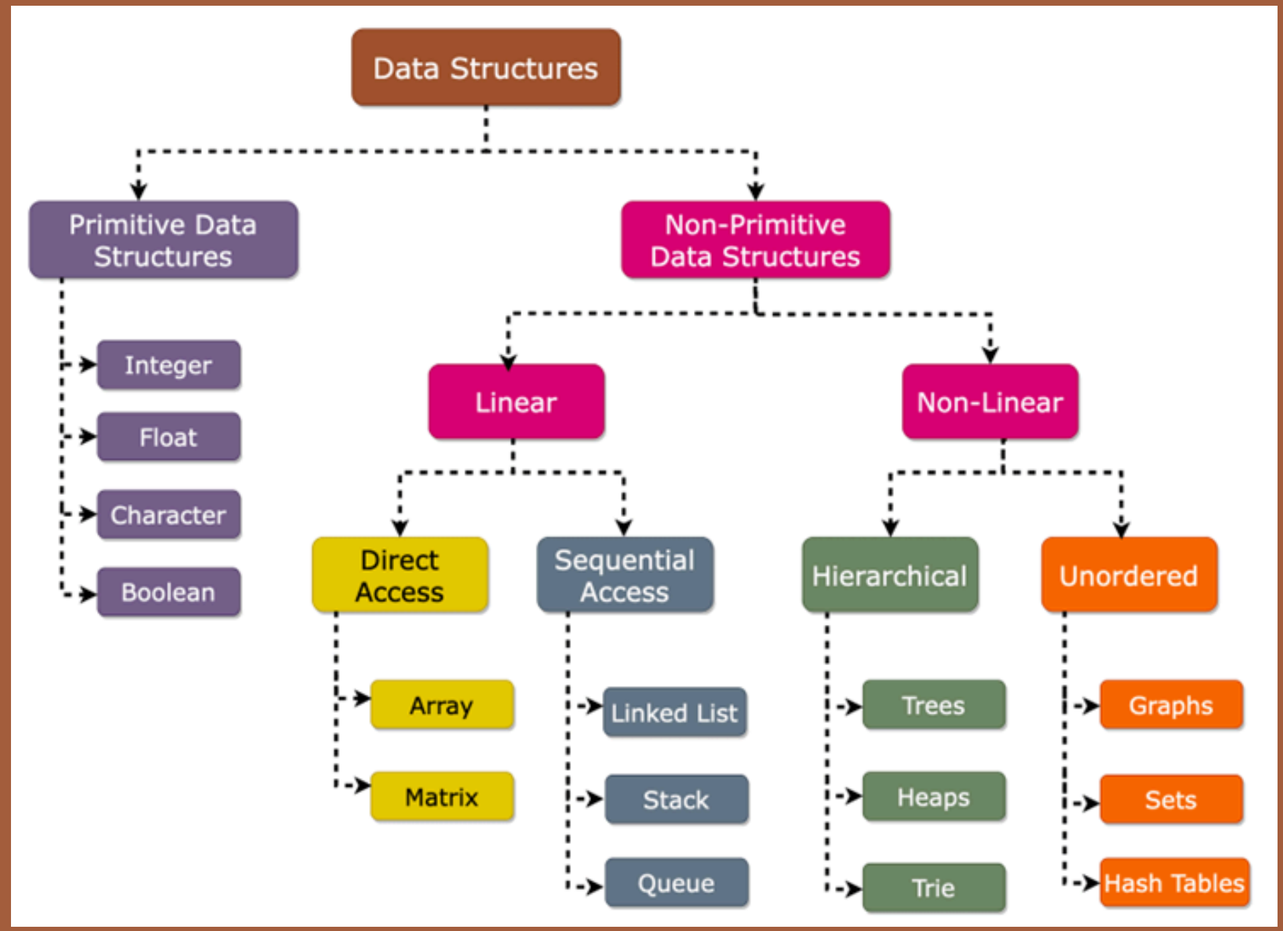
1.1, Identify the Data Structures.

1.1.1, What is a Data Structure?

A data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.

A data structure is not only used for organizing the data. It is also used for processing, retrieving, and storing data. There are different basic and advanced types of data structures that are used in almost every program or software system that has been developed. So we must have good knowledge about data structures.





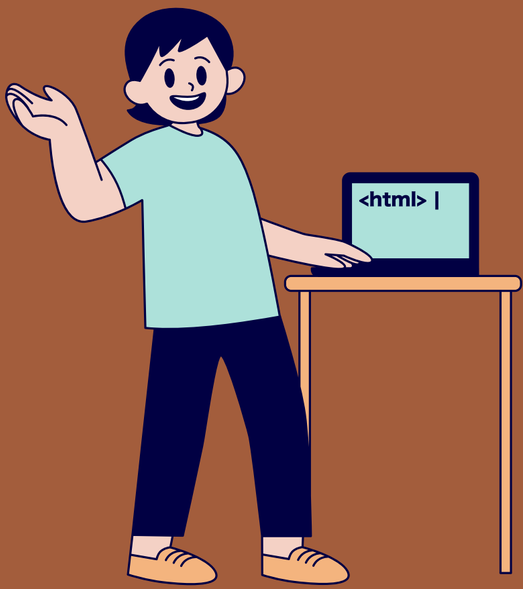
1.2, Provide Examples and Code Snippets (if applicable).

1.2.1, An example of an array in a data structure.

```java
package org.example;

public class Array {

    public static void main(String[] args) {
        // Initialize an array with 5 elements of type int
        int[] arr = {1, 2, 3, 4, 5};

        // Access the 3rd element (index = 2)
        System.out.println("3rd element: " + arr[2]);

        // Modify the value of the element at the 4th position
        arr[3] = 10;
        System.out.println("New element at position 4: " + arr[3]);

        // Iterate and print all elements in the array
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }
    }
}
```

1.2.2, Results when running the program.

```
C:\Users\ADMIN\.jdks\corretto-17.0.12\bin\java.exe
3rd element: 3
New element at position 4: 10
1 2 3 10 5
Process finished with exit code 0
```



2, MEMORY STACK.

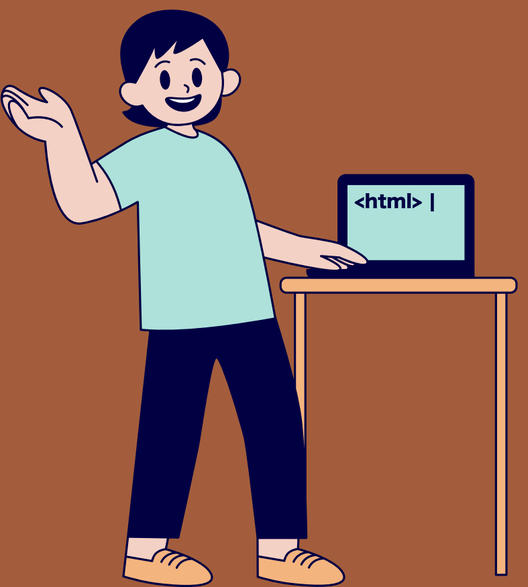
2.1, Define a Memory Stack.

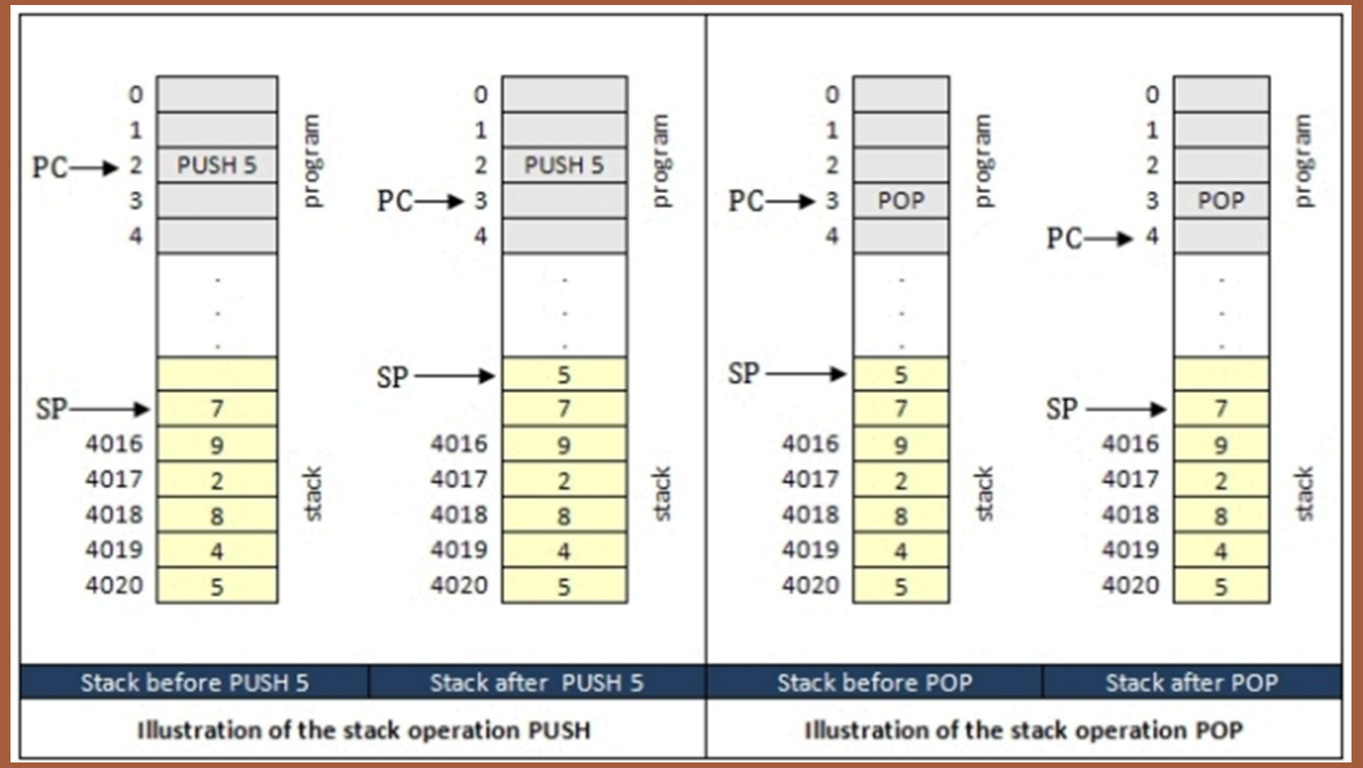
Stack memory is a temporary data storage system that follows the Last-In-First-Out (LIFO) principle. A crucial component of its operation is the Stack Pointer (SP), which keeps track of the current position in the stack and adjusts automatically with each stack operation.

In Cortex®-M processors, the Stack Pointer is register R13. There are two stack pointers, but only one is active at a time, depending on the CONTROL register and processor state.

Stack operations include:

- Pushing (storing data) using the PUSH instruction.
- Popping (restoring data) using the POP instruction.



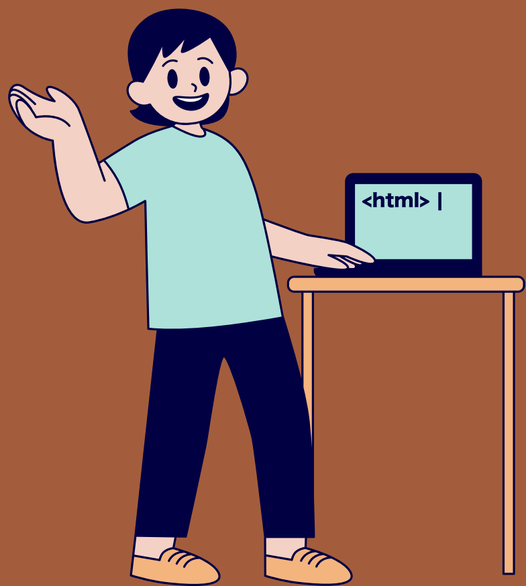


2.2, Function Call Implementation.

```java
package org.example;

public class MemoryStackExample {

    public static void main(String[] args) {
        int x = 5;
        int y = 10;
        int result = multiply(x, y);
        System.out.println("Result: " + result);
    }

    public static int multiply(int a, int b) {
        return a * b;
    }
}
```

Results when running the program:

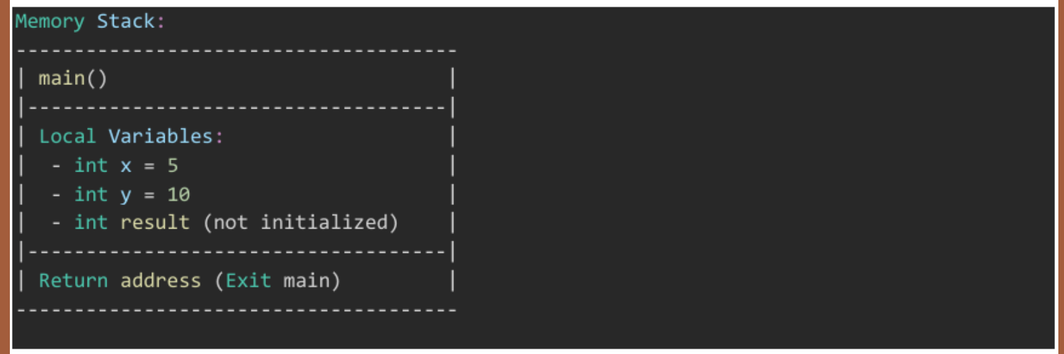
```
C:\Users\ADMIN\.jdks\corretto-17.0.12\bin\java.exe
Result: 50

Process finished with exit code 0
```



2.3, Demonstrate Stack Frames.

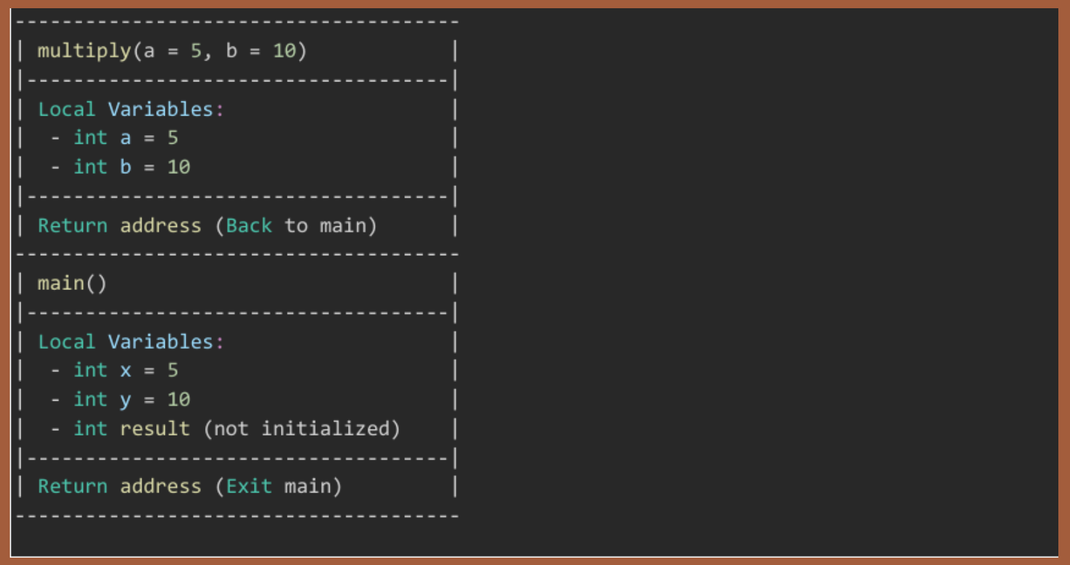
a, Starting the main() method:

- The JVM creates a stack frame for the main() method and pushes it onto the memory stack.



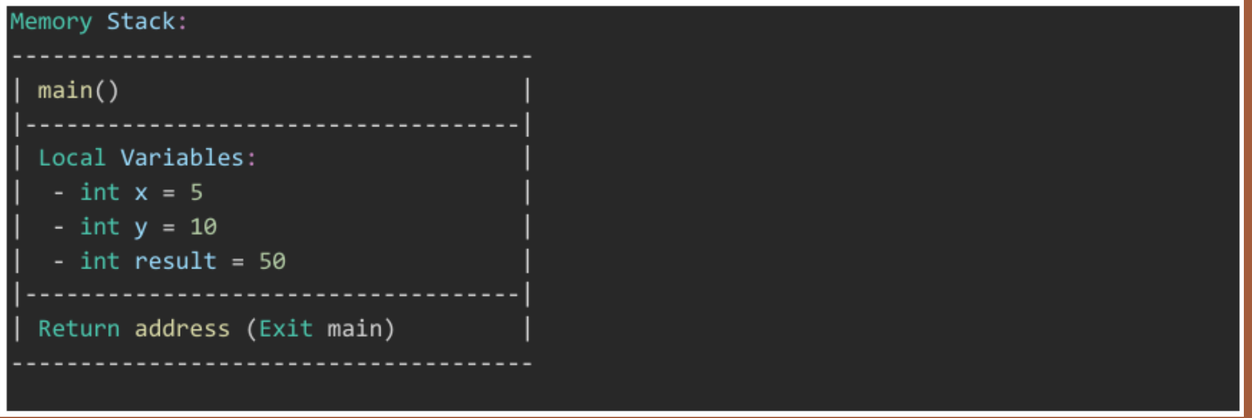
b, Calling multiply(x, y) (multiply(5, 10)):

- A new stack frame is created for the “multiply()” method when it’s called by “main()”. This stack frame contains the parameters “a = 5 and b = 10”. This frame is pushed onto the memory stack above the “main()” stack frame.



c, Returning from multiply() (result = 50):

- The “multiply()” method calculates “5 * 10 = 50”, then returns this value to “main()”. The “multiply()” stack frame is popped off the stack since the method has completed execution, and the result is passed back to the “main()” method.



d, Exiting main() method:

- After the result is printed, the “main()” method finishes. The stack frame for “main()” is popped off the memory stack, and the program terminates.

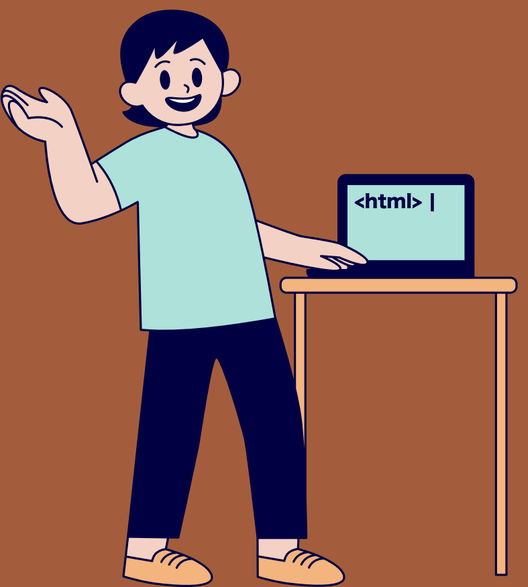
Explanation of the Stack Frames:

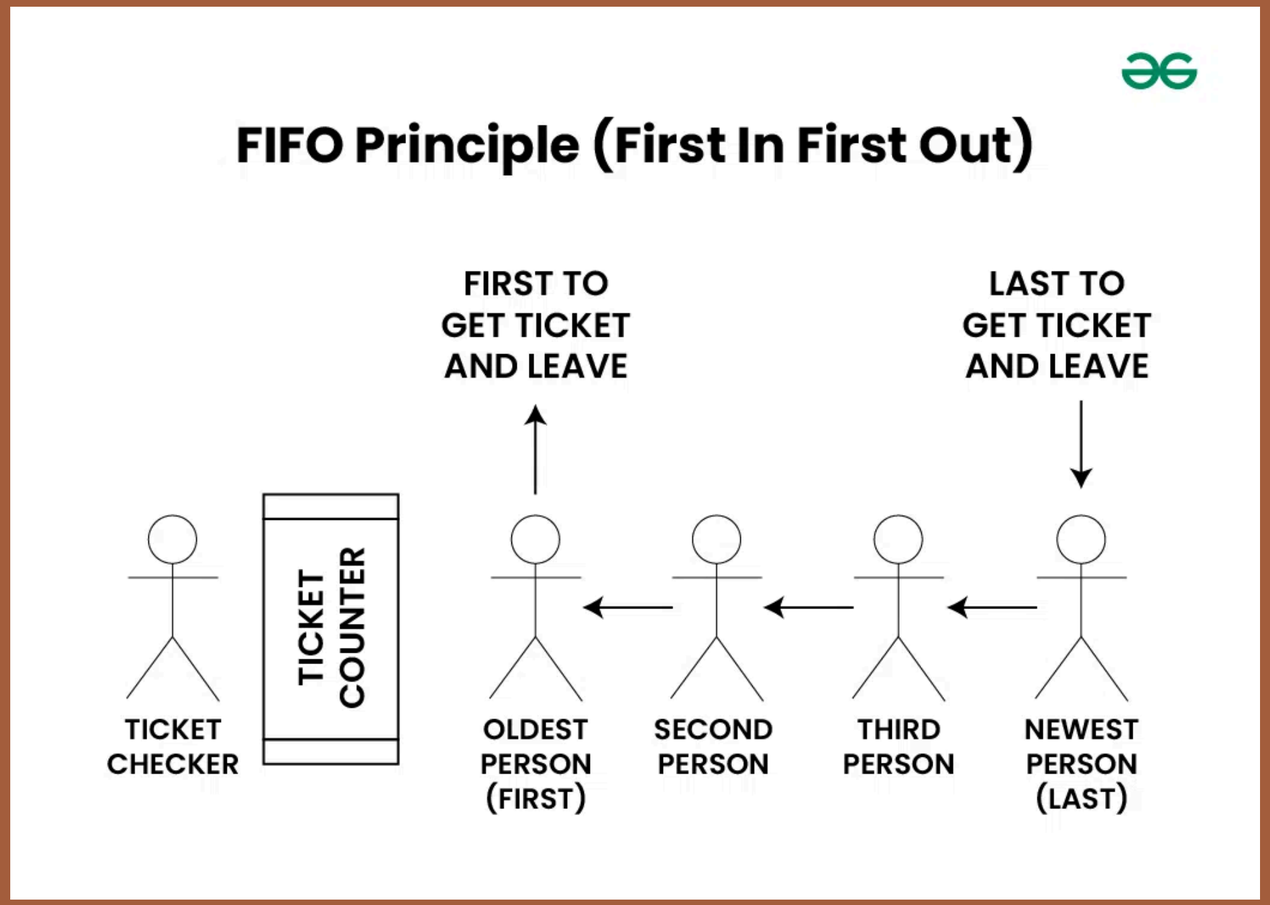
- “main()” Stack Frame: Contains local variables (x, y, and result), and the return address for exiting the program.
- “multiply()” Stack Frame: Contains the method parameters (a and b) and the return address to “main()”.

3, Queue.

3.1,Introduction FIFO.

FIFO (First In, First Out) is an inventory management method and accounting principle that assumes the items purchased or produced first are sold or used first. In this system, the oldest inventory items are recorded as sold before newer ones, which helps determine the cost of goods sold (COGS) and remaining inventory value.





3.2, Array-Based Implementation.

```java
package org.example;

public class ArrayQueue {
    private int[] queue;
    private int front, rear, size, capacity;

    // Constructor to initialize the queue
    public ArrayQueue(int capacity) {
        this.capacity = capacity;
        queue = new int[capacity];
        front = 0;
        size = 0;
        rear = capacity - 1;
    }

    public boolean isFull() {
        return size == capacity;
    }

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

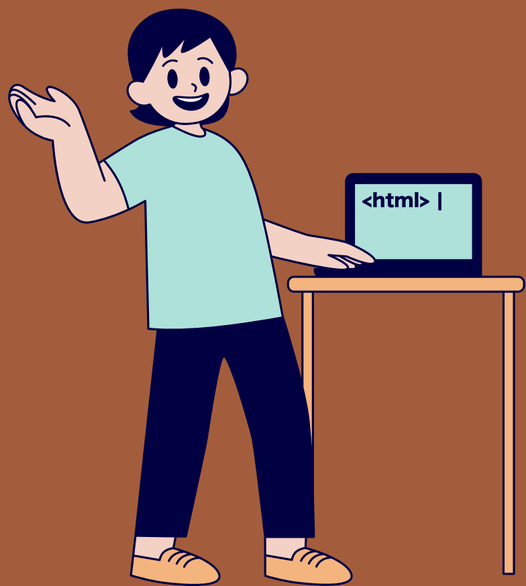
```java
public void enqueue(int item) {
    if (isFull()) {
        System.out.println("Queue is full!");
        return;
    }
    rear = (rear + 1) % capacity;
    queue[rear] = item;
    size++;
}
```

```java
public int dequeue() {
    if (isEmpty()) {
        System.out.println("Queue is empty!");
        return -1;
    }
    int item = queue[front];
    front = (front + 1) % capacity;
    size--;
    return item;
```

```java
public void display() {
    if (isEmpty()) {
        System.out.println("Queue is empty!");
        return;
    }
    System.out.print("Queue contents: ");
    for (int i = 0; i < size; i++) {
        System.out.print(queue[(front + i) % capacity] + " ");
    }
    System.out.println();
}

public static void main(String[] args) {
    ArrayQueue queue = new ArrayQueue(5);

    queue.enqueue(10);
    queue.enqueue(20);
    queue.enqueue(30);
    queue.display(); // Output: Queue contents: 10 20 30

    System.out.println("Dequeued: " + queue.dequeue()); // Output: Dequeued: 10
    queue.enqueue(40);
    queue.enqueue(50);
    queue.display(); // Output: Queue contents: 20 30 40 50

    queue.enqueue(60); // Output: Queue is full!
    System.out.println("Dequeued: " + queue.dequeue()); // Output: Dequeued: 20
    queue.enqueue(60);
    queue.display(); // Output: Queue contents: 30 40 50 60
}
```

Results when running the program:

```
C:\Users\ADMIN\.jdks\corretto-17.0.12\bin\java.exe
Queue contents: 10 20 30
Dequeued: 10
Queue contents: 20 30 40 50
Dequeued: 20
Queue contents: 30 40 50 60 60

Process finished with exit code 0
```



3.3, Linked List-Based Implementation.

```java
package org.example;

public class LinkedListQueue {
    private class Node {
        int data;
        Node next;
        public Node(int data) { this.data = data; this.next = null; }
    }

    private Node front, rear;

    public LinkedListQueue() {
        this.front = null;
        this.rear = null;
    }

    public boolean isEmpty() {
        return front == null;
    }

    public void enqueue(int item) {
        Node newNode = new Node(item);
        if (isEmpty()) {
            front = rear = newNode;
        } else {
            rear.next = newNode;
            rear = newNode;
        }
    }
```

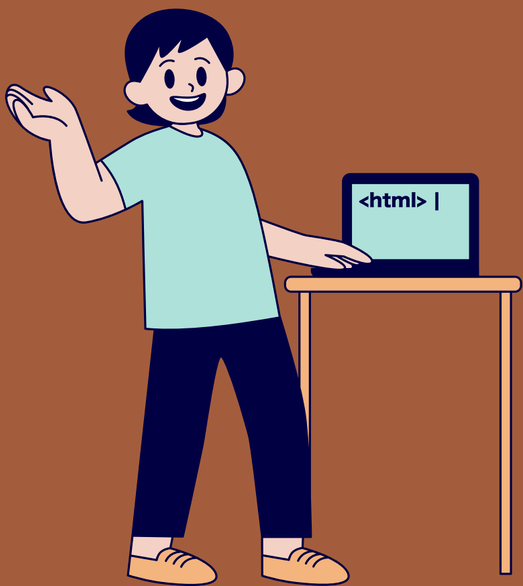
```java
public int dequeue() {
    if (isEmpty()) {
        System.out.println("Queue is empty!");
        return -1;
    }
    int item = front.data;
    front = front.next;
    if (front == null) rear = null; // Queue is empty now
    return item;
}
```

```java
public int peek() {
    if (isEmpty()) {
        System.out.println("Queue is empty!");
        return -1;
    }
    return front.data;
}

public void display() {
    if (isEmpty()) {
        System.out.println("Queue is empty!");
        return;
    }
    System.out.print("Queue contents: ");
    Node current = front;
    while (current != null) {
        System.out.print(current.data + " ");
        current = current.next;
    }
    System.out.println();
}
```

```java
public static void main(String[] args) {
    LinkedListQueue queue = new LinkedListQueue();

    queue.enqueue(10);
    queue.enqueue(20);
    queue.enqueue(30);
    queue.display(); // Output: Queue contents: 10 20 30

    System.out.println("Dequeued: " + queue.dequeue()); // Output: Dequeued: 10
    queue.enqueue(40);
    queue.enqueue(50);
    queue.display(); // Output: Queue contents: 20 30 40 50

    System.out.println("Dequeued: " + queue.dequeue()); // Output: Dequeued: 20
    queue.enqueue(60);
    queue.display(); // Output: Queue contents: 30 40 50 60
}
```

Results when running the program:

```
C:\Users\ADMIN\.jdks\corretto-17.0.12\bin\java.exe
Queue contents: 10 20 30
Dequeued: 10
Queue contents: 20 30 40 50
Dequeued: 20
Queue contents: 30 40 50 60 60

Process finished with exit code 0
```



3.4, Provide a concrete example to illustrate how the FIFO queue works.

a, Array-Based Queue State.

Operation	Queue (Array)	Front	Rear	Size	Status
Enqueue 10 :	[10, _, _, _, _]	0	0	1	Queue: [10]
Enqueue 20 :	[10, 20, _, _, _]	0	1	2	Queue: [10, 20]
Enqueue 30 :	[10, 20, 30, _, _]	0	2	3	Queue: [10, 20, 30]
Dequeue :	[_, 20, 30, _, _]	1	2	2	Queue: [20, 30]
Enqueue 40 :	[_, 20, 30, 40, _]	1	3	3	Queue: [20, 30, 40]
Enqueue 50 :	[_, 20, 30, 40, 50]	1	4	4	Queue: [20, 30, 40, 50]
Dequeue :	[_, _, 30, 40, 50]	2	4	3	Queue: [30, 40, 50]
Enqueue 60 :	[60, _, 30, 40, 50] (wrap)	2	0	4	Queue: [30, 40, 50, 60]
Display :	**[30, 40, 50, 60]**				

b, Linked List-Based Queue State.

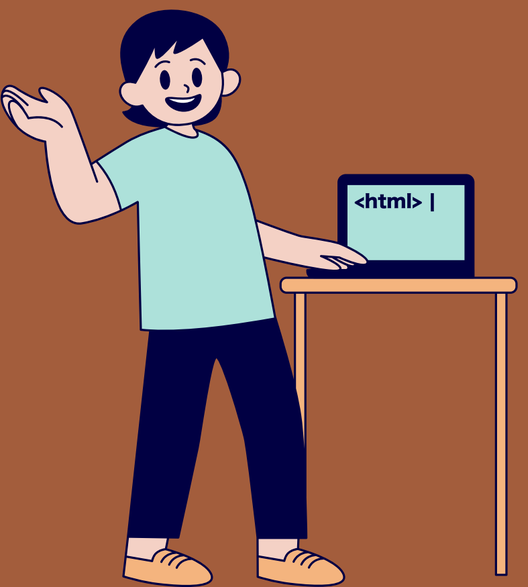
Operation	Queue (Array)	Front	Rear	Status
Enqueue 10 :	10 -> null	10	10	Queue: [10]
Enqueue 20 :	10 -> 20 -> null	10	20	Queue: [10, 20]
Enqueue 30 :	10 -> 20 -> 30 -> null	10	30	Queue: [10, 20, 30]
Dequeue :	20 -> 30 -> null	20	30	Queue: [20, 30]
Enqueue 40 :	20 -> 30 -> 40 -> null	20	40	Queue: [20, 30, 40]
Enqueue 50 :	20 -> 30 -> 40 -> 50 -> null	20	50	Queue: [20, 30, 40, 50]
Dequeue :	30 -> 40 -> 50 -> null	30	50	Queue: [30, 40, 50]
Enqueue 60 :	30 -> 40 -> 50 -> 60 -> null	30	60	Queue: [30, 40, 50, 60]
Display :	**[30, 40, 50, 60]**			

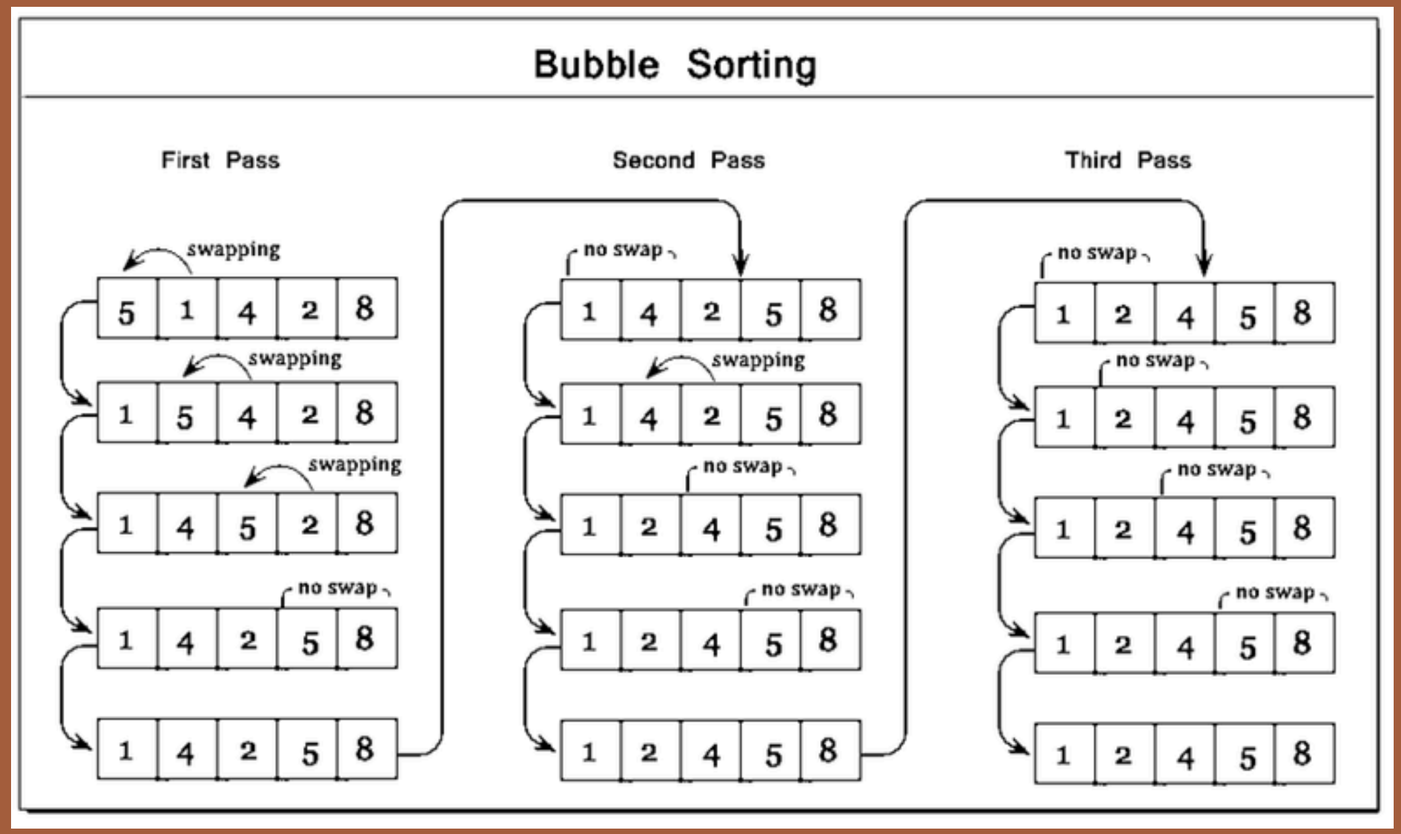
4. Compare the performance of two sorting algorithms.

4.1, Introducing the two sorting algorithms you will be comparing.

4.1.1, Bubble Sort.

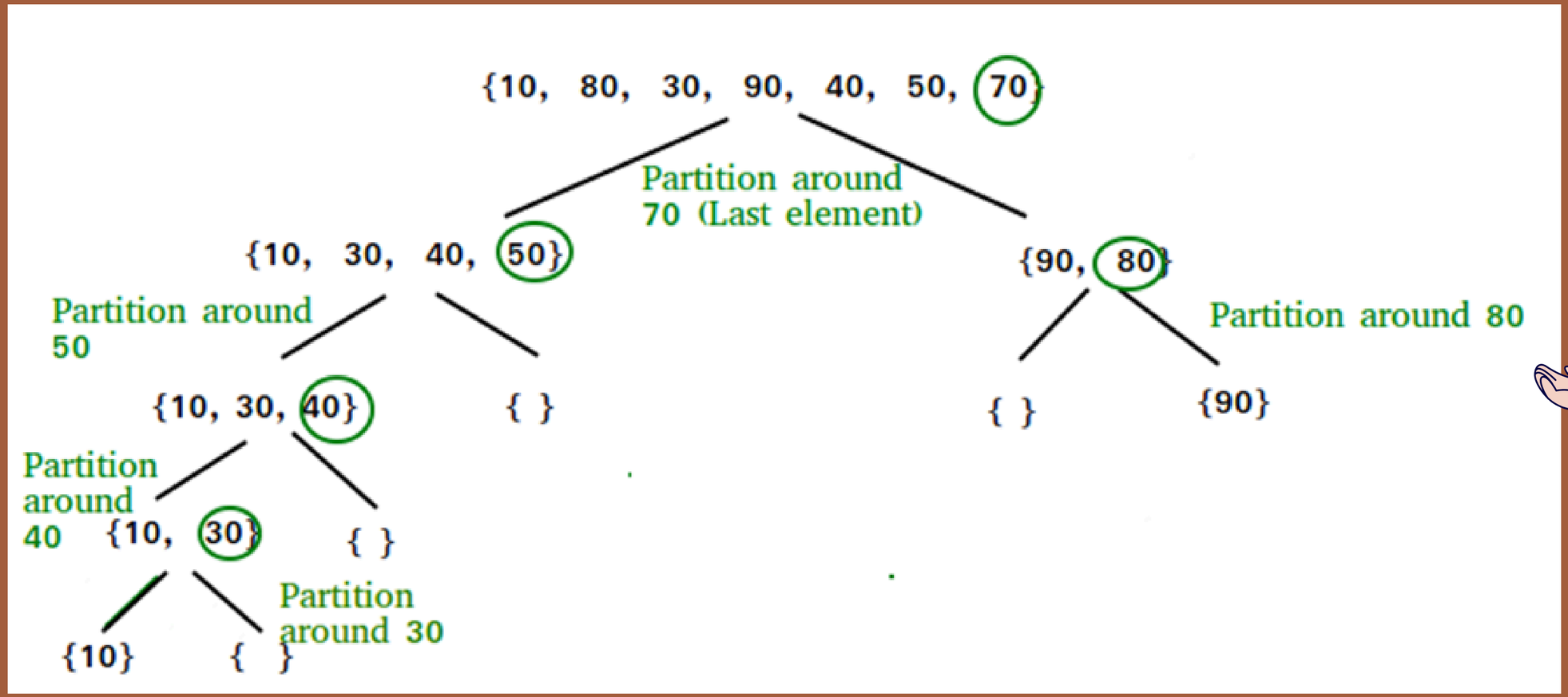
Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst case time complexity are quite high.





4.1.2, Quick Sort.

QuickSort is a sorting algorithm based on the Divide and Conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.



4.2, Provide a concrete example to demonstrate the differences in performance between the two algorithms.

4.2.1, An example of the performance between the two algorithms.

```java
package org.example;
import java.util.Arrays;
import java.util.Random;

public class SortComparison {
    // Bubble Sort implementation
    public static void bubbleSort(int[] arr) {
        int n = arr.length;
        for (int i = 0; i < n - 1; i++) {
            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;
                }
            }
        }
    }
```

```java
    public static void main(String[] args) {
        // Array sizes to test
        int[] smallArray = generateRandomArray(10);      // Small array
        int[] mediumArray = generateRandomArray(100);     // Medium array
        int[] largeArray = generateRandomArray(1000);     // Large array

        // Test Bubble Sort
        System.out.println("Bubble Sort Performance:");
        testSortPerformance(smallArray.clone(), "Bubble Sort");
        testSortPerformance(mediumArray.clone(), "Bubble Sort");
        testSortPerformance(largeArray.clone(), "Bubble Sort");

        // Test Quick Sort
        System.out.println("\nQuick Sort Performance:");
        testSortPerformance(smallArray.clone(), "Quick Sort");
        testSortPerformance(mediumArray.clone(), "Quick Sort");
        testSortPerformance(largeArray.clone(), "Quick Sort");
    }
```

```java
    // Quick Sort implementation
    public static void quickSort(int[] arr, int low, int high) {
        if (low < high) {
            int pi = partition(arr, low, high);
            quickSort(arr, low, pi - 1);   // Sort elements before partition
            quickSort(arr, pi + 1, high);  // Sort elements after partition
        }
    }

    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;
    }
```

```java
    // Method to test and time each sort
    public static void testSortPerformance(int[] array, String sortType) {
        long startTime = System.nanoTime();

        if (sortType.equals("Bubble Sort")) {
            bubbleSort(array);
        } else if (sortType.equals("Quick Sort")) {
            quickSort(array, 0, array.length - 1);
        }

        long endTime = System.nanoTime();
        long duration = (endTime - startTime) / 1000000; // Convert to milliseconds
        System.out.println(sortType + " (Array size " + array.length + "): " + duration
+ " ms");
    }

    // Helper method to generate a random array of specified size
    public static int[] generateRandomArray(int size) {
        Random rand = new Random();
        int[] array = new int[size];
        for (int i = 0; i < size; i++) {
            array[i] = rand.nextInt(1000);
        }
        return array;
    }
}
```

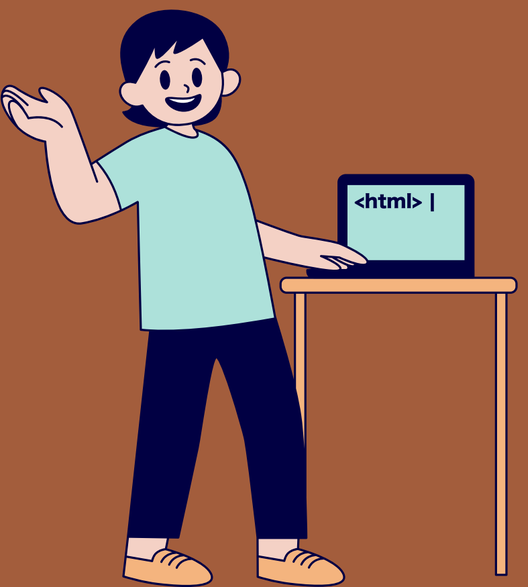
4.2.2, Results when running the program:

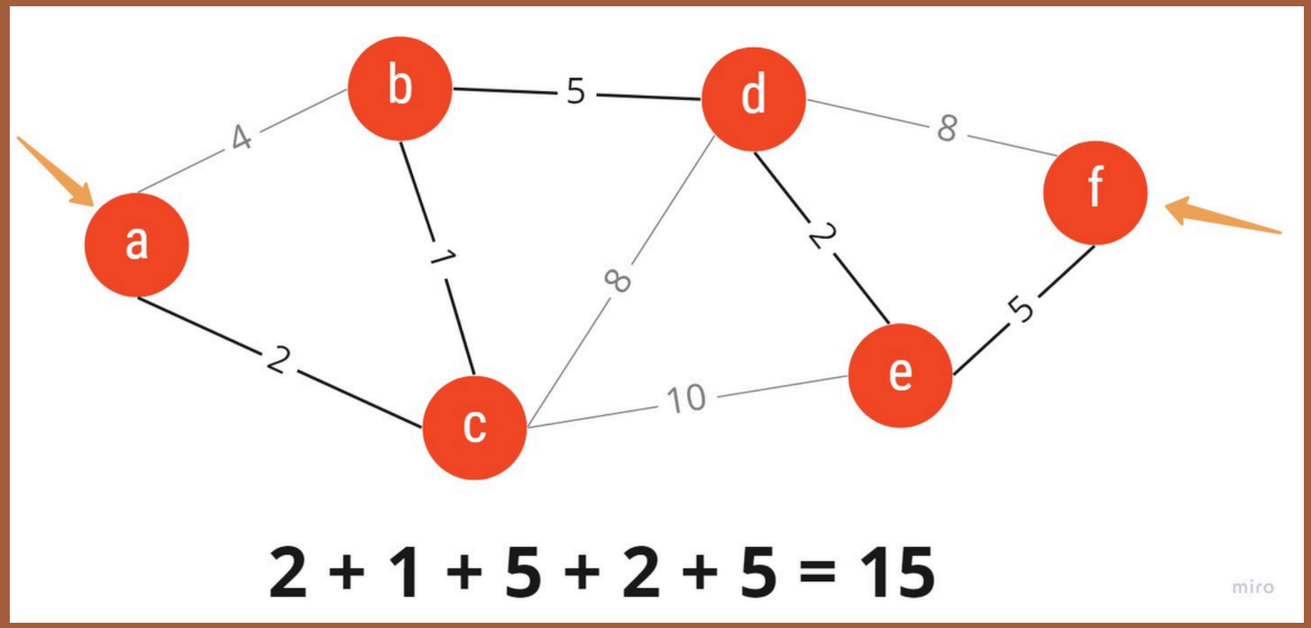
```
C:\Users\ADMIN\.jdks\corretto-17.0.12\bin\java.exe
Bubble Sort Performance:
Bubble Sort (Array size 10): 0 ms
Bubble Sort (Array size 100): 0 ms
Bubble Sort (Array size 1000): 10 ms

Quick Sort Performance:
Quick Sort (Array size 10): 0 ms
Quick Sort (Array size 100): 0 ms
Quick Sort (Array size 1000): 1 ms

Process finished with exit code
```

5, Network shortest path algorithms.

5.1, Introducing the concept of network shortest path algorithms.

Network shortest path algorithms are designed to find the most efficient route from a source node to a destination node within a weighted graph or network. These algorithms play a crucial role in various applications, including routing in computer networks, transportation systems, and logistics. The objective is to minimize the total weight (or cost) of the path, which can represent distance, time, or any other metric.



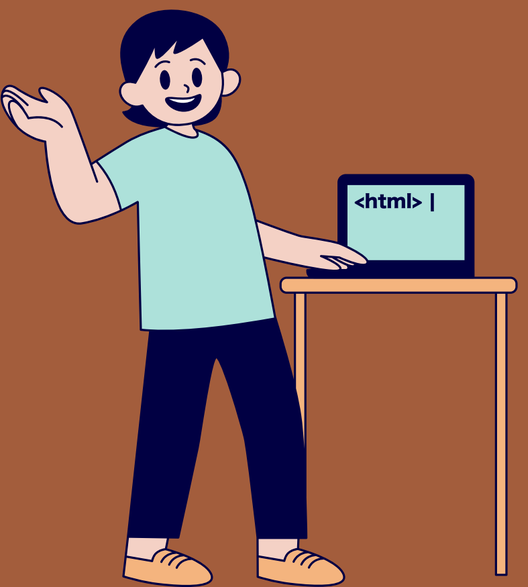


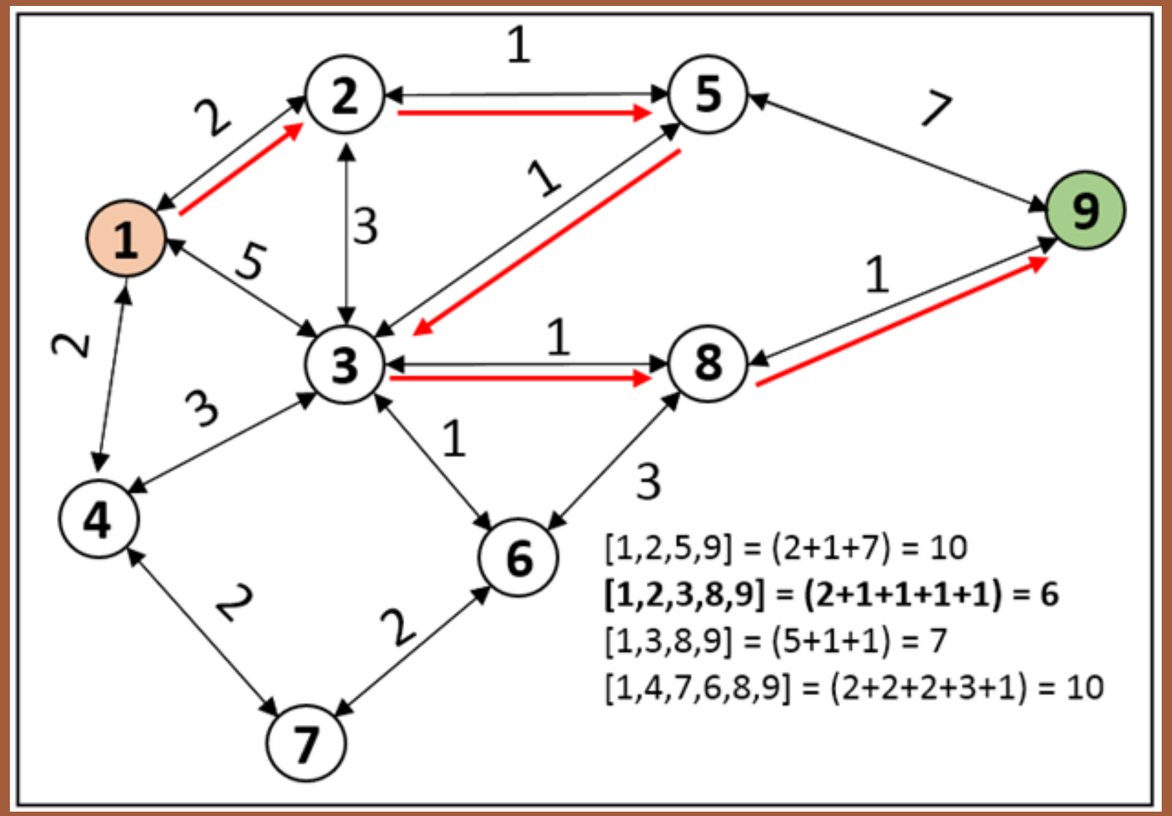
5.2, Algorithm 1: Dijkstra's Algorithm.

5.2.1, Dijkstra's Algorithm.

Dijkstra’s algorithm is a well-known method for finding the shortest path from a single source node to all other nodes in a graph with non-negative edge weights. It operates on a greedy approach, expanding paths from the source node one step at a time, always choosing the path with the smallest cumulative weight.

- **Process:** Starting from the source node, Dijkstra’s algorithm repeatedly selects the node with the lowest cumulative distance, marks it as visited, and updates the shortest known distances to neighboring nodes. This process continues until the shortest path to each reachable node is identified.
- **Applications in Networking:** Widely used in routing protocols like OSPF (Open Shortest Path First), which determines the most efficient path for data packet travel through an IP network.





5.2.2, Example of Dijkstra's algorithm.

```java
package org.example;
import java.util.*;

public class DijkstraGraph {
    private final Map<Integer, List<Edge>> adjacencyList;

    public DijkstraGraph() {
        adjacencyList = new HashMap<>();
    }

    public void addEdge(int source, int destination, int weight) {
        adjacencyList.putIfAbsent(source, new ArrayList<>());
        adjacencyList.putIfAbsent(destination, new ArrayList<>());
        adjacencyList.get(source).add(new Edge(destination, weight));
        adjacencyList.get(destination).add(new Edge(source, weight)); // if it's an undirected graph
    }
```

```java
public Map<Integer, Integer> dijkstra(int start) {
    Map<Integer, Integer> distances = new HashMap<>();
    for (int node : adjacencyList.keySet()) {
        distances.put(node, Integer.MAX_VALUE); // Start with "infinity"
    }
    distances.put(start, 0); // Distance to start node is 0

    PriorityQueue<Edge> priorityQueue = new PriorityQueue<>(Comparator.comparingInt(edge -> edge.weight));
    priorityQueue.add(new Edge(start, 0));

    while (!priorityQueue.isEmpty()) {
        Edge currentEdge = priorityQueue.poll();
        int currentNode = currentEdge.destination;

        for (Edge neighbor : adjacencyList.getOrDefault(currentNode, Collections.emptyList())) {
            int newDist = distances.get(currentNode) + neighbor.weight;
            if (newDist < distances.get(neighbor.destination)) {
                distances.put(neighbor.destination, newDist);
                priorityQueue.add(new Edge(neighbor.destination, newDist));
            }
        }
    }

    return distances;
}
```

```java
static class Edge {
    int destination;
    int weight;

    public Edge(int destination, int weight) {
        this.destination = destination;
        this.weight = weight;
    }
}

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

    int startNode = 0;
    Map<Integer, Integer> distances = graph.dijkstra(startNode);

    System.out.println("Shortest distances from node " + startNode + ":");
    for (Map.Entry<Integer, Integer> entry : distances.entrySet()) {
        System.out.println("Node " + entry.getKey() + " - Distance: " + entry.getValue());
    }
}
```

Results when running the program:

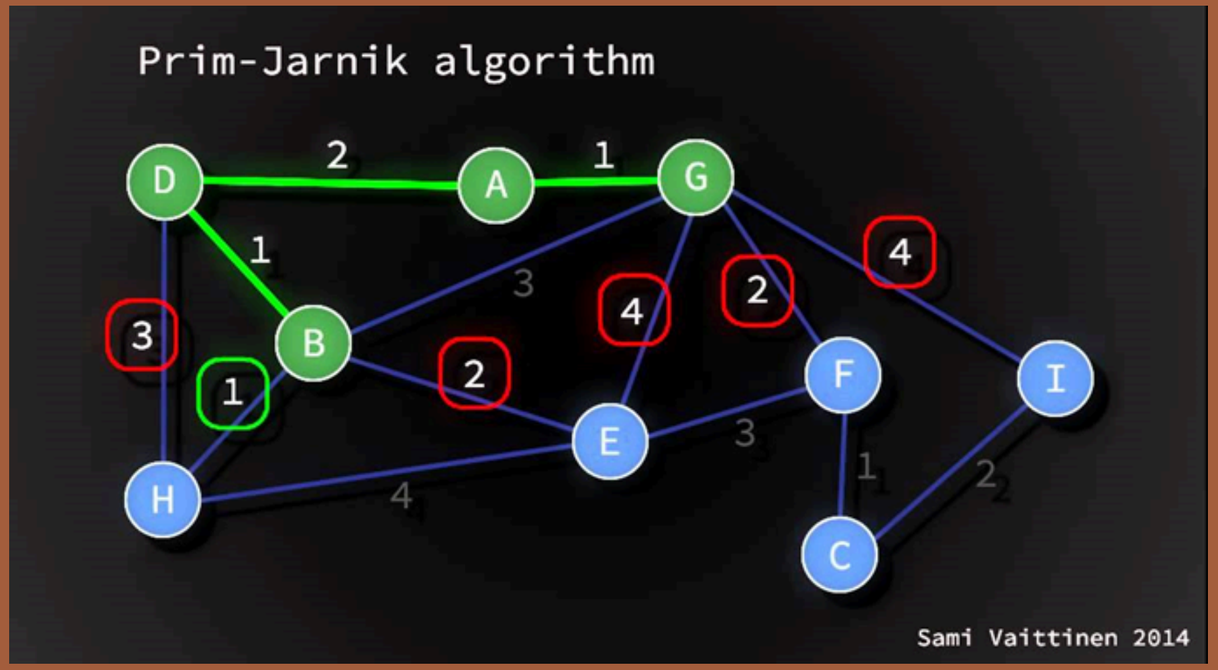
```
C:\Users\ADMIN\.jdks\corretto-17.0.12\bin\java.exe
Shortest distances from node 0:
Node 0 - Distance: 0
Node 1 - Distance: 3
Node 2 - Distance: 1
Node 3 - Distance: 4

Process finished with exit code 0
```

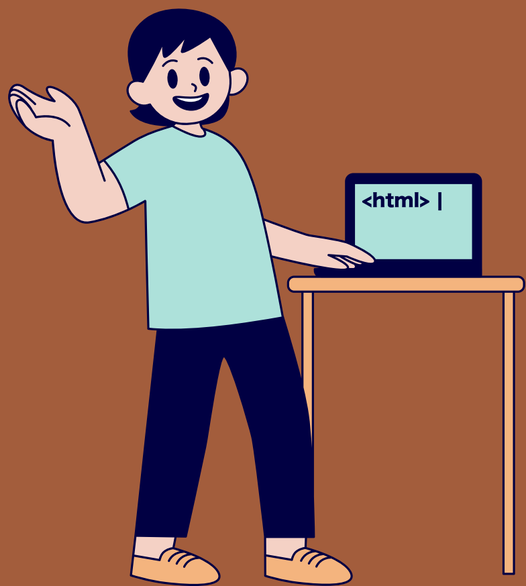
5.3, Algorithm 2: Prim-Jarnik Algorithm.

5.3.1, Prim-Jarnik's algorithm.

Prim-Jarnik, or Prim's algorithm, is typically associated with finding the Minimum Spanning Tree (MST) of a graph rather than directly solving shortest path problems. However, it's relevant in network design when connecting nodes with the minimum total weight without creating loops, ensuring connectivity across the entire network with minimal cost.

- **Process:** Starting from an arbitrary node, Prim's algorithm expands the MST by repeatedly adding the smallest edge that connects a node inside the tree to a node outside it. The algorithm continues until all nodes are included, resulting in a tree that spans all nodes with minimal weight.
- **Applications in Networking:** Used for network layout and cable routing, Prim's algorithm helps minimize total wiring costs in physical networks and serves in network clustering and backbone design for efficient data transmission.





5.3.2, Example of Prim-Jarnik's algorithm.

```java
package org.example;
import java.util.*;

public class PrimGraph {
    private final Map<Integer, List<Edge>> adjacencyList;

    public PrimGraph() {
        adjacencyList = new HashMap<>();
    }

    public void addEdge(int source, int destination, int weight) {
        adjacencyList.putIfAbsent(source, new ArrayList<>());
        adjacencyList.putIfAbsent(destination, new ArrayList<>());
        adjacencyList.get(source).add(new Edge(destination, weight));
        adjacencyList.get(destination).add(new Edge(source, weight));
    }
```

```java
static class Edge {
    int destination;
    int weight;

    public Edge(int destination, int weight) {
        this.destination = destination;
        this.weight = weight;
    }

    @Override
    public String toString() {
        return String.format("Edge to node %d with weight %d", destination, weight);
    }
}
```

```java
public List<Edge> prim(int start) {
    List<Edge> mst = new ArrayList<>();
    Set<Integer> visited = new HashSet<>();
    PriorityQueue<Edge> priorityQueue = new PriorityQueue<>
    (Comparator.comparingInt(edge -> edge.weight));

    visited.add(start);
    priorityQueue.addAll(adjacencyList.get(start));

    while (!priorityQueue.isEmpty() && visited.size() < adjacencyList.size()) {
        Edge edge = priorityQueue.poll();

        if (!visited.contains(edge.destination)) {
            visited.add(edge.destination);
            mst.add(edge);

            for (Edge neighbor : adjacencyList.get(edge.destination)) {
                if (!visited.contains(neighbor.destination)) {
                    priorityQueue.add(neighbor);
                }
            }
        }
    }
    return mst;
}
```

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

    int startNode = 0;
    List<Edge> mst = graph.prim(startNode);

    System.out.println("Minimum Spanning Tree starting
    from node " + startNode + ":");

    for (Edge edge : mst) {
        System.out.println(edge);
    }
}
```

Results when running the program:

```
C:\Users\ADMIN\.jdks\corretto-17.0.12\bin\java.exe
Minimum Spanning Tree starting from node 0:
Edge to node 2 with weight 3
Edge to node 1 with weight 1
Edge to node 3 with weight 2
Edge to node 4 with weight 2
Edge to node 5 with weight 6

Process finished with exit code 0
```

Thank you for listening to my presentation.