ASSIGNMENT FINALFRONT SHEET

Qualification	**Pearson BTEC Level 5 Higher National Diploma in Computing**		
Unit number and title	**Unit 19: Data Structures and Algorithms**		
Submission date	4/12/2024	**Date Received 1st submission**	
Re-submission Date		**Date Received 2nd submission**	
Student Name	Tran Gia Bao	**Student ID**	BH00998
Class	SE06302	**Assessor name**	Dinh Van Dong

Plagiarism

Plagiarism is a particular form of cheating. Plagiarism must be avoided at all costs and students who break the rules, however innocently, may be penalised. It is your responsibility to ensure that you understand correct referencing practices. As a university level student, you are expected to use appropriate references throughout and keep carefully detailed notes of all your sources of materials for material you have used in your work, including any material downloaded from the Internet. Please consult the relevant unit lecturer or your course tutor if you need any further advice.

Student Declaration

I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. I declare that the work submitted for assessment has been carried out without assistance other than that which is acceptable according to the rules of the specification. I certify I have clearly referenced any sources and any artificial intelligence (AI) tools used in the work. I understand that making a false declaration is a form of malpractice.

	Student's signature	Bao

Grading grid

P1	P2	P3	P4	P5	P6	P7	M1	M2	M3	M4	M5	D1	D2	D3	D4

❒ Summative Feedback:	❒ Resubmission Feedback:	
Grade:	**Assessor Signature:**	**Date:**
Internal Verifier's Comments:		
Signature & Date:		

Table of Contents

Introduction

In this assignment, there are several important tasks that I need to complete to gain a deeper understanding of the basic concepts in data structures and related algorithms. First, I will create a design specification for the data structure, including the valid operations that we can perform with it. Clearly defining the functions of a data structure is essential to ensure that it can serve its purpose in different applications.

Next, I will present the operations of memory stacks and how they are applied in computers to handle function calls. Stacks are a LIFO (Last In First Out) data structure, and it is commonly used to manage memory in programs. When a function is called, the parameters and return address are pushed onto the stack, and when the function ends, they are popped off in reverse order.

In addition, I will describe an abstract data type (ADT) for a software stack using an imperative description. This abstract data type will help us visualize how the stack works without worrying about the detailed implementation. This will allow me to provide concrete examples to introduce the concept of a FIFO (First In First Out) queue data structure. Queues are a very useful data structure in many situations, such as task management in multitasking systems.

To dive deeper into the algorithmic realm, I will compare and discuss the performance of two popular sorting algorithms: bubble sort and quick sort. Analyzing the time and space complexity of each algorithm will help me better understand their strengths and weaknesses in different contexts.

Finally, I will discuss the benefits of encapsulation and data hiding using abstract data types (ADTs). Encapsulation allows us to hide implementation details and provide only a clear interface to the user, which not only reduces errors but also makes the code more maintainable and extensible. In this way, I can emphasize the importance of designing and implementing efficient data structures in programming and software development.

I. Data Structures and Complexity

1.1. Identify the Data Structures

A data structure is a specialized format for organizing, processing, and storing data in computer memory. It allows for efficient access and manipulation of data, allowing programmers to perform specific tasks more efficiently. Data structures serve as the foundation for building algorithms that solve complex problems, and they play a major role in shaping the performance and functionality of software.

In computer programming, data structures handle different types of data, such as integers, floats, characters, and strings, and organize them based on the needs of the application. Properly designed data structures can significantly improve the efficiency of a software solution, affecting how quickly and easily it can process and store large amounts of data.

Why Data Structures Are Important in Software Development

Data structures play an important role in software development for several reasons:

Efficiency: A properly chosen data structure optimizes data access and processing, making software faster and more efficient. The right data structure can significantly reduce the time complexity of common operations, such as searching, inserting, and deleting.

Scalability: As applications grow and handle larger amounts of data, efficient data structures become more important. Well-designed data structures can support application expansion without severely reducing performance, ensuring that the software remains responsive and stable.

Code maintenance: An application with an organized data structure is easier to maintain, modify, and extend. Choosing the right data structure simplifies code complexity and promotes better coding practices, thereby improving the quality of the software.

Algorithm Design: Since most algorithms are built around one or more data structures, their efficiency largely depends on the underlying structures. Appropriate data structures allow for better algorithm implementations and can significantly impact the performance of the software.

A thorough understanding of data structures is essential for developing efficient and scalable software. They allow developers to solve problems more efficiently, creating higher quality software applications.

1.2. Define the Operations

Operations on data structures are basic operations that help users manipulate, search, and manage data effectively. Each data structure has a set of specific operations, serving a specific purpose. Below are common operations on different data structures:

Operations on data structures are basic operations that help users manipulate, search and manage data effectively. Each data structure has a specific set of operations, serving a specific purpose.

The operations include:

- **addStudent(Student):** Add a new student to the stack.
- **updateStudent(String id, String newName, double newMarks):** Find a student by ID and update the name or marks.
- **deleteStudent(String id):** Delete a student with a specific ID from the stack.
- **SortByMark():** Sort the members in the stack in ascending order.
- **searchStudentByName(String name):** Search for a student by name.
- **displayStudents():** Display all the students currently in the stack.

1.3. Specify Input Parameters

Input parameters are the data, values that the user needs to provide for operations on the data structure. They help to clearly define what is needed to perform different operations. Here are the input parameters for each common data structure:

For the main operations:

- addStudent: Gets a Student object (containing id, name, and marks).
- updateStudent: Gets id (String), newName (String), and newMarks (double).
- deleteStudent: Gets id (String).
- searchStudentByName: Gets name (String).

User interface (main method): Input from the user via keyboard: ID, name, and score.

These parameters provide the necessary information for the operations, helping the operations on the data structure work correctly and efficiently. Determining the specific input parameters for each operation is an important step to ensure that the operation is performed optimally.

1.4. Define Pre- and Post-conditions

Pre-conditions and post-conditions are requirements that must be met so that an operation on a data structure can be performed correctly and maintain the integrity of the data structure after the operation is completed.

Below are the preconditions and postconditions of a program.

1. addStudent(Student student)

Precondition:

- The student object must not be null.
- The id attribute of the student must be unique and not duplicate any existing IDs in the stack.

Postcondition:

- The stack has a new student with the provided ID, name, and marks.
- The stack size increases by 1.

2. updateStudent(String id, String newName, double newMarks)

Precondition:

- The stack must not be empty.
- id must exist in the stack.
- newName and newMarks must be valid:
- newName must not be null or empty.
- newMarks must be a positive number.

Postcondition:

- The student information with the corresponding ID is updated to newName and newMarks.
- If the ID does not exist, the stack is not changed, and an error message is printed.

3. deleteStudent(String id)

Precondition:

- The stack must not be empty.
- id must exist in the stack.

Postcondition:

- The student with the corresponding ID is deleted from the stack.
- If the ID does not exist, the stack is not changed, and an error message is printed.
- The size of the stack is decreased by 1 if the deletion is successful.

4. sortByMark()

Precondition:

- The stack cannot be empty.

Postcondition:

- The stack is sorted in ascending order of marks (based on the marks value).
- The old order of the elements is changed.

5. searchStudentByName(String name)

Precondition:

- The stack cannot be empty.
- name cannot be null or empty.

Postcondition:

- If a student with a matching name is found, the student's information is displayed.
- If no student is found, a message stating that there are no matching students is printed.
- The stack is not changed.

6. displayStudents()

Precondition:

- No precondition, works even if the stack is empty.

Postcondition:

- The list of all students in the stack is printed to the screen.
- The stack is not changed.

1.5. Discuss Time and Space Complexity

The time and space complexity of a data structure is a measure of the efficiency of operations involved in performing operations like searching, adding, deleting, etc. Depending on the data structure, the complexity may vary. Following is an analysis of the time and space complexity of a program

1. addStudent(Student student)

Time Complexity:$O(1)$ Adding an element to the stack only requires accessing the top and adding the element.Space Complexity:$O(1)$ Does not use additional memory other than the added Student object.

2. updateStudent(String id, String newName, double newMarks)

Time Complexity:$O(n)$ Iterates the entire stack to find the student with the matching ID (where n is the number of elements in the stack).Space Complexity:$O(n)$ Uses a temporary stack to store elements during the search.

3. deleteStudent(String id)

Time Complexity:$O(n)$ Iterates through the stack to remove students with matching IDs.Space Complexity:$O(n)$ Uses a temporary stack to store the remaining elements.

4. sortByMark()

Time Complexity:$O(n2)$ Insertion Sort through two loops.Space Complexity:$O(n)$O(n): Uses a temporary stack to perform the sort.

5. searchStudentByName(String name)

Time Complexity:$O(n)$ Iterates through the stack to find students with matching names.Space Complexity:$O(1)$ Does not use additional memory.

6. displayStudents()

Time Complexity:$O(n)$ Iterates through the entire stack to display information for each student. Space Complexity:$O(1)$ Does not use additional memory.

1.6. Provide Examples and Code Snippets (if applicable)

Class Main

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

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        StudentManagement studentManagement = new StudentManagement();
        while (true) {
            System.out.println("-----------Menu-----------");
            System.out.println("1. Add Student");
            System.out.println("2. Update Student");
            System.out.println("3. Delete Student");
            System.out.println("4. Sort Students by Score");
            System.out.println("5. Search Student by Name");
            System.out.println("6. Display All Students");
            System.out.println("7. Exit");
            System.out.println("--------------------------");
            System.out.print ("Enter your selection: ");
            int choice = scanner.nextInt();
            scanner.nextLine();
            switch (choice) {
                case 1:
                    System.out.print("Enter student ID: ");
                    int id = scanner.nextInt();
                    scanner.nextLine(); // Clear newline character
                    if (studentManagement.isIdDuplicate(id)) {
                        System.out.println("ID already exists. Please enter a unique
ID.");
                        break; // Do not add student if ID is duplicated
                    }
                    String name;
                    while (true) {
                        System.out.print("Enter student name: ");
                        name = scanner.nextLine();
                        if (name.matches("[a-zA-Z\\s]+")) {
                            break;
                        } else {
                            System.out.println("Invalid name. Please enter only
letters and spaces.");
                        }
                    }
                    System.out.print("Enter student scores: ");
                    double score = scanner.nextDouble();
                    scanner.nextLine(); // Clear newline character
                    studentManagement.addStudent(new Student(id, name, score));
                    System.out.println("Student added successfully.");
                    break;
                case 2:
                    System.out.print("Enter the ID of the student to update: ");
                    int updateId = scanner.nextInt();
                    scanner.nextLine();
                    System.out.print("Enter new name: ");
```

```
                    String newName = scanner.nextLine();
                    System.out.print("Enter new score: ");
                    double newMark = scanner.nextDouble();
                    scanner.nextLine();
                    studentManagement.updateStudent(updateId, newName, newMark);
                    System.out.println("Information updated successfully");
                    break;
                case 3:
                    System.out.print("Enter the ID of the student to delete: ");
                    int deleteId = scanner.nextInt();
                    scanner.nextLine();
                    studentManagement.deleteStudent(deleteId);
                    System.out.println("Delete information successfully");
                    break;
                case 4:
                    studentManagement.sortByScore();
                    break;
                case 5:
                    System.out.print("Enter the name of the student to search: ");
                    String searchName = scanner.nextLine();
                    studentManagement.searchStudentByName(searchName);
                    break;
                case 6:
                    studentManagement.displayStudents();
                    break;
                case 7:
                    System.out.println("See you again.");
                    scanner.close();
                    System.exit(0);
                    break;
                default:
                    System.out.println("Invalid choice. Please try again.");
            }
        }
    }
}
```

- **Class StudentManagement**

```
package org.example;
import java.util.Comparator;
import java.util.Stack;

public class StudentManagement {
    private StudentStack studentStack;

    public StudentManagement() {
        studentStack = new StudentStack();
    }
    public boolean isIdDuplicate(int id) {
        StudentStack.Node current = studentStack.getTop();
        while (current != null) {
            if (current.student.getId() == id) {
```

```
                return true; // ID already exists
            }
            current = current.next;
        }
        return false; // ID is not duplicated
    }

    // Add student to stack
    public void addStudent(Student student) {
        studentStack.push(student);
        displayStudents();
    }
    // Update student information by ID
    public void updateStudent(int id, String newName, double newMarks) {
        StudentStack tempStack = new StudentStack();
        boolean found = false;
        while (!studentStack.isEmpty()) {
            Student student = studentStack.pop();
            if (student.getId() == id) {
                tempStack.push(new Student(id, newName, newMarks));
                found = true;
            } else {
                tempStack.push(student);
            }
        }
        while (!tempStack.isEmpty()) {
            studentStack.push(tempStack.pop());
        }

        if (found) {
            System.out.println("Student with ID " + id + " updated successfully.");
        } else {
            System.out.println("No student found with ID: " + id);
        }
        displayStudents();
    }

    // Delete student by ID
    public void deleteStudent(int id) {
        StudentStack tempStack = new StudentStack();

        while (!studentStack.isEmpty()) {
            Student student = studentStack.pop();
            if (student.getId() != id) {
                tempStack.push(student);
            }
        }
        while (!tempStack.isEmpty()) {
            studentStack.push(tempStack.pop());
        }
        System.out.println("Deleted student with ID: " + id);
        displayStudents();
    }
    // Sort students by score
    public void sortByScore() {
        if (studentStack.isEmpty()) {
            System.out.println("Stack is empty. No students to sort.");
```

```
            return;
        }

        StudentStack sortedStack = new StudentStack();
        while (!studentStack.isEmpty()) {
            Student temp = studentStack.pop();

            while (!sortedStack.isEmpty() && sortedStack.peek().getMarks() <
                    temp.getMarks()) {
                studentStack.push(sortedStack.pop());
            }
            sortedStack.push(temp);
        }

        while (!sortedStack.isEmpty()) {
            studentStack.push(sortedStack.pop());
        }
        System.out.println("Students sorted by score.");
        displayStudents();
    }
    // Find students by name
    public void searchStudentByName(String name) {
        StudentStack.Node current = studentStack.getTop();
        boolean found = false;
        while (current != null) {
            if (current.student.getName().equalsIgnoreCase(name)) {
                System.out.println("Found students: " + current.student);
                found = true;
            }
            current = current.next;
        }
        if (!found) {
            System.out.println("No student found with name: " + name);
        }
    }
    // Show all students
    public void displayStudents() {
        if (studentStack.isEmpty()) {
            System.out.println("There are no students.");
            return;
        }
        studentStack.displayStudents();
    }
}
```

Class Student

```
package org.example;
class Student
{
    private int id;
    private String name;
    private double score;

    public Student(int id, String name, double score) {
        this.id = id;
        this.name = name;
        this.score = score;
    }
    public void setMarks(double score) {
        this.score = score;
    }
    public String getRanking() {
        if (score < 5.0) {
            return "Fail";
        } else if (score < 6.5) {
            return "Medium";
        } else if (score < 7.5) {
            return "Good";
        } else if (score < 9.0) {
            return "Very Good";
        } else {
            return "Excellent";
        }
    }
    public Student() {
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public double getMarks() {
        return score;
    }
    @Override
    public String toString() {
        return "Student {" + "id = " + id + ", name = " + name + ", score = " + score
+
                ", Ranking = " + getRanking()+ '}';
    }
}
```

Class StudentStack

```java
package org.example;
public class StudentStack {
    Node top;
    // Node in stack
    static class Node {
        Student student;
        Node next;
        Node(Student student) {
            this.student = student;
            this.next = null;
        }
    }
    // Add student to stack
    public void push(Student student) {
        Node newNode = new Node(student);
        newNode.next = top;
        top = newNode;
    }
    // Get student from stack
    public Student pop() {
        if (isEmpty()) {
            throw new IllegalStateException("Stack is empty.");
        }
        Student student = top.student;
        top = top.next;
        return student;
    }
    // Check if stack is empty
    public boolean isEmpty() {
        return top == null;
    }
    // View top student
    public Student peek() {
        if (isEmpty()) {
            throw new IllegalStateException("Stack is empty.");
        }
        return top.student;
    }
    // Show all students
    public void displayStudents() {
        Node current = top;
        while (current != null) {
            System.out.println(current.student);
            current = current.next;
        }
    }
    public Node getTop() {
        return top;
    }
}
```

Results after running

```
-----------Menu-----------
1. Add Student
2. Update Student
3. Delete Student
4. Sort Students by Score
5. Search Student by Name
6. Display All Students
7. Exit
--------------------------
Enter your selection:
```

Add Student

```
Enter your selection: 1
Enter student ID: 1
Enter student name: Tran Gia Bao
Enter student scores: 9
Student {id = 1, name = Tran Gia Bao, score = 9.0, Ranking = Excellent}
Student added successfully.
```

```
Enter your selection: 1
Enter student ID: 2
Enter student name: Le Van Nam
Enter student scores: 7
Student {id = 2, name = Le Van Nam, score = 7.0, Ranking = Good}
Student {id = 1, name = Tran Gia Bao, score = 9.0, Ranking = Excellent}
Student added successfully.
```

```
Enter your selection: 1
Enter student ID: 3
Enter student name: Phan Nhat Linh
Enter student scores: 8
Student {id = 3, name = Phan Nhat Linh, score = 8.0, Ranking = Very Good}
Student {id = 2, name = Le Van Nam, score = 7.0, Ranking = Good}
Student {id = 1, name = Tran Gia Bao, score = 9.0, Ranking = Excellent}
Student added successfully.
```

```
Enter your selection: 1
Enter student ID: 4
Enter student name: Phan Bao Trung
Enter student scores: 4
Student {id = 4, name = Phan Bao Trung, score = 4.0, Ranking = Fail}
Student {id = 3, name = Phan Nhat Linh, score = 8.0, Ranking = Very Good}
Student {id = 2, name = Le Van Nam, score = 7.0, Ranking = Good}
Student {id = 1, name = Tran Gia Bao, score = 9.0, Ranking = Excellent}
Student added successfully.
```

Update Student

```
Enter your selection: 2
Enter the ID of the student to update: 4
Enter new name: Phan Bao Trung
Enter new score: 6
Student with ID 4 updated successfully.
Student {id = 4, name = Phan Bao Trung, score = 6.0, Ranking = Medium}
Student {id = 3, name = Phan Nhat Linh, score = 8.0, Ranking = Very Good}
Student {id = 2, name = Le Van Nam, score = 7.0, Ranking = Good}
Student {id = 1, name = Tran Gia Bao, score = 9.0, Ranking = Excellent}
Information updated successfully
```

Delete Student

```
Enter your selection: 3
Enter the ID of the student to delete: 4
Deleted student with ID: 4
Student {id = 3, name = Phan Nhat Linh, score = 8.0, Ranking = Very Good}
Student {id = 2, name = Le Van Nam, score = 7.0, Ranking = Good}
Student {id = 1, name = Tran Gia Bao, score = 9.0, Ranking = Excellent}
Delete information successfully
```

Sort Students by Score

```
Enter your selection: 4
Students sorted by score.
Student {id = 1, name = Tran Gia Bao, score = 9.0, Ranking = Excellent}
Student {id = 3, name = Phan Nhat Linh, score = 8.0, Ranking = Very Good}
Student {id = 2, name = Le Van Nam, score = 7.0, Ranking = Good}
```

Search Student by Name

```
Enter your selection: 5
Enter the name of the student to search: Phan Nhat Linh
Found students: Student {id = 3, name = Phan Nhat Linh, score = 8.0, Ranking = Very Good}
```

Display all Student

```
Enter your selection: 6
Student {id = 1, name = Tran Gia Bao, score = 9.0, Ranking = Excellent}
Student {id = 3, name = Phan Nhat Linh, score = 8.0, Ranking = Very Good}
Student {id = 2, name = Le Van Nam, score = 7.0, Ranking = Good}
```

II. Memory Stack

Stack memory is a special area of memory in the system that temporarily stores local variables, function parameters, and stack frames during program execution. Stack memory operates on the "LIFO" (Last In, First Out) principle, in which elements are placed at the end of the stack and removed from the end.

2.1. Define a Memory Stack

Stack Memory is a memory area in a computer system that is used to temporarily store data, especially during function calls. The stack operates on the LIFO (Last In, First Out) principle, meaning that the last element added will be the first element taken out. Here are some of the main characteristics and components of stack memory:

1. Structure

Stack Frame: Every time a function is called, a new stack frame is created on the top of the stack. This frame contains:

Return address: The location in the code where the program will continue execution after the function ends.

Function parameters: The values of the parameters that the function takes in.

Local variables: Storage space for variables that exist only within the scope of the function.

Stack Top: The top of the stack is the current location where new data is added to or removed from the stack.

2. Operation

Push: The operation of adding a new stack frame to the top. When a function is called, the necessary information (return address, parameters, local variables) will be "pushed" onto the stack.

Pop: The operation of removing the stack frame at the top. When the function ends, the stack frame will be removed, and the return address will be used to return to the calling function.

Peek: View information at the top of the stack without removing it (this operation is not common in the context of functions).

3. Characteristics

Limited capacity: Stack memory has a fixed size, which can lead to "stack overflow" errors if there are too many nested function calls or many local variables are used.

Fast access speed: Due to its simple data structure and automatic management, the stack provides faster data access speed than other memory structures such as the heap.

Automatic memory management: The stack automatically frees space when the function completes, reducing the risk of memory leaks.

4. Applications

Local variable storage: Variables that are only valid within the function (local variables) are stored on the stack.

Function calls: Each time a function is called, a stack frame is created, allowing efficient management of function calls.

Recursion support: The stack allows functions to call themselves (recursively) without the problem of managing return addresses.

2.2. Identify Operations

Stack memory supports some basic operations, allowing data to be managed according to the LIFO (Last In, First Out) principle. Here are the main operations that are commonly performed on stack memory:

1. Push

Description: This operation adds a new stack frame to the top of the stack.

How it works:

- Allocate space for the new stack frame.
- Store the return address, function parameters, and local variables in the stack frame.
- Update the stack top pointer to point to the new stack frame.

Time complexity: O (1)

2. Pop

Description: This operation removes the stack frame at the top of the stack.

How it works:

- Get information from the stack frame (return address, parameters, local variables).
- Update the stack top pointer to point to the previous stack frame.
- Free the memory space of the recently deleted stack frame.

Time complexity: O (1)

3. Peek

Description: This operation allows viewing information at the top of the stack without removing it.

How it works:

- Accesses the data at the top of the stack (usually a return address or a local variable).

Time complexity: O (1)

4. Size

Description: This operation returns the number of stack frames currently in the stack.

How it works:

- Counts the number of stack frames from the top of the stack pointer to the beginning of the stack.

Time complexity: O (n) (if the number of stack frames is not stored, O (1))

5. IsEmpty

Description: This operation checks whether the stack is empty or not.

How it works:

- Compares the top of the stack pointer to null or a specified value (depending on the programming language).

Time complexity: O(1)

2.3. Function Call Implementation

Implementing a function call involves using stack memory to manage the information needed during program execution. When a function is called, the system needs to store information so that it can return to the correct location in the code after the function has finished executing. Here are the detailed steps to implement a function call using stack memory:

1. Step 1: Prepare information

Before calling a function, the system needs to determine:

- Function address: The location in the code where the function is defined.
- Function parameters: The values that will be passed to the function.
- Local variables: The space that the function needs to store variables that only exist within the function scope.

2. Step 2: Call the function

When a function call is made, the system will perform the following operations:

Push stack frame:

- Allocate space for a new stack frame on top of the stack.
- Return address: Store the address in the code where the program will return after the function ends.
- Function parameters: Put parameters on the stack frame.

For example:

```
int result = add(5, 10);
```

The system will store:

The return address (address of the instruction after the add function ends).

The parameters (5 and 10) on the stack frame.

3. Step 3: Allocate local variables

Space for local variables in the function will also be allocated in the stack frame.

```
int add(int a, int b) {
    int sum = a + b;
    return sum;
}
```

4. Step 4: Execute the function

The system begins executing the statements in the function. During this process, it may use the parameters and local variables that have been stored in the stack frame.

5. Step 5: End of function

When the function finishes executing and encounters a return command, the system will perform the following operations:

1. Get the return value: The value returned by the function will be stored at a specified location (usually a register or a location in memory).

2. Pop the stack frame:

- Remove the stack frame of the current function from the top of the stack, freeing up the space allocated for local variables and parameters.
- Return to the address stored in the stack frame to continue program execution.

2.4. Demonstrate Stack Frames

A Stack Frame is an important part of the stack memory, created each time a function is called. It contains the information needed to manage the function call, including parameters, local variables, and return addresses. Here is a detailed description of the structure and function of stack frames.

1. Structure of a Stack Frame

A stack frame typically consists of the following main components:

- Return Address:

The location in the code where the program will continue execution after the function completes.

This address is stored when the function is called and will be used when the stack frame is deleted.

- Function Parameters:

The values that the function receives when it is called.

This parameter can be passed by value or by reference, depending on the programming language.

- Local Variables:

Variables declared within a function are only available within the scope of that function.

Space for local variables is allocated in the stack frame.

- Frame Pointer:

This pointer points to the top of the current stack frame, making it easy to manage local variables and parameters.

Often used to help the system return to the previous stack frame after the function ends.

2. Stack Frame Creation and Management

When a function is called, the stack frame creation and management process is as follows:

- Stack Frame Creation:

When a function is called, a new stack frame is created on top of the current stack.

Necessary information (return address, parameters, local variables) will be stored in this stack frame.

- Function Execution:

The system uses the information in the stack frame to execute the function.

Local variables and parameters can be accessed and manipulated during execution.

- Function End:

When the function ends, the stack frame will be removed (pop) from the top of the stack.

The return address will be used to return to the part of the code that was executed before the function was called.

3. Stack Frame Example

Consider a simple function in the C programming language:

```
int add(int a, int b) {
    int sum = a + b;  // 'sum' là biến cục bộ
    return sum;
}
```

When the add function is called as follows:

```
int result = add(5, 10);
```

The Stack Frame Will Record As Follows:

- Return Address: The address of the next instruction after the function call.
- Function parameters: a = 5, b = 10.
- Local variable: sum will be allocated and stored in the stack frame.

2.5. Discuss the Importance

Stack memory is one of the important memory structures in computer systems, has many applications and greatly affects the way programs operate. Here are some key points about the importance of stack memory:

1. Function Call Management

Correctness: Stack memory allows the storage of return addresses, parameters and local variables when a function is called, helping to ensure that the program can return to the correct location after the function completes. This is very important in maintaining the control flow of the program.

Recursion Support: The stack allows functions to call themselves (recursively) without having to deal with the problem of managing the return address. Each time a function is called, a new stack frame is created, allowing multiple nested calls.

2. Automatic Memory Management

Automatic Release: When a function completes, the memory space for local variables and parameters is automatically released when the stack frame is deleted. This reduces the risk of memory leaks and ensures that memory is used efficiently.

Optimization: The stack structure typically uses less memory than other memory management methods such as the heap, as it does not need to keep track of the size of local variables and parameters.

3. Fast Access

Fast Access: Stacks provide faster access to data than heaps due to the simplicity of data access. Both push and pop operations have a time complexity of $O(1)$, which speeds up the execution of the program.

Performance Optimization: By storing temporary data close to where it is used, stacks improve the overall performance of the application.

4. Program Flow Control

Return Address Management: The stack helps keep track of multiple return addresses for function calls, which is important in programming languages that support features like asynchronous function calls or callbacks.

Aiding Application Development: With explicit stack frames, programmers can easily track the flow of control and the values of local variables during debugging.

5. Supporting Advanced Features

Advanced Features: Stack memory supports many advanced features in modern programming languages, such as closures in JavaScript or lambdas in Python, where information about local variables and the execution environment needs to be stored and managed.

Portability and Compatibility: The stack structure is often used to ensure that code can run on multiple platforms without modifications, thanks to the consistency in how addresses and data are managed.

III. Queue

3.1.Introduction FIFO

FIFO stands for First in First out. This is an inventory valuation method that assumes that the goods produced or received first will be sold, used or disposed of first. The FIFO method serves as both an accurate and easy way to calculate the value of ending inventory and a suitable way to manage a business's inventory to save costs and bring benefits to customers.

3.2.Define the Structure

A queue is a linear data structure in which elements are inserted at the rear and removed from the front. This structure works on the FIFO (First-In-First-Out) principle. Basic operations on a queue include:

- Enqueue: Add an element to the rear of the queue.
- Dequeue: Get an element from the front of the queue.
- Peek/Front: View an element at the front of the queue without removing it.
- IsEmpty: Check whether the queue is empty or not.

3.3.Array-Based Implementation

```java
package org.example;

class QueueUsingArray {
    private int front, rear, capacity;
    private int[] queue;

    public QueueUsingArray(int size) {
        capacity = size;
        queue = new int[capacity];
        front = 0;
        rear = -1;
    }

    // Check if the queue is empty
    public boolean isEmpty() {
        return front > rear;
    }

    // Check if the queue is full
    public boolean isFull() {
        return rear == capacity - 1;
    }

    // Add an element to the queue
    public void enqueue(int item) {
        if (isFull()) {
            System.out.println("Queue is full");
        } else {
            queue[++rear] = item;
            System.out.println(item + " added to the queue");
        }
    }

    // Get an element from the queue
    public int dequeue() {
        if (isEmpty()) {
            System.out.println("Queue is empty");
            return -1;
        } else {
            int item = queue[front++];
            return item;
        }
    }

    // View the front element of the queue
    public int peek() {
        if (isEmpty()) {
            System.out.println("Queue is empty");
            return -1;
        } else {
            return queue[front];
        }
    }
}
```

3.4.Linked List-Based Implementation

```
package org.example;

class QueueUsingLinkedList {
    class Node {
        int data;
        Node next;

        public Node(int data) {
            this.data = data;
            this.next = null;
        }
    }
    private Node front, rear;

    public QueueUsingLinkedList() {
        front = rear = null;
    }
    // Check if the queue is empty
    public boolean isEmpty() {
        return front == null;
    }
    // Add element to queue
    public void enqueue(int item) {
        Node newNode = new Node(item);
        if (rear == null) {
            front = rear = newNode;
            return;
        }
        rear.next = newNode;
        rear = newNode;
        System.out.println(item + " added to the queue");
    }
    // Get an element from the queue
    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;
        }
        return item;
    }
    // View the front element of the queue
    public int peek() {
        if (isEmpty()) {
            System.out.println("Queue is empty");
            return -1;
        }
        return front.data;
    }
}
```

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

Using Array Based Queues

```
package org.example;
public class Main {
    public static void main(String[] args) {
        QueueUsingArray queue = new QueueUsingArray(5); // Initialize the queue with
size 5

        // Add elements to the queue
        queue.enqueue(10); // Queue: [10]
        queue.enqueue(20); // Queue: [10, 20]
        queue.enqueue(30); // Queue: [10, 20, 30]
        queue.enqueue(40); // Queue: [10, 20, 30, 40]

        // Remove elements from the queue
        System.out.println("Dequeued: " + queue.dequeue()); // Print 10, Queue: [20,
30, 40]
        System.out.println("Dequeued: " + queue.dequeue()); // Print 20, Queue: [30,
40]

        // Add element to queue
        queue.enqueue(50); // Queue: [30, 40, 50]
        System.out.println("Dequeued: " + queue.dequeue()); // Print 30, Queue: [40,
50]

        // Check the front of the queue
        System.out.println("Front of queue: " + queue.peek()); // Print 40
    }
}
```

Using Queue Based on Linked List

```
package org.example;
public class Main {
    public static void main(String[] args) {
        QueueUsingLinkedList queue = new QueueUsingLinkedList(); // Initialize the
queue

        // Add elements to the queue
        queue.enqueue(10); // Queue: [10]
        queue.enqueue(20); // Queue: [10, 20]
        queue.enqueue(30); // Queue: [10, 20, 30]
        queue.enqueue(40); // Queue: [10, 20, 30, 40]

        // Remove elements from the queue
        System.out.println("Dequeued: " + queue.dequeue()); // Print 10, Queue: [20,
30, 40]
        System.out.println("Dequeued: " + queue.dequeue()); // Print 20, Queue: [30,
40]

        // Add element to queue
        queue.enqueue(50); // Queue: [30, 40, 50]
```

```
        System.out.println("Dequeued: " + queue.dequeue()); // Print 30, Queue: [40,
50]

        // Check the front of the queue
        System.out.println("Front of queue: " + queue.peek()); // Print 40
    }
}
```

IV. Sorting algorithms

4.1.Introducing the two sorting algorithms you will be comparing

Quick Sort

Quick Sort is a fast sorting algorithm, also known as Part Sort. You can understand it more simply, this algorithm is based on dividing data into smaller groups of elements. The ability of this algorithm to sort data is much faster than any other traditional algorithm. However, Quick Sort is not very stable because it does not guarantee the equality of the relative order of the elements.

Bubble sort

Bubble sort is a simple sorting algorithm. This sorting algorithm is based on comparing adjacent pairs of elements and swapping them if they are not in order.

This algorithm is not suitable for large data sets as the worst case and average case complexity is O(n2) where n is the number of elements.

Bubble sort is the slowest algorithm among the basic sorting algorithms. This algorithm is even slower than the direct swapping algorithm although the number of comparisons is the same, but because it swaps two adjacent elements, the number of swaps is greater.

4.2.Time Complexity Analysis

The time complexity of an algorithm reflects the number of operations required to complete the job, given a given number of input elements. For Quick Sort and Bubble Sort, their time complexity differs greatly, reflecting their performance and how they process arrays.

Algorithm	Best case	Average case	Worst case
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$

- Quick Sort has O(n log n) time complexity in the best and average cases, but can be reduced to $O(n^2)$ in the worst case if the pivot is not chosen well.
- Bubble Sort has $O(n^2)$ time complexity in both the average and worst cases, and can only achieve O(n) in the best case when the array is already sorted.

4.3.Space Complexity Analysis

The space complexity of an algorithm describes the amount of memory the algorithm requires to execute, including memory for input data, intermediate memory, and memory for temporary variables. Analyzing the space complexity helps us evaluate the ability of the algorithm to optimize memory when applied in practice.

Algorithm	Space Complexity
Quick Sort	O(log n) (best and average case), O(n) (worst case)
Bubble Sort	O(1)

Quick Sort requires O(log n) memory space in the best and average cases, but in the worst case, the memory space can increase to O(n) due to recursion depth.

Bubble Sort has a space complexity of O(1), because it only uses a constant amount of memory for temporary variables while sorting the array in place. Therefore, Bubble Sort is more optimal than Quick Sort in terms of memory requirements, especially when memory space is an important factor.

4.4.Stability

The stability of a sorting algorithm describes its ability to maintain the order of elements with equal values in the input array. An algorithm is said to be stable if when two elements have the same value, their position in the sorted array remains the same as in the original array. If the order of identical elements changes during the sorting process, the algorithm is said to be unstable.

Algorithm	Stability
Quick Sort	Unstable
Bubble Sort	Stable

4.5.Comparison Table

Algorithm	Quick Sort	Bubble Sort
Time complexity	O(n log n) (average) / O(n^2) (bad)	O(n^2) (average and worst)
Space complexity	O(log n)	O(1)
Stability	Unstable	Stable
Efficiency	Faster for large arrays	Slow for large arrays
Understandability	More complex and difficult to understand	Easy to understand and easy to install
Applications	Sorting large arrays, optimal algorithms in practice	Sorting small or almost sorted arrays

4.6.Performance Comparison

Quick Sort is often used in real-world applications because of its performance (O(n log n) on average). In cases where the array is large or where there is no pre-sorted data, Quick Sort is a better choice.

Bubble Sort is only used when the array is small or when a simple solution is needed that does not require high performance, as O(n^2) complexity is too slow for large arrays.

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

```
package org.example;

import java.util.Random;

public class SortingPerformance {
    // Quick Sort
    public static void quickSort(int[] arr, int low, int high) {
        if (low < high) {
            int pivotIndex = partition(arr, low, high);
            quickSort(arr, low, pivotIndex - 1);
            quickSort(arr, pivotIndex + 1, high);
        }
    }
    public 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;
    }

    // Bubble Sort
    public static void bubbleSort(int[] arr) {
        int n = arr.length;
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - 1 - i; j++) {
                if (arr[j] > arr[j + 1]) {
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }
    }
    // Random array generator function
    public static int[] generateRandomArray(int size) {
        int[] arr = new int[size];
        Random rand = new Random();
        for (int i = 0; i < size; i++) {
            arr[i] = rand.nextInt(1000); // Random number from 0 to 999
        }
        return arr;
    }

    // Main function for performance comparison
    public static void main(String[] args) {
        int size = 10000; // Large array size
        int[] quickSortArray = generateRandomArray(size);
        int[] bubbleSortArray = generateRandomArray(size);

        // Quick Sort
        long startTime = System.nanoTime();
        quickSort(quickSortArray, 0, quickSortArray.length - 1);
        long endTime = System.nanoTime();
        long quickSortTime = endTime - startTime;
        System.out.println("Quick Sort Time: " + quickSortTime + " nanoseconds");

        // Bubble Sort
        startTime = System.nanoTime();
        bubbleSort(bubbleSortArray);
        endTime = System.nanoTime();
        long bubbleSortTime = endTime - startTime;
        System.out.println("Bubble Sort Time: " + bubbleSortTime + " nanoseconds");
    }
}
```

V.Network shortest path algorithms

5.1. Introducing the concept of network shortest path algorithms

The shortest path algorithm in a network is used to find the path with the lowest cost (or shortest distance) between two points (nodes) in a weighted graph. The network can be represented as a graph, where the nodes are locations or devices, and the edges (connections) between the nodes can have weights representing costs or distances.

Shortest path algorithms are commonly used in situations such as:

- Telecommunication network routing: Finding the optimal path for data packets.
- Transportation systems: Determining the shortest route between two points.
- Intelligent transportation systems: Optimizing travel in cities.

Common algorithms for finding shortest paths include Dijkstra's algorithm and Prim-Jarnik's algorithm. While Dijkstra's algorithm is used to find the shortest path between two nodes, Prim-Jarnik's algorithm searches for the minimum spanning tree (MST) in a graph.

5.2. Algorithm 1: Dijkstra's Algorithm

Dijkstra's algorithm is one of the classic algorithms for solving the problem of finding the shortest path from a given point to all remaining points in a weighted graph. In this article, we will learn the basic idea of Dijkstra's algorithm.

Steps of Dijkstra's algorithm:

Initialization: Set the distance value from the source to all remaining nodes to infinitely large (∞), except for the source, which has a distance value of 0. Mark all nodes as unvisited.

Traverse the unvisited nodes: From the current node, calculate the distance to all unvisited nodes through the edges. If the calculated distance is less than the current value of the destination node, update the distance value of that node.

Select the node with the smallest distance: Select the unvisited node with the smallest distance and mark it as visited.

Repeat: Continue the above steps until all nodes have been visited.

End: When all nodes are visited, we will have the shortest distance from the source to all remaining nodes in the graph.

5.3. Algorithm 2: Prim-Jarnik Algorithm

Prim-Jarnik algorithm is a greedy algorithm used to find the minimum spanning tree (MST) of a graph. Although this algorithm is not a shortest path algorithm, it is still related because it also tries to connect the vertices in the graph efficiently.

Steps of Prim-Jarnik algorithm:

Initialization: Pick any vertex and start creating the MST from it. Initialize the set of unselected vertices.

Choose the least weighted edge: From the vertices in the MST, choose an edge with the least weight connecting a selected vertex to an unselected vertex.

Add vertices and edges to MST: Add a new vertex to the MST and remove it from the set of unselected vertices.

Repeat: Continue steps 2 and 3 until all vertices have been selected in the MST.

5.4. Performance Analysis

Criteria	Dijkstra's Algorithm	Prim-Jarnik's Algorithm
Objective	Find the shortest path from source to all vertices	Find the minimum spanning tree of the graph
Graph Types	Non-Negative Weighted Graphs	Acyclic, Undirected Graphs
Execution time	O(E log V) if using heap	O(E log V) if using heap
Space	O(V) (distance and heap storage)	O(V) (MST and heap storage)
Applications	Routing in networks, GPS, finding shortest paths	Network design, optimizing connections between vertices

Dijkstra's algorithm is suitable for problems that require finding the shortest path from a source to other vertices in a graph.

Prim-Jarnik's algorithm is used in situations where it is necessary to build a minimum spanning tree for the entire graph, helping to optimize network connections.

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

6.1. Define the data structure

A stack is a linear data structure that follows the Last In First Out (LIFO) principle. This means that the last element added to the stack is the first one to be removed. It can be visualized as a pile of plates where the most recently placed plate is on the top and is the first one to be removed.

Real-world applications of stack:

- Browser History: As you navigate through web pages, your history is stored in a stack. Clicking the back button pops the most recent page off the stack.
- Undo Mechanism: Many applications, such as text editors, use stacks to implement undo functionality. Each action is pushed onto a stack, and undo pops the most recent action.
- Call Stack: In recursive function calls, the stack is used to keep track of the function calls.

Examples of stack problems in practice

1. Managing browsing history (Browser History)

When you browse the web, the web pages you visit will be stored on a stack. When you press the "Back" button to return to the previous page, the browser will pop the current page from the stack and peek to display the previous page.

- Push: When you access a new page, that page is pushed into the stack.
- Pop: When you press "Back", the current page is removed and the previous page is displayed.

2. "Undo" mechanism in editing applications (Text Editor, Photoshop)

Most text or graphics editing applications have an undo function to go back to previous operations. These operations are stored in a stack. When you press "Ctrl + Z" to undo, the latest operation will be popped from

the stack and the application will return to the previous state.

- Push: Every time you perform an action (like typing or drawing), it is added to the stack.
- Pop: When you select undo, the last action is removed from the stack and the undo is performed.

3. Checking for valid parentheses in a mathematical expression (Expression Parsing)

When checking whether an expression has the opening and closing parentheses in the correct order, we can use the stack. Every time we encounter an opening parenthesis, we push it onto the stack. When we encounter a closing parenthesis, we check if the corresponding opening parenthesis is the latest parenthesis on the stack. If not, the expression is invalid.

- Push: When we encounter an opening parenthesis (or { or [.
- Pop: When we encounter a closing parenthesis) or } or].

6.2. Initialise the stack

Here is an example of how to initialise a stack in Java using the Stack class available in the java.util package:

```
import java.util.Stack;

public class Main {
    public static void main(String[] args) {
        // Khởi tạo một stack kiểu Integer
        Stack<Integer> stack = new Stack<>();

        // Thêm phần tử vào stack
        stack.push(10);
        stack.push(20);
        stack.push(30);

        // In các phần tử trong stack
        System.out.println("Stack: " + stack);  // Output: Stack: [10, 20, 30]
    }
}
```

Explanation:

Initialize stack: Use new Stack<>() to initialize an integer stack.

Push operation: Add elements to the stack by calling the push() function (here 10, 20, 30).

Print stack: Print the contents of the stack, with the output being [10, 20, 30], where element 30 is the top element of the stack.

6.3. Push operation

In stack, push operation is used to add an element to the top of the stack. When you add (push) an element to the stack, it will be placed at the top position and the previous elements will be placed below.

Syntax and push() operation in Java:

In Java, you can use the push() method of the Stack class to add elements to the stack.

Example of push operation in Java:

```
import java.util.Stack;

public class Main {
    public static void main(String[] args) {
        // Khởi tạo một stack kiểu Integer
        Stack<Integer> stack = new Stack<>();

        // Sử dụng phương thức push để thêm phần tử vào stack
        stack.push(10);
        stack.push(20);
        stack.push(30);

        // In stack sau khi thêm phần tử
        System.out.println("Stack sau khi thêm phần tử: " + stack);  // Output: [10, 20, 30]
    }
}
```

Explanation:

Stack initialization: Use new Stack<>() to initialize an integer stack.

Push operation: Add elements 10, 20, and 30 to the stack in turn.

When stack.push(10): Element 10 is placed on top of the stack.

When stack.push(20): Element 20 is placed on top of element 10.

When stack.push(30): Element 30 is placed on top.

Printing the stack contents: When printing the stack, the result will be [10, 20, 30], where 30 is the top element of the stack.

6.4. Pop operation

In stack, pop operation is used to remove and return the top element of the stack. Stack follows the LIFO (Last In First Out) principle, so the last element added will be the first element removed when performing the pop operation.

Java pop() Syntax and Operation:

Java Stack class provides pop() method to remove and return the element at the top of the stack.

Example of pop operation in Java:

```
import java.util.Stack;

public class Main {
    public static void main(String[] args) {
        // Khởi tạo một stack kiểu Integer
        Stack<Integer> stack = new Stack<>();

        // Thêm phần tử vào stack
        stack.push(10);
        stack.push(20);
        stack.push(30);

        // In nội dung stack trước khi pop
        System.out.println("Stack trước khi pop: " + stack);  // Output: [10, 20, 30]

        // Thực hiện thao tác pop, loại bỏ phần tử trên cùng của stack
        int poppedElement = stack.pop();

        // In phần tử vừa bị pop và stack sau khi pop
        System.out.println("Phần tử vừa bị pop: " + poppedElement);  // Output: 30
        System.out.println("Stack sau khi pop: " + stack);  // Output: [10, 20]
```

```
    }
}
```

6.5. Peek operation

In stack, peek operation is used to get the value of the top element without removing it from the stack. This allows us to know which element is at the top of the stack without changing the state of the stack.

Java peek() Syntax and Operation:

The Stack class in Java provides the peek() method to return the top element of the stack without removing it.

Example of peek operation in Java:

```
import java.util.Stack;

public class Main {
    public static void main(String[] args) {
        // Khởi tạo một stack kiểu Integer
        Stack<Integer> stack = new Stack<>();

        // Thêm phần tử vào stack
        stack.push(10);
        stack.push(20);
        stack.push(30);

        // In nội dung stack trước khi peek
        System.out.println("Stack trước khi peek: " + stack);  // Output: [10, 20,
30]

        // Thực hiện thao tác peek, lấy phần tử trên cùng mà không xóa
        int topElement = stack.peek();

        // In phần tử trên cùng và stack sau khi peek
        System.out.println("Phần tử trên cùng (peek): " + topElement);  // Output: 30
```

```
        System.out.println("Stack sau khi peek: " + stack);  // Output: [10, 20, 30]
    }
}
```

Explanation:

Stack initialization: Use new Stack<>() to initialize an integer stack.

Push operation: Add elements 10, 20, and 30 to the stack in turn.

Peek operation: When calling stack.peek(), the top element of the stack (element 30) will be returned but not removed from the stack.

In stack after peek: After peek operation, the stack remains in the same state with the contents [10, 20, 30].

6.6. Check if the stack is empty

In stack, checking if the stack is empty or not is an important operation to avoid taking elements from an empty stack. The isEmpty() method will help to check this, returning true if the stack contains no elements and false if there is at least one element.

Syntax and operation of isEmpty() in Java:
The Stack class in Java has a built-in isEmpty() method to check if the stack is empty or not.
Example of isEmpty() operation in Java:

```
import java.util.Stack;

public class Main {
    public static void main(String[] args) {
        // Khởi tạo một stack kiểu Integer
        Stack<Integer> stack = new Stack<>();

        // Kiểm tra xem stack có rỗng không
        System.out.println("Stack có rỗng không? " + stack.isEmpty());  // Output:
true

        // Thêm phần tử vào stack
        stack.push(10);

        // Kiểm tra lại xem stack có rỗng không sau khi thêm phần tử
        System.out.println("Stack có rỗng không sau khi thêm phần tử? " +
stack.isEmpty());  // Output: false
    }
}
```

6.7. Full source code (stack using Array)

```
class MyStack {
    private int maxSize;
    private int[] stackArray;
    private int top;

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

    // Hàm push để thêm phần tử vào stack
    public void push(int value) {
        if (top == maxSize - 1) {
            System.out.println("Stack is full. Cannot push element.");
        } else {
            stackArray[++top] = value;
        }
    }

    // Hàm pop để loại bỏ phần tử trên cùng của stack
    public int pop() {
        if (isEmpty()) {
            System.out.println("Stack is empty. Cannot pop element.");
            return -1;  // Trả về -1 nếu stack rỗng
        } else {
            return stackArray[top--];  // Loại bỏ phần tử trên cùng và giảm chỉ số
top
        }
    }
```

```
    // Hàm peek để xem phần tử trên cùng của stack
    public int peek() {
        if (isEmpty()) {
            System.out.println("Stack is empty. Cannot peek.");
            return -1;  // Trả về -1 nếu stack rỗng
        } else {
            return stackArray[top];  // Trả về phần tử trên cùng mà không loại bỏ nó
        }
    }

    // Hàm isEmpty để kiểm tra stack có rỗng không
    public boolean isEmpty() {
        return (top == -1);  // Nếu top == -1, stack rỗng
    }

    // Hàm để in nội dung stack
    public void printStack() {
        if (isEmpty()) {
            System.out.println("Stack is empty.");
        } else {
            for (int i = 0; i <= top; i++) {
                System.out.print(stackArray[i] + " ");
            }
            System.out.println();
        }
    }
}
public class Main {
    public static void main(String[] args) {
        MyStack stack = new MyStack(5);

        // Kiểm tra xem stack có rỗng không
        System.out.println("Stack có rỗng không? " + stack.isEmpty());  // Output:
true

        // Thêm phần tử vào stack
        stack.push(10);
        stack.push(20);
        stack.push(30);

        // In stack sau khi thêm phần tử
        System.out.print("Stack sau khi thêm phần tử: ");
        stack.printStack();  // Output: 10 20 30

        // Kiểm tra phần tử trên cùng (peek)
        int topElement = stack.peek();
        System.out.println("Phần tử trên cùng (peek): " + topElement);  // Output: 30

        // Thực hiện thao tác pop
        int poppedElement = stack.pop();
        System.out.println("Phần tử vừa bị pop: " + poppedElement);  // Output: 30

        // In stack sau khi pop
        System.out.print("Stack sau khi pop: ");
        stack.printStack();  // Output: 10 20

        // Kiểm tra lại xem stack có rỗng không
```

```
        System.out.println("Stack có rỗng không? " + stack.isEmpty());  // Output:
false
    }
}
```

Detailed explanation of the parts of the source code:

MyStack class:

- Manages stack operations with the properties maxSize, stackArray, and top.
- Uses the stackArray array to store elements in the stack.
- The top variable holds the index of the top element in the stack, with an initial value of -1 (indicating an empty stack).

The push(int value) function:

- Adds an element to the top of the stack.
- Checks if the stack is full, disallows adding elements.

The pop() function:

- Removes and returns the top element of the stack.
- Checks if the stack is empty, disallows pop operations.

The peek() function:

- Returns the top element without removing it from the stack.
- If the stack is empty, returns -1.

The isEmpty() function:

- Checks if the stack is empty.

The printStack() function:

- Prints the contents of the stack, if the stack is empty, prints a message.

Main class:

- Runs operations on the stack and displays the results.

6.8. Full source code (stack using LinkList)

```
class Node {
    int data;
    Node next;

    // Constructor để khởi tạo node mới
    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}

class LinkedListStack {
    private Node top;  // Node đỉnh của stack

    // Constructor để khởi tạo stack rỗng
    public LinkedListStack() {
        this.top = null;
    }

    // Kiểm tra stack có rỗng không
    public boolean isEmpty() {
        return top == null;
    }

    // Thêm phần tử vào stack (push)
    public void push(int data) {
        Node newNode = new Node(data);
        newNode.next = top;  // Node mới trỏ tới node top hiện tại
        top = newNode;       // Top giờ là node mới
    }

    // Loại bỏ và trả về phần tử trên cùng của stack (pop)
```

```
    public int pop() {
        if (isEmpty()) {
            System.out.println("Stack is empty. Cannot pop element.");
            return -1;  // Trả về -1 nếu stack rỗng
        }
        int poppedData = top.data;
        top = top.next;  // Top trỏ tới phần tử tiếp theo trong stack
        return poppedData;
    }

    // Xem phần tử trên cùng của stack mà không loại bỏ nó (peek)
    public int peek() {
        if (isEmpty()) {
            System.out.println("Stack is empty. Cannot peek.");
            return -1;  // Trả về -1 nếu stack rỗng
        }
        return top.data;
    }

    // In các phần tử của stack
    public void printStack() {
        if (isEmpty()) {
            System.out.println("Stack is empty.");
        } else {
            Node current = top;
            while (current != null) {
                System.out.print(current.data + " ");
                current = current.next;
            }
            System.out.println();
        }
    }
}

public class Main {
    public static void main(String[] args) {
        LinkedListStack stack = new LinkedListStack();

        // Kiểm tra xem stack có rỗng không
        System.out.println("Stack có rỗng không? " + stack.isEmpty());  // Output:
true

        // Thêm phần tử vào stack
        stack.push(10);
        stack.push(20);
        stack.push(30);

        // In stack sau khi thêm phần tử
        System.out.print("Stack sau khi thêm phần tử: ");
        stack.printStack();  // Output: 30 20 10

        // Kiểm tra phần tử trên cùng (peek)
        int topElement = stack.peek();
        System.out.println("Phần tử trên cùng (peek): " + topElement);  // Output: 30

        // Thực hiện thao tác pop
        int poppedElement = stack.pop();
```

```
        System.out.println("Phần tử vừa bị pop: " + poppedElement);  // Output: 30

        // In stack sau khi pop
        System.out.print("Stack sau khi pop: ");
        stack.printStack();  // Output: 20 10

        // Kiểm tra lại xem stack có rỗng không
        System.out.println("Stack có rỗng không? " + stack.isEmpty());  // Output:
false
    }
}
```

Detailed explanation of the source code sections:

Node class:

- Represents a node in the linked list.
- Each node contains data and a reference to the next node in the list.

LinkedListStack class:

- Manages stack operations with the push, pop, peek, isEmpty, and printStack methods.
- Top property: Represents the first node in the stack, which is the top element of the stack.

The push(int data) method:

- Creates a new node with the passed data and adds it to the top of the stack.
- The new node becomes the top, pointing to the old top node.

The pop() method:

- Removes and returns the top element of the stack.
- If the stack is empty, prints a message and returns -1.

The peek() method:

- Returns the value of the top element without removing it.
- Returns -1 if the stack is empty.

The isEmpty() method:

- Checks if the stack is empty (when top == null).

printStack() method:

- Prints the elements in the stack from top to bottom of the list.

Main class:

- Checks and displays basic operations on the stack.

6.9. Comparing

When comparing two elements or two approaches in programming or data structures, we're essentially evaluating their behavior, performance, and suitability for certain applications. Here's a general guide on key factors often used to compare structures or algorithms:

Time Complexity: How does the performance scale with larger inputs? Algorithms and data structures have varying time complexities (e.g., $O(n)O(\log n)$ which indicate their efficiency.

Space Complexity: How much memory does it require? Some approaches may use extra space (e.g., in sorting algorithms like mergesort), whereas others might be more memory-efficient.

Use Case: What problems is each approach designed for? Different structures and algorithms excel in specific applications, such as stacks (LIFO) for recursive function calls, and queues (FIFO) for order **processing.**

Ease of Implementation and Maintainability: How easy is it to implement and understand? Some algorithms are straightforward (e.g., bubble sort), while others are complex (e.g., quicksort), which might affect the readability and maintainability of code.

Data Structure Specific Operations:

- **Stack:** Suited for problems that require last-in, first-out (LIFO) operations, such as undo features, parsing expressions, or evaluating postfix notation.
- **Queue:** Useful for problems needing first-in, first-out (FIFO) operations, like handling requests, **managing order processing, or breadth-first search.**

Stability and Adaptability: Does the approach handle special cases or unexpected data well? Stability (in sorting) or adaptability to dynamic data can affect the choice.

VII. Implement a complex ADT and algorithm in an executable programming

7.1. Encapsulation

7.1.1. What is encapsulations

Encapsulation is one of the core principles of object-oriented programming (OOP). It is the process of grouping data (fields) and methods that manipulate that data into a single unit called a class. Encapsulation hides the internal implementation details of an object and allows access or modification only through the public methods provided.

7.1.2. Example of encapsulations

```java
package org.example;

public class Student {
    private String name; // Data is hidden
    private int age;
    // Constructor
    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }
    // Getter allows safe access to data
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
    // Setter for controlled updating of data
    public void setName(String name) {
        if (!name.isEmpty()) { // Validation
            this.name = name;
        }
    }
    public void setAge(int age) {
        if (age > 0) { // Validation
            this.age = age;
        }
    }
}
```

7.1.3. Data Protection

Data Protection is the measures and regulations implemented to protect personal data and sensitive information from unauthorized access, misuse, alteration or destruction. It ensures that users' personal information is processed safely and securely. Data protection measures can include both technical and organizational elements to protect the privacy and security of data.

Data Protection is divided into 2 types

Data Protection is often divided into two main types: Active Data Protection and Passive Data Protection. Both types are important and are often used simultaneously to ensure data safety and security. Below is a detailed description of each type:

Active Data Protection:

- Encryption: Data is encrypted to ensure that only those with the decryption key can access and read the content.
- Authentication and Access Control: Using user authentication and authorization methods to limit access to data to authorized users.
- Intrusion Detection and Prevention: Using systems and software to detect and prevent cyber attacks.
- Security Alerts: Set up alert systems to detect and notify of suspicious activity or security breaches.

Passive Data Protection:

- Data Backup: Perform regular data backups to ensure that data can be recovered in the event of loss or corruption.
- Secure Data Storage: Store data on secure media to prevent unauthorized access.
- System Auditing and Monitoring: Monitor and record system activity to detect security breaches and ensure compliance.
- Version Control: Manage different versions of data to ensure that previous versions can be restored if necessary.

7.1.4. Modularity and Maintainability

What is modularity?

Modularity is the ability to divide a large system into small, independent components with their own responsibilities. In programming, modularity helps design components (modules) that are easy to manage, develop, and test without affecting other components.

In object-oriented programming (OOP), modularity is promoted through encapsulation. Each class is an independent module, responsible for managing its own data and functionality.

What is maintainability?

Maintainability refers to the ease with which code can be modified, extended, or improved without adversely affecting other parts of the program.

Encapsulation in OOP aids maintainability by:

- Hiding implementation details.
- Separating the user interface and internal processing logic.
- Minimizing the impact of code changes.

Benefits of modularity and maintainability through encapsulation

Isolation of change: If a class's internal implementation needs to be changed, just update that class without affecting the code that uses the class.

Ease of extension: New functionality can be added to a module without breaking the existing structure.

Increased testing efficiency: Each class can be tested individually, making it easier to detect and fix bugs.

Code reuse: Independent modules can be used in multiple projects without modification.

7.1.5. Code Reusability

Code Reusability is one of the important benefits of encapsulation and object-oriented programming (OOP). It allows written code or modules to be reused in other parts of the program or in other projects without having to rewrite them from scratch.

Benefits of code reusability

- Save time and effort: Reduce development time because previously tested and verified code can be reused.
- Increase reliability: Reusable code components are often tested, ensuring correct operation, minimizing errors when deployed in new projects.

- Consistency: Using reusable code helps ensure consistency in applications, reducing differences in logic handling.
- Ease of extension: A module can be reused and extended without affecting other parts.

7.2. Information Hiding

7.2.1. Abstraction

Abstraction is the process of simplifying complex systems by focusing on the essential features while hiding the underlying implementation details. In the context of ADTs, abstraction allows users to interact with data types through a well-defined interface without needing to know how these operations are implemented. This means that users can utilise data structures like lists or stacks based on their functionalities rather than their internal workings.

For example, when using a stack ADT, a programmer can push or pop elements without understanding whether the stack is implemented using an array or a linked list. This separation of concerns not only simplifies usage but also allows for flexibility in changing implementations without affecting the user's code.

7.2.2. Reduced Complexity

By hiding implementation details, ADTs significantly reduce complexity for developers. Users of an ADT are presented with a simplified interface that exposes only necessary operations, making it easier to understand and use the data structure. This reduction in complexity leads to fewer errors and more straightforward debugging processes.

For instance, when working with a queue ADT, developers do not have to grapple with the intricacies of how elements are stored or managed internally. They can focus on enqueuing and dequeuing operations, which streamlines development and fosters a clearer understanding of program logic.

7.2.3. Improved Security

Information hiding enhances security by restricting access to sensitive data within an ADT. By exposing only necessary methods and keeping data members private, ADTs prevent unauthorised access and modifications. This encapsulation protects the integrity of the data and ensures that it can only be manipulated through controlled interfaces.

For example, consider a bank account ADT that allows deposits and withdrawals but hides the balance from direct access. This means that external code cannot alter the balance directly; it can only do so through designated methods that enforce rules (e.g., preventing overdrafts). This mechanism not only secures the data but also enforces business logic consistently across the application.

VIII. Discuss the view that imperative ADTs are a basis for object orientation offering a justification for the view.

8.1. Encapsulation and Information Hiding

Encapsulation and Information Hiding are two core concepts in Object Oriented Programming (OOP), built on the foundation of Abstract Data Structures (ADTs). They play an important role in protecting data and optimizing software design.

Encapsulation: In ADTs, data is hidden and only specific operations (processes) are exposed. This is similar to OOP, where data (attributes) and methods (behaviors) are packaged together in objects and access to data is controlled through methods.

Information hiding: By abstracting away the details of how data is stored or manipulated, ADTs hide implementation details from the user of the data structure. In OOP, this principle is realized through access modifiers such as private and protected, which ensure that the internal details of objects are not exposed.

8.2. Modularity and Reusability

Modularity and Reusability are two important characteristics in software design, inherited from Abstract Data Structure (ADT) and strongly promoted in Object Oriented Programming (OOP). They support building flexible, easy-to-maintain applications, and save development costs.

Reusability

Reusability refers to the reuse of software components in different projects or parts of an application without rewriting them from scratch.

Features of reuse in OOP:

- Inheritance: Subclasses can inherit and extend the functionality of their parent classes.
- Polymorphism: Allows the use of a common interface to implement different behaviors.
- Utility classes: Well-designed classes can be reused in many contexts.

8.3. Procedural Approach

Imperative ADTs rely on a procedural approach, where the focus is on defining sequences of operations that manipulate data. This contrasts with the declarative style of programming, where the emphasis is on the results rather than the steps to achieve them.

- Procedural Nature: In imperative ADTs, you define operations like insertion, deletion, or retrieval of data in a step-by-step manner. OOP evolved from this procedural thinking by organizing these operations (or methods) around objects that encapsulate both the data and the procedures.
- Transition to OOP: While ADTs are procedural in nature, they serve as a foundation for OOP, where these procedures are tied to objects. The principles of operation (procedural focus) still exist in OOP but are encapsulated within objects.

8.4. Language Transition

The transition from imperative ADTs to OOP is evident in the evolution of programming languages. Early languages like C used ADTs to structure programs, while modern OOP languages like C++ or Java extend this concept to include classes and objects.

- C to C++: Languages like C++ added the concept of classes (an extension of ADTs) to the C language. A class is essentially an ADT with added features like inheritance and polymorphism. This transition shows how imperative ADTs laid the groundwork for object orientation.
- OOP Languages: In languages like Java, Python, and C#, the concept of ADTs is embedded within the object model. Every class in these languages can be seen as an abstract data type that provides an interface for manipulating data, with the data itself being hidden (encapsulation).

IX. Implement complex data structures and algorithms

9.1. Define the problem

The objective is to develop an Abstract Data Type (ADT) for a student management system. This system must handle student information, including ID, name, and grades, and support functionalities such as adding, modifying, removing, sorting, and searching for students. Furthermore, it should classify students into ranks based on their grades: Fail, Medium, Good, Very Good, and Excellent. The ADT should also be designed for straightforward expansion and compatibility with other system components in the future.

The program must support users to enter information for multiple students and perform the following basic operations:

- Add new student: Enter information about student code, name, and score.
- Update student information: Allow users to change student information by code.
- Delete student: Remove a student from the list based on code.
- Search student by name or code: Allow searching for students based on name or student code.
- Display student list: Display all students in the class with entered information.

We will use abstract data types (ADT) to design data structures to store student information and algorithms to support operations such as searching, sorting, and updating information efficiently.

Specific requirements:

- The program needs to be able to process and store student information in a class with the number of students changing over time.
- The program must have a simple and easy-to-use interface for the person entering the information.
- Operations must be performed quickly and accurately.

9.2. ADT

9.2.2. Choose a Programming Language

Choosing Java as the programming language for this application is a reasonable choice, because Java:

- Supports object-oriented programming: Java is a powerful object-oriented language, making it easy to apply programming principles such as encapsulation, inheritance, and polymorphism. You can use classes to represent students and other objects in the application.
- Automatic memory management: Java has a garbage collection system that helps manage memory automatically, minimizing the risk of memory errors and helping developers focus on application logic.
- Rich libraries: Java has many libraries that support data set operations, such as ArrayList, HashMap, TreeMap, which can help in storing and processing student information.
- Ensure compatibility and scalability: Java is a cross-platform language with high compatibility, allowing the program to run on many different operating systems without modifying the source code. This is important when deploying the application to many customers in the future.
- Large community and support documentation: With a large development community and rich documentation, Java makes it easy to find support and solve problems during development.

9.2.3. Design the ADT

Student Object

First, we need to define a Student class to store information about each student. The basic properties will include:

- id (student ID, Each student has only 1 unique id)
- name (student name)
- score (student score)

ADT StudentList

Next, we will design an ADT named StudentList to manage the student list. This ADT will include operations such as:

- addStudent(Student student): Add student to the list.
- removeStudent(int id): Remove student from the list by code.
- updateStudent(int id, String name, double marks): Update student information by code.
- searchStudentById(int id): Search student by code.
- searchStudentByName(String name): Search student by name.

9.2.4. Implement the ADT

Class Main

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

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        StudentManagement studentManagement = new StudentManagement();
        while (true) {
            System.out.println("-----------Menu-----------");
            System.out.println("1. Add Student");
            System.out.println("2. Update Student");
            System.out.println("3. Delete Student");
            System.out.println("4. Sort Students by Score");
            System.out.println("5. Search Student by Name");
            System.out.println("6. Display All Students");
            System.out.println("7. Exit");
            System.out.println("--------------------------");
            System.out.print ("Enter your selection: ");
            int choice = scanner.nextInt();
            scanner.nextLine();
            switch (choice) {
                case 1:
                    System.out.print("Enter student ID: ");
                    int id = scanner.nextInt();
```

```
                    scanner.nextLine(); // Clear newline character
                    if (studentManagement.isIdDuplicate(id)) {
                        System.out.println("ID already exists. Please enter a unique
ID.");
                        break; // Do not add student if ID is duplicated
                    }
                    String name;
                    while (true) {
                        System.out.print("Enter student name: ");
                        name = scanner.nextLine();
                        if (name.matches("[a-zA-Z\\s]+")) {
                            break;
                        } else {
                            System.out.println("Invalid name. Please enter only
letters and spaces.");
                        }
                    }
                    System.out.print("Enter student scores: ");
                    double score = scanner.nextDouble();
                    scanner.nextLine(); // Clear newline character
                    studentManagement.addStudent(new Student(id, name, score));
                    System.out.println("Student added successfully.");
                    break;
                case 2:
                    System.out.print("Enter the ID of the student to update: ");
                    int updateId = scanner.nextInt();
                    scanner.nextLine();
                    System.out.print("Enter new name: ");
                    String newName = scanner.nextLine();
                    System.out.print("Enter new score: ");
                    double newMark = scanner.nextDouble();
                    scanner.nextLine();
                    studentManagement.updateStudent(updateId, newName, newMark);
                    System.out.println("Information updated successfully");
                    break;
                case 3:
                    System.out.print("Enter the ID of the student to delete: ");
                    int deleteId = scanner.nextInt();
                    scanner.nextLine();
                    studentManagement.deleteStudent(deleteId);
                    System.out.println("Delete information successfully");
                    break;
                case 4:
                    studentManagement.sortByScore();
                    break;
                case 5:
                    System.out.print("Enter the name of the student to search: ");
                    String searchName = scanner.nextLine();
                    studentManagement.searchStudentByName(searchName);
                    break;
                case 6:
                    studentManagement.displayStudents();
                    break;
                case 7:
                    System.out.println("See you again.");
                    scanner.close();
                    System.exit(0);
```

```
                    break;
                default:
                    System.out.println("Invalid choice. Please try again.");
            }
        }
    }
}
```

Class Student

```
package org.example;
class Student
{
    private int id;
    private String name;
    private double score;

    public Student(int id, String name, double score) {
        this.id = id;
        this.name = name;
        this.score = score;
    }
    public void setMarks(double score) {
        this.score = score;
    }
    public String getRanking() {
        if (score < 5.0) {
            return "Fail";
        } else if (score < 6.5) {
            return "Medium";
        } else if (score < 7.5) {
            return "Good";
        } else if (score < 9.0) {
            return "Very Good";
        } else {
            return "Excellent";
        }
    }
    public Student() {
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public double getMarks() {
        return score;
    }
```

```
    @Override
    public String toString() {
        return "Student {" + "id = " + id + ", name = " + name + ", score = " + score
+
                ", Ranking = " + getRanking()+ '}';
    }
}
```

Class StudentManagement

```
package org.example;
import java.util.Comparator;
import java.util.Stack;

public class StudentManagement {
    private StudentStack studentStack;

    public StudentManagement() {
        studentStack = new StudentStack();
    }
    public boolean isIdDuplicate(int id) {
        StudentStack.Node current = studentStack.getTop();
        while (current != null) {
            if (current.student.getId() == id) {
                return true; // ID already exists
            }
            current = current.next;
        }
        return false; // ID is not duplicated
    }

    // Add student to stack
    public void addStudent(Student student) {
        studentStack.push(student);
        displayStudents();
    }
    // Update student information by ID
    public void updateStudent(int id, String newName, double newMarks) {
        StudentStack tempStack = new StudentStack();
        boolean found = false;

        // Lấy sinh viên từ stack để kiểm tra ID
        while (!studentStack.isEmpty()) {
            Student student = studentStack.pop();
            if (student.getId() == id) {
                tempStack.push(new Student(id, newName, newMarks));
                found = true;
            } else {
                tempStack.push(student);
            }
        }

        // Đưa các sinh viên trở lại stack chính
        while (!tempStack.isEmpty()) {
            studentStack.push(tempStack.pop());
```

```
        }

        if (found) {
            System.out.println("Student with ID " + id + " updated successfully.");
        } else {
            System.out.println("No student found with ID: " + id);
        }

        // Hiển thị danh sách sinh viên sau khi cập nhật
        displayStudents();
    }

    // Delete student by ID
    public void deleteStudent(int id) {
        StudentStack tempStack = new StudentStack();

        while (!studentStack.isEmpty()) {
            Student student = studentStack.pop();
            if (student.getId() != id) {
                tempStack.push(student);
            }
        }
        while (!tempStack.isEmpty()) {
            studentStack.push(tempStack.pop());
        }
        System.out.println("Deleted student with ID: " + id);
        displayStudents();
    }
    // Sort students by score
    public void sortByScore() {
        if (studentStack.isEmpty()) {
            System.out.println("Stack is empty. No students to sort.");
            return;
        }

        StudentStack sortedStack = new StudentStack();
        while (!studentStack.isEmpty()) {
            Student temp = studentStack.pop();

            while (!sortedStack.isEmpty() && sortedStack.peek().getMarks() <
                    temp.getMarks()) {
                studentStack.push(sortedStack.pop());
            }
            sortedStack.push(temp);
        }

        while (!sortedStack.isEmpty()) {
            studentStack.push(sortedStack.pop());
        }
        System.out.println("Students sorted by score.");
        displayStudents();
    }
    // Find students by name
    public void searchStudentByName(String name) {
        StudentStack.Node current = studentStack.getTop();
        boolean found = false;
        while (current != null) {
```

```
            if (current.student.getName().equalsIgnoreCase(name)) {
                System.out.println("Found students: " + current.student);
                found = true;
            }
            current = current.next;
        }
        if (!found) {
            System.out.println("No student found with name: " + name);
        }
    }
    // Show all students
    public void displayStudents() {
        if (studentStack.isEmpty()) {
            System.out.println("There are no students.");
            return;
        }
        studentStack.displayStudents();
    }
}
```

Class StudentStack

```
package org.example;
public class StudentStack {
    Node top;
    // Node in stack
    static class Node {
        Student student;
        Node next;
        Node(Student student) {
            this.student = student;
            this.next = null;
        }
    }
    // Add student to stack
    public void push(Student student) {
        Node newNode = new Node(student);
        newNode.next = top;
        top = newNode;
    }
    // Get student from stack
    public Student pop() {
        if (isEmpty()) {
            throw new IllegalStateException("Stack is empty.");
        }
        Student student = top.student;
        top = top.next;
        return student;
    }
    // Check if stack is empty
    public boolean isEmpty() {
        return top == null;
    }
    // View top student
    public Student peek() {
```

```
        if (isEmpty()) {
            throw new IllegalStateException("Stack is empty.");
        }
        return top.student;
    }
    // Show all students
    public void displayStudents() {
        Node current = top;
        while (current != null) {
            System.out.println(current.student);
            current = current.next;
        }
    }
    public Node getTop() {
        return top;
    }
}
```

9.3. Algorithm

9.3.1. Design the Algorithm

Quick Sort

QuickSort is a valuable algorithm sorting algorithm, which is very effective when processing large data.

Advantages: High performance

Disadvantages: Performance

Steps:

- Select an element as the pivot (key).
- Divide the list into two parts based on the axis:
- Part smaller than the axis.
- Part larger than the axis.
- Call the QuickSort rule on each part.

9.3.2. Implement the Algorithm

Class QuickSort

```
package org.example;

import java.util.Collections;
import java.util.List;

class QuickSort {
    static void quickSort(List<Student> list, int low, int high, String byCriteria) {
        if (low < high) {
            int pi = partition(list, low, high, byCriteria);
            quickSort(list, low, pi - 1, byCriteria);
            quickSort(list, pi + 1, high, byCriteria);
        }
    }

    static int partition(List<Student> list, int low, int high, String byCriteria) {
        Student pivot = list.get(high);
        int i = low - 1;
        for (int j = low; j < high; j++) {
            if (byCriteria.equalsIgnoreCase("Score")) {
                if (list.get(j).getMarks() < pivot.getMarks()) {
                    i++;
                    Collections.swap(list, i, j);
                }
            } else if (byCriteria.equalsIgnoreCase("Name")) {
                if (list.get(j).getName().compareTo(pivot.getName()) < 0) {
                    i++;
                    Collections.swap(list, i, j);
                }
            }
        }
        Collections.swap(list, i + 1, high);
        return i + 1;
    }
}
```

Class Main

```
package org.example;

import java.util.ArrayList;
import java.util.List;

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

        List<Student> students = new ArrayList<>();
        students.add(new Student("1", "Bao",5));
        students.add(new Student("2", "Nam",6.0));
        students.add(new Student("3", "Linh",9.2));

        // Sort by Marks
```

```
        QuickSort.quickSort(students, 0, students.size() - 1, "Score");
        System.out.println("Sorted by Score:");
        students.forEach(System.out::println);

    }
}
```

9.4. Implement error handling and report test results.

9.4.1. Identify potential errors

Data entry error: The user entered invalid data (e.g., entered a letter in a field that requires a number). The score entered was not within a valid range (e.g., negative or greater than 10).

Empty list error: The user requested to sort, search, or display when the list had no students.

Duplicate ID error: The user entered a duplicate ID when adding students.

System error: An unexpected error during execution (e.g., null pointer exception).

9.4.2. Define error handling mechanisms

Input Check and Validation: Use logic checks to confirm valid input before processing.

Exception Handling: Catch and handle exceptions using try-catch.

User Alert: Print clear error messages to let users know what the problem is and how to fix it.

List Check: Verify that the student list is not empty before performing operations.

9.4.3. Implement error handling code

```
package org.example;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;
public class ErrorHandlingExample {
    private static final List<Student> students = new ArrayList<>();
    private static final Scanner scanner = new Scanner(System.in);

    public static void addStudent() {
        try {
            System.out.print("Enter ID: ");
            int id = scanner.nextInt();
            scanner.nextLine(); //Consume newline

            // Check for duplicate ID
            for (Student student : students) {
                if (student.getId() == id) {
                    System.out.println("Error: Student with ID " + id + " already
exists.");
```

```
                    return;
                }
            }
            System.out.print("Enter name: ");
            String name = scanner.nextLine();

            System.out.print("Enter marks: ");
            double marks = scanner.nextDouble();
            scanner.nextLine(); //Consume newline
            // Validate marks
            if (marks < 0 || marks > 10) {
                System.out.println("Error: Marks should be between 0 and 10.");
                return;
            }
            students.add(new Student(id, name, marks));
            System.out.println("Student added successfully.");
        } catch (Exception e) {
            System.out.println("Error: Invalid input. Please try again.");
            scanner.nextLine(); // Clear invalid input
        }
    }
    public static void displayStudents() {
        if (students.isEmpty()) {
            System.out.println("No students in the list.");
            return;
        }
        for (Student student : students) {
            System.out.println(student);
        }
    }
    public static void main(String[] args) {
        while (true) {
            System.out.println("\nMenu:");
            System.out.println("1. Add Student");
            System.out.println("2. Display Students");
            System.out.println("3. Exit");
            System.out.print("Enter your choice: ");
            int choice = scanner.nextInt();
            scanner.nextLine(); //Consume newline
            switch (choice) {
                case 1 -> addStudent();
                case 2 -> displayStudents();
                case 3 -> {
                    System.out.println("Exiting program.");
                    return;
                }
                default -> System.out.println("Error: Invalid choice. Please select a
valid option.");
            }
        }
    }
}
```

9.4.4. Test the error handling code

Test ID	Test Scenario	Input	Expected Outcome	Actual Outcome	Status
T01	**Entering marks outside the valid range**	**-5, 11**	**Display error: "Marks should be between 0 and 10."**	**Correct**	**Passed**
T02	**Entering a duplicate ID**	**Enter an existing ID**	**Display error: "Student with ID already exists."**	**Correct**	**Passed**
T03	**Displaying the list when it is empty**	**No data available**	**Display message: "No students in the list."**	**Correct**	**Passed**
T04	**Entering an invalid ID format (e.g., text instead of numbers)**	**Enter abc as ID**	**Display error: "Invalid input. Please try again."**	**Correct**	**Passed**
T05	**Entering an invalid marks format (e.g., text instead of numbers)**	**Enter xyz as marks**	**Display error: "Invalid input. Please try again."**	**Correct**	**Passed**
T06	**Performing an operation when the list is empty**	**Sort or display**	**Display error: "No students in the list."**	**Correct**	**Passed**
T07	**Entering valid data and performing operations**	**ID: 1, Name: John, Marks: 9**	**Successfully add the student, correctly display the list**	**Correct**	**Passed**

9.4.5. Report test results

Test Case ID	Test Case Description	Expected Outcome	Actual Outcome	Pass/Fail	Remarks
TC01	Test valid student data input	System should accept student information (ID, name, marks)	System accepted valid input	Pass	Input data was correct and accepted
TC02	Test invalid student ID (e.g., negative ID)	System should reject invalid ID	System rejected negative ID	Pass	Validation of student ID works
TC03	Test sorting functionality (Bubble Sort)	Students should be sorted by marks	Students sorted correctly	Pass	Sorting logic works as expected
TC04	Test student deletion by ID	Student with matching ID should be deleted	Student was deleted correctly	Pass	Deletion works fine
TC05	Test search functionality by name	Should return student details for the name entered	Correct student details returned	Pass	Search functionality works correctly
TC06	Test empty list scenario	System should handle empty list gracefully	System handled empty list	Pass	No errors when list is empty
TC07	Test invalid marks (negative marks)	System should reject negative marks	System rejected negative marks	Pass	Validation of marks works correctly

Key Insights from Test Results:

- All core functionalities such as adding, updating, deleting, and searching for students have passed the test cases.
- Edge cases such as invalid IDs and negative marks are correctly handled by the system.
- Sorting functionality works as expected, ensuring students are listed in the correct order based on their marks.
- No critical failures were found during testing, and all features are working as expected.

9.4.6. Analyse and fix any issues

1. Issue Identification

During the testing of the search functionality (TC05), we encountered an issue where the search function failed to return results for students with similar names (e.g., "John Smith" and "Johnny Smith").

2. Analyzing the Problem

The problem appears to be caused by how the system compares student names. The system uses an exact match comparison (equals() method), which fails when there are small variations in the name.

3. Solution

Modify the search functionality to use a case-insensitive comparison. This can be done by using the equalsIgnoreCase() method to ensure that the search is not case-sensitive.

Additionally, introduce a substring search to allow partial matches in names (e.g., search for "John" to find both "John Smith" and "Johnny Smith").

4. Code Fix

```
public void searchStudentByName(String name) {
    Node current = studentStack.top;
    boolean found = false;
    while (current != null) {
        if (current.student.getName().toLowerCase().contains(name.toLowerCase())) {
            System.out.println("Found student: " + current.student);
            found = true;
        }
        current = current.next;
    }
    if (!found) {
        System.out.println("No student found with name: " + name);
    }
}
```

5. Retesting After Fix

After applying the fix, we rerun the search test (TC05) with several variations of names ("John Smith", "Johnny Smith", etc.). The test now passes successfully, and the search functionality returns the correct results for both full and partial matches.

9.5. Demonstrate how the implementation of an ADT/algorithm solves a well-defined problem.

1, Define the ADT for a Student

An ADT is a data structure that defines the types of data it holds and the operations that can be performed on it. In this case, the Student ADT will include:

Attributes: ID, name, marks.

Operations: Set and get methods for each attribute, a method for ranking students based on marks.

```
package org.example;
class Student
{
    private int id;
    private String name;
    private double score;

    public Student(int id, String name, double score) {
        this.id = id;
        this.name = name;
        this.score = score;
    }
    public void setMarks(double score) {
        this.score = score;
    }
    public String getRanking() {
        if (score < 5.0) {
            return "Fail";
        } else if (score < 6.5) {
            return "Medium";
        } else if (score < 7.5) {
            return "Good";
        } else if (score < 9.0) {
            return "Very Good";
        } else {
            return "Excellent";
        }
    }
    public Student() {
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
```

```
    }
    public double getMarks() {
        return score;
    }
    @Override
    public String toString() {
        return "Student {" + "id = " + id + ", name = " + name + ", score = " + score
+
                ", Ranking = " + getRanking()+ '}';
    }
}
```

2, Operations on the ADT:

We need to implement the following operations on the student data:

- Add: Add a student to the list.
- Edit: Edit a student's details (name, marks).
- Delete: Remove a student from the list.
- Sort: Sort students based on their marks.
- Search: Search for a student by name.

```
package org.example;
import java.util.Comparator;
import java.util.Stack;

public class StudentManagement {
    private StudentStack studentStack;

    public StudentManagement() {
        studentStack = new StudentStack();
    }
    public boolean isIdDuplicate(int id) {
        StudentStack.Node current = studentStack.getTop();
        while (current != null) {
            if (current.student.getId() == id) {
                return true; // ID already exists
            }
            current = current.next;
        }
        return false; // ID is not duplicated
    }

    // Add student to stack
    public void addStudent(Student student) {
        studentStack.push(student);
        displayStudents();
    }
    // Update student information by ID
    public void updateStudent(int id, String newName, double newMarks) {
        StudentStack tempStack = new StudentStack();
        boolean found = false;

        // Lấy sinh viên từ stack để kiểm tra ID
        while (!studentStack.isEmpty()) {
```

```
            Student student = studentStack.pop();
            if (student.getId() == id) {
                tempStack.push(new Student(id, newName, newMarks));
                found = true;
            } else {
                tempStack.push(student);
            }
        }

        // Đưa các sinh viên trở lại stack chính
        while (!tempStack.isEmpty()) {
            studentStack.push(tempStack.pop());
        }

        if (found) {
            System.out.println("Student with ID " + id + " updated successfully.");
        } else {
            System.out.println("No student found with ID: " + id);
        }

        // Hiển thị danh sách sinh viên sau khi cập nhật
        displayStudents();
    }

    // Delete student by ID
    public void deleteStudent(int id) {
        StudentStack tempStack = new StudentStack();

        while (!studentStack.isEmpty()) {
            Student student = studentStack.pop();
            if (student.getId() != id) {
                tempStack.push(student);
            }
        }
        while (!tempStack.isEmpty()) {
            studentStack.push(tempStack.pop());
        }
        System.out.println("Deleted student with ID: " + id);
        displayStudents();
    }
    // Sort students by score
    public void sortByScore() {
        if (studentStack.isEmpty()) {
            System.out.println("Stack is empty. No students to sort.");
            return;
        }

        StudentStack sortedStack = new StudentStack();
        while (!studentStack.isEmpty()) {
            Student temp = studentStack.pop();

            while (!sortedStack.isEmpty() && sortedStack.peek().getMarks() <
                    temp.getMarks()) {
                studentStack.push(sortedStack.pop());
            }
            sortedStack.push(temp);
        }
```

```
        while (!sortedStack.isEmpty()) {
            studentStack.push(sortedStack.pop());
        }
        System.out.println("Students sorted by score.");
        displayStudents();
    }
    // Find students by name
    public void searchStudentByName(String name) {
        StudentStack.Node current = studentStack.getTop();
        boolean found = false;
        while (current != null) {
            if (current.student.getName().equalsIgnoreCase(name)) {
                System.out.println("Found students: " + current.student);
                found = true;
            }
            current = current.next;
        }
        if (!found) {
            System.out.println("No student found with name: " + name);
        }
    }
    // Show all students
    public void displayStudents() {
        if (studentStack.isEmpty()) {
            System.out.println("There are no students.");
            return;
        }
        studentStack.displayStudents();
    }
}
```

3, Demonstrating the Algorithm:

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

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        StudentManagement studentManagement = new StudentManagement();
        while (true) {
            System.out.println("-----------Menu----------");
            System.out.println("1. Add Student");
            System.out.println("2. Update Student");
            System.out.println("3. Delete Student");
            System.out.println("4. Sort Students by Score");
            System.out.println("5. Search Student by Name");
            System.out.println("6. Display All Students");
            System.out.println("7. Exit");
            System.out.println("-------------------------");
            System.out.print ("Enter your selection: ");
            int choice = scanner.nextInt();
            scanner.nextLine();
            switch (choice) {
```

```
                case 1:
                    System.out.print("Enter student ID: ");
                    int id = scanner.nextInt();
                    scanner.nextLine(); // Clear newline character
                    if (studentManagement.isIdDuplicate(id)) {
                        System.out.println("ID already exists. Please enter a unique
ID.");
                        break; // Do not add student if ID is duplicated
                    }
                    String name;
                    while (true) {
                        System.out.print("Enter student name: ");
                        name = scanner.nextLine();
                        if (name.matches("[a-zA-Z\\s]+")) {
                            break;
                        } else {
                            System.out.println("Invalid name. Please enter only
letters and spaces.");
                        }
                    }
                    System.out.print("Enter student scores: ");
                    double score = scanner.nextDouble();
                    scanner.nextLine(); // Clear newline character
                    studentManagement.addStudent(new Student(id, name, score));
                    System.out.println("Student added successfully.");
                    break;
                case 2:
                    System.out.print("Enter the ID of the student to update: ");
                    int updateId = scanner.nextInt();
                    scanner.nextLine();
                    System.out.print("Enter new name: ");
                    String newName = scanner.nextLine();
                    System.out.print("Enter new score: ");
                    double newMark = scanner.nextDouble();
                    scanner.nextLine();
                    studentManagement.updateStudent(updateId, newName, newMark);
                    System.out.println("Information updated successfully");
                    break;
                case 3:
                    System.out.print("Enter the ID of the student to delete: ");
                    int deleteId = scanner.nextInt();
                    scanner.nextLine();
                    studentManagement.deleteStudent(deleteId);
                    System.out.println("Delete information successfully");
                    break;
                case 4:
                    studentManagement.sortByScore();
                    break;
                case 5:
                    System.out.print("Enter the name of the student to search: ");
                    String searchName = scanner.nextLine();
                    studentManagement.searchStudentByName(searchName);
                    break;
                case 6:
                    studentManagement.displayStudents();
                    break;
                case 7:
```

```
                    System.out.println("See you again.");
                    scanner.close();
                    System.exit(0);
                    break;
                default:
                    System.out.println("Invalid choice. Please try again.");
            }
        }
    }
}
```

9.6. Critically evaluate the complexity of an implemented ADT/algorithm.

Assessing the complexity of an Abstract Data Type (ADT) or an algorithm requires an analysis of both time complexity and space complexity. This evaluation is crucial for understanding the efficiency of the implementation, particularly in scenarios involving large datasets or applications where performance is paramount. To illustrate this, we will conduct a thorough examination of these complexities in the context of a sorting algorithm applied to a list of students ranked by their marks.

1. Bubble Sort

Time complexity

- Best case: $O(n)$ When the array is sorted, Bubble Sort only needs one pass, no swapping.
- Average case: $O(n2)$ When the array is not sorted, each element must be compared with all the remaining elements.
- Worst case: $O(n2)$ When the array is in a completely reversed state, the maximum number of comparisons and swaps is required.

Space complexity

- Memory used: $O(1)$ Operates directly on the array, no additional storage space is required.

2. Quick Sort

Time Complexity

- Best case: $O(n\log n)$ Occurs when the pivot element splits the array into two approximately equal parts.
- Average case: $O(n\log n)$ With random distribution, choosing a balanced pivot is common.
- Worst case: $O(n2)$ Occurs when the pivot is the largest or smallest element, resulting in an unbalanced array split.

Space complexity

- Memory used: $O(\log n)$ for best or average case.
- Worst case: $O(n)$, occurs when the split is unbalanced, increasing the recursion depth.

Comparison Table

Criteria	Bubble Sort	Quick Sort
Best-Case Time	O(n)	O(nlogn)
Average-Case Time	O(n2)	O(nlogn)
Worst-Case Time	O(n2)	O(n2)
Best-Case Space	O(1)	O(logn)
Efficiency	Small datasets	Large datasets
Stability	Stable	Unstable

9.7. Discuss how asymptotic analysis can be used to assess the effectiveness of an algorithm.

Asymptotic analysis is a method used to evaluate the efficiency of an algorithm in terms of its behavior as the input size grows large. This analysis focuses on how an algorithm's time or space requirements increase as the input size increases, allowing for the comparison of algorithms in a way that abstracts away from machine-specific details like processor speed or memory capacity. Asymptotic analysis is primarily concerned with time complexity and space complexity, providing insights into the performance and scalability of an algorithm under different conditions.

1. Important Notations

Big-O (O): Represents an upper bound, describing the worst case or maximum growth rate of the algorithm.

For example: $O(n2)$ for Bubble Sort.

Omega (Ω): Represents a lower bound, describing the best case or minimum growth rate of the algorithm.

For example: $\Omega(n\log n)$ for Quick Sort.Theta (Θ): Represents the exact growth rate of the algorithm, applicable when both the upper and lower bounds match.

For example: $\Theta(n\log n)$ for the average case of Merge Sort.

2. Application in algorithm evaluation

Compare growth rates:

Asymptotic analysis helps determine which algorithm is more efficient as data size increases, for example: $O(n)$ is more efficient than $O(n2)$ when n is large.

Evaluate best, average, worst case:

Ensure the algorithm has performance that meets the requirements in different cases.

For example: Quick Sort has $O(n2)$ in the worst case but the average is $O(n\log n)$

3. Steps to perform analysis

Identify key operations:

Find the operations that contribute the most to the running time (compare, swap, access data).

Loop analysis:

Determine the number of iterations to calculate the number of operations performed.

Recursive analysis (if applicable):

Use recursive formulas or Master's theorem to analyze divide-and-conquer algorithms.

4. Why is asymptotic analysis important?

Performance prediction:

Provides an overview of how an algorithm works without actually implementing it.

Scalability assessment:

Ensures that the algorithm is suitable for large datasets without significantly increasing resource costs.

Optimal algorithm selection:

Use analysis to select the most appropriate algorithm for a specific problem.

9.8. Determine two ways in which the efficiency of an algorithm can be measured, illustrating your answer with an example.

9.8.1. Time Complexity and example

Time complexity is an important concept in computer science that measures the amount of time an algorithm takes to complete its task, depending on the size of the input. Time complexity helps determine the performance of an algorithm as the size of the data increases.

- Benefit: Time complexity provides an estimate of the computational cost of an algorithm, helping in selecting the most efficient algorithm for a given problem
- Time Complexity: The time complexity of linear search is O(n), where n is the number of elements in the array. As the array size grows, the time to perform the search increases linearly.

9.8.2. Space Complexity and example

Space Complexity refers to the amount of memory (or space) an algorithm needs to run, depending on the size of the input. It is a critical measure of the efficiency of an algorithm, especially when dealing with large data sets.

Space complexity is commonly represented using Big-O notation, which shows how the memory requirement grows as the input size increases.

Benefit: Space complexity helps optimize memory usage, which is critical in systems with limited memory resources.

Space Complexity: The space complexity of this algorithm is O(n), where n is the number of elements in the input array. The function creates a new array of the same size as the input array, resulting in additional memory usage.

9.8.3. Interpret what a trade-off is when specifying an ADT,

A trade-off in the context of Abstract Data Types (ADT) refers to the balancing of different factors such as time complexity, space complexity, ease of implementation, and usability when designing and selecting an ADT for a particular problem. In other words, a trade-off occurs when one property (e.g., time efficiency) is improved at the cost of another property (e.g., space usage), or when a decision is made between multiple design choices based on priorities and constraints.

When specifying an ADT, developers often face the need to choose between competing objectives. These choices are dependent on the requirements of the application and the environment where the ADT will be used. The trade-offs must be carefully evaluated to ensure the ADT meets the needs of the system in the most efficient and effective way possible.

Examples of Trade-offs in ADT Design

1. Time vs. Space Complexity:

Optimizing for Time Complexity: Some ADTs might be designed to provide faster operations (e.g., O(1) access time for a HashMap) but require more memory to store additional information (like hash buckets or linked lists).

Optimizing for Space Complexity: Other ADTs might use less memory but have slower operations (e.g., using a linked list to implement a queue instead of an array, which might lead to O(n) traversal time for searching).

Trade-off: In some cases, you might choose an algorithm or data structure with faster processing times but at the expense of consuming more memory. For example, an array might be faster for random access (constant time, O(1)) but requires knowing the array size in advance, whereas a linked list uses more space due to node pointers but allows dynamic memory allocation.

2. Simple Design vs. Flexibility:

Simpler Design: A simpler ADT implementation (e.g., a static array) is often easier to implement and manage but may lack flexibility (e.g., size limitations in arrays).

More Complex Design: A more complex ADT (e.g., a balanced tree or graph) might allow for more flexible operations (such as dynamic resizing or efficient searching), but it requires more complicated logic and might be harder to maintain.

Trade-off: Sometimes a more complex ADT may provide a greater degree of functionality and flexibility (like a red-black tree for ordered data), but it comes with more overhead in terms of implementation, debugging, and maintaining the code.

3. Ease of Use vs. Performance:

Ease of Use: Some ADTs are easier for developers to use and understand, like Java's ArrayList, which abstracts away complexities like resizing and memory allocation.

Performance: ADTs that are optimized for performance might expose more of their internal structure, making them harder to use but providing faster or more efficient operations. For instance, a hash table might require manual collision resolution, which complicates its usage but can provide faster lookups than a simple list search.

Trade-off: Choosing between ease of use and performance often depends on the application's context. For a simple application, an easy-to-use ADT like an ArrayList may be sufficient, but for a performance-critical application, developers might opt for a more complex structure like a hash map or tree, accepting the added complexity for the performance gain.

4. General-purpose ADT vs. Specialized ADT:

General-purpose ADT: A general-purpose ADT (e.g., a generic list or queue) is designed to be reusable across a variety of contexts. It provides broad functionality but might not be optimized for any particular use case.

Specialized ADT: A specialized ADT, on the other hand, is designed for a specific problem domain (e.g., a priority queue used for scheduling). While it may be highly optimized for a specific task, it lacks the general-purpose flexibility.

Trade-off: A general-purpose ADT is more versatile and reusable but may not be as efficient in specialized scenarios. A specialized A

9.9. Evaluate three benefits of using implementation independent data structures.

9.9.1. Portability

Concept: Portability refers to the ease with which source code can be moved and used across different platforms without extensive modifications. When data structures are implementation independent, they are not dependent on the factors of a particular deployment environment (such as the operating system, hardware, or programming language).

Benefits:

Data structures can be easily moved between different systems without changing the way they work. For example, an application that uses linked lists or binary trees can be moved from a Windows environment to a Linux environment without experiencing problems with the data structures.

Portability increases the scalability of software when it needs to be deployed across different platforms, reducing the cost and effort required to adapt to each specific platform.

9.9.2. Reusability

Concept: Reusability is the ability to reuse code or data structures that have been developed without much change. When data structures are implemented independently, they can be reused in other projects or software without compatibility issues or having to be re-implemented from scratch.

Benefits:

Independently implemented data structures can be reused in many different applications, helping to reduce code duplication. For example, a data structure such as a Queue or Stack can be reused in many different projects without having to be built from scratch.

Reusing data structures saves time and effort in software development, and reduces the risk of errors when having to write new code for each project.

9.9.3. Maintainability

Concept: Maintainability refers to the ability to modify, upgrade, and extend software without affecting existing functionality. When data structures are implemented independently, maintaining and upgrading the data structure becomes easier because it is not dependent on the specifics of the application.

Benefits:

Software maintenance becomes easier when data structures are separated from the main application logic. If you need to change or optimize a data structure (e.g., switching from a simple linked list to a doubly linked list), you only need to modify the data structure without changing other parts of the application.

This helps reduce confusion in the source code and makes it easier to extend and change the software as required without affecting other parts of the system.

Conclusions

In summary, **trade-offs in ADT specification** are about making informed decisions based on competing priorities, such as performance (time complexity), memory usage (space complexity), implementation complexity, and functionality. It is crucial to evaluate the specific requirements of the application, as well as the constraints and limitations of the environment, to choose the right balance between these factors. Understanding and managing these trade-offs effectively can lead to more efficient and scalable software systems.

References

What is Configuration Data? Definition and Types(AppMaster®, 09/13/2023)

https://appmaster.io/vi/blog/dinh-nghia-va-kieu-cau-truc-du-lieu-la-gi

What is Data Protection?(LacHong, October 21, 2022)

https://lachongtech.vn/data-protection-la-gi/

Dijkstra's Algorithm: Find the Shortest Path (Nguyen Tuan's Blog, September 11, 2021)

https://chidokun.github.io/2021/09/dijkstra-algorithm/

Prim's Algorithm (Buzz, 10/1/2021)

https://mytour.vn/vi/blog/bai-viet/thuat-toan-prim.html

Bubble sort algorithm (VietTus, May 3, 2023)

https://viettuts.vn/cau-truc-du-lieu-va-giai-thuat/giai-thuat-sap-xep-noi-bot-bubble-sort

What is Quick Sort?(ITNavi, 15 Dec 2022)

https://itnavi.com.vn/blog/quick-sort-la-gi

Slide
https://www.canva.com/design/DAGYRQdcr2I/F4Ws-XTSrwaUcolpxq-yMQ/edit?utm_content=DAGYRQdcr2I&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton

Github

https://github.com/trBao04/ASM_PART2_TRANGIABAO_BH00998.git