APPLICATION OF ABSTRACT DATA TYPES (ADTS) IN DEVELOPING STUDENT MANAGEMENT SYSTEM

Objective & Scope

Objective:

- Explain the role of ADTs in software design.
- Illustrate through Student Management System: add, edit, delete, sort, search.
- Evaluate and compare the effectiveness of algorithms, data structures.

Scope:

- Manage student information (ID, Name, Score, Ranking).
- Functions: Add, Edit, Delete, Sort by score, Search by ID, Save & Load data.

Introduction to Abstract Data Types (ADTs)

What are ADTs?

Definition: Describes data and operations on data without worrying about implementation details.

Benefits:

- Separate interface and implementation.
- Easy to change internally without affecting the source code using ADT.
- Increased flexibility, reuse, easy to maintain.

ADTs & Applied Data Structures

ADTs used:

- List

Illustrative data structure:

- `ArrayList` to store the list of students, convenient for sorting AND access students by ID.

Real Problem & System Requirements

Context: Managing a class with a changing number of students.

Data:

Student ID (unique), Name, Score (0-10), Rating (Fail → Excellent)

Functions:

- Add, Edit, Delete students
- Sort by score
- Search students by ID
- Save & load data for long-term use

System Overview Architecture

System architecture diagram:

- Class `Student`: Represents student information (ID, Name, Score).
- Class `StudentStack`: Manages student stacks (push, pop, peek).
- Class `Student Management`: Coordinates student management operations (add, edit, delete, sort, search).
- Class `Main`: User interface commands.

Amount of operations:

- User input command → `Main` calls `StudentManager` → `StudentManager` interacts with `StudentStack`.

Student Class - Structure & Function

Attributes:
- `id` (String): Student code, unique.
- `name` (String): Full name of student.
- `marks` (double): Student score.

Methods:
- `getId()`, `getName()`, `getMarks()`
- `setName(String name)`, `setMarks(double quotes)`
- `getRanking()`: Calculates the type based on the score.

Constraints:
- ID is not empty.
- Score from 0 to 10.

```java
private String id;
private String name;
private double marks;

public Student(String id, String name, double marks) {
    this.id = id;
    this.name = name;
    this.marks = marks;
}

public String getId() { return id; }

public String getName() { return name; }

public double getMarks() { return marks; }

public void setMarks(double marks) {
    this.marks = marks;
}

public String getRanking(){
    if (marks < 5) return "Fail";
    else if (marks < 6.5) return "Medium";
    else if (marks < 7.5) return "Good";
    else if (marks < 9.0) return "Very Good";
    else return "Excellent";
}
```

StudentsStack Class - Managing Student Rankings

Properties:
- `maxSize` (int): Maximum size of the stack.
- `stackArray` (Student[]): Table containing students.
- `top` (int): Index of the top element in the stack.

Methods:
- `push(Student)`: Add students to the stack.
- `pop()`: Removes and returns student members.
- `peek()`: Views the top student without skipping.
- `isEmpty()`: Checks if the classification is empty.
- `isFull()`: Checks if the traf is full.
- `size()`: Returns the number of students in the stack.
- `clear()`: Clears all students in the stack.

```java
public class StudentStack {

    int maxSize;
    private static Student[] stackArray;
    private static int top;

    public StudentStack(int size) {
        this.maxSize = size;
        this.stackArray = new Student[maxSize];
        this.top = -1; // stack is initially empty
    }

    public void push(Student student){

        if(top < maxSize - 1){
            stackArray[++top] = student;
        }
        else {
            System.out.println("Stack is full. Cannot add more students");
        }
    }
   public Student pop(){
       if (top >= 0){
           return stackArray[top--];
       }else {
           System.out.println("Stack is empty. Cannot pop.");
           return null;
       }
   }

   public Student peek(){
       if (top >= 0){
           return stackArray[top];
       }else {
           System.out.println("Stack is empty. Nothing to peek.");
           return null;
       }
   }

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

   public int size() { return top + 1; }
}
```

Student Management Class - Coordinating Student Management Operations

Properties: `studentStack` (StudentStack): Student management solution.
Methods:
`addStudent(Student)`: Adds a student to the stack.
`removeStudent()`: Removes a student from the top of the stack.
`updateStudent(String id, double newMark)`: Updates a student's score by ID.
`searchStudent(String id)`: Searches for students by ID.
`sortStudents()`: Sorts students by score.
`displayStudents()`: Displays the list of students.

```java
public void displayStudent(){
    Student[] tempArr = new Student[studentStack.size()];
    int count = 0;

    while (!studentStack.isEmpty()){
        Student student = studentStack.pop();
        System.out.println(student);
        tempArr[count++] = student;
    }

    for (int i = count - 1; i >= 0; i--){
        studentStack.push(tempArr[i]);
    }
}

public boolean isEmpty() {
    return studentStack.isEmpty();
}
```

```java
public class StudentManagement {

    private StudentStack studentStack;

    public StudentManagement(int size) {
        studentStack = new StudentStack(size);
    }



    public void addStudent(Student student) {
        if (studentStack.size() >= studentStack.maxSize) {
            System.out.println("Stack is full. Cannot add more students.");
            return; // Exit without adding the student
        }
        studentStack.push(student);
    }

    public Student removeStudent() {
        if (studentStack.isEmpty()) {
            System.out.println("No student to remove. The stack is empty.");
            return null;
        }
        return studentStack.pop();
    }
```

```java
public void updateStudent(String id, double newMark) {
    // temporary store students in an array
    boolean found = false;
    Student[] tempArr = new Student[studentStack.size()];
    int count = 0;

    while (!studentStack.isEmpty()) {
        Student student = studentStack.pop();
        if (student.getId().equals(id)) {
            student.setMarks(newMark); // update mark
            found = true;
        }
        tempArr[count++] = student; // store student in tempArr
    }

    for (int i = count - 1; i >= 0; i--) {
        studentStack.push(tempArr[i]);
    }

    if (found) {
        System.out.println("Student marks updated successfully");
    } else {
        System.out.println("Student with ID " + id + " not found.");
    }
}
```

```java
public Student searchStudent(String id) {
    Student[] tempArr = new Student[studentStack.size()];
    int count = 0;

    while (!studentStack.isEmpty()) {
        Student student = studentStack.pop();
        if (student.getId().equals(id)) {
            // Found the student, push it back to the stack and return it
            for (int i = count - 1; i >= 0; i--) {
                studentStack.push(tempArr[i]);
            }
            return student; // Return the found student
        }
        tempArr[count++] = student; // Store student in tempArr
    }

    // If not found, restore all students back to the stack
    for (int i = count - 1; i >= 0; i--) {
        studentStack.push(tempArr[i]);
    }

    System.out.println("Student with ID " + id + " not found.");
    return null;
}
```

Functions: Add, Edit, Delete Students

Add Student:
- Check for incomplete classification.
- Use `push()` to add students.

Edit Student:
- Search for students by ID.
- Update scores if found.

Delete Student:
- Use `pop()` to remove students.
- Notify if not sorted empty.

Process operation:
- Call the corresponding method in `Student Management`.
- Notify the user of the result.

```java
public void addStudent(Student student) {
    if (studentStack.size() >= studentStack.maxSize) {
        System.out.println("Stack is full. Cannot add more students.");
        return; // Exit without adding the student
    }
    studentStack.push(student);
}

public Student removeStudent() {
    if (studentStack.isEmpty()) {
        System.out.println("No student to remove. The stack is empty.");
        return null;
    }
    return studentStack.pop();
}
```

```java
public void updateStudent(String id, double newMark) {
    // temporary store students in an array
    boolean found = false;
    Student[] tempArr = new Student[studentStack.size()];
    int count = 0;

    while (!studentStack.isEmpty()) {
        Student student = studentStack.pop();
        if (student.getId().equals(id)) {
            student.setMarks(newMark); // update mark
            found = true;
        }
        tempArr[count++] = student; // store student in tempArr
    }

    for (int i = count - 1; i >= 0; i--) {
        studentStack.push(tempArr[i]);
    }

    if (found) {
        System.out.println("Student marks updated successfully");
    } else {
        System.out.println("Student with ID " + id + " not found.");
    }
}
```


```java
public Student removeStudent() {
    if (studentStack.isEmpty()) {
        System.out.println("No student to remove. The stack is empty.");
        return null;
    }
    return studentStack.pop();
}
```

Big O Theory & Notation Analysis

Asymptotic Analysis (Asymptotic Analysis): Evaluate the performance of the algorithm when increasing the size of data (N).

Big O Notation: Describe the rate of increase or space of time for the input size.
Examples: $O(1)$, $O(n)$, $O(n \log n)$, $O(n^2)$
Meaning: Helps choose the optimal solution when the data is large.

Complexity of Functions in the System

Add Student: Check incomplete classification: O(1)

- Add to stack: O(1)

Edit Student:

- Search member: O(n)
- Update number: O(1)

Delete Student:

- Remove top member: O(1)

Sort Biome:

- Use Quick Sort: O(n log n)

Search Student:

- Search by ID: O(n)

Real-world Performance Testing (Benchmark)

Purpose: Measure real-world time instead of just theoretical.
Method:
- Randomly generate 10,000 students.
- Time the execution of Bubble Sort and Quick Sort.

Expected results:
- Quick Sort is significantly faster than Bubble Sort when N is large.

```
/Library/Java/JavaVirtualMachines/jdk-21.jdk/Contents/Home/bin,
Bubble Sort Execution Time: 422015077 nanoseconds
Quick Sort Execution Time: 3024296 nanoseconds
```

```java
public void measureSortTime() {
    int numStudents = 10000;  // Ví dụ số lượng học sinh

    // Tạo sinh viên ngẫu nhiên
    Student[] students = generateRandomStudents(numStudents);

    // Đo thời gian sắp xếp Bubble Sort
    long bubbleStartTime = System.nanoTime();
    bubbleSort(students);  // Chạy Bubble Sort
    long bubbleEndTime = System.nanoTime();
    long bubbleSortDuration = bubbleEndTime - bubbleStartTime;

    // Tạo lại sinh viên ngẫu nhiên cho Quick Sort
    students = generateRandomStudents(numStudents);

    // Đo thời gian sắp xếp Quick Sort
    long quickStartTime = System.nanoTime();
    quickSort(students, low: 0, high: students.length - 1);  // Chạy Quick Sort
    long quickEndTime = System.nanoTime();
    long quickSortDuration = quickEndTime - quickStartTime;

    // Hiển thị kết quả
    System.out.println("Bubble Sort Execution Time: " + bubbleSortDuration + " nanoseconds");
    System.out.println("Quick Sort Execution Time: " + quickSortDuration + " nanoseconds");
}

 // Tạo sinh viên ngẫu nhiên cho học sinh
 private Student[] generateRandomStudents(int numStudents) {
     Random random = new Random();
     Student[] students = new Student[numStudents];
     for (int i = 0; i < numStudents; i++) {
         double marks = 0 + (10 - 0) * random.nextDouble();  // Điểm số từ 0 đến 10
         students[i] = new Student( id: "ID" + i,  name: "Student" + i, marks);
     }
     return students;
 }
```

Benchmark Results - Comparing Bubble Sort & Quick Sort

Actual Results:

- Bubble Sort: 422 milliseconds
- Quick Sort: 30 milliseconds

Comments: Quick Sort is much faster than Bubble Sort.

Conclusion: Choosing Quick Sort to sort students is the optimal type.

```
Bubble Sort Execution Time: 422015077 nanoseconds
Quick Sort Execution Time: 3024296 nanoseconds
```

Balancing Resources & System Performance

Memory (Cache):
`StudentStack` uses continuous memory in the array.
Size limit.
CPU & Runtime:
- Optimal time processing of Quick Sort.
- Quick addition and deletion operations.

Can be used:
- Consumes memory to ensure performance.
- Suitable for current data.

Advice:
- Increase size if needed.
- Consider using other structured data if increasing capacity.

Testing & Debugging Strategy

Important testing scope:
- Ensure the system is working correctly.
- Detect and fix errors before developing the declaration.

Type checking:
- Unit testing (Unit testing): Test each individual method.
- Validity checking (Integration testing): Test the interaction between classes and components.
- System testing (System testing): Test the entire system under real-world conditions.

Debugging method:
- Use debug tool in IDE.
- Add message log to trace the execution of this flow.
- Check input data validation.

Handle user error:
- Check input data validity.
- Clear message and error correction user guide.

Using Data Structure Library (DSL): Welding & Manual Pressing

What is a data structure library (DSL)?
- A collection of classes and methods for managing data.
Advantages of DSL:
- Saves development time.
- Tested and optimized.
DSL mode:
- Limited in customization.
- Performance may not be optimal for all cases.
Reasons for manual selection:
- Full control over configuration data.
- Performance optimization for specific operations.
- Flexibility in extension and change.
Example in project:
- Use `ArrayList` for lists students.
- Using `StudentStack` to manage student stacks.

Algorithm Efficiency: Evaluation & Benchmarking

Target value: Compare the performance of sorting algorithms.
Algorithms used: Bubble sort: Simple but efficient for large data.
Quick sort: More efficient than O(n log n) complexity.
Benchmark of the method:
- Randomly generate 10,000 students with numbers from 0 to 10.
- Execute and measure the time of both algorithms.
Result:
- Quick Sort is significantly faster than Bubble Sort.
Comments:
- Choose the appropriate algorithm to improve the system efficiency.

Alternative Algorithm & Performance Evaluation

Output alternative algorithm:
Merge Sort: O(n log n) complexity, more stable than Quick Sort.
Reasons for choosing Merge Sort:
- Stable in sorting elements with equal values.
- Optimized for large and distributed data.
Cost efficiency:
- Time comparison between Quick Sort and Merge Sort on the same data.
- Merge Sort may consume more memory but provides stability.
Conclusion:
- Choosing Merge Sort is appropriate when stability and correct sorting of duplicate elements is required.

Conclusion

Summary:
- Application of ADTs improves system design and performance.
- Selection of appropriate data configuration and algorithms for improved efficiency.
- Testing and debugging to ensure software quality.

Benefits achieved:
- Effective and maintainable student management system.
- High performance in key operations.
- Scalability and operability.

THANK YOU