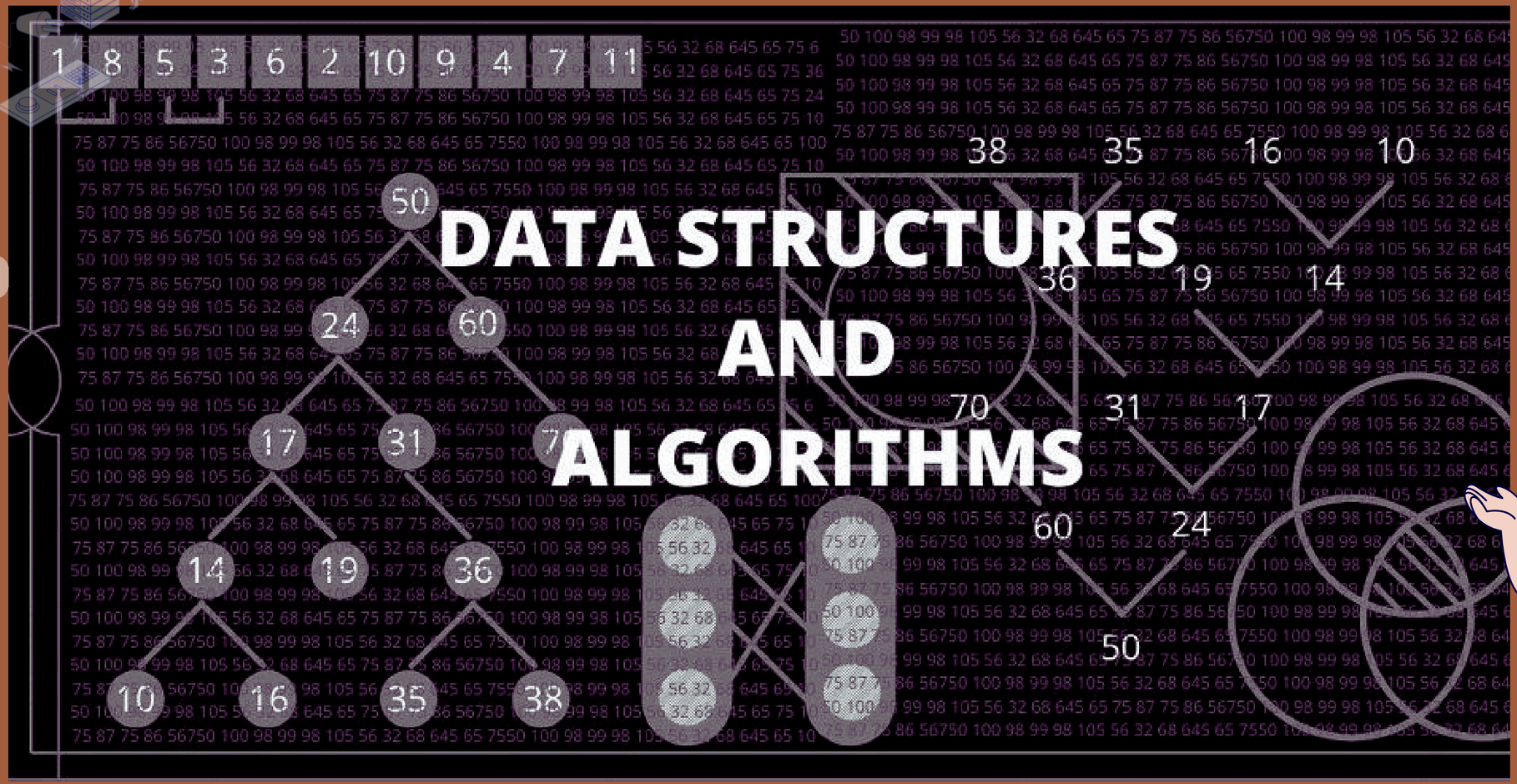
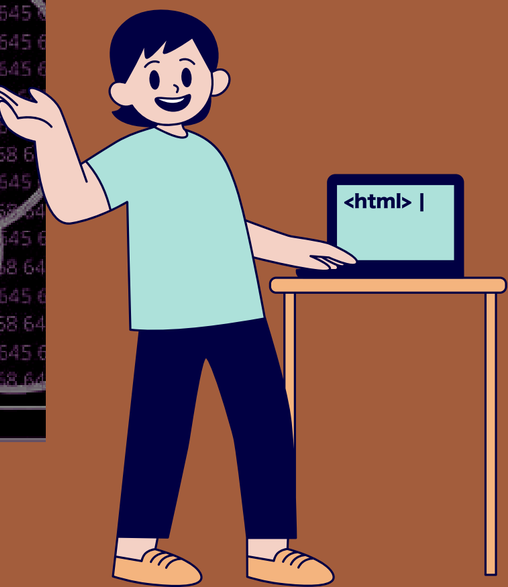
Implement complex data structures and algorithms

(Chu Viet Chien - BH00915.)





The content in this process is:

1, Define the problem.
2, ADT.
3, Algorithm.
4, Implement error handlling and report test results.
5, Demonstrate how the implementation of an ADT/algorithm solves a well-defined problem.
6, Critically evaluate the complexity of an implemented ADT/algorithm.
7, Discuss how asymptotic can be used to assess the effectiveness of an algorithm.
8, Determine two ways in which the effectiveness of an algorithm can be measured, illustrating your answer with an example.
9, Evaluate three benefits of using implementation independent data structures.

1, Define the problem.

First, it is necessary to define the specific problem that the data structure and algorithm will solve. This helps guide the design and selection of appropriate tools. For example:

Problem: Managing student information with the following main requirements:

- Add, update, and delete student records.
- Search for students by ID or name.
- Sort the list of students by marks.
- Rank students based on criteria such as "Good" or "Excellent."

Objective: Design and implement a system that uses suitable data structures and algorithms to process data efficiently and accurately.

2, ADT.

2.1, Choose a Programming Language.

Choose a programming language based on the problem's requirements and criteria such as performance, ease of implementation, and integration capabilities.

Selected Language: Java

Reason:

- Java offers robust object-oriented programming support, suitable for ADT design.
- It provides extensive libraries for data management and algorithm implementation.
- It is easy to maintain and widely used in software projects.

2.1, Design the ADT.

ADT (Abstract Data Type) is a logical modeling of data and the operations performed on that data without focusing on how it is implemented.

Designing the Student ADT:

a, Data:

- id (integer): The student's unique identifier.
- name (string): The student's name.
- contactNumber (string): The student's phone number.
- address (string): The student's address.
- marks (string): The student's marks.
- rank (string): The student's rank.

b, Operations:

Class StudentStack:

- Add (push): Adds a student to the stack.
- Remove (pop): Removes a student from the stack.
- View (peek): Retrieves the student at the top of the stack.
- Check empty (isEmpty): Determines if the stack is empty.
- Get size (size): Retrieves the number of elements in the stack.
- Display (displayStudents): Traverses and prints the list of students in the stack.

Class StudentManagement:

- AddStudent(id, name, contactNumber, address, marks, rank): Add a new student.
- UpdateStudent(id, newName, newContact, newAddress, newMarks, newRank): Update student information.
- DeleteStudent(id): Delete a student by ID.
- SearchStudentById(id): Search for a student by ID.
- bubbleSort() and quicksort(): Sort the list of students in descending order of marks.

2.1, Implement the ADT.

2.1.1, The Student Class.

```java
package org.example;

public class Student {
    private int id;
    private String name;
    private String contactNumber;
    private String address;
    private double marks;
    private String rank;

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


    public void setRank() {
        if (marks > 90) {
            this.rank = "Excellent";
        } else if (marks >= 75) {
            this.rank = "Good";
        } else if (marks >= 60) {
            this.rank = "Medium";
        } else if (marks >= 50) {
            this.rank = "Fall";
        } else {
            this.rank = "Fail";
        }
    }
```

```java
    // Getter và Setter
    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public String getContactNumber() {
        return contactNumber;
    }

    public String getAddress() {
        return address;
    }

    public double getMarks() {
        return marks;
    }

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

    public String getRank() {
        return rank;
    }
```

```java
    @Override public String toString() {
        return "Student{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", contactNumber='" + contactNumber + '\'' +
                ", address='" + address + '\'' +
                ", marks=" + marks +
                ", rank='" + rank + '\'' +
                '}';
    }
}
```

2.1.2, The StudentStack Class.

```java
package org.example;

public class StudentStack {
    private Node top; // Top of the stack
    private int size; // To keep track of the number of elements

    // Constructor
    public StudentStack() {
        top = null; // Stack is initially empty
        size = 0;   // Initial size is 0
    }

    // Push a student onto the stack
    public void push(Student student) {
        Node newNode = new Node(student);
        newNode.next = top; // Point new node to the previous top
        top = newNode;      // Update top to be the new node
        size++;
        System.out.println("Inserted: " + student);
    }
```

```java
// Pop a student from the stack
public Student pop() {
    if (isEmpty()) {
        System.out.println("Stack Underflow! No students to remove.");
        return null;
    }
    Student poppedStudent = top.student; // Get the student from the top node
    top = top.next;                      // Move top to the next node
    size--;
    return poppedStudent;
}

// Peek at the top student
public Student peek() {
    if (isEmpty()) {
        System.out.println("Stack is empty!");
        return null;
    }
    return top.student; // Return the student at the top node
}

// Check if the stack is empty
public boolean isEmpty() {
    return top == null; // Stack is empty if top is null
}
```

```java
// Get the size of the stack
public int size() {
    return size; // Return current size of the stack
}

// Display all students in the stack
public void displayStudents() {
    if (isEmpty()) {
        System.out.println("No students in the stack.");
        return;
    }

    System.out.println("Students in Stack:");
    Node current = top;  // Start from the top node
    while (current != null) {
        System.out.println(current.student);
        current = current.next; // Move to the next node
    }
}
```

2.1.3, The Node Class.

```java
package org.example;

public class Node {
    Student student;
    Node next;

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

2.1.4, The StudentManagement Class.

```java
package org.example;

public class StudentManagement {
    private StudentStack students;

    public StudentManagement() {
        this.students = new StudentStack();
    }

    public void addStudent(Student student) {
        try {
            if (student == null) {
                throw new IllegalArgumentException("Student object cannot be null.");
            }
            students.push(student);
        } catch (Exception e) {
            System.out.println("Error adding student: " + e.getMessage());
        }
    }
```

```java
public void deleteStudent(int id) {
    StudentStack tempStack = new StudentStack();
    try {
        while (!students.isEmpty()) {
            Student student = students.pop();
            if (student.getId() != id) {
                tempStack.push(student);
            }
        }

        while (!tempStack.isEmpty()) {
            students.push(tempStack.pop());
        }

        System.out.println("Student with ID " + id + " has been deleted (if existed).");
    } catch (Exception e) {
        System.out.println("Error deleting student: " + e.getMessage());
    }
}
```

```java
public void updateStudent(int id, String newName, String newContactNumber, String address, double marks, String rank) {
    StudentStack tempStack = new StudentStack();
    boolean found = false;

    try {
        while (!students.isEmpty()) {
            Student student = students.pop();

            if (student.getId() == id) {
                tempStack.push(new Student(id, newName, newContactNumber, address, marks, rank));
                found = true;
            } else {
                tempStack.push(student);
            }
        }

        while (!tempStack.isEmpty()) {
            students.push(tempStack.pop());
        }

        if (!found) {
            System.out.println("Student with ID " + id + " not found.");
        }
    } catch (Exception e) {
        System.out.println("Error updating student: " + e.getMessage());
    }
}
```

```java
public void bubbleSort() {
    if (students.isEmpty()) {
        System.out.println("No students to sort.");
        return;
    }

    try {
        Student[] studentArray = new Student[students.size()];
        StudentStack tempStack = new StudentStack();
        int index = 0;

        while (!students.isEmpty()) {
            Student student = students.pop();
            studentArray[index++] = student;
            tempStack.push(student);
        }

        while (!tempStack.isEmpty()) {
            students.push(tempStack.pop());
        }

        long startTime = System.nanoTime();
        for (int i = 0; i < studentArray.length - 1; i++) {
            for (int j = 0; j < studentArray.length - 1 - i; j++) {
                if (studentArray[j].getMarks() > studentArray[j + 1].getMarks()) {
                    Student temp = studentArray[j];
                    studentArray[j] = studentArray[j + 1];
                    studentArray[j + 1] = temp;
                }
            }
        }
        long endTime = System.nanoTime();

        for (int i = studentArray.length - 1; i >= 0; i--) {
            students.push(studentArray[i]);
        }

        System.out.println("Students sorted using Bubble Sort.");
        System.out.println("Bubble Sort Time: " + (endTime - startTime) + " nanoseconds.");
    } catch (Exception e) {
        System.out.println("Error in Bubble Sort: " + e.getMessage());
    }
}
```

```java
public void quickSort() {
    if (students.isEmpty()) {
        System.out.println("No students to sort.");
        return;
    }

    try {
        Student[] studentArray = new Student[students.size()];
        StudentStack tempStack = new StudentStack();
        int index = 0;

        while (!students.isEmpty()) {
            Student student = students.pop();
            studentArray[index++] = student;
            tempStack.push(student);
        }

        while (!tempStack.isEmpty()) {
            students.push(tempStack.pop());
        }

        long startTime = System.nanoTime();
        quickSortHelper(studentArray, 0, studentArray.length - 1);
        long endTime = System.nanoTime();

        for (int i = studentArray.length - 1; i >= 0; i--) {
            students.push(studentArray[i]);
        }

        System.out.println("Students sorted using Quick Sort.");
        System.out.println("Quick Sort Time: " + (endTime - startTime) + " nanoseconds.");
    } catch (Exception e) {
        System.out.println("Error in Quick Sort: " + e.getMessage());
    }
}

private void quickSortHelper(Student[] studentArray, int low, int high) {
    if (low < high) {
        int pi = partition(studentArray, low, high);
        quickSortHelper(studentArray, low, pi - 1);
        quickSortHelper(studentArray, pi + 1, high);
    }
}
```

```java
private int partition(Student[] studentArray, int low, int high) {
    try {
        Student pivot = studentArray[high];
        int i = low - 1;

        for (int j = low; j < high; j++) {
            if (studentArray[j].getMarks() <= pivot.getMarks()) {
                i++;
                Student temp = studentArray[i];
                studentArray[i] = studentArray[j];
                studentArray[j] = temp;
            }
        }

        Student temp = studentArray[i + 1];
        studentArray[i + 1] = studentArray[high];
        studentArray[high] = temp;

        return i + 1;
    } catch (Exception e) {
        System.out.println("Error in partitioning: " + e.getMessage());
        return -1;
    }
}
```

```java
public void displayAllStudents() {
    if (students.isEmpty()) {
        System.out.println("No students to display.");
        return;
    }

    try {
        System.out.println("Current student list:");
        StudentStack tempStack = new StudentStack();
        while (!students.isEmpty()) {
            Student student = students.pop();
            System.out.println(student);
            tempStack.push(student);
        }

        while (!tempStack.isEmpty()) {
            students.push(tempStack.pop());
        }
    } catch (Exception e) {
        System.out.println("Error displaying students: " + e.getMessage());
    }
}
```

```java
public void searchStudentById(int id) {
    StudentStack tempStack = new StudentStack();
    try {
        while (!students.isEmpty()) {
            Student student = students.pop();
            if (student.getId() == id) {
                System.out.println("Found student: " + student.getName() +
                        ", Contact: " + student.getContactNumber() +
                        ", Address: " + student.getAddress() +
                        ", Marks: " + student.getMarks() +
                        ", Rank: " + student.getRank());
                tempStack.push(student);
                while (!tempStack.isEmpty()) {
                    students.push(tempStack.pop());
                }
                return;
            }
            tempStack.push(student);
        }

        System.out.println("Student with ID " + id + " not found.");
    } catch (Exception e) {
        System.out.println("Error searching for student: " + e.getMessage());
    } finally {
        while (!tempStack.isEmpty()) {
            students.push(tempStack.pop());
        }
    }
}
```

3, Algorithm.

a, Add Student.

b, Update Student.

c, Delete Student.

d, Bubble Sort.

e, Quick Sort.

f, Search Student by ID.

4, Implement error handlling and report test results.

4.1, Identify protential errors.

Here are some potential errors in the provided StudentManagement code:

a, Empty Stack Errors:

- Operations like pop, peek, bubbleSort, and quickSort may not function properly if the stack is empty. These operations would either return incorrect results or result in errors (e.g., accessing null values).

b, Invalid Student ID for Update or Delete:

- When trying to update or delete a student by ID, the operation may fail silently if the student with the specified ID does not exist. No error feedback may be provided to the user.

c, Sorting Algorithm Errors:

- Empty Stack: Sorting operations like Bubble Sort and Quick Sort may fail or behave incorrectly if the stack is empty.
- Identical Marks: Sorting algorithms may not handle cases where students have identical marks correctly, potentially leading to inefficient sorting or undesired behavior.

d, Inconsistent Stack State After Operations:

- During operations like updating or deleting students, the state of the stack may become inconsistent if students are not properly restored after temporary removal. This may result in missing or misplaced students in the stack.

e, Array Index Out of Bounds:

- When converting the stack to an array for sorting, an error may occur if the array is not sized properly, or if there is an attempt to access an index beyond the bounds of the array.

f, Memory Overflows/Heap Space Issues:

- Handling large numbers of students in the stack, especially with sorting algorithms that require additional memory, could potentially cause memory issues or stack overflows, especially in resource-constrained environments.

4.2, Define error handling mechanisms.

a, Handling Empty Stack Errors:

- Ensure that any operation like pop, peek, or sorting algorithms check if the stack is empty before proceeding. If the stack is empty, provide a clear error message and prevent further operations.

b, Handling Invalid Student ID for Update or Delete:

- Before performing update or delete operations, verify if the student with the given ID exists in the stack. If not, return a clear error message indicating that the student was not found.

c, Handling Sorting Algorithm Errors:

- Empty Stack: Check if the stack is empty before attempting to sort. If it is, return an appropriate message indicating that sorting cannot be performed on an empty stack.
- Identical Marks: Ensure that sorting algorithms account for ties in student marks. Sorting should be stable, meaning students with the same marks retain their original order relative to each other.

d, Consistent Stack State After Operations:

- When performing operations that modify the stack (such as update or delete), ensure that the stack is always restored to its original state, even if an error occurs. This prevents any inconsistencies or loss of data.

e, Handling Array Index Out of Bounds:

- Carefully handle the conversion of the stack to an array. Ensure that the array is correctly sized to match the number of students in the stack, and that the sorting operations respect the array bounds.

f, Handling Memory Overflows/Heap Space Issues:

- For operations that involve large amounts of data (such as sorting), monitor memory usage and ensure that algorithms are optimized for space efficiency. In case of potential memory issues, handle them gracefully by providing feedback to the user and avoiding operations that exceed system resources.

4.3, Identify error handling code.

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

public class Main3 {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Random rand = new Random();
        StudentManagement studentManagement = new StudentManagement();

        System.out.print("Enter the number of students to generate: ");
        int numStudents = readInteger(scanner); // Prompt user to input the number of students

        // Generate random students with increasing ID
        for (int i = 0; i < numStudents; i++) {
            String name = "Student" + i;
            String contactNumber = "ContactNumber" + (1000 + rand.nextInt(9000));
            String address = "Address " + (rand.nextInt(100) + 1);
            double marks = rand.nextDouble() * 100;
            String rank = "";
            // Create a new student with increasing ID
            Student student = new Student(i, name, contactNumber, address, marks, rank);
            // Set rank based on marks (this method should be defined in Student class)
            student.setRank();
            // Add student to the management system
            studentManagement.addStudent(student);
        }

        // You can now test student management, for example:
        studentManagement.displayAllStudents();
```

```java
    // Menu chính
    while (true) {
        System.out.println("\nStudent Management System");
        System.out.println("1. Display all students");
        System.out.println("2. Add a new student");
        System.out.println("3. Update a student");
        System.out.println("4. Delete a student");
        System.out.println("5. Quicksort");
        System.out.println("6. Bubble sort");
        System.out.println("7. Search student by ID");
        System.out.println("8. Exit");
        System.out.print("Choose an option: ");

        int choice = scanner.nextInt();
        scanner.nextLine(); // Consume the newline character
```

```java
switch (choice) {
    case 1:
        studentManagement.displayAllStudents();
        break;
    case 2:
        System.out.print("Enter ID: ");
        int id = scanner.nextInt();
        scanner.nextLine(); // Consume the newline character
        System.out.print("Enter Name: ");
        String name = scanner.nextLine();
        System.out.print("Enter Contact Number: ");
        String contactNumber = scanner.nextLine();
        System.out.print("Enter Address: ");
        String address = scanner.nextLine();
        System.out.print("Enter Marks: ");
        double marks = readDouble(scanner);
        System.out.print("Enter Rank: ");
        String rank = scanner.nextLine();
        studentManagement.addStudent(new Student(id, name, contactNumber, address, marks, rank));
        break;
```

```java
case 3:
    System.out.print("Enter Student ID to update: ");
    int updateId = readInteger(scanner);
    scanner.nextLine(); // Consume the newline character
    System.out.print("Enter new Name: ");
    String newName = scanner.nextLine();
    System.out.print("Enter new Contact Number: ");
    String newContactNumber = scanner.nextLine();
    System.out.print("Enter new Address: ");
    String newAddress = scanner.nextLine();
    System.out.print("Enter new Marks: ");
    double newMarks = readDouble(scanner);
    System.out.print("Enter new Rank: ");
    String newRank = scanner.nextLine();
    studentManagement.updateStudent(updateId, newName, newContactNumber, newAddress, newMarks, newRank);
    break;
case 4:
    System.out.print("Enter Student ID to delete: ");
    int deleteId = scanner.nextInt();
    studentManagement.deleteStudent(deleteId);
    break;
case 5:
    studentManagement.quickSort();
    break;
```

```java
case 6:
    studentManagement.bubbleSort();
    break;
case 7:
    System.out.print("Enter Student ID to search: ");
    int searchId = readInteger(scanner);
    studentManagement.searchStudentById(searchId);
    break;
case 8:
    System.out.println("Exiting the program.");
    scanner.close();
    return;
default:
    System.out.println("Invalid option. Please try again.");
```

```java
// Helper method to safely read an integer from the user input
private static int readInteger(Scanner scanner) {
    while (true) {
        try {
            return scanner.nextInt();
        } catch (InputMismatchException e) {
            System.out.print("Invalid input. Please enter an integer: ");
            scanner.next(); // Consume the invalid input
        }
    }
}

// Helper method to safely read a double from the user input
private static double readDouble(Scanner scanner) {
    while (true) {
        try {
            return scanner.nextDouble();
        } catch (InputMismatchException e) {
            System.out.print("Invalid input. Please enter a valid number for marks: ");
            scanner.next(); // Consume the invalid input
        }
    }
}
```

4.4, Identify error handling code.

ID	Question	Process	Expected Result	Actual Result	Status
HC001	Identify if non-numeric characters are entered into the menu	Open the application: Enter a non-numeric character when selecting an option.	Display an error message and prompt for re-entry.		
HC002	Identify if the user inputs a number outside the range 1-8	Open the application: Enter a number outside the 1-8 range.	Display "Invalid option" and prompt for re-entry.		
HC003	Identify if a non-numeric value is entered for ID when adding a student	Select option 2 (Add student): Enter a non-numeric ID.	Display an error message and prompt for re-entry.		
HC004	Identify if an empty string is entered for student name	Select option 2 (Add student): Leave the name field empty.	Display an error message and prompt for re-entry.		
HC005	Identify if non-numeric characters are entered for marks	Select option 2 (Add student): Enter non-numeric characters for marks.	Display an error message and prompt for re-entry.		
HC006	Identify if a non-existing ID is entered when updating a student	Select option 3 (Update student): Enter an ID that doesn't exist in the system.	Display "Student not found."		
HC007	Identify if invalid data is entered when updating a student	Select option 3 (Update student): Enter invalid data (e.g., non-numeric marks).	Display an error message and prompt for re-entry.		
HC008	Identify if a non-existing ID is entered to delete a student	Select option 4 (Delete student): Enter an ID that doesn't exist in the system.	Display "Student not found."		
HC009	Identify if a user can't be deleted (unexpected error)	Select option 4 (Delete student): Attempt to delete a student that cannot be deleted.	Display an error message or handle gracefully.		
HC010	Identify if searching for a student by a non-existent ID works	Select option 6 (Search by ID): Enter a non-existent ID.	Display "Student not found."		
HC011	Identify if searching for a student by name that's not in the list works	Select option 7 (Search by Name): Enter a non-existing name.	Display "No student found with this name."		
HC012	Identify if sorting works when there are invalid marks in the list	Add students with invalid marks: Select option 5 (Sort students by marks in descending order).	Ignore invalid marks or display an error message.		
HC013	Identify if sorting works in descending order	Select option 5 (Sort students by marks in descending order).	Students should be sorted correctly based on marks in descending order.		
HC014	Identify if large values for ID or marks are handled properly	Select option 2 or 3 (Add/Update student): Enter extremely large numbers for ID or marks.	Accept input or provide a warning if values exceed acceptable limits.		
HC015	Identify if repeated wrong inputs are handled correctly	Continuously enter invalid data (e.g., non-numeric value for ID) when prompted.	After multiple invalid inputs, display an error message and prompt for re-entry.		
HC016	Identify if the scanner is unexpectedly closed	Close the scanner unexpectedly (e.g., system shutdown, exit command).	Display an error message or handle the unexpected closure gracefully.		
HC017	Identify if option 8 is used to exit the program	Select option 8 (Exit) from the menu.	Exit the program.		

4.5, Report test results

Date	Tester	ID	Actual Result	Status
28/11/2024	Do Ngoc Trung	HC001	The system crashes and exits the program.	Fail
28/11/2024	Do Ngoc Trung	HC002	Displays "Invalid option" and prompts to re-enter.	Pass
28/11/2024	Do Ngoc Trung	HC003	The system crashes and exits the program.	Fail
29/11/2024	Nguyen Le Nam	HC004	No error message displayed, updates system as usual.	Fail
29/11/2024	Nguyen Le Nam	HC005	No error message displayed, updates system as usual.	Fail
29/11/2024	Nguyen Le Nam	HC006	Displays "Student not found."	Pass
30/11/2024	Do Ngoc Thuc	HC007	No error message displayed, updates system as usual.	Fail
30/11/2024	Do Ngoc Thuc	HC008	No "Student not found" message displayed, system works normally.	Fail
30/11/2024	Do Ngoc Thuc	HC009	No students are non-deletable in the system.	Pass
30/11/2024	Do Ngoc Thuc	HC010	Displays "Student not found."	Pass
30/11/2024	Do Ngoc Thuc	HC011	Displays "No student found with this name."	Pass
01/12/2024	Nguyen Vu Cuong	HC012	Sorting caused an error and the system exited.	Fail
01/12/2024	Nguyen Vu Cuong	HC013	Sorts students correctly by descending marks.	Pass
01/12/2024	Nguyen Vu Cuong	HC014	System handled large values without crashing.	Pass
01/12/2024	Nguyen Vu Cuong	HC015	No error message, just crashes and exits the program.	Fail
01/12/2024	Nguyen Vu Cuong	HC016	No error message, system handled unexpected close smoothly.	Pass
01/12/2024	Nguyen Vu Cuong	HC017	The application exits safely.	Pass

4.6, Identify error handling code.

a. BC001: System crashes and exits the program.

- Analysis: This issue indicates that the system crashes when invalid input is provided. The root cause could be a lack of proper exception handling in critical sections, leading to unhandled exceptions and program termination.
- Fix: Add appropriate exception handling for input validation, especially for invalid characters or out-of-range values. Ensure that the program does not terminate unexpectedly and provides user-friendly error messages.

b. BC003: System crashes and exits the program.

- Analysis: Similar to BC001, this suggests an error in the system during a specific action (likely when a menu option is selected).
- Fix: Review the switch case structure in the menu system. Ensure that for every user input, the system validates it before performing any action, and appropriate exception handling is applied.

c. BC004: No error message displayed; updates system as usual.

- Analysis: When adding a new student, the system should validate the inputs (e.g., ensure that all fields are filled and that the input data is in the correct format). However, it appears the system is not providing any error messages.
- Fix: Add input validation checks for all fields when adding a new student. Ensure that error messages are displayed when invalid data is entered.

d. BC005: No error message displayed; updates system as usual.

- Analysis: This issue occurs when updating a student's information, and no error message is displayed if the input is invalid.
- Fix: Implement input validation to check for valid input for each field. For example, ensure the contact number is numeric, the marks are in the correct format, etc. Display error messages if the inputs do not meet the expected format.

e. BC007: No error message displayed; updates system as usual.

- Analysis: Similar to BC005, this occurs when trying to update student information, and the system does not show an error message when the input is incorrect.
- Fix: Implement proper error handling and validation in the update logic to ensure that invalid inputs trigger appropriate error messages.

f. BC008: No "Student not found" message displayed.

- Analysis: When searching for a student, the system does not display an appropriate message when the student is not found.
- Fix: In the searchStudentById or searchStudentByName method, add checks to verify if a student exists. If not, display the message "Student not found."

g. BC009: No students are non-deletable in the system.

- Analysis: The test case expected that there would be students that cannot be deleted, but the system does not support this feature.
- Fix: If this is a feature requirement, add logic to the student deletion function that prevents deletion of certain students (e.g., administrators or students with certain statuses). Implement checks to prevent deletion.

h. BC012: Sorting causes an error and the system exits.

- Analysis: This issue arises during sorting and likely results from improper handling of null or invalid values in the student data.
- Fix: Before sorting, ensure that all values used for comparison (e.g., marks) are valid and non-null. Add error handling to catch any unexpected issues during sorting.

i. BC016: No error message, system just crashes.

- Analysis: Similar to other crash-related issues, the system is not handling errors gracefully, leading to an abrupt exit.
- Fix: Review all user input and data processing areas to ensure that invalid or unexpected inputs trigger proper error messages without crashing the program.

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

4.5, Identify error handling code.

Problem: The goal is to manage a collection of student records, where each record includes attributes like student ID, name, contact details, marks, and rank. The system should allow for easy manipulation of these records through operations such as:

- Add New Students: Store student information upon enrollment.
- Remove Students: Remove students from the list when they drop out.
- Update Student Information: Modify data when there are personal or academic changes.
- Search Students: Retrieve information quickly based on student ID.
- Sort Students: Rank students by grades or other criteria.
- Display List: Review and manage the student list.

4.5, Identify error handling code.

1, Stack for Data Management:

- The StudentManagement class uses a Stack to hold student records. A stack operates on a LastIn-First-Out (LIFO) principle, meaning that the most recently added student is the first to be removed. This approach can work well for scenarios where recent records are more likely to be modified or removed.

2, Core Operations and Their Processes:

a, Adding a Student to the List (Adding a Student to the Stack)

The addStudent method in the StudentManagement class uses the StudentStack to add students to the list. When a new student object is created, it is added to the top of the stack. The StudentStack ADT manages this stack.

- Process: A new Student object is created and pushed onto the stack using the push() method of the StudentStack.
- You can see the algorithm code in sentence **A** (9.3.2, Implement the Algorithm).

b, Updating a Student's Information (Updating Student Information by ID)

The algorithm uses a temporary stack to move all students out of the StudentStack. Then, if a student with a matching ID is found, their information is updated and pushed back onto the stack. If no student is found, an error message is displayed.

- Process: Traverse through the stack, find the student with the matching ID, update their information, and push them back into the stack.
- You can see the algorithm code in sentence **B** (9.3.2, Implement the Algorithm).

c, Deleting a Student from the List (Deleting a Student by ID)

The algorithm uses a temporary stack to keep track of the students who are not being deleted. If a student with the matching ID is found, that student is removed. Afterward, the stack is restored by pushing the remaining students back into the stack.

- Process: Traverse through the stack, find the student with the matching ID, remove them, and restore the stack by pushing the remaining students back.
- You can see the algorithm code in sentence **C** (9.3.2, Implement the Algorithm).

d, Searching for a Student by ID (Searching for a Student by ID)

The searchStudentById method traverses the stack and searches for a student with a matching ID. If found, the student's information is printed. After searching, the stack is restored to its original state.

- Process: Traverse through the stack, find the student with the matching ID, print their information, and restore the stack by pushing the students back.
- You can see the algorithm code in sentence **F** (9.3.2, Implement the Algorithm).

e, Sorting Students by Grades (Sorting Students)

The bubbleSort and quickSort methods are used to sort students based on their grades. First, the student data is transferred to an array, sorted by grades, and then transferred back to the stack after sorting.

- Process: Transfer students from the stack to an array, apply a sorting algorithm (Bubble Sort or Quick Sort), and transfer the sorted students back into the stack.
- You can see the algorithm code in sentence **D and E** (9.3.2, Implement the Algorithm).

f, Display All Students (Display List)

To implement the display function, the system iterates through the stack and prints each student's information. After displaying, the stack is restored to its original state.

- Process: Traverse through the stack, print the details of each student (such as ID, name, contact details, address, marks, and rank), and restore the stack to maintain the integrity of the data.
- You can see the algorithm code in sentence **G** (9.3.2, Implement the Algorithm).

3, Why the ADT Solves the Problem:

- The Stack provides an efficient way to manage students in a LIFO manner, which works for adding and removing students efficiently. However, for more complex operations like searching and sorting, additional structures like a list are used temporarily, showing that a hybrid approach combining the Stack ADT with other data structures (e.g., Lists) can solve the problem more efficiently.

6, Critically evaluate the complexity of an implemented ADT/algorithm.

When evaluating the complexity of an implemented ADT/algorithm, we consider both time complexity and space complexity.

a, Adding a Student to the List (Adding a Student to the Stack) - Method: addStudent.

Time Complexity: O(1)

- The operation of adding a student to the list (stack) using push() takes constant time since adding to the top of the stack involves a simple operation.

Space Complexity: O(1)

- Only one element is added to the stack, so the space required increases by one unit.

b, Updating Student Information (Updating Student Information by ID) - Method: updateStudent

Time Complexity: O(n)

- In the worst case, the system needs to traverse all students in the stack to find a student with the matching ID. This search operation takes O(n) time, where n is the number of students in the stack. After finding the student, updating their information and restoring the stack involves constant time operations.

Space Complexity: O(n)

- A temporary stack is created to hold the students while traversing, so the space required is O(n) in the worst case.

c, Deleting a Student from the List (Deleting a Student by ID) - Method: deleteStudent

Time Complexity: O(n)

- Similar to the update operation, we must traverse all students in the stack to find and delete the student with the matching ID. Each student in the stack is checked once.

Space Complexity: O(n)

- A temporary stack is also used to store the students that are not being deleted. Therefore, the space required is O(n).

d, Searching for a Student by ID (Searching for a Student by ID) - Method: searchStudentById

Time Complexity: O(n)

- Similar to the update and delete operations, the system must traverse all students in the stack to search for a student with the matching ID. The time complexity for this search is O(n).

Space Complexity: O(n)

- This method also uses a temporary stack to store students during traversal, so the space required is O(n).

e, Sorting Students by Grades (Sorting Students) - Method: bubbleSort or quickSort

Time Complexity:

Bubble Sort: O(n^2)

- Bubble Sort has a time complexity of O(n^2) in the worst case because the algorithm requires two nested loops to compare and swap elements in the array.

Quick Sort: O(n log n)

- Quick Sort is a divide-and-conquer algorithm with an average time complexity of O(n log n), but in the worst case (with already sorted or nearly sorted data), the time complexity can degrade to O(n^2).

Space Complexity:

Bubble Sort: O(n)

- Bubble Sort does not require significant auxiliary space as it sorts the array in place.

Quick Sort: O(log n)

- Quick Sort requires O(log n) space for recursion in the best and average cases.

f, Displaying the List of Students (Display List) - Method: displayList

Time Complexity: O(n)

- To display the list of students, the system must traverse all students in the stack and print each student's information. This operation takes O(n) time.

Space Complexity: O(1)

- The space required for this operation is constant because no new data is created during the process.

7, Discuss how asymptotic 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 time and space complexity as the input size grows. It helps in understanding how the performance of an algorithm scales and whether it is suitable for handling large inputs. Asymptotic analysis provides a way to describe the behavior of an algorithm in the long run, regardless of hardware or specific implementation details. Here's how asymptotic analysis can be used to assess the effectiveness of an algorithm:

a, Time Complexity.

Time complexity refers to the amount of time an algorithm takes to complete its task as a function of the input size. Asymptotic analysis uses big-O notation to describe the upper bound of an algorithm's time complexity, which helps us understand how the execution time increases as the input size increases.

Big-O Notation (O): Describes the worst-case upper bound of an algorithm. It provides an upper limit on the running time, ensuring the algorithm will not take longer than a certain amount of time in the worst-case scenario.

- O(1): Constant time, the algorithm's running time doesn't depend on the input size.
- O(n): Linear time, the running time increases linearly with the input size.
- O(n^2): Quadratic time, the running time increases quadratically as the input size grows (e.g., bubble sort).
- O(log n): Logarithmic time, the running time increases logarithmically with the input size (e.g., binary search).
- O(n log n): Log-linear time, typical of efficient sorting algorithms like quicksort and mergesort.

b, Space Complexity.

Space complexity measures the amount of memory an algorithm uses as a function of the input size. Asymptotic analysis also uses big-O notation to assess space complexity.

- O(1): Constant space, the algorithm uses a fixed amount of memory regardless of input size.
- O(n): Linear space, the memory required increases linearly with the input size.
- O(n^2): Quadratic space, the memory usage grows quadratically with input size.

c, Asymptotic Classes and Their Implications.

By analyzing an algorithm's time and space complexity, we can classify it into different asymptotic classes, which help in determining its practical performance:

- Best-case: The minimum time required to run an algorithm for the best possible input.
- Worst-case: The maximum time required for the algorithm to run, typically used in big-O notation.
- Average-case: The expected time for a random input of a given size, though this is more complex to calculate and may require probabilistic models.

d, Comparing Algorithms.

- Asymptotic analysis allows for a fair comparison of different algorithms. For instance, when choosing between a quicksort (O(n log n)) and a bubble sort (O(n^2)) algorithm, asymptotic analysis shows that quicksort is generally more efficient for large inputs. Even if bubble sort performs well for small data sets, asymptotic analysis reveals that it will not scale well as the input size grows.

e, Scalability.

- Asymptotic analysis helps assess how well an algorithm will perform as the input size increases. An algorithm with a higher complexity (such as O(n^2)) may be ineffective for large inputs, while an algorithm with lower complexity (like O(log n) or O(n)) will scale much better. This is particularly important in applications where the data set can grow substantially, such as big data or real-time systems.

f, Practical Effectiveness.

- Although asymptotic analysis gives a theoretical view of an algorithm's performance, it is important to note that it doesn't account for factors like constant factors, lower-order terms, or the specifics of hardware. In practice, an algorithm with a better asymptotic complexity may not always perform better in real-world applications due to factors like caching, memory usage, and parallelization. Hence, while asymptotic analysis is valuable for understanding an algorithm's theoretical behavior, empirical testing and profiling are still necessary for practical performance assessment.

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

4.8, Example of code illustration.

The sorting results of the QuickSort and BubbleSort algorithms:

a, arrange 10 students.

QuickSort:

```
Students sorted using Quick Sort.
Quick Sort Time: 14700 nanoseconds.
```

BubbleSort:

```
Students sorted using Bubble Sort.
Bubble Sort Time: 51600 nanoseconds.
```

b, arrange 100 students.

QuickSort:

```
Students sorted using Quick Sort.
Quick Sort Time: 103100 nanoseconds.
```

BubbleSort:

```
Students sorted using Bubble Sort.
Bubble Sort Time: 938200 nanoseconds.
```

c. Arrange 1000 students.

QuickSort:

```
Students sorted using Quick Sort.
Quick Sort Time: 822800 nanoseconds.
```

BubbleSort:

```
Students sorted using Bubble Sort.
Bubble Sort Time: 28714600 nanoseconds.
```

d. Arrange 10000 students.

QuickSort:

```
Students sorted using Quick Sort.
Quick Sort Time: 4321800 nanoseconds.
```

BubbleSort:

```
Students sorted using Bubble Sort.
Bubble Sort Time: 1084480800 nanoseconds.
```

9.8.2, Time complexity.

In this example, two sorting algorithms are used: Quick Sort and Bubble Sort. Below is a detailed analysis

a, Quick Sort.

Average complexity: O(n log n).

- By dividing the list into two roughly equal parts and processing recursively, the number of required comparisons decreases exponentially, making it more efficient than $O(n^2)$ algorithms.

Worst-case complexity: $O(n^2)$.

- This occurs if the pivot is chosen poorly (e.g., always the largest or smallest element), resulting in unbalanced partitions.

Observed times from the example:

- 10 students: 14,700 nanoseconds
- 100 students: 103,100 nanoseconds
- 1,000 students: 822,800 nanoseconds
- 10,000 students: 4,321,800 nanoseconds

Observation: As data size increases, the processing time scales linearly with **n log n**, demonstrating the efficiency of this algorithm.

b, Bubble Sort.

Complexity: $O(n^2)$.

- Every element must be compared with all others in the list, causing the number of comparisons to grow quadratically as data size increases.

Observed times from the example:

- 10 students: 51,600 nanoseconds
- 100 students: 938,200 nanoseconds
- 1,000 students: 28,714,600 nanoseconds
- 10,000 students: 1,084,480,800 nanoseconds

Observation**:** The execution time grows quadratically with data size, making this algorithm unsuitable for large datasets

9.8.3, Spatial complexity.

a, Quick Sort:

- **Space Complexity**: O(log n), due to recursion stack usage.
- This is acceptable for small to moderately large datasets but can cause stack overflow for extremely large datasets if not managed properly.

b, Bubble Sort:

- **Space Complexity**: O(1), as it operates in-place without requiring additional memory.
- This makes it more suitable for scenarios with memory constraints

9.8.4, Explain what the trade-off is when appointing an ADT.

a, Time Complexity vs. Spatial Complexity.

Quick Sort:

- Prioritizes faster execution at the cost of higher space usage (due to recursion).
- Efficient for large datasets due to its logarithmic depth but might cause memory issues with extremely large datasets or limited stack sizes.

Bubble Sort:

- Minimizes space usage but requires significantly more time, especially for large datasets.Can be useful for very small datasets or situations where memory is extremely constrained.

b, Flexibility vs. Performance.

Flexibility of ADT:

- The StudentManagement ADT is flexible, allowing operations like adding, updating, deleting, and sorting. This makes it suitable for various scenarios.

Performance:

- Sorting algorithms like Quick Sort offer higher performance but require more careful memory management.
- Bubble Sort is simple and space-efficient but lacks the performance needed for scalability.

c, Balance.

- Flexibility vs. Performance: To enhance performance without sacrificing flexibility, hybrid approaches like combining Quick Sort with other algorithms (e.g., Insertion Sort for small partitions) can be considered.
- Time vs. Space: For large-scale applications, choosing Quick Sort or other efficient algorithms is typically better, while ensuring that the recursion depth is manageable.

9, Evaluate three benefits of using implementation independent data structures.

9.9.1, Portability.

Portability refers to the ability of a data structure to be used across different platforms, environments, or programming languages without modification. Implementation-independent data structures abstract away the low-level details of how the data is stored or manipulated, making it easier to adapt the software for different systems or compilers. This means that:

- Developers can use the same abstract data structure across various platforms (e.g., from Windows to Linux or between 32-bit and 64-bit systems) without needing to rewrite or adjust the structure.
- The same logic can be applied to different hardware or architectures, improving compatibility.

Example: A queue data structure defined as an ADT can be implemented with arrays in one environment and with linked lists in another, without changing the program that uses the ADT.

9.9.2, Reusability.

Reusability means that a data structure can be reused in different programs, modules, or components. When data structures are defined independently of specific implementation details, they become general-purpose and flexible, making it possible to use them across different projects without modification. This leads to:

- Reduced development time: Developers don’t need to rewrite commonly used data structures; they can simply reuse existing abstractions.
- Increased consistency: Using the same data structure in multiple parts of the codebase or across different projects ensures consistency in behavior, improving the quality of software.

Example: A "Stack" ADT can be reused across various applications that require LIFO (Last In First Out) functionality, such as expression evaluation, backtracking algorithms, or undo features.

9.9.3, Maintainability.

Maintainability refers to how easy it is to modify or extend the codebase as requirements evolve. By decoupling the data structure's logic from its implementation details, maintainability is improved because:

- Developers can modify the internal workings of the data structure without affecting the rest of the application. Since the interface remains consistent, changes to the implementation do not require changes to other parts of the system that interact with the data structure.
- Bug fixes and performance improvements can be made to the data structure independently, making it easier to manage long-term changes.

Example: If a performance bottleneck is discovered in the implementation of a linked list (such as slow traversal), it can be optimized (e.g., by implementing a doubly linked list) without changing the software that uses the linked list ADT.

Thank you for listening to my presentation.