ASSIGNMENT 1 FRONT SHEET

Qualification	**BTEC Level 5 HND Diploma in Computing**		
Unit number and title	Unit 19: Data Structures and Algorithms		
Submission date	19/6/2023	**Date Received 1st submission**	
Re-submission Date		**Date Received 2nd submission**	
Student Name	Hoang Van Quyet	**Student ID**	BH00711
Class	IT0603	**Assessor name**	Vu An Tu

Student declaration

I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. I understand that making a false declaration is a form of malpractice.

Student's signature	Quyet

Grading grid

P1	P2	P3	M1	M2	M3	D1	D2

❐ Summative Feedback:		❐ Resubmission Feedback:
Grade:	Assessor Signature:	Date:
Internal Verifier's Comments:		
IV Signature:		

Table of content

I. Introduction 6

II. Body 7

P1 Create a design specification for data structures, explaining the valid operations that can be carried out on the structures 7

1. Definition of Data structure 7

1.1. Linear data structures 7

1.2. Nonlinear data structures 7

2. Defining Operations 7

3. Specifying Input Parameters 8

3.1. Linked lists 8

3.2. Arrays 9

3.3. Stacks 9

4. Defining Pre- and Post-conditions 10

4.1. Pre-conditions 10

4.2. Post-conditions 10

5. Time and Space Complexity 10

5.1. Time Complexity 10

5.2. Space Complexity 11

5.3. Difference Between Time Complexity and Space Complexity 11

6. Examples and Code Snippets 11

P2 Determine the operations of a memory stack and how it is used to implement function calls in a computer. 13

1. Definition of a Memory Stack 13

1.1. LIFO (Last In First Out) 13

2. Operations of a Memory Stack 14

2.1. Push Operation 14

2.2. Pop Operation 14

2.3. Peek Operation 14

3. Implementation of Function Calls using Memory Stack 14

4. Stack Frames and Their Components 14

M1 Illustrate, with an example, a concrete data structure for a First in First out (FIFO) queue. 16

1. First in First out (FIFO) Queue 16
1.1. Enqueue Operation in Queue Data Structure: 16
1.2. Dequeue Operation in Queue Data Structure 17
2. Define the Structure 17
3. Array-Based Implementation 17
4. Linked List-Based Implementation 18
5. Example to Illustrate FIFO Queue 20
M2 Compare the performance of two sorting algorithms. 22
1. Quick Sort 22
2. Selection sort 22
3. Time Complexity Analysis 23
4. Space Complexity Analysis 24
5. Stability 24
6. Comparison Table 24
7. Example 24
7.1. Quick Sort Execution 24
7.2. Selection Sort Execution 25
P3 Specify the abstract data type for a software stack using an imperative definition. 26
1. Definition of the Data Structure 26
2. Initialization of the Stack 26
3. Push operation 26
4. Pop operation 26
5. Peek operation 27
6. IsEmpty 27
7. Imperative Definition with Examples 27
III. Conclusion 29
IV. References 30

Table of figure

Figure 1: Memory Stack ____ 13
Figure 2: LIFO ____ 13
Figure 3: Queue data structure ____ 16
Figure 4: Quick sort ____ 22
Figure 5: Selection sort ____ 23

I. Introduction

Abstract Data Types (ADTs) are theoretical concepts in computer science that define data types by their behavior from the user's perspective, focusing on possible values, operations, and the behavior of these operations. Unlike concrete data structures, which are actual implementations (e.g., arrays, linked lists, stacks, queues), ADTs emphasize operations and mathematical principles describing their functionality. ADTs play a crucial role in software development by encapsulating data and operations, which enhances modularity and maintainability, as changes to the implementation do not affect the user of the ADT if the abstract interface remains unchanged. They promote code reusability by defining a clear and consistent interface, allowing developers to use the same ADT across different programs or projects without rewriting code or delving into implementation details. ADTs contribute to designing reliable and robust software by providing a formal basis for data types and their operations, ensuring expected behavior and reducing bugs. They also improve the design phase by allowing developers to focus on high-level system organization and data manipulation before implementation, leading to cleaner and more efficient code. Furthermore, ADTs facilitate testing and verification by enabling developers to create test cases based on abstract behavior, ensuring implementation adherence to specified operations and constraints. In summary, ADTs are fundamental to effective software development, providing a framework for data and operations, promoting encapsulation and reusability, enhancing robustness and reliability, and improving design, development, testing, and verification processes, essential for building efficient, maintainable, and scalable software systems.

II. Body

P1 Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

1. Definition of Data structure

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

   Data structures are an integral part of computers used for the arrangement of data in memory. They are essential and responsible for organizing, processing, accessing, and storing data efficiently. But this is not all. Various types of data structures have their characteristics, features, applications, advantages, and disadvantages.

   Data structures can be divided into several types, including linear and nonlinear structures. Within each type, subtypes suit different purposes and provide unique ways of arranging and linking data.

1.1. Linear data structures

- **Linked lists**: A linked list is a sequence of elements where each one contains reference information for the element next to it. With linked lists, you can efficiently insert and delete elements, easily adjusting the size of your list.
- **Arrays**: An array is a group of data elements arranged in adjacent memory locations. The index gives the direct address of each element, making arrays a highly efficient data structure for accessing different data pieces. Arrays are common across computer science functions as a convenient way to store accessible data.
- **Stacks**: Stack structures follow the last in, first out (LIFO) principle. The addition (push) and removal (pop) of elements happen only at one end, referred to as the top of the stack. For example, if you enter a new element, it appears on top. This element addition is a "push," as it pushes all the previous elements down. If you then remove the elements, this is a "pop." One of the most common uses of this algorithm is the "undo" function in text editors.
- **Queue**: A queue structure uses the first in, first out (FIFO) principle, meaning that you add elements from the back (enqueue) and remove them from the front (dequeue).

1.2. Nonlinear data structures

- **Graphs**: Graphs are fundamental types of nonlinear data structures. A graph is a collection of nodes connected by edges. It can represent networks, such as social networks or transportation systems. However, graphs are often chosen to represent networks of information, such as social media relationships or geographical maps.
- **Trees**: A tree is a graph structure with a hierarchy of nodes. The top node is the "root," and the descendants of this node are "children." Nodes with no children are "leaves." You can use trees in various applications, including hierarchical data representation, databases, and searching algorithms.

2. Defining Operations

Data structure operations are the methods used to manipulate the data in a data structure. The most common data structure operations are:

- **Insertion**: Insertion operations add new data elements to a data structure. You can do this at the data structure's beginning, middle, or end.
- **Deletion**: Deletion operations remove data elements from a data structure. These operations are typically performed on nodes that are no longer needed.
- **Search**: Search operations are used to find a specific data element in a data structure. These operations typically employ a compare function to determine if two data elements are equal.
- **Sort**: Sort operations are used to arrange the data elements in a data structure in a specific order. This can be done using various sorting algorithms, such as insertion sort, bubble sort, merge sort, and quick sort.
- **Merge**: Merge operations are used to combine two data structures into one. This operation is typically used when two data structures need to be combined into a single structure.
- **Copy**: Copy operations are used to create a duplicate of a data structure. This can be done by copying each element in the original data structure to the new one.

3. Specifying Input Parameters

3.1. Linked lists

- **Insert Operation**:
- Input Parameters: Data to be inserted.
- Usage Example: insert(data)
- Example: linked_list.insert(5)
- Explanation: The insert operation in a linked list typically requires the data to be inserted. This could be a single value or a complex data structure depending on the implementation. For example, inserting an integer value 5 into a linked list adds a new node containing 5 at the specified position.
- **Delete Operation**:
- Input Parameters: Key or data to be deleted.
- Usage Example: delete(key)
- Example: linked_list.delete(5)
- Explanation: Deleting an element from a linked list usually involves specifying the key or data of the node to be removed. In this case, 5 would indicate deleting the node containing the data 5 from the linked list.
- **Search Operation:**
- Input Parameters: Key or data to search for.
- Usage Example: search(key)
- Example: linked_list.search(5)
- Explanation: Searching in a linked list typically requires specifying the key or data to look for. For instance, search(5) would traverse the linked list to find the node containing the data 5 and return information about its position or existence.

3.2. Arrays

- **Insert Operation:**
  - Input Parameters: Index position and data to be inserted.
  - Usage Example: insert(index, data)
  - Example: array.insert(2, 10)
  - Explanation: Inserting into an array involves specifying both the index position where the data should be inserted and the actual data itself. For instance, inserting 10 at index 2 would shift existing elements and place 10 at the specified position.
- Delete Operation:
  - Input Parameters: Index position of the element to be deleted.
  - Usage Example: delete(index)
  - Example: array.delete(2)
  - Explanation: Deleting from an array typically requires specifying the index of the element to be removed. For example, delete(2) would remove the element at index 2 from the array and adjust the positions of subsequent elements accordingly.
- Search Operation:
  - Input Parameters: Key or data to search for.
  - Usage Example: search(key)
  - Example: array.search(10)
  - Explanation: Searching in an array involves specifying the key or data to look for. For instance, search(10) would iterate through the array to find the first occurrence of 10 and return its index or other relevant information.

3.3. Stacks

- **Insert Operation:**
  - Input Parameters: Index position and data to be inserted.
  - Usage Example: insert(index, data)
  - Example: array.insert(2, 10)
  - Explanation: Inserting into an array involves specifying both the index position where the data should be inserted and the actual data itself. For instance, inserting 10 at index 2 would shift existing elements and place 10 at the specified position.
- **Delete Operation:**
  - Input Parameters: Index position of the element to be deleted.
  - Usage Example: delete(index)
  - Example: array.delete(2)
  - Explanation: Deleting from an array typically requires specifying the index of the element to be removed. For example, delete(2) would remove the element at index 2 from the array and adjust the positions of subsequent elements accordingly.
- **Search Operation:**

- Input Parameters: Key or data to search for.
- Usage Example: search(key)
- Example: array.search(10)
- Explanation: Searching in an array involves specifying the key or data to look for. For instance, search(10) would iterate through the array to find the first occurrence of 10 and return its index or other relevant information.

4. Defining Pre- and Post-conditions

4.1. Pre-conditions

Pre-conditions are conditions that must be true before an operation on a data structure can be executed. They ensure that the operation can proceed without encountering errors or inconsistencies.

Here is Array Insertion Example:

- When inserting an element into an array, a common pre-condition is that the array must have sufficient capacity to accommodate the new element. If the array is full, additional memory allocation or resizing might be necessary.
- Before inserting an element at index i in an array arr of size n, the condition 0 <= i <= n should hold true to prevent out-of-bounds errors.

4.2. Post-conditions

Post-conditions specify what must be true after the completion of an operation on a data structure. They ensure that the operation has been executed correctly and that the data structure maintains its expected state.

Here is Array Insertion Example:

- After inserting an element into an array at a specific index, the array should reflect the addition of the new element at that position.
- After inserting an element element at index i in an array arr, the post-condition ensures that arr[i] == element.

5. Time and Space Complexity

5.1. Time Complexity

The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Note that the time to run is a function of the length of the input and not the actual execution time of the machine on which the algorithm is running on.

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. Time complexity is very useful measure in algorithm analysis.

It is the time needed for the completion of an algorithm. To estimate the time complexity, we need to consider the cost of each fundamental instruction and the number of times the instruction is executed.

5.2. Space Complexity

Problem-solving using computer requires memory to hold temporary data or final result while the program is in execution. The amount of memory required by the algorithm to solve given problem is called space complexity of the algorithm.

The space complexity of an algorithm quantifies the amount of space taken by an algorithm to run as a function of the length of the input. Consider an example: Suppose a problem to find the frequency of array elements.

5.3. Difference Between Time Complexity and Space Complexity

Aspect	Time Complexity	Space Complexity
Definition	Measures the execution time of an algorithm relative to input size.	Measures the memory usage of an algorithm relative to input size.
Focus	Amount of time an algorithm takes to complete.	Amount of memory (RAM) an algorithm requires for execution.
Primary Concern	Efficiency in terms of time.	Efficiency in terms of memory usage.
Analysis Approach	Number of operations or computational steps relative to input size.	Memory allocation for variables, data structures, and function calls relative to input size.
Input Size Impact	Larger inputs usually lead to longer execution times.	Larger inputs may require more memory for storing data structures or managing recursion.
Optimization Goal	Reducing the number of computational operations or steps.	Minimizing the amount of memory needed for data storage and processing.
Typical Example	Finding an element using binary search (O(log n) time complexity).	Creating a 2D array of size n x n (O(n^2) space complexity).

6. Examples and Code Snippets

Providing practical examples and code snippets can significantly enhance the understanding of data structures and their operations. For instance, an example of array insertion in Java could be:

```java
package asm;

public class ArrayExample {
    public static void main(String[] args) {
        int[] arr = new int[5]; // Initialize an array of size 5
        arr[0] = 1; // Inserting element 1 at index 0
        arr[1] = 2; // Inserting element 2 at index 1
        arr[2] = 3; // Inserting element 3 at index 2
        // Printing the array elements
        System.out.println("Array elements after insertion:");
        for (int i = 0; i < arr.length; i++) {
            System.out.println("Element at index " + i + ": " + arr[i]);
        }
    }
}
```

This Java program demonstrates how to insert elements into an array and then print all elements. The array arr is initialized with a size of 5, and elements are inserted at specific indices using assignment statements (arr[0] = 1, arr[1] = 2, etc.). Finally, it iterates through the array to print all elements.

P2 Determine the operations of a memory stack and how it is used to implement function calls in a computer.

1. Definition of a Memory Stack

Stack memory is a memory usage mechanism that allows the system memory to be used as temporary data storage that behaves as a first-in, last-out buffer. An example of a stack is illustrated on the below:

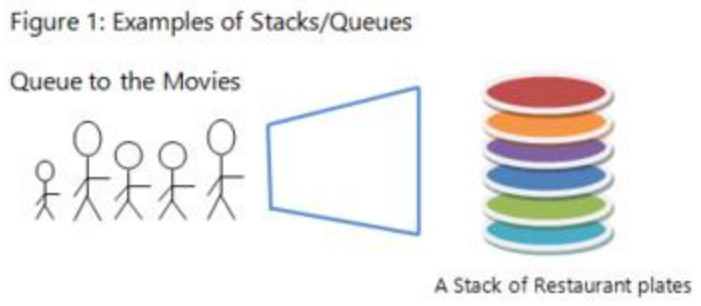
Figure 1: Examples of Stacks/Queues
Queue to the Movies
A Stack of Restaurant plates

Figure 1: Memory Stack

Items in a stack are inserted or removed in a linear order and not in any random sequence. As in any queue or collection that is assembled, the data items in a stack are stored and accessed in a specific way. In this case, a technique called LIFO (Last In First Out) is used. This involves a series of insertion and removal operations which we will discuss in the following section

1.1. LIFO (Last In First Out)

When a stack is accessed (there are inserted or deleted items), it is done in an orderly manner by LIFO. LIFO stands for Last In First Out. Computers use this method to request service data that the computer's memory receives. LIFO dictates that data is last inserted or stored in any particular stack, must be the first data to be deleted. If that applies to our queue for movies in Figure 1 above, there will be chaos! The operations on the stack are discussed in the following sections with image on the below:

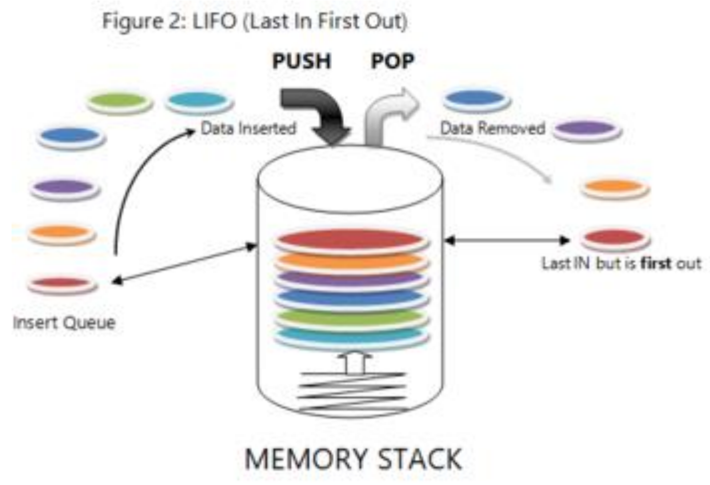

Figure 2: LIFO

2. Operations of a Memory Stack

2.1. Push Operation

- **Definition**: Adds a new element (often referred to as a stack frame or activation record) onto the top of the stack.
- **Functionality**: Allocates memory for the new stack frame and adjusts the stack pointer to reflect the new top position. This operation is essential when a function is called or when a new scope is entered, such as in block structures.

2.2. Pop Operation

- **Definition**: Removes the top element from the stack.
- **Functionality**: Deallocates the memory occupied by the top stack frame and updates the stack pointer to the next frame below. This operation occurs when a function returns or when a scope/block is exited.

2.3. Peek Operation

- **Definition**: Retrieves the value of the top element without removing it from the stack.
- **Functionality**: Allows inspection of the top stack frame's data, such as parameters, local variables, and return addresses. This operation is useful for debugging and for accessing context-specific information during runtime.

3. Implementation of Function Calls using Memory Stack

Function calls in programming languages utilize the memory stack to manage the execution flow and maintain local variables:

- **Function Call Sequence:**

- When a function is called, a new stack frame is pushed onto the stack to store its parameters, local variables, return address, and other necessary information.
- As the function executes, it accesses and modifies its stack frame's data.
- Upon completion, the function's stack frame is popped off the stack, returning control to the caller function or to the program's entry point.

4. Stack Frames and Their Components

Stack frames (or activation records) are structured components within the stack that hold crucial information during function execution:

- **Components of a Stack Frame:**

- Parameters: Values passed to the function.
- Local Variables: Variables declared within the function scope.
- Return Address: Address of the instruction to resume execution after the function completes.
- Saved Registers: Register values that need to be preserved across function calls.

- **Importance of Stack Frames:**

- Scope Management: Ensures each function call has its isolated workspace for parameters and local variables, preventing unintended interactions between different parts of the program.
- Execution Flow Control: Facilitates orderly function invocation and return, maintaining the correct sequence of operations within the program.
- Memory Efficiency: Efficiently allocates and deallocates memory for function contexts, optimizing resource usage and preventing memory leaks.

Example Scenario:

Consider a simple Java program demonstrating function calls and stack operations:

```java
package asm;

public class StackFrame {
    public static void main(String[] args) {
        int result = addNumbers(3, 5);
        System.out.println("Result: " + result);
    }

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

M1 Illustrate, with an example, a concrete data structure for a First in First out (FIFO) queue.

1. First in First out (FIFO) Queue

   A FIFO (First-In-First-Out) queue is a linear data structure where elements are inserted at the end (enqueue operation) and removed from the front (dequeue operation). It follows the principle that the first element added to the queue will be the first one to be removed, similar to how a queue operates in real-world scenarios.

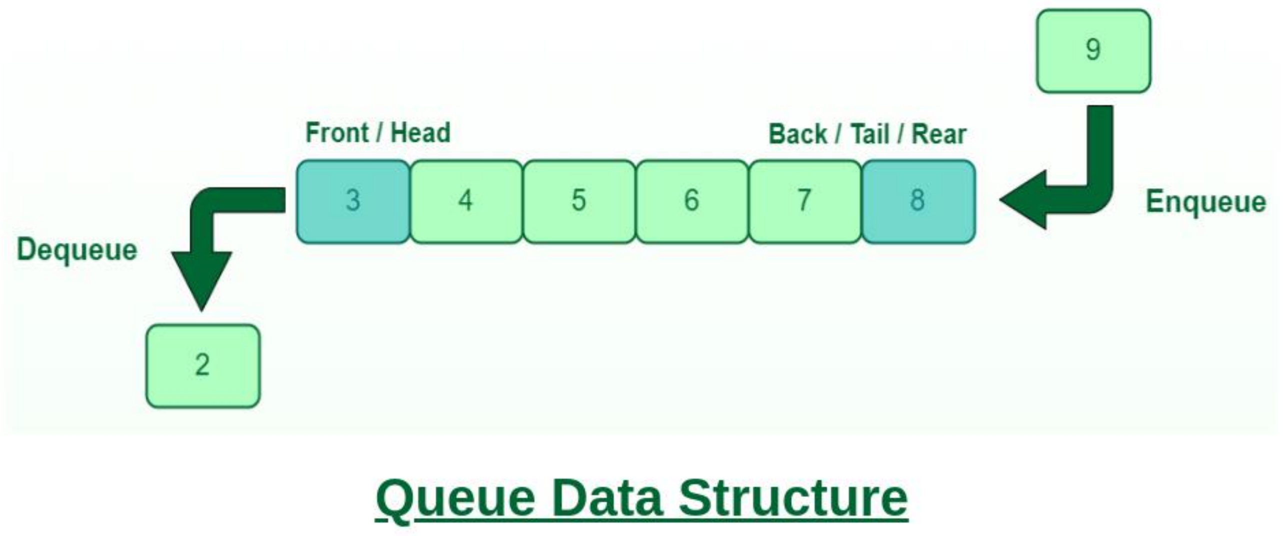

Figure 3: Queue data structure

FIFO Queue basic operations:

Linear operations may involve starting or defining a queue, using it, and then deleting it completely from memory. Here we will try to understand the basic operations of a line -

enqueue () - add (save) items to a queue.

dequeue () - removes (access) items from the queue.

1.1. Enqueue Operation in Queue Data Structure:

Enqueue() operation in Queue adds (or stores) an element to the end of the queue.

The following steps should be taken to enqueue (insert) data into a queue:

- Step 1: Check if the queue is full.
- Step 2: If the queue is full, return overflow error and exit.

- Step 3: If the queue is not full, increment the rear pointer to point to the next empty space.
- Step 4: Add the data element to the queue location, where the rear is pointing.
- Step 5: return success.

1.2. Dequeue Operation in Queue Data Structure

Removes (or access) the first element from the queue.

The following steps are taken to perform the dequeue operation:

- Step 1: Check if the queue is empty.
- Step 2: If the queue is empty, return the underflow error and exit.
- Step 3: If the queue is not empty, access the data where the front is pointing.
- Step 4: Increment the front pointer to point to the next available data element.
- Step 5: The Return success.

2. Define the Structure

   A FIFO queue is a linear data structure where elements are added at one end (rear) and removed from the other end (front). It follows the principle that the first element added to the queue will be the first one to be removed, similar to how a queue operates in real-world scenarios (e.g., a queue at a ticket counter).

3. Array-Based Implementation

   In an array-based implementation, the queue's elements are stored in a fixed-size array. Here's how it can be structured in Java:

```java
package asm;

public class ArrayQueue {
    private int[] queueArray;
    private int front;
    private int rear;
    private int capacity;

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

    // Method to check if the queue is empty
    public boolean isEmpty() {
        return rear == -1;
    }

    // Method to check if the queue is full
    public boolean isFull() {
        return rear == capacity - 1;
    }
    // Method to add an element to the rear of the queue
    public void enqueue(int item) {
        if (isFull()) {
            System.out.println("Queue is full. Cannot enqueue.");
            return;
        }
        queueArray[++rear] = item;
    }

    // Method to remove an element from the front of the queue
    public int dequeue() {
        if (isEmpty()) {
            System.out.println("Queue is empty. Cannot dequeue.");
            return -1; // Or throw an exception
        }
        int item = queueArray[front++];
        if (front > rear) { // Reset front and rear when queue becomes empty
            front = 0;
            rear = -1;
        }
        return item;
    }


    public int peek() {
        if (isEmpty()) {
            System.out.println("Queue is empty. Cannot peek.");
            return -1; // Or throw an exception
        }
        return queueArray[front];
    }
}
```

Explanation:

- ArrayQueue Class: Manages the queue using an array.
- Constructor (ArrayQueue(int size)): Initializes the queue with a specified size.
- isEmpty(): Checks if the queue is empty.
- isFull(): Checks if the queue is full.
- enqueue(int item): Adds an element to the rear of the queue.
- dequeue(): Removes an element from the front of the queue.
- peek(): Retrieves the element at the front without removing it.

4. Linked List-Based Implementation

   In a linked list-based implementation, each element in the queue is stored as a node in a linked list. Here’s a simplified Java implementation:

```
package asm;

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

public class LinkedListQueue {
    private Node front;
    private Node rear;


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

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

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

    public int dequeue() {
        if (isEmpty()) {
            System.out.println("Queue is empty. Cannot dequeue.");
            return -1;
        }
        int item = front.data;
        front = front.next;
        if (front == null) {
            rear = null;
        }
        return item;
    }
    public int peek() {
        if (isEmpty()) {
            System.out.println("Queue is empty. Cannot peek.");
            return -1;
        }
        return front.data;
    }
}
```

Explanation:

- Node Class: Represents each element (node) in the linked list.
- LinkedListQueue Class: Manages the queue using a linked list.
- Constructor (LinkedListQueue()): Initializes an empty queue.
- isEmpty(): Checks if the queue is empty.
- enqueue(int item): Adds an element to the rear of the queue by creating a new node.
- dequeue(): Removes an element from the front of the queue.
- peek(): Retrieves the element at the front without removing it.

5. Example to Illustrate FIFO Queue

   Let's consider an example scenario where we use the ArrayQueue class to manage a queue of integers:

```java
package asm;

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

        queue.enqueue(10);
        queue.enqueue(20);
        queue.enqueue(30);

        int removedElement = queue.dequeue();
        System.out.println("Removed Element: " + removedElement);


        int frontElement = queue.peek();
        System.out.println("Front Element: " + frontElement);


        queue.enqueue(40);
        queue.enqueue(50);


        while (!queue.isEmpty()) {
            int element = queue.dequeue();
            System.out.println("Removed Element: " + element);
        }
    }
}
```

And here is output:

```
Removed Element: 10
Front Element: 20
Removed Element: 20
Removed Element: 30
Removed Element: 40
Removed Element: 50
```

Explanation:

- The QueueExample class demonstrates various operations on the ArrayQueue implementation:
- Enqueue Operations: Adds elements (10, 20, 30) to the queue.
- Dequeue Operation: Removes the first element (10) from the queue.
- Peek Operation: Retrieves the front element (20) without removing it.
- Enqueue More Elements: Adds more elements (40, 50) to the queue.
- Dequeue Remaining Elements: Removes all elements (20, 30, 40, 50) from the queue in FIFO order.

M2 Compare the performance of two sorting algorithms.

1. Quick Sort

   Quick Sort is a highly efficient sorting algorithm that uses the divide-and-conquer approach. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

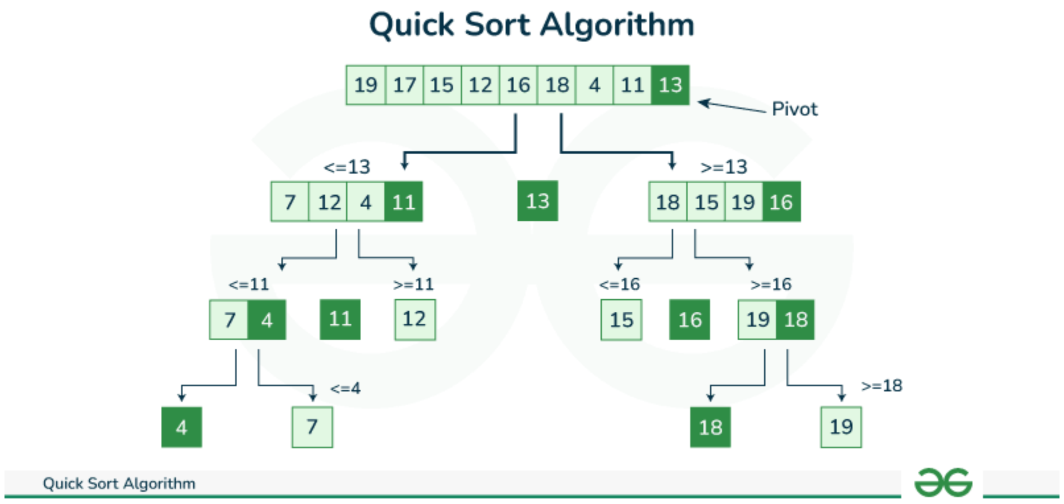

Figure 4: Quick sort

   The quick sort algorithm is performed using the following steps:

   - **Step 1**: Initialize two pointers, i and j, at the beginning and end of the array (excluding the pivot).
   - **Step 2**: Increment i until you find an element greater than or equal to the pivot.
   - **Step 3**: Decrement j until you find an element less than or equal to the pivot.
   - **Step 4**: If i is less than j, swap the elements at i and j.
   - **Step 5**: Repeat steps 2-4 until i and j cross each other.
   - **Step 6**: Swap the pivot element with the element at j. This places the pivot in its final position, dividing the array into two partitioned subarrays.
   - **Step 7**: Apply Quick Sort to the sub-arrays (Continue recursion until each sub-array becomes sorted)

2. Selection sort

   Selection Sort is a simple comparison-based sorting algorithm. It works by repeatedly selecting the smallest (or largest, depending on the order) element from the unsorted portion of the array and swapping it with the first unsorted element.

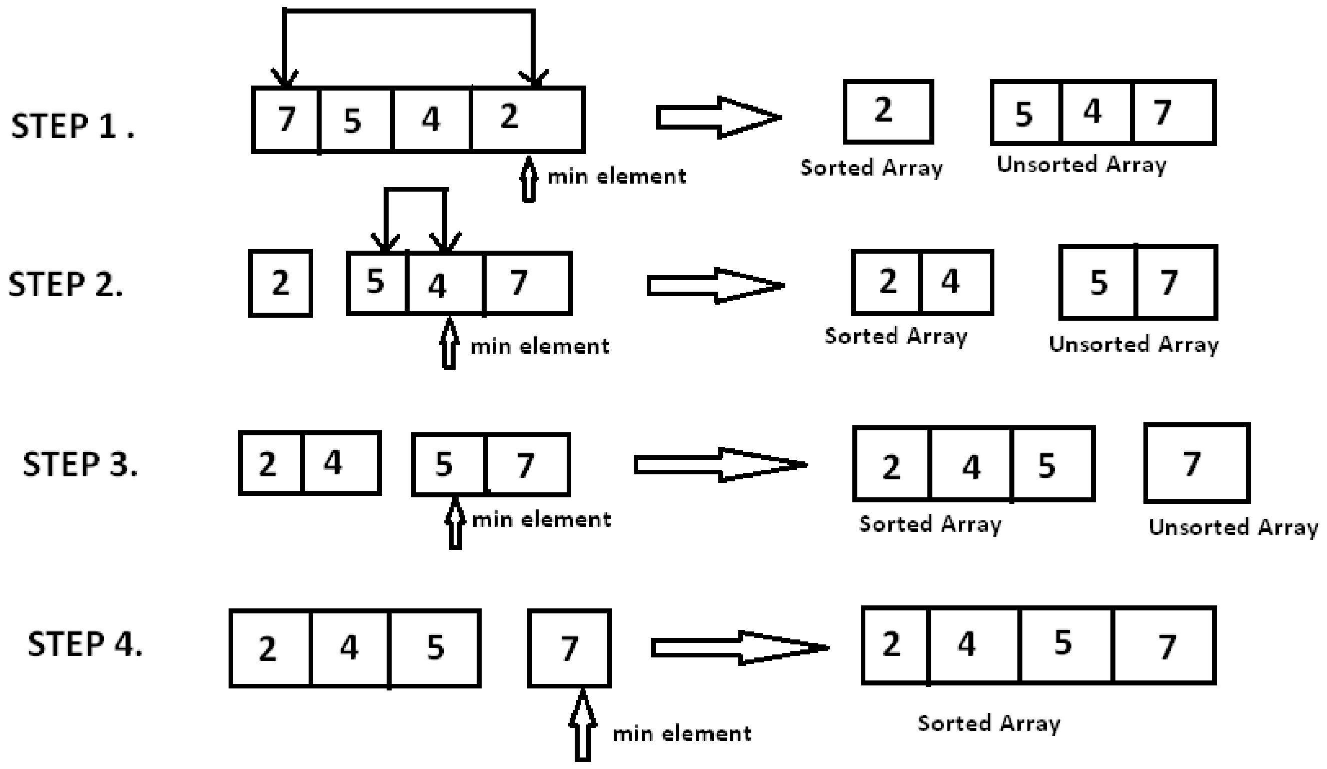

Figure 5: Selection sort

The selection sort algorithm is performed using the following steps:

- **Step 1**: Select the first element in the list (that is, Elements in the first position in the list).
- **Step 2**: Compare selected elements with all other elements in the list.
- **Step 3**: In each comparison, if any element is found to be smaller than the selected element (for Ascending order), then both are changed.
- **Step 4**: Repeat the same procedure with the element in the next position in the list until the whole list is sorted.

3. Time Complexity Analysis

Quick Sort Time Complexity:

- Best Case: O(n log n), occurs when the pivot splits the array into two nearly equal halves.
- Average Case: O(n log n), typical performance with random pivot selection.
- Worst Case: O(n^2), occurs when the smallest or largest element is always chosen as the pivot, resulting in unbalanced partitions (e.g., when the array is already sorted).

Selection Sort Time Complexity:

- Best Case: O(n^2), the algorithm always makes n^2 comparisons regardless of the input order.
- Average Case: O(n^2), as it consistently performs the same number of comparisons.

- Worst Case: O(n^2), similar to the best and average cases, as it doesn't depend on the initial order of the elements.\

4. Space Complexity Analysis

   Quick Sort Space Complexity:
   - In-place Version: O(log n) due to the stack space used by recursive calls. The in-place version requires constant space for the elements being swapped.
   - Non In-place Version: O(n) if additional arrays are used to store partitions, but typically the in-place version is preferred for efficiency.

   Selection Sort Space Complexity:
   - In-place: O(1) since it only requires a constant amount of additional memory for swapping elements.

5. Stability
   - **Quick Sort Stability**: Quick Sort is not inherently stable. Stability can be maintained by modifying the partitioning process, but this comes at the cost of additional complexity.
   - **Selection Sort Stability**: Selection Sort is not stable because swapping elements can change the relative order of equal elements.

6. Comparison Table

Algorithm	**Time Complexity (Best)**	**Time Complexity (Average)**	**Time Complexity (Worst)**	**Space Complexity**	**Stability**
Quick Sort	O(n log n)	O(n log n)	O(n^2)	O(log n)	Not Stable
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)	Not Stable

7. Example

   I have a example: Consider the array [64, 25, 12, 22, 11].

   7.1. Quick Sort Execution

   Initial Array: [64, 25, 12, 22, 11]

   First Partition:

   - Choose 11 as the pivot.
   - Partition the array into elements less than 11 and greater than or equal to 11.
   - Since 11 is the smallest element, the array after partitioning is: [11, 64, 25, 12, 22].
   - The pivot 11 is now in its correct position.

   Second Partition:
   - Focus on the sub-array [64, 25, 12, 22].
   - Choose 64 as the pivot.

- Partition into elements less than 64 and greater than or equal to 64.
- Array after partitioning: [11, 25, 12, 22, 64].
- The pivot 64 is now in its correct position.

Third Partition:

Focus on the sub-array [25, 12, 22].

- Choose 25 as the pivot.
- Partition into elements less than 25 and greater than or equal to 25.
- Array after partitioning: [11, 12, 22, 25, 64].
- The pivot 25 is now in its correct position.

Fourth Partition:

- Focus on the sub-array [12, 22].
- Choose 12 as the pivot.
- Partition into elements less than 12 and greater than or equal to 12.
- Array after partitioning: [11, 12, 22, 25, 64].
- The pivot 12 is now in its correct position.

The final sorted array is: [11, 12, 22, 25, 64].

7.2.Selection Sort Execution

Initial Array: [64, 25, 12, 22, 11]

First Iteration:

- Find the minimum element in the array [64, 25, 12, 22, 11], which is 11.
- Swap 11 with the first element 64.
- Array after first iteration: [11, 25, 12, 22, 64].

Second Iteration:

- Find the minimum element in the sub-array [25, 12, 22, 64], which is 12.
- Swap 12 with the second element 25.
- Array after second iteration: [11, 12, 25, 22, 64].

Third Iteration:

- Find the minimum element in the sub-array [25, 22, 64], which is 22.
- Swap 22 with the third element 25.
- Array after third iteration: [11, 12, 22, 25, 64].

Fourth Iteration:

- The sub-array [25, 64] is already sorted.
- No swap needed.
- Array after fourth iteration: [11, 12, 22, 25, 64].

The final sorted array is: [11, 12, 22, 25, 64].

P3 Specify the abstract data type for a software stack using an imperative definition.

1. Definition of the Data Structure

   A stack can be defined as a collection of elements with two primary operations:
   - Push: Adds an element to the top of the stack.
   - Pop: Removes the top element from the stack.

   Other auxiliary operations include:
   - Peek: Retrieves the top element without removing it.
   - IsEmpty: Checks if the stack is empty.

2. Initialization of the Stack

   To initialize a stack, we need to set up the data structure that will hold the elements. In an imperative programming language like Java, this typically involves creating an array or a linked list and a pointer or index to keep track of the top element.

```
package asm;

public class Stack {
    private int maxSize;
    private int[] stackArray;
    private int top;

    // Constructor
    public Stack(int size) {
        maxSize = size;
        stackArray = new int[maxSize];
        top = -1;
    }
}
```

3. Push operation

   The push operation adds an element to the top of the stack. It first checks if there is enough space to add the element, and if so, it increments the top pointer and adds the element at the new top position.

```
public void push(int value) {
    if (top == maxSize - 1) {
        System.out.println("Stack is full");
    } else {
        stackArray[++top] = value;
    }
```

   Steps:
   - Check if the stack is full.
   - If full, print an error message or handle overflow.
   - If not full, increment the top index.
   - Add the new element at the position indicated by top.

4. Pop operation

The pop operation removes the top element from the stack. It first checks if the stack is empty, and if not, it retrieves the element at the top position, decrements the top pointer, and returns the element.

```
public int pop() {
    if (top == -1) {
        System.out.println("Stack is empty");
        return -1;
    } else {
        return stackArray[top--];
    }
```

Steps:

- Check if the stack is empty.
- If empty, print an error message or handle underflow.
- If not empty, retrieve the element at the position indicated by top.
- Decrement the top index.

5. Peek operation

The peek operation retrieves the top element without removing it. It checks if the stack is empty, and if not, it returns the element at the top position.

```
public int peek() {
    if (top == -1) {
        System.out.println("Stack is empty");
        return -1;
    } else {
        return stackArray[top];
    }
}
```

Steps:

- Check if the stack is empty.
- If empty, print an error message or handle underflow.
- If not empty, return the element at the position indicated by top.

6. IsEmpty

The isEmpty operation checks if the stack is empty by verifying if the top pointer is at its initial position.

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

Steps:

- Return true if top is -1, indicating the stack is empty.
- Return false otherwise.

7. Imperative Definition with Examples

Let's consider an example where we use the stack to perform a series of operations.

```
package asm;

public class StackExample {
    public static void main(String[] args) {
```

```java
        Stack stack = new Stack(5);

        stack.push(10);
        stack.push(20);
        stack.push(30);
        System.out.println("Top element: " + stack.peek());
        System.out.println("Removed element: " + stack.pop());
        System.out.println("Removed element: " + stack.pop());
        System.out.println("Is stack empty? " + stack.isEmpty());
        stack.pop();
        System.out.println("Is stack empty? " + stack.isEmpty());
        stack.pop();
    }
}
```

III. Conclusion

In this analysis, we have explored the significance of Abstract Data Types (ADTs) in software development, focusing on their role in encapsulating data and operations, promoting code reusability, and enhancing software robustness and reliability. We have demonstrated how ADTs provide a high-level framework for designing and managing data structures, allowing developers to focus on abstract behavior without getting bogged down by implementation details. By leveraging ADTs, developers can create more maintainable and modular code, which is easier to test and verify, ensuring that software systems behave as expected.

In our case study of the Student Management System, we illustrated how ADTs can be applied to solve real-world problems effectively. Through the use of various operations such as adding, editing, deleting, sorting, and searching for student records, we showcased how ADTs facilitate efficient data management. Furthermore, we compared linear search and binary search algorithms, highlighting the advantages of selecting the appropriate algorithm for specific use cases to improve performance.

In conclusion, ADTs are indispensable tools in the software development process. They provide the necessary abstraction to manage complexity, improve code quality, and ensure that software systems are both reliable and scalable. By incorporating ADTs into the design and development phases, developers can build robust applications that meet user needs and stand the test of time. As we continue to advance in the field of computer science, the principles and practices surrounding ADTs will remain foundational to the creation of efficient and effective software solutions.

IV. References

Link github: https://github.com/quyet11/DSA

Coursera. (2024). Types of Data Structures. [online] Available at: https://www.coursera.org/articles/types-of-data-structures [Accessed 18 Jun. 2024].

GeeksforGeeks. (2022). What is Data Structure: Types, Classifications and Applications. [online] Available at: https://www.geeksforgeeks.org/what-is-data-structure-types-classifications-and-applications/.

GeeksforGeeks. (2020). Common operations on various Data Structures. [online] Available at: https://www.geeksforgeeks.org/common-operations-on-various-data-structures/.

adservio.fr. (n.d.). Data Structure | Types | Operations. [online] Available at: https://www.adservio.fr/post/data-structure-types-operations#el6.

GeeksforGeeks (2021). Time Complexity and Space Complexity. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/time-complexity-and-space-complexity/.

Gupta, E. (2024). Difference Between Time Complexity and Space Complexity. [online] Shiksha.com. Available at: https://www.shiksha.com/online-courses/articles/difference-between-time-complexity-and-space-complexity-blogId-151433.

GeeksforGeeks. (2022). Introduction to Queue - Data Structure and Algorithm Tutorials. [online] Available at: https://www.geeksforgeeks.org/introduction-to-queue-data-structure-and-algorithm-tutorials/.

Shiksha Online (2022). QuickSort Algorithm (With Code) - Shiksha Online. [online] Shiksha.com. Available at: https://www.shiksha.com/online-courses/articles/quicksort-algorithm-with-code/#:~:text=Quick%20Sort%20Algorithm [Accessed 18 Jun. 2024].