Phạm Quang Huy

BH01568

1 Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

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 2 types of data structure: Primitive Data Structures and Abstract Data Structure.

1 Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

What is Data Structure

Abstract Data Structures 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, queue...

1 Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

What is Pre and Post Condition

Pre condition

A precondition is a requirement that must be true before a method executes. The method expects certain conditions to be met beforehand; otherwise, it may not function correctly.

Post condition

A postcondition is a condition that is guaranteed to be true after a method finishes. If the operation is correct and the preconditions are met, the postcondition holds.

1 Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

What is Time and Space Complexity

Time Complexity

The valid algorithm takes a finite amount of time for execution. The time required by the algorithm to solve given problem is called time complexity of the algorithm.

Space Complexity

Problem-solving with a computer requires memory to store temporary data or final results during execution. The memory needed by an algorithm to solve a problem is called its space complexity.

1 Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

Array

Operations

- **Access**: Retrieve an element at a specific index.
- **Insert**: Add an element at a specific index.
- **Delete**: Remove an element at a specific index.
- **Search**: Find an element in the array.

Input Parameters

- **For Access, Insert, and Delete**: index, element
- **For Search**: element

1 Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

Array

Pre and Post Condition

- **Pre-conditions:**
  - **For Access, Insert, and Delete:** 0 ≤ index < array_size
- **Post-conditions:**
  - **For Access:** The element at the given index is returned.
  - **For Insert:** The array grows by one element, and the element is placed at the specified index.
  - **For Delete:** The array shrinks by one element, and elements are shifted accordingly.
  - **For Search:** The index of the found element is returned or -1 if not found.

Time and Space Complexity

- **Time Complexity:**
  - **Access: O(1)**
  - **Insert: O(n)**
  - **Delete: O(n)**
  - **Search: O(n)**
- **Space Complexity: O(n)**

Array _ Example Code

```
no usages
public class Array {
    14 usages
    private int[] arr;

    no usages
    public Array(int size) {
        arr = new int[size];
    }
```

- **Attributes**:
  - **arr**: An array of integers.
- **Constructor**: Initializes the array with a specified size.

Array _ Example Code

```
no usages
public int access(int index) {
    if (index >= 0 && index < arr.length) {
        return arr[index];
    } else {
        throw new IndexOutOfBoundsException("Index out of bounds");
    }
}

no usages
public void insert(int index, int element) {
    if (index >= 0 && index <= arr.length) {
        int[] newArr = new int[arr.length + 1];
        for (int i = 0, j = 0; i < newArr.length; i++) {
            if (i == index) {
                newArr[i] = element;
            } else {
                newArr[i] = arr[j++];
            }
        }
        arr = newArr;
    } else {
        throw new IndexOutOfBoundsException("Index out of bounds");
    }
}
```

- **Methods:**
  - **access(int index):** Retrieves the element at the specified index, throwing an exception if the index is out of bounds.
  - **insert(int index, int element):** Inserts an element at the specified index. It creates a new array with an extra space, shifts elements, and adds the new element.

Array _ Example Code

```
    no usages
    public void delete(int index) {
        if (index >= 0 && index < arr.length) {
            int[] newArr = new int[arr.length - 1];
            for (int i = 0, j = 0; i < arr.length; i++) {
                if (i != index) {
                    newArr[j++] = arr[i];
                }
            }
            arr = newArr;
        } else {
            throw new IndexOutOfBoundsException("Index out of bounds");
        }
    }

    no usages
    public int search(int element) {
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == element) {
                return i;
            }
        }
        return -1;
    }
}
```

- **Methods:**
  - **delete(int index):** Removes the element at the specified index. It creates a new array without the deleted element and copies over the remaining elements.
  - **search(int element):** Searches for an element in the array and returns its index or -1 if not found.

1 Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

Linked List

Operations

- **Insert:** Add an element at the beginning, end, or middle.
- **Delete:** Remove an element by value or position.
- **Search:** Find an element by value.
- **Traversal:** Visit each node sequentially.

Input Parameters

- **For Insert:**
  element, position
- **For Delete:**
  position or element
- **For Search:**
  element

1 Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

Linked List

Pre and Post Condition

- **Pre-conditions:**
  - **For Insert, Delete:** The linked list should be initialized.
  - **For Search:** The linked list should contain elements.
- **Post-conditions:**
  - **For Insert:** The element is added to the list.
  - **For Delete:** The element is removed from the list, or no action if the element doesn't exist.
  - **For Search:** The node is returned, or null if not found.

Time and Space Complexity

- **Time Complexity:**
  - **Insert: O(1) (beginning), O(n) (any position)**
  - **Delete: O(n)**
  - **Search: O(n)**
  - **Traversal: O(n)**
- **Space Complexity: O(n)**

Linked List _ Example Code

```
10 usages
class Node {
    5 usages
    int data;
    14 usages
    Node next;

    2 usages
    Node(int data) {
        this.data = data;
        this.next = null;
    }
}
```

- **Class: Node**
  - **Attributes:**
    - **data:** The value stored in the node.
    - **next:** A reference to the next node in the list.

Linked List _ Example Code

```java
public class LinkedList {
    12 usages
    private Node head;

    // Insert at the beginning
    no usages
    public void insertAtBeginning(int data) {
        Node newNode = new Node(data);
        newNode.next = head;
        head = newNode;
    }

    // Insert at the end
    no usages
    public void insertAtEnd(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            return;
        }
        Node current = head;
        while (current.next != null) {
            current = current.next;
        }
        current.next = newNode;
    }
```

- **Class: LinkedList**
  - **Attributes:**
    - **head: A reference to the first node in the list.**
  - **Methods:**
    - **insertAtBeginning(int data):** Creates a new node and inserts it at the beginning by updating the head reference.
    - **insertAtEnd(int data):** Adds a new node at the end of the list by traversing to the last node and setting its next reference to the new node.

Linked List _ Example Code

```java
public void deleteByValue(int data) {
    if (head == null) return;

    if (head.data == data) {
        head = head.next;
        return;
    }

    Node current = head;
    while (current.next != null) {
        if (current.next.data == data) {
            current.next = current.next.next;
            return;
        }
        current = current.next;
    }
}

// Search
no usages
public boolean search(int data) {
    Node current = head;
    while (current != null) {
        if (current.data == data) {
            return true;
        }
        current = current.next;
    }
    return false;
}
```

- **Class: LinkedList**
  - **Methods:**
    - **deleteByValue(int data):** Removes a node by searching for the node with the specified value, adjusting pointers accordingly to bypass the deleted node.
    - **search(int data): Traverses the list to check if a node with the specified value exists.**

Linked List _ Example Code

```java
// Traversal
no usages
public void traverse() {
    Node current = head;
    while (current != null) {
        System.out.print(current.data + " ");
        current = current.next;
    }
    System.out.println();
}
```

- **Class: LinkedList**
  - **Methods:**
    - **traverse()**: Prints all the elements in the list.

1 Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

Stack

Operations

- **Push:** Insert an element at the top of the stack.
- **Pop:** Remove the top element from the stack.
- **Peek:** Access the top element without removing it.
- **isEmpty:** Check if the stack is empty.

Input Parameters

- **For Push:** element

1 Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

Stack

Pre and Post Condition

- **Pre-conditions:**
  - The stack must be initialized, and its size must not exceed the maximum capacity (if defined).
- **Post-conditions:**
  - **For Push:** The element is added to the top of the stack.
  - **For Pop:** The top element is removed, and the stack is updated.
  - **For Peek:** The top element is returned without being removed.

Time and Space Complexity

- **Time Complexity:**
  - **Push: O(1)**
  - **Pop: O(1)**
  - **Peek: O(1)**
  - **isEmpty: O(1)**
- **Space Complexity: O(n)**

Stack _ Example Code

```java
no usages
public class Stack {
    4 usages
    private int[] arr;
    6 usages
    private int top;
    2 usages
    private int capacity;

    no usages
    public Stack(int size) {
        arr = new int[size];
        capacity = size;
        top = -1;
    }
```

- **Class: Stack**
  - **Attributes:**
    - **arr:** An array to store stack elements.
    - **top:** The index of the top element in the stack.
    - **capacity:** The maximum size of the stack.

Stack _ Example Code

```
no usages
public void push(int element) {
    if (top == capacity - 1) {
        throw new StackOverflowError( s: "Stack is full");
    }
    arr[++top] = element;
}

// Pop
no usages
public int pop() {
    if (isEmpty()) {
        throw new EmptyStackException();
    }
    return arr[top--];
}
```

- **Methods:**
  - **push(int element):** Adds an element to the top of the stack by incrementing top and assigning the value to the array. Throws an exception if the stack is full.
  - **pop():** Removes and returns the top element. Throws an exception if the stack is empty.

Stack _ Example Code

```
no usages
public int peek() {
    if (isEmpty()) {
        throw new EmptyStackException();
    }
    return arr[top];
}

// isEmpty
2 usages
public boolean isEmpty() {
    return top == -1;
}
```

- **Methods:**
  - **peek():** Returns the top element without removing it, throwing an exception if the stack is empty.
  - **isEmpty():** Checks if the stack is empty by verifying if top is -1.

1 Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

Queue

Operations

- **Enqueue:** Add an element to the rear of the queue.
- **Dequeue:** Remove an element from the front of the queue.
- **Peek:** Access the front element without removing it.
- **isEmpty:** Check if the queue is empty.
-

Input Parameters

- **For Enqueue:**
  element

1 Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

Queue

Pre and Post Condition

- **Pre-conditions:**
  - The queue should be initialized, and its size should not exceed the maximum capacity (if defined).
- **Post-conditions:**
  - **For Enqueue:** The element is added to the rear**.**
  - **For Dequeue:** The front element is removed.
  - **For Peek:** The front element is returned without being removed.

Time and Space Complexity

- **Time Complexity:**
  - **Enqueue: O(1)**
  - **Dequeue: O(1)**
  - **Peek: O(1)**
  - **isEmpty: O(1)**
- **Space Complexity: O(n)**

Queue _ Example Code

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

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

- **Class: Queue**
  - **Attributes:**
    - **arr:** An array to store queue elements.
    - **front:** The index of the front element.
    - **rear:** The index of the last element.
    - **capacity:** The maximum size of the queue.
    - **size:** The current number of elements in the queue.

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++;
}

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

- **Methods:**
  - **enqueue(int element):** Adds an element to the rear of the queue. Increments the rear index and wraps around using modulo to maintain circularity. Throws an exception if the queue is full.
  - **dequeue():** Removes and returns the front element. Updates the front index and decrements size. Throws an exception if the queue is empty.

Queue _ Example Code

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

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

- **Methods:**
  - **peek()**: Returns the front element without removing it, throwing an exception if the queue is empty.
  - **isEmpty()**: Checks if the queue is empty by verifying if size is 0.

2 Determine the operations of a memory stack and how it is used to implement function calls in a computer.

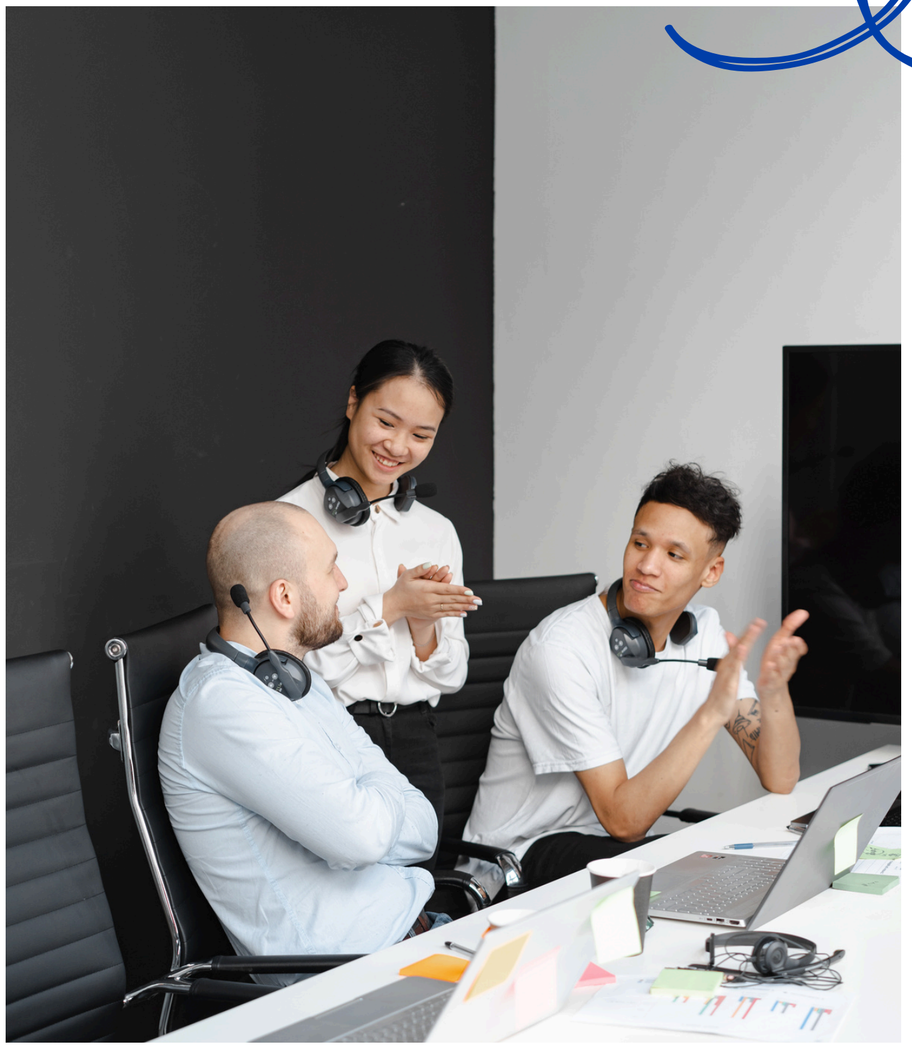
2.1. Define a Memory Stack:

Stack memory is a section in memory used by functions to store data such as local variables and parameters that will be used by the malware to perform its nefarious activity on a compromised device.

2.2. Identify Operation

There are 2 fundamental operations performed on a memory stack:

- Push:
  - Definition: Adds (or pushes) an item onto the top of the stack.
  - Use: When a function is called, the return address (where to continue after the function ends), local variables, and parameters are pushed onto the stack.
- Pop:
  - Definition: Removes (or pops) the item from the top of the stack.
  - Use: When a function completes, its return address, local variables, and parameters are popped from the stack, restoring the state of the previous function.



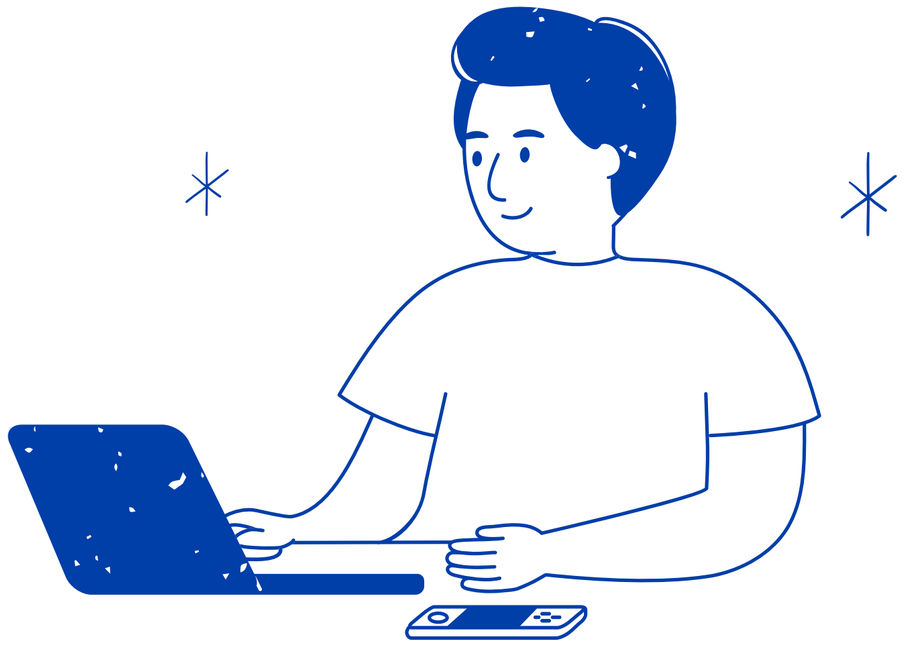
2.3. Function call implementaion

Whenever a function is called, a new stack frame is created with all the function’s data, and this stack frame is pushed into the program stack, and the stack pointer that always points to the top of the program stack points to the stack frame pushed as it is on the top of the program stack.

2.3. Function call implementaion

The series of operations when we call a function are as follows:

1. Stack Frame is pushed into the stack.
2. Sub-routine instructions are executed.
3. Stack Frame is popped from the stack.
4. Now Program Counter is holding the return address.



2.4. Demonstrate Stack Frames

A stack frame is the buffer memory that is the element of the call stack. It is a collection of all the local variables, function parameters, return address, and all the data related to the called function.

2.4. Demonstrate Stack Frames

```
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the value of a: ");
        int a = scanner.nextInt();

        System.out.print("Enter the value of b: ");
        int b = scanner.nextInt();

        int product = findProduct(a, b);
        System.out.println("Product is : " + product);

        int sum = findSum(a, b);
        System.out.println("Sum is : " + sum);

        scanner.close();
    }
}
```

Main Function's Stack Frame:

- When the program starts, the main method is called, and a stack frame is created for the main function. Inside the stack frame:
  - Local variables a and b are stored.
    - The stack also holds information about the program's execution, like the return address (where control goes after main finishes).

Calling findProduct(a, b):

- A new stack frame is created when findProduct is called.
- Inside the findProduct stack frame, the parameters a and b (passed from main) are stored.
- A local variable product is created inside this stack frame.
- After the product is computed and returned, the stack frame for findProduct is popped (deleted), and control returns to main

.

Calling findSum(a, b):

- A new stack frame is created for findSum.
- Parameters a and b are passed and stored in this new stack frame.
- A local variable sum is created inside the stack frame.
- After sum is computed and returned, the stack frame for findSum is popped (removed), and control returns to main.

```java
package MemoryStack;
import java.util.Scanner;

public class StackFrame {
    1 usage
    public static int findProduct(int a, int b) {
        int product = a * b;
        return product;
    }

    1 usage
    public static int findSum(int a, int b) {
        int sum = a + b;
        return sum;
    }
```

2.5. The imprtance of Memory Stack

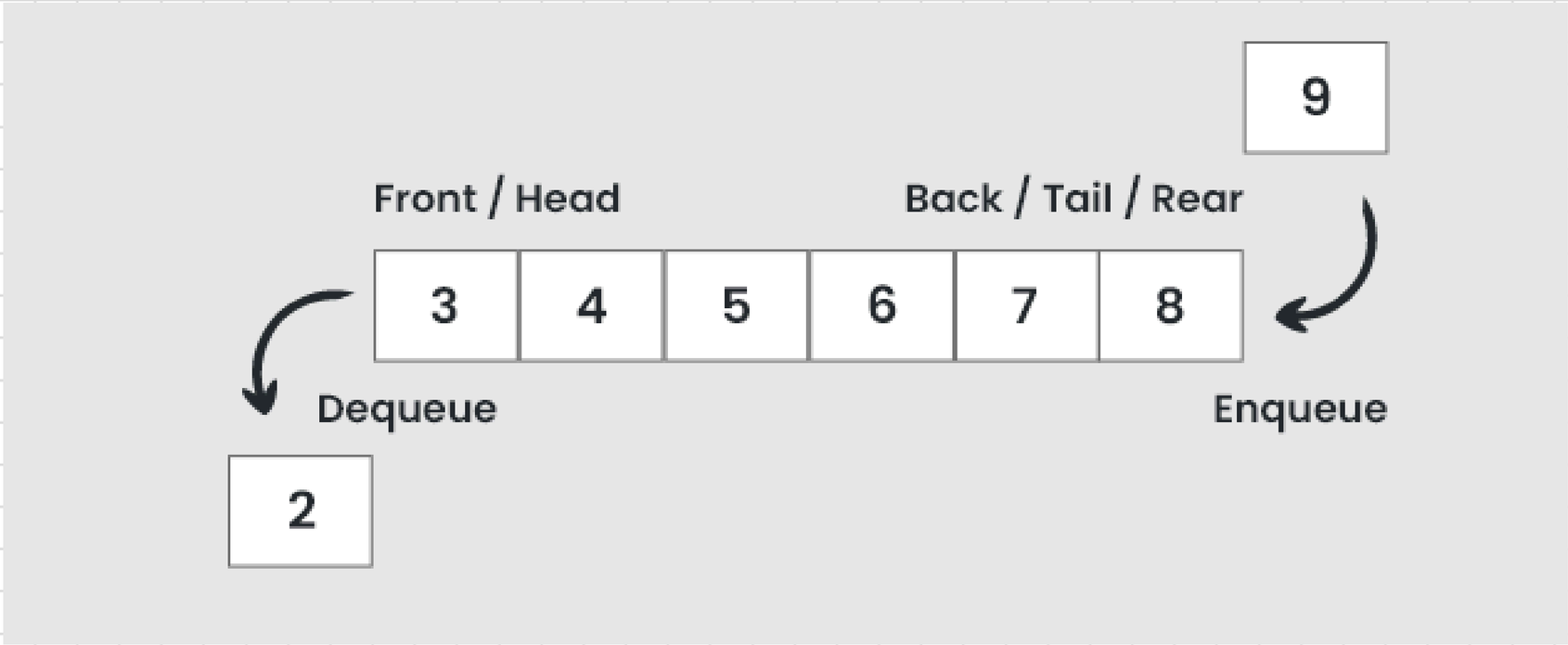
Efficient Function Call Management	Support for Recursion	Memory Isolation for Local Variables	Control Flow Maintenance
It handles function calls by storing return addresses, parameters, and local variables. The stack ensures proper execution order for nested or sequential calls.	Each recursive call gets its own stack frame, allowing multiple instances of the same function to run simultaneously and ensuring correct return order.	Local variables are isolated within stack frames, preventing interference between functions. Memory is automatically freed when functions return.	The stack tracks return addresses, maintaining correct execution flow during function calls and returns.

3 Illustrate, with an example, a concrete data structure for a First in First out (FIFO) queue.

3.1. Introduction FIFO

FIFO stands for First In, First Out. It is a method for organizing and managing data that is based on the principle that the item that is stored first is the item that is retrieved first.

3 Illustrate, with an example, a concrete data structure for a First in First out (FIFO) queue.



3.2. Define the Structure of (FIFO) Queue

A queue typically has two main operations:

- **Enqueue:** Add an element to the rear of the queue.
- **Dequeue:** Remove and return the element from the front of the queue.

Additionally, it often includes methods to check if the queue is empty and to view the front element without removing it.

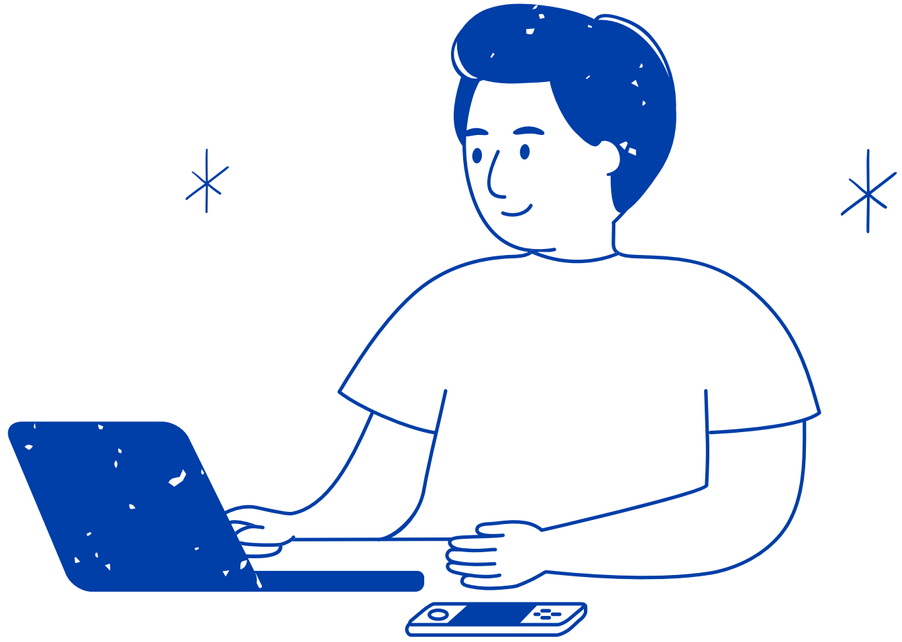
3.3. Array-based Queue and Linked List-based Queue

Array-based queues use an array to store queue elements, adding to the rear and removing from the front (enqueue and dequeue). They offer constant-time access to any element, but require a fixed size and may need costly resizing when full.

List-based queues utilize a linked list, where each node represents an element. The first node is the front and the last is the rear. They can grow dynamically without a fixed size, but accessing middle elements can be slower than in array-based queues.

3.3.1.

Array-based Queue _ Example Code



3.3.1. 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.

3.3.1. 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.

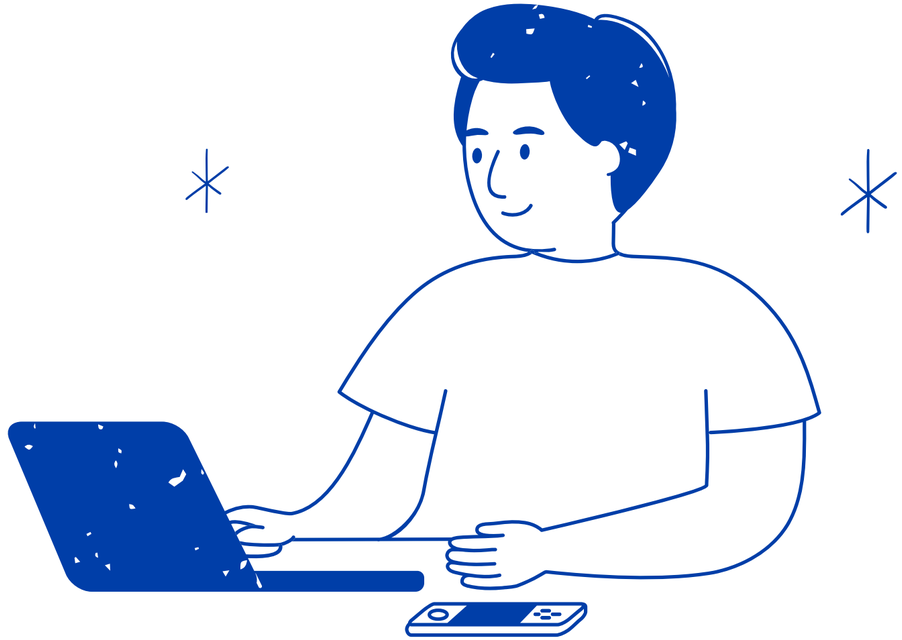
3.3.1. Array-based Queue _ Example Code

```
    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.

3.3.2.

Linked List-based Queue _ Example Code



3.3.2. 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.

3.3.2. 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.

3.3.2. 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.

3.3.2. 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.

4 Compare the performance of two sorting algorithms.

4.1 Introducing the two sorting algorithms you will be comparing

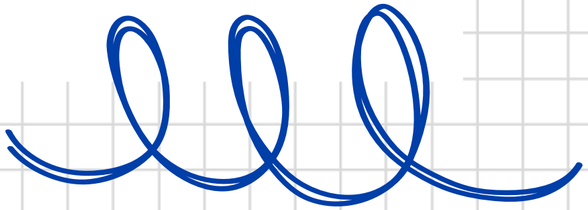
Bubble Sort

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

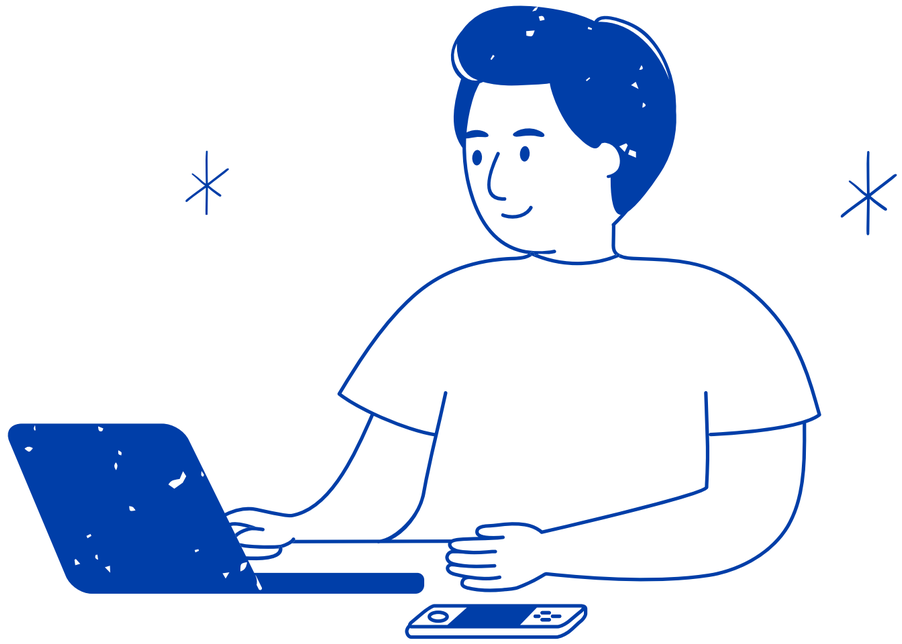
Merge sort is a sorting algorithm that follows the divide-and-conquer approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.

4.2 Time and Space Complexity Analysis



4.2.1

Bubble Sort



4.2.1. 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).

4.2.1. Bubble Sort

Time Complexity

Worst Case Time Complexity: O(N²)

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

4.2.1. 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

4.2.1. Bubble Sort

Time Complexity

Average Case Time Complexity Analysis of Bubble Sort: O(N2)

- ***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)*
  - *= N2 – (2 x N2 / 4)*
  - *= N2 – N2 / 2*
  - *= N2 / 2*
- ***Therefore, in average case the number of comparisons is O(N2)***

4.2.1. 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.

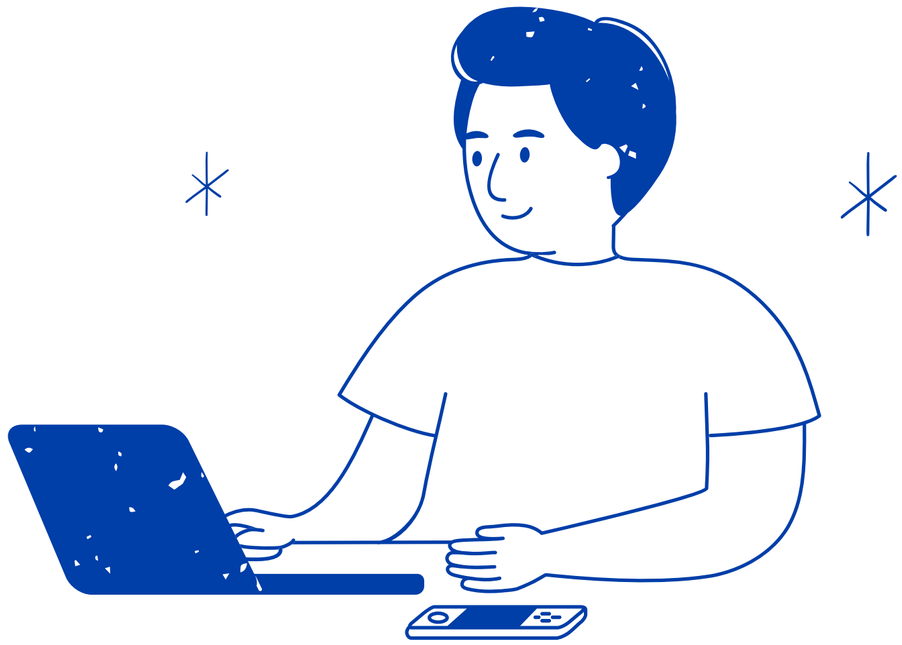
4.2.1. Bubble Sort

Stability

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

4.2.2

Merge Sort



4.2.2. 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).

4.2.2. 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).

4.2.2. 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).

4.2.2. 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.

4.2.2. Merge Sort

Stability

- Merge Sort also 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.

4.3. 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)
Space Complexity	O(1)	O(n)
Stability	Stable	Stable
In-Place Sorting	Yes	No
Practical Use	Small datasets	Large datasets, stable sorts

4.4 differences in performance between the two algorithms

4.4.1 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.

4.4.1 Bubble Sort

```java
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.

4.4.1 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 $O(n^2)$.

4.4.2 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.

4.4.2 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.

4.4.2 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 (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.

4.4.2 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.

4.4.2 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.

End