DATA STRUCTURE

SE06304
Part 2

IMPLEMENT COMPLEX DATA STRUCTURES AND ALGORITHMS

Define the problem

The problem involves implementing a Student Management System where students' details are stored in a stack data structure, and various operations like adding, updating, deleting, searching, and sorting students based on marks are required. The challenge includes sorting students using different algorithms (Bubble Sort, Merge Sort, Quick Sort, Selection Sort) and ensuring that the system is flexible and efficient in handling large amounts of data.

Choose a Programming Language

Choosing Java for the student management software is a popular and sensible decision for several reasons:

1.Reliability and Stability: Java has been around for a long time and is widely used in large enterprise systems. This proves Java's reliability and stability in management applications.

2.Security: Java offers robust security features, such as access management and data protection, which are critical for storing and managing personal and sensitive student information.

3.Cross-Platform Compatibility: Java can run on multiple operating systems (Windows, Mac, Linux, etc.) without needing to change the source code. This makes the software easy to deploy and use on various devices.

4.Strong Community Support and Rich Libraries: There is a large community of Java developers and extensive documentation, open-source libraries, and development tools that help speed up the software development process.

5.Performance: Although it might not be as fast as some other programming languages in specific cases, Java still offers good performance to meet the needs of a student management system.

Design the ADT

Student class: Represents a student with attributes like ID, name, marks, and rank (based on marks).

StudentStack class: A stack data structure used to store the list of students. It allows efficient addition and removal of students following the LIFO (Last In, First Out) principle.

StudentManagement class: Manages operations on students, including adding, updating, deleting, searching, sorting, and displaying the list of students.

Design the Algorithm

The algorithm involves:
· Bubble Sort: Repeatedly compares adjacent students and swaps them if they are in the wrong order, making multiple passes through the list until the list is sorted.
· Merge Sort: Divides the list into two halves, recursively sorts each half, and then merges the sorted halves.
· Quick Sort: Partitions the list around a pivot and recursively sorts the two partitions.
· Selection Sort: Iterates through the list, selects the smallest element, and swaps it with the current element.
These algorithms are designed to handle sorting within a linked list structure efficiently.

Implement the ADT

```java
public class Student {
    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 String getRank() {
        if (marks >= 9.0) return "Excellent";
        else if (marks >= 7.5) return "Very Good";
        else if (marks >= 6.5) return "Good";
        else if (marks >= 5.0) return "Medium";
        else return "Fail";
    }

    @Override
    public String toString() {
        return "Student{" +
                "ID='" + id + '\'' +
                ", Name='" + name + '\'' +
                ", Marks=" + marks +
                ", Rank=" + getRank() +
                '}';
    }
}
```

```java
public class StudentStack {
    private Node top;

    private class Node {
        Student student;
        Node next;

        Node(Student student) {
            this.student = student;
            this.next = null;
        }
    }

    public StudentStack() {
        this.top = null;
    }

    public void push(Student student) {
        Node newNode = new Node(student);
        newNode.next = top;
        top = newNode;
    }

    public Student pop() {
        Student student = top.student;
        top = top.next;
        return student;
    }

    public Student peek() {
        return top.student;
    }

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

    public void bubbleSortStudents() {
        if (top == null || top.next == null) {
            System.out.println("Stack is empty or contains only one student
            return;
        }

        boolean swapped;
        do {
            swapped = false;
            Node current = top;
```

```java
util.Random;
util.function.Consumer;
util.HashSet;
util.*;

 StudentManagement {
 StudentStack students;

tudentManagement() {
.students = new StudentStack();


oid addStudent(Student student) {
searchStudent(student.getId()) != null) {
System.out.println("Student with ID " + student.getId() + " already exists. Can
return;

dents.push(student);
em.out.println("Added Student: ID=" + student.getId() + ", Name=" + student.getN


oid updateStudent(String id, String newName, double newMarks) {
dentStack tempStack = new StudentStack();
ean found = false;
e (!students.isEmpty()) {
Student student = students.pop();
if (student.getId().equals(id)) {
    tempStack.push(new Student(id, newName, newMarks));
    found = true;
} else {
    tempStack.push(student);
}

e (!tempStack.isEmpty()) {
students.push(tempStack.pop());

!found) {
System.out.println("Student with ID " + id + " not found.");



oid deleteStudent(String id) {
dentStack tempStack = new StudentStack();
ean found = false;
e (!students.isEmpty()) {
Student student = students.pop();
if (!student.getId().equals(id)) {
    tempStack.push(student);
} else {
```

IMPLEMENT ERROR HANDLING AND REPORT TEST RESULTS.

Identify potential errors

1.Invalid data input:
o Input data is in the wrong format (e.g., entering characters instead of numbers).
o Marks are outside the required range (0–10).
2.Operations on an empty list:
o Attempting to delete, search, or sort when the stack is empty.
3.Student not found:
o A student ID does not exist in the list but is used in operations such as updateStudent or deleteStudent.
4.Logical errors:
o Incorrect sorting.
o Operations on data that has already been deleted.
5.Runtime exceptions:
o Errors caused by runtime or system issues (e.g., NullPointerException, InputMismatchException).

2 Define error handling mechanisms

1. Invalid data input:
Validate user input with checks and error messages.
Use try-catch blocks for input parsing and provide feedback to the user.
2. Operations on an empty list:
Add pre-condition checks before performing operations on the list.
Inform the user that the operation cannot proceed.
3. Student not found
Search for the student ID first.
Display an appropriate error message if the student is not found.
4. Logical Errors
Validate the correctness of sorting by comparing the results.
Ensure that deleted nodes are properly removed.
5. Runtime Exceptions
Use try-catch blocks to catch runtime exceptions.
Log the error details for debugging.

Implement Error Handling Code

1. Input Validation (e.g., Marks Range):

Already implemented in the Main class by checking marks between 0 and 10.

```
System.out.print("Enter student marks (0 to 10): ");
while (!scanner.hasNextDouble()) {
    System.out.println("Invalid input. Please enter a number.");
    scanner.next(); // Skip invalid data
}
marks = scanner.nextDouble();
if (marks < 0 || marks > 10) {
    System.out.println("Marks must be between 0 and 10.");
```

2. Check stack state

Add checks before popping or peeking:

```
public Student pop() {
    if (isEmpty()) {
        throw new IllegalStateException("Cannot pop from an empty stack.");
    }
    Student student = top.student;
    top = top.next;
    return student;
}
public Student peek() {
    if (isEmpty()) {
        throw new IllegalStateException("Cannot peek from an empty stack.");
    }
    return top.student;
```

Implement Error Handling Code

1.Adding a Student:
o Input valid details: Check addition.
o Input duplicate ID: Ensure error message appears.
2.Updating a Student:
o Valid ID: Verify update.
o Invalid ID: Ensure error message.
3.Deleting a Student:
o Valid ID: Verify deletion.
o Invalid ID: Check error message.
4. Search a Student:
o Valid ID: Verify search.
o Invalid ID: Check error message.
5.Sorting Students:
o Empty stack: Confirm "too few students" message.
o Stack with multiple students: Verify correct sorting.
6.Displaying Students:
o Empty stack: Ensure no crash and correct message.

DEMONSTRATE HOW THE IMPLEMENTATION OF AN ADT/ALGORITHM SOLVES A WELL-DEFINED PROBLEM

Design the ADT/Algorithm

To design a solution, we need to implement an Abstract Data Type (ADT) that provides the necessary operations to manage students. In this case, the ADT is represented by the StudentStack class, which models a stack (LIFO structure) of students, and several sorting algorithms for arranging students based on their marks.

Key operations (ADT):

1.push(Student student): Adds a student to the top of the stack.
2.pop(): Removes and returns the student from the top of the stack.
3.peek(): Returns the student at the top of the stack without removing them.
4.isEmpty(): Checks whether the stack is empty.
5.bubbleSortStudents(): Sorts the students by their marks using the Bubble Sort algorithm.
6.mergeSortStudents(): Sorts the students by their marks using the Merge Sort algorithm.
7.quickSortStudents(): Sorts the students by their marks using the Quick Sort algorithm.
8.selectionSortStudents(): Sorts the students by their marks using the Selection Sort algorithm.

Each sorting algorithm is designed to handle the stack of students differently, but all aim to order the students based on their marks.

Test cases

Test Case for Add Student

No.	Feature	Description	Input Data	Expected Result	Notes
1	Add Student	Add a new student to the system.	ID: 123, Name: John, Marks: 8.5	Student added successfully with information: ID = 123, Name = John, Marks = 8.5.	Check add student functionality.
2	Add Student	Add another student to the system.	ID: 124, Name: Alice, Marks: 9.0	Student added successfully with information: ID = 124, Name = Alice, Marks = 9.0.	Check adding student with higher marks.
3	Add Student	Add a student with marks on the boundary (0 to 10).	ID: 125, Name: Bob, Marks: 0.0	Student added successfully with information: ID = 125, Name = Bob, Marks = 0.0.	Test for lower boundary marks.
4	Add Student	Add a student with marks on the boundary (0 to 10).	ID: 126, Name: Eve, Marks: 10.0	Student added successfully with information: ID = 126, Name = Eve, Marks = 10.0.	Test for upper boundary marks.

Test cases

Test Case for Sort Students

No.	Feature	Description	Input Data	Expected Result	Notes
9	Sort Students	Sort students by Bubble Sort (ascending marks).	[Student ID: 123, Marks: 8.5, Student ID: 124, Marks: 9.0]	Students sorted in ascending order based on marks: ID = 123, Marks = 8.5, ID = 124, Marks = 9.0.	Test sorting with Bubble Sort.
10	Sort Students	Sort students by Merge Sort (ascending marks).	[Student ID: 125, Marks: 7.2, Student ID: 126, Marks: 8.5]	Students sorted in ascending order based on marks: ID = 125, Marks = 7.2, ID = 126, Marks = 8.5.	Test sorting with Merge Sort.
11	Sort Students	Sort students by Quick Sort (ascending marks).	[Student ID: 127, Marks: 6.0, Student ID: 128, Marks: 9.2]	Students sorted in ascending order based on marks: ID = 127, Marks = 6.0, ID = 128, Marks = 9.2.	Test sorting with Quick Sort.
12	Sort Students	Sort students by Selection Sort (ascending marks).	[Student ID: 129, Marks: 5.0, Student ID: 130, Marks: 7.5]	Students sorted in ascending order based on marks: ID = 129, Marks = 5.0, ID = 130, Marks = 7.5.	Test sorting with Selection Sort.
13	Sort Students	Try sorting an empty stack of students.	[]	No students to sort. Error message: "Stack is empty."	Check sort on an empty stack.
14	Sort Students	Sort students when all students have the same marks.	[Student ID: 131, Marks: 8.0, Student ID: 132, Marks: 8.0]	Students remain in original order: ID = 131, Marks = 8.0, ID = 132, Marks = 8.0.	Check sort when students have equal marks.

DISCUSS HOW ASYMPTOTIC ANALYSIS CAN BE USED TO ASSESS THE EFFECTIVENESS OF AN ALGORITHM

Time complexity analysis

Definition: Asymptotic Analysis is the big idea that handles the above issues in analyzing algorithms. In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size (we don't measure the actual running time). We calculate, order of growth of time taken (or space) by an algorithm in terms of input size. For example linear search grows linearly and Binary Search grows logarithmically in terms of input size.

Big O notation is a powerful tool used in computer science to describe the time complexity or space complexity of algorithms. It provides a standardized way to compare the efficiency of different algorithms in terms of their worst-case performance. Understanding Big O notation is essential for analyzing and designing efficient algorithms. Common notations include:

·Big-O (O): Upper bound, worst-case performance.
·Big-Ω (Ω): Lower bound, best-case performance.
·Big-Θ (Θ): Tight bound, average-case or typical performance.

Example:

Best Case (O(n)): When the list is already sorted, the loop runs only once, with no swaps needed.

Worst Case (O(n2)): When the list is completely unordered, the loop runs n times, and each time compares all n - i elements.

Bubble Sort is inefficient for large datasets due to its quadratic time complexity.

```java
public void bubbleSortStudents() {
    do {
        swapped = false;
        Node current = top;

        while (current != null && current.next != null) {
            if (current.student.getMarks() > current.next.student.getMarks()) {
                Student temp = current.student;
                current.student = current.next.student;
                current.next.student = temp;
                swapped = true;
            }
            current = current.next;
        }
    } while (swapped);
}
```

Growth Rate Comparison

Comparing Algorithms: By comparing the growth rates of different algorithms, we can determine which one is more efficient. For example, an algorithm with O(n log n) is generally more efficient than one with O(n^2) for large input sizes.

Impact of Input Size: Asymptotic analysis helps in understanding how algorithms perform relative to each other as the input size grows.

Comparing Bubble Sort and Merge Sort

Bubble Sort (O(n2)O(n^2)O(n2)): For a stack of 1,000 students, Bubble Sort would require 1,000,000 operations in the worst case.

Merge Sort (O(nlogn)O(n \log n)O(nlogn)): For the same input size, Merge Sort would need around 10,000 operations.

Efficiency and scalability

Scalability: Asymptotic analysis helps in assessing whether an algorithm can handle large input sizes efficiently. Algorithms with lower time complexity are typically more scalable.

Resource Utilization: It provides insights into the amount of computational resources (time and space) that an algorithm will consume.

Comparing Selection Sort and Quick Sort

·Selection Sort: O(n2)O(n^2)O(n2). It doesn't scale well with large nnn because it finds the smallest element in the remaining unsorted part in every step.

·Quick Sort: O(nlogn)O(n \log n)O(nlogn) (on average). It divides the data, which reduces the processing load at each step.

Example:

Quick Sort for 1,000 students will need approximately $1{,}000 \times \log_2(1{,}000) \approx 10{,}000$ operations.

Selection Sort for 1,000 students will need approximately $1{,}000 \times 999 = 999{,}000$ operations.

Thus, Quick Sort is more scalable than Selection Sort as the dataset grows.

Space complexity

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

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

It is the amount of memory needed for the completion of an algorithm.

To estimate the memory requirement we need to focus on two parts:

(1) A fixed part: It is independent of the input size. It includes memory for instructions (code), constants, variables, etc.

(2) A variable part: It is dependent on the input size. It includes memory for recursion stack, referenced variables, etc.

Example: Consider the mergeSortStudents() method in your code, which implements Merge Sort:

Merge Sort works by recursively dividing the list into two halves. For each recursive call, additional memory is needed to store the left and right sublists. At each recursive level, the space needed to store the sublists grows.

Space Complexity of Merge Sort: O(n), where n is the number of students. This is because the algorithm requires extra space to hold the left and right sublists during the merging phase. It also uses recursion, which consumes stack space.

In this case, the space complexity is O(n) because the algorithm needs to create temporary sublists for merging.

DISCUSS HOW ASYMPTOTIC ANALYSIS CAN BE USED TO ASSESS THE EFFECTIVENESS OF AN ALGORITHM

Analyze the Time and Space Complexity

1. Stack Operations (Push, Pop, Peek, IsEmpty):
·Push: O(1) - Inserting a new node at the top is a constant time operation.
·Pop: O(1) - Removing the top node and adjusting the pointer is constant time.
·Peek: O(1) - Accessing the top student is constant time.
·IsEmpty: O(1) - Checking if the stack is empty is a constant time operation.

2. Bubble Sort
Time Complexity:
oBest Case (O(n)): If the list is already sorted, it only takes one pass to confirm no swaps are needed.
oWorst Case ($O(n^2)$): If the list is sorted in reverse order, every pair must be compared and swapped.
oAverage Case ($O(n^2)$): The typical case still involves comparing each pair of elements repeatedly.
Space Complexity: O(1) - Bubble Sort is an in-place sorting algorithm that doesn't require extra space apart from the input list.
Evaluation:
o Limitations: Inefficient for large datasets due to its $O(n^2)$ time complexity in both average and worst cases.
o Recommendations: For larger datasets, Merge Sort or Quick Sort would be better choices.

Execution Results with 10 students:
Execution Time: 120000 nanoseconds (0.12 milliseconds)
Result: For a small number of students, Bubble Sort runs quickly, but its $O(n^2)$ time complexity makes it inefficient as the number of students increases.
Sorting students using Bubble Sort...
Time taken to sort 10 students using Bubble Sort: 120000 nanoseconds.

Execution Results with 1000 students:
Execution Time: 320000000 nanoseconds (320 milliseconds)
Result: Becomes very slow with 1000 students due to its $O(n^2)$ time complexity.
Sorting students using Bubble Sort...
Time taken to sort 1000 students using Bubble Sort: 320000000 nanoseconds.

Execution Results with 10000 students:
Execution Time: 5000000000 nanoseconds (5 seconds)
Result: Becomes impractical with 10000 students, and the execution time increases drastically.
Sorting students using Bubble Sort...
Time taken to sort 10000 students using Bubble Sort: 5000000000 nanoseconds.

. **MERGE SORT**

TIME COMPLEXITY:
O BEST CASE (O(N LOG N)): EVEN IF THE LIST IS ALREADY SORTED, THE MERGING PROCESS STILL TAKES O(N LOG N).
O WORST CASE (O(N LOG N)): THE WORST CASE REMAINS O(N LOG N) AS THE LIST IS ALWAYS DIVIDED AND MERGED BACK.
O AVERAGE CASE (O(N LOG N)): CONSISTENTLY PERFORMS O(N LOG N) COMPARISONS REGARDLESS OF THE ORDER OF ELEMENTS.
SPACE COMPLEXITY: O(N) - REQUIRES EXTRA SPACE FOR MERGING THE SUBLISTS.
EVALUATION:
LIMITATIONS: HIGHER SPACE COMPLEXITY DUE TO AUXILIARY ARRAYS USED FOR MERGING.
RECOMMENDATIONS: IF MEMORY USAGE IS A CONCERN, CONSIDER QUICK SORT OR INSERTION SORT FOR SMALLER DATASETS.

EXECUTION RESULTS WITH 10 STUDENTS:
EXECUTION TIME: 70000 NANOSECONDS (0.07 MILLISECONDS)
RESULT: MERGE SORT IS FASTER AND MORE EFFICIENT THAN BUBBLE SORT WITH A TIME COMPLEXITY OF O(N LOG N).
SORTING STUDENTS USING MERGE SORT...
TIME TAKEN TO SORT 10 STUDENTS USING MERGE SORT: 70000 NANOSECONDS.

EXECUTION RESULTS WITH 1000 STUDENTS:
EXECUTION TIME: 2000000 NANOSECONDS (2 MILLISECONDS)
RESULT: MERGE SORT PERFORMS BETTER THAN BUBBLE SORT AND IS MORE EFFICIENT WITH O(N LOG N).
SORTING STUDENTS USING MERGE SORT...
TIME TAKEN TO SORT 1000 STUDENTS USING MERGE SORT: 2000000 NANOSECONDS.

EXECUTION RESULTS WITH 10000 STUDENTS:
EXECUTION TIME: 15000000 NANOSECONDS (15 MILLISECONDS)
RESULT: MERGE SORT REMAINS EFFICIENT WITH LARGE NUMBERS OF STUDENTS, AND THE EXECUTION TIME DOESN'T INCREASE DRASTICALLY.

```
SORTING STUDENTS USING MERGE SORT...
TIME TAKEN TO SORT 10000 STUDENTS USING MERGE SORT: 15000000 NANOSECONDS.
```

INTERPRET WHAT A TRADE-OFF IS WHEN SPECIFYING AN ADT, USING AN EXAMPLE TO SUPPORT YOUR ANSWER

Time Complexity vs. Space Complexity

In data structures, optimizing for time complexity (speed of operations) often results in higher space complexity (memory usage) and vice versa.
This trade-off is evident when comparing the implemented sorting algorithms in StudentStack:
Bubble Sort:
oTime complexity: O(n^2) - It iterates through the stack n times, performing comparisons within each iteration. This nested loop structure leads to a quadratic increase in time with the number of elements (n).
oSpace complexity: O(1) - It only uses a few temporary variables for comparisons and swapping, making its space usage constant regardless of the number of elements.
Merge Sort and Quick Sort:
oTime complexity: O(n log n) on average - These algorithms employ a divide-and-conquer approach, breaking down the problem into smaller sub-problems and merging/partitioning them efficiently. This results in a more efficient time complexity compared to Bubble Sort.
oSpace complexity: O(log n) - They utilize recursion with a call stack for dividing the problem. The depth of the call stack grows logarithmically with the number of elements.

Flexibility vs. Performance

The trade-off between flexibility and performance occurs when we choose a data structure or approach that provides adaptability for various use cases at the cost of raw performance.
Trade-off:

- A stack is ideal when you need performance for specific operations like push, pop, or peek, but it is less flexible if you need more complex operations like sorting, searching, or random access.
- If the system needs more flexibility (e.g., efficient searching by ID or frequent updates to the student list), you would need to use a more complex data structure, like a linked list or hash map, but these come with trade-offs in terms of performance for stack-like operations.

Flexibility vs. Performance: The stack is optimal for simple operations like push and pop but is not flexible for tasks like searching or sorting. If we need greater flexibility (for example, searching by student ID), a hash map would be more suitable, but it would compromise the stack's straightforward performance in terms of order-based operations.

THANK YOU