Higher Nationals - Summative Assignment Feedback Form

Student Name/ID	MOHAMMED MAHROOF MOHAMMED AASHIK/E230667		
Unit Title	Unit 19: Data Structures and Algorithms		
Assignment Number	1 of 1	**Assessor**	Miss. Sasini
Submission Date	31.07.2025	**Date Received 1st submission**	31.07.2025
Re-submission Date		**Date Received 2nd submission**	

Assessor Feedback:		
Grade:	**Assessor Signature:**	**Date:**
Resubmission Feedback: *Please note resubmission feedback is focussed only on the resubmitted work		
Grade:	**Assessor Signature:**	**Date:**
Internal Verifier's Comments:		
Signature & Date:		

* Please note that grade decisions are provisional. They are only confirmed once internal and external moderation has taken place and grades decisions have been agreed at the assessment board.

Important Points:

1. It is strictly prohibited to use textboxes to add texts in the assignments, except for the compulsory information. eg: Figures, tables of comparison etc. Adding text boxes in the body except for the before mentioned compulsory information will result in rejection of your work.
2. Avoid using page borders in your assignment body.
3. Carefully check the hand in date and the instructions given in the assignment. Late submissions will not be accepted.
4. Ensure that you give yourself enough time to complete the assignment by the due date.
5. Excuses of any nature will not be accepted for failure to hand in the work on time.
6. You must take responsibility for managing your own time effectively.
7. If you are unable to hand in your assignment on time and have valid reasons such as illness, you may apply (in writing) for an extension.
8. Failure to achieve at least PASS criteria will result in a REFERRAL grade .
9. Non-submission of work without valid reasons will lead to an automatic RE FERRAL. You will then be asked to complete an alternative assignment.
10. If you use other people's work or ideas in your assignment, reference them properly using HARVARD referencing system to avoid plagiarism. You have to provide both in-text citation and a reference list.
11. If you are proven to be guilty of plagiarism or any academic misconduct, your grade could be reduced to A REFERRAL or at worst you could be expelled from the course
12. Use word processing application spell check and grammar check function to help editing your assignment.
13. Use **footer function in the word processor to insert Your Name, Subject, Assignment No, and Page Number on each pag**e. This is useful if individual sheets become detached for any reason.

STUDENT ASSESSMENT SUBMISSION AND DECLARATION

When submitting evidence for assessment, each student must sign a declaration confirming that the work is their own.

Student name: MOHAMMED AASHIK	Assessor name:	
Issue date:	Submission date:	Submitted on: **31.07.2025**
Programme: **BTEC HND in Computing**		
Unit: Unit 19: Data Structures and Algorithms		
Assignment number and title: Specification, Implementation, and Assessment of Data Structures for a sample scenario.		

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.

Guidelines for incorporating AI-generated content into assignments:

The use of AI-generated tools to enhance intellectual development is permitted; nevertheless, submitted work must be original. It is not acceptable to pass off AI-generated work as your own.

Student Declaration

Student declaration

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

Student signature: E230667 Date:

Turnitin Similarity Report

Plagiarism Percentage

Assessor Comments on Turnitin report, Report content, and Grade	

Unit 19: Data Structures and Algorithms

Assignment Brief

Student Name/ID Number	MOHAMMED MAHROOF MOHAMMED AASHIK/E230667
Unit Number and Title	**Unit 19: Data Structures and Algorithms**
Academic Year	2024/25
Unit Tutor	
Assignment Title	**Specification, Implementation, and Assessment of Data Structures for a sample scenario.**
Issue Date	
Submission Date	

Submission Format

The submission should be in the form of a report, which contains code snippets (which must be described well), text-based descriptions, and diagrams where appropriate. References to external sources of knowledge must be cited (reference list supported by in-text citations) using the Harvard Referencing style.

Please note that to pass the module, all deliverables must be completed.

Unit Learning Outcomes

LO1. Examine abstract data types, concrete data structures and algorithms.
LO2. Specify abstract data types and algorithms in a formal notation.
LO3. Implement complex data structures and algorithms.
LO4. Assess the effectiveness of data structures and algorithms.

Transferable skills and competencies developed

Cognitive Thinking

Demonstration of familiarity with and comprehension of key ideas, theories, facts, and concepts pertaining to computing and computer applications.

Understanding the moral, ethical, social, economic, professional, and environmental concerns associated with the sustainable use of computer technology.

Analyze how well a computer-based system satisfies the requirements established for its present use and future advancement through critical review and testing.

Use the right concepts, procedures, and resources for computer-based system design, implementation, and assessment.

Intellectual Skills

Problem-solving and logical reasoning.

Adaptability to new computational paradigms.

Effective decision-making based on data-driven insights.

Self-Control

The capacity to effectively plan and complete projects by the deadline.

Efficient prioritizing and time management.

Self-directed education and the capacity to adjust to new developments in technology.

Contextual Awareness

Understanding how software solutions impact individuals, businesses, and society.

Vocational scenario

Scenario

Digital solutions in the healthcare sector have advanced rapidly, making it simpler for people to make doctor's appointments. It is still difficult to effectively manage patient records, doctor availability, and appointment scheduling.

A doctor channelling system is a software program created to give patients a simple way to schedule and manage their appointments. By facilitating smooth communication between physicians, patients, and hospital management, the system lowers wait times and raises patient satisfaction.

Assume you work as a software engineer for XYZ Pvt Ltd. Your organization has been asked to create a doctor channelling system that meets the following specifications:

1. **Patient Registration** (Patient name, Mobile number, Email ID, City, Age, Medical History)
2. **Doctor Registration** (Doctor ID, Name, Specialization, Available Time Slots, Consultation Fee)
3. **Patients can search for available doctors**
4. **Patients can book an appointment.** As soon as an appointment is made, the respective patient should be notified through a message. All the patient's appointments should be saved for future reference.
5. **Patients can cancel an appointment at any time.** Upon cancellation, the respective patient should be notified, and the next patient on the waiting list should be assigned the slot and notified accordingly.
6. **Patients can request a reschedule.** During this time, the respective patient should wait in queue.
7. **All scheduled appointments should be displayed.**

Assignment activity and guidance

Activity 1:

- Examine and create data structures by analyzing the above scenario and explain the valid operations that can be carried out on these data structures. Determine the operations of a queue and critically review how it is used to implement different operations.
- XYZ Pvt Ltd also provides home visit services for patients who cannot visit the hospital. To optimize the scheduling of these visits, the company wants to determine the shortest route for doctors to travel between patient locations. Analyze this routing problem using two shortest path algorithms, illustrating how each algorithm processes a sample graph to find the most efficient path.
- Additionally, to ensure priority-based patient handling, sort the patients based on age using two different sorting algorithms. Critically compare the performance of these algorithms in terms of time complexity and suitability for large datasets.

Activity 2:

- Implement the above scenario using the selected data structure and its valid operations for the design specification given in Task 1 by using Java programming. Use suitable error handling and test the application using appropriate test cases. Illustrate the system and provide evidence of the test cases and test results.
- "Imperative ADTs are the basis for object orientation." Discuss this statement, stating whether you agree or not. Justify your answer.

Activity 3:

Registered patient details are stored from the oldest to the newest. The management of XYZ Pvt Ltd should be able to retrieve patient details from the newest to the oldest. Using an imperative definition, specify the abstract data type (ADT) for the above scenario and

implement the specified ADT using Java programming. Critically analyze the complexity of the chosen ADT algorithm. Examine the advantages of Encapsulation and Information Hiding when using the selected ADT.

Activity 4:

Evaluate how Asymptotic Analysis can be used to assess the effectiveness of an algorithm and discuss at least two ways in which the efficiency of an algorithm can be measured with relevant examples. Explain the trade-offs that exist when using an ADT for implementing programs by supporting your answer with specific examples. Evaluate the benefits of using independent data structures for implementing programs.

Recommended Resources
Please note that the resources listed are examples for you to use as a starting point in your research – the list is not definitive.

General Introduction to Data Structures and Algorithms

- GeeksforGeeks (n.d.) Data Structures Basics [online]. Available at: https://www.geeksforgeeks.org/data-structures/ [Accessed 27 March 2025]
- W3Schools (n.d.) Java Data Structures [online]. Available at: https://www.w3schools.com/java/java_data_types.asp [Accessed 27 March 2025]
- TutorialsPoint (n.d.) Introduction to Data Structures [online]. Available at: https://www.tutorialspoint.com/data_structures_algorithms/data_structures_basics.htm [Accessed 27 March 2025]

Shortest Path Algorithms

- GeeksforGeeks (n.d.) Shortest Path Algorithms – Dijkstra and Floyd-Warshall [online]. Available at: https://www.geeksforgeeks.org/shortest-path-algorithms-dijkstra-floyd-warshall/ [Accessed 27 March 2025]
- Brilliant (n.d.) Graph Theory and Shortest Path Algorithms [online]. Available at: https://brilliant.org/wiki/shortest-path/ [Accessed 27 March 2025]

Sorting Algorithms

- TutorialsPoint (n.d.) Java Sorting Algorithms [online]. Available at:

https://www.tutorialspoint.com/java/java_quick_sort_algorithm.htm [Accessed 27 March 2025]
- GeeksforGeeks (n.d.) Sorting Algorithms – Bubble, Quick, Merge, and Insertion Sort [online]. Available at: https://www.geeksforgeeks.org/sorting-algorithms/ [Accessed 27 March 2025]

Java Programming Resources

- Oracle (n.d.) Java Documentation [online]. Available at: https://docs.oracle.com/javase/tutorial/ [Accessed 27 March 2025]
- Codecademy (n.d.) Learn Java [online]. Available at: https://www.codecademy.com/learn/learn-java [Accessed 27 March 2025]

Learning Outcomes and Assessment Criteria

Pass	Merit	Distinction
LO1. Examine abstract data types, concrete data structures and algorithms.		**D1** Analyse the operation, using illustrations, of two network shortest path algorithms, providing an example of each.
P1 Create a design specification for data structures explaining the valid operations that can be carried out on the structures. **P2** Determine the operations of a memory stack and how it is used to implement function calls in a computer.	**M1** Illustrate, with an example, a concrete data structure for a First In First out (FIFO) queue. **M2** Compare the performance of two sorting algorithms.	
LO2. Specify abstract data types and algorithms in a formal notation.		

P3 Using an imperative definition, specify the abstract data type for a software stack.	**M3** Examine the advantages of encapsulation and information hiding when using an ADT.	**D2** Discuss the view that imperative ADTs are a basis for object orientation and, with justification, state whether you agree.

LO3. Implement complex data structures and algorithms.		
P4 Implement a complex ADT and algorithm in an executable programming language to solve a well-defined problem. **P5** Implement error handling and report test results.	**M4** Demonstrate how the implementation of an ADT/algorithm solves a well-defined problem.	**D3** Critically evaluate the complexity of an implemented ADT/algorithm.
LO4. Assess the effectiveness of data structures and algorithms.		
P6 Discuss how asymptotic analysis can be used to assess the effectiveness of an algorithm. **P7** Determine two ways in which the efficiency of an algorithm can be measured, illustrating your answer with an example.	**M5** Interpret what a trade-off is when specifying an ADT using an example to support your answer.	**D4** Evaluate three benefits of using implementation independent data structures.

ACKNOWLEDGEMENT

I am deeply grateful for the assistance and guidance I received from numerous esteemed individuals, which was instrumental in the successful completion of my task. I would like to express my sincere appreciation to ESOFT for providing a conducive workspace that facilitated the completion of my task. I am delighted to announce the successful completion of the assignment. I am particularly indebted to Mrs. **Sasini** for his invaluable guidance throughout my fourth semester assignments. Lastly, I extend my heartfelt gratitude to my family members and classmates whose unwavering support greatly contributed to the timely completion of this project. Thank you all for your immense contribution!

CONTENT

Contents

Plagiarism 3

Unit 19: Data Structures and Algorithms 5

Please note that the resources listed are examples for you to use as a starting point in your research – the list is not definitive. 8

General Introduction to Data Structures and Algorithms 8

Shortest Path Algorithms 8

Sorting Algorithms 8

Java Programming Resources 9

Activity 01 30

Indroduction 30

In today's world, people want quick and easy access to healthcare services. With more hospitals and clinics competing to offer the best care, it has become important to provide fast and convenient ways for patients to make appointments. Many people now use smartphones and the internet, so they prefer booking appointments online rather than waiting in long lines or making phone calls. To meet this demand, the Doctor Channeling System was created. It is a simple and easy-to-use software that lets patient's book appointments with doctors through an online platform. 30

This system handles important information such as patient details, doctor schedules, and appointment times. It helps reduce human errors that can happen when booking is done manually. It also saves time for both the patients and the staff. With everything managed through the system, appointments can be scheduled faster and more accurately. This helps medical centers improve their daily operations and provide better service to their patients. 30

The system reduces waiting times at the clinic or hospital by allowing patients to select available time slots beforehand. It also helps doctors and staff plan their schedules better. Overall, the Doctor Channeling System improves the booking process, makes it more efficient, and ensures a better experience for everyone involved. By using this system, healthcare providers can offer a modern and professional service that meets the expectations of today's patients while also improving the quality and reliability of care. 30

Introduction to Data Structures 30

In computer science, a data structure is a special way of organizing and storing data so that it can be used efficiently. When we build programs, we often need to work with large amounts of data. A data structure helps us manage this data so we can access, update, or remove it easily whenever needed. Whether we are building a small mobile app or a complex software system, data structures help in making the program faster and more efficient. 30

Different types of data structures are used depending on the nature of the data and the operations we want to perform. For example, if we need to keep track of items in order, we might use an array or a list. If we want to manage tasks that follow a "first come, first serve" approach, a queue would be more suitable. Choosing the right data structure is very important because it directly affects the performance of the application. Good knowledge of data structures helps programmers write better code that runs faster and uses less memory. 31

Explanation of Each Data Structure 35

Edge Cases and Error Handling 49

Important of Edge cases and error handling 50

Edge cases and error handling are important for several reasons: 50

- Reliability: The system keeps working even if something unexpected happens. 50
- Better User Experience: Users receive clear messages instead of confusing errors. 50
- Security: Handling errors properly can stop hackers from taking advantage of weak points. 50
- Easier Maintenance: If errors are reported clearly, it becomes easier for developers to fix them. 50

Implementation of Function Calls Related to a Developed to the doctor channeling system 55

Developed a doctor channeling system in Java. When a patient books an appointment, the system may call several functions one after the other 55

1. registerPatient() 55
2. checkDoctorAvailability() 55
3. createAppointment() 55
4. sendConfirmationMessage() 55

Each time one of these functions is called, the system uses the memory stack to store the function's details: 55

- The location to return after the function ends (return address) 55
- The values passed to the function (parameters) 55
- Any temporary values used inside the function (local variables) 55

Example Flow Using Memory Stack 55

1. Main () function starts the program. 55
2. RegisterPatient () is called a stack frame is pushed onto the memory stack. 55

3. Inside it, checkDoctorAvailability () is called a new stack frame is pushed. 55
4. Then, createAppointment () is called another stack frame is pushed. 55
5. Finally, sendConfirmationMessage () is called another frame is added. 55

Once sendConfirmationMessage () finishes, its stack frame is popped off. Then control returns to createAppointment (), which also finishes, and its frame is removed. This continues until we return to main () and the stack is empty again. 55

Stack 56

Implementation of a Stack in Java 59

XYZ Pvt Ltd Doctor Channelling System 70

Linked list 71

Patient Linked list 72

Doctor Linked list 72

Appointments Linked list 72

Appointment data will be stored in this Linked List based Queue. Each node will consist of patient's mobile number, doctor id and Available time slots. 72

Implementing the system using Java Language 74

Class patient 74

```
class Patient
{
   String name;
   String mobile;
   String email;
   String city;
   int age;
   String medicalhistory;
   Patient next;
```

```
public   Patient(String name, String mobile, String email, String city, int age, String history) {
    this.name = name;
    this.mobile = mobile;
    this.email = email;
    this.city = city;
    this.age = age;
    this.medicalhistory = medicalhistory;
 }

 public  String getName() {
    return name;
 }

 public String toString() {
    return "Name: " + name + ", Mobile: " + mobile + ", Email: " + email + ", City: " + city + ", Age: " + age + ",medicalHistory: " + medicalhistory;
 }
}
```

..... 74

Class doctor 74

```
class Doctor {
    String id;
    String name;
    String specialization;
    String[] slots;
    double fee;
    Doctor next;
```

..... 74

```
public Doctor(String id, String name, String specialization, String[] slots, double fee) {
    this.id = id;
    this.name = name;
    this.specialization = specialization;
    this.slots = slots;
    this.fee = fee;
}

public String getId() {
    return id;
}

public String getSpecialization() {
    return specialization;
}

public String toString() {
    return "Doctor ID: " + id + ", Name: " + name + ", Specialization: " + specialization + ", Fee: " + fee;
}
}
```

..... 75

Class appointment..... 75

```
class Appointment {
    Patient patient;
    Doctor doctor;
    String timeSlot;
    Appointment next;
```

..... 75

```
    public Appointment(Patient patient, Doctor doctor, String timeSlot) {
        this.patient = patient;
        this.doctor = doctor;
        this.timeSlot = timeSlot;
    }

    public Patient getPatient() {
        return patient;
    }

    public String toString() {
        return "Patient: " + patient.getName() + ", Doctor: " + doctor.toString() + ", Slot: " + timeSlot;
    }
}
```

..... 75

```
import java.util.*;
import java.util.ArrayList;
import java.util.Scanner;
```

..... 76

Doctor channeling system 76

```java
//doctor channeling system of xyz private limited
class Doctorchanelingsystem {

    static Scanner input = new Scanner(System.in);
    static List<Patient> patients = new ArrayList<>();
    static List<Doctor> doctors = new ArrayList<>();
    static Queue<Appointment> waitingList = new LinkedList<>();
    static List<Appointment> appointments = new ArrayList<>();

  public static void main(String[] args) {
Doctorchanelingsystem system = new Doctorchanelingsystem();
 Scanner input = new Scanner(System.in);
      while (true) {
         System.out.println("");
         System.out.println("$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$");
         System.out.println("Doctor Channeling System of XYZ Private Limited");
         System.out.println("1. Register Patient");
         System.out.println("2. Register Doctor");
         System.out.println("3. Search Doctor");
         System.out.println("4. Book Appointment");
         System.out.println("5. Cancel Appointment");
         System.out.println("6. Request Reschedule");
         System.out.println("7. Show All Appointments");
         System.out.println("8. Exit");
         System.out.println("9. Display All Registered Patients");
         System.out.println("10. Display All Registered Doctors");
         System.out.print("Enter your choice:1  |  2|  3|  4|  5|  6|  7|  8|  9|  10   ");
         int choice = input.nextInt();
         input.nextLine();  // clear buffer
```

.... 76

```
        System.out.print("Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10  ");
        int choice = input.nextInt();
        input.nextLine();  // clear buffer

        if (choice == 1) {
            registerPatient();
        } else if (choice == 2) {
            registerDoctor();
        } else if (choice == 3) {
            searchDoctor();
        } else if (choice == 4) {
            bookAppointment();
        } else if (choice == 5) {
            cancelAppointment();
        } else if (choice == 6) {
            requestReschedule();
        } else if (choice == 7) {
            displayAppointments();
        } else if (choice == 8) {
            System.out.println("YOU ARE EXIT FROM THE XYZ CHANELLING SYSTEM. THANK YOU, WELCOME!");
            break;
        } else if (choice == 9) {
            displayAllPatients();
        } else if (choice == 10) {
            displayAllDoctors();
        } else {
            System.out.println("Invalid choice. Try again.");
        }
    }

    input.close();
}
```

..... 77

```
    }
//DISPLAYING THE REGISTERED PATIENT
    public static void displayAllPatients() {
        System.out.println("**** **** Registered Patients **** ****");
        if (patients.isEmpty()) {
            System.out.println("**** No patients registered ****.");
        } else {
            for (int i = 0; i < patients.size(); i++) {
                System.out.println((i + 1) + ". " + patients.get(i));
            }
        }
    }
```

..... 78

```
    public static void displayAllDoctors() {
        System.out.println("**** **** Registered Doctors **** ****");
        if (doctors.isEmpty()) {
            System.out.println("**** No doctors registered ****.");
        } else {
            for (int i = 0; i < doctors.size(); i++) {
                System.out.println((i + 1) + ". " + doctors.get(i));
            }
        }
    }
```

..... 78

```
//register a doctor in xyz private limited
    static void registerDoctor() {
        Scanner input = new Scanner(System.in);
        System.out.println(" ");
        System.out.println("**** **** Register Doctor **** ****");
        System.out.print("Doctor ID: ");
        String id = input.nextLine();
        System.out.print("Name: ");
        String name = input.nextLine();
        System.out.print("Specialization: ");
        String specialization = input.nextLine();
        System.out.print("Available Time Slots (comma-separated): ");
        String[] slots = input.nextLine().split(", ");
        System.out.print("Consultation Fee: ");
        double fee = input.nextDouble();
        input.nextLine();

        doctors.add(new
         Doctor(id, name, specialization, slots, fee));
        System.out.println("Doctor registered successfully for the XYZ private limited!");
    }
```

..... 79

```
//register a patient in xyz private limited
    public static void registerPatient() {
Scanner input = new Scanner(System.in);
        System.out.println("");
        System.out.println("**** **** Register Patient **** ****");
        System.out.print("Name: ");
        String name = input.nextLine();
        System.out.print("Mobile: ");
        String mobile = input.nextLine();
        System.out.print("Email: ");
        String email = input.nextLine();
        System.out.print("City: ");
        String city = input.nextLine();
        System.out.print("Age: ");
        int age = input.nextInt();
        input.nextLine(); // consume newline
        System.out.print("Medical History: ");
        String history = input.nextLine();

        patients.add(new
         Patient(name, mobile, email, city, age, history));
        System.out.println("Patient registered successfully for the XYZ private limited!");
    }
```

..... 80

```
// searching the doctor in xyz limited
    static void searchDoctor() {
        System.out.print(" ");
        System.out.print("Enter specialization to search: ");
        System.out.print("Prefered day for visit the chaneling: ");
        String spec = input.nextLine();
        for (Doctor doc : doctors) {
            if (doc.getSpecialization().equalsIgnoreCase(spec)) {
                System.out.println(doc);
            }
        }
    }
//book and appoinment in xyz private limited
    static void bookAppointment() {
        System.out.print(" ");
        System.out.print("Enter patient name: ");
        String patientName = input.nextLine();
        System.out.print("Enter doctor ID: ");
        String docId = input.nextLine();

        Patient foundPatient = null;
        for (Patient p : patients) {
            if (p.getName().equalsIgnoreCase(patientName)) {
                foundPatient = p;
                break;
            }
        }
```

..... 81

```
        Doctor foundDoctor = null;
        for (Doctor d : doctors) {
            if (d.getId().equalsIgnoreCase(docId)) {
                foundDoctor = d;
                break;
            }
        }

        if (foundPatient != null && foundDoctor != null) {
            System.out.print("Enter preferred time slot: ");
            String slot = input.nextLine();

            appointments.add
        (new Appointment(foundPatient, foundDoctor, slot));
            System.out.println("Appointment booked and message sent to patient!");
        } else {
            System.out.println("Patient or Doctor not found.");
        }
    }
```

..... 82

```
    //cancel the doctor appoinment
    public static void cancelAppointment() {
        System.out.print("");
        System.out.print("Enter patient name to cancel appointment: ");
        String name = input.nextLine();

        Iterator<Appointment> iter = appointments.iterator();
        while (iter.hasNext()) {
            Appointment a = iter.next();
            if (a.getPatient().getName().equalsIgnoreCase(name)) {
                iter.remove();
                System.out.println("Appointment cancelled. Message sent to patient.");
                if (!waitingList.isEmpty()) {
                    Appointment next = waitingList.poll();
                    appointments.add(next);
                    System.out.println("Next patient in queue notified for slot.");
                }
                return;
            }
        }

        System.out.println("No appointment found for that name.");
    }
```

..... 82

```
            System.out.println("No appointment found for that name.");
        }
    //resedule
        static void requestReschedule() {
            System.out.print(" ");
            System.out.print("Enter patient name for rescheduling: ");
            String name = input.nextLine();
            for (Appointment a : appointments) {
                if (a.getPatient().getName().equalsIgnoreCase(name)) {
                    waitingList.add(a);
                    System.out.println("Patient added to queue for rescheduling.");
                    return;
                }
            }
            System.out.println("Appointment not found.");
        }
    // display all reservations
        static void displayAppointments() {
            System.out.println("");
            System.out.println("Scheduled Appointments for the XYZ private limited:");
            for (Appointment a : appointments) {
                System.out.println(a);
            }
        }
    }
```

..... 83

```
        // Consultation fee input and validation
        System.out.print("Enter Consultation Fee: ");
        double fee = input.nextDouble();
        input.nextLine(); // Consume the newline character
        if (fee < 0) {
            throw new IllegalArgumentException("Consultation fee cannot be negative.");
        }

        // Add doctor to the list
        doctors.add(new Doctor(id, name, specialization, timeSlot, fee));
        System.out.println(" Doctor registered successfully.");

    } catch (InputMismatchException e) {
        System.out.println(" Invalid input. Please enter correct numeric values.");
        input.nextLine(); // clear the buffer
    } catch (IllegalArgumentException e) {
        System.out.println(" " + e.getMessage());
    } catch (Exception e) {
        System.out.println(" Unexpected error: " + e.getMessage());
    }
```

..... 83

```
public static void registerDoctor() {
    try {
        // Doctor ID input and validation
        System.out.print("Enter Doctor ID: ");
        String id = input.nextLine();
        if (id.isEmpty()) {
            throw new IllegalArgumentException("Doctor ID cannot be empty.");
        }

        // Doctor name input and validation
        System.out.print("Enter Doctor Name: ");
        String name = input.nextLine();
        if (name.isEmpty()) {
            throw new IllegalArgumentException("Doctor name cannot be empty.");
        }

        // Specialization input and validation
        System.out.print("Enter Specialization: ");
        String specialization = input.nextLine();
        if (specialization.isEmpty()) {
            throw new IllegalArgumentException("Specialization cannot be empty.");
        }
```

..... 84

```
        System.out.print("Enter Age: ");
        int age = input.nextInt();
        input.nextLine();
        if (age <= 0) {
            throw new IllegalArgumentException("Age must be greater than 0.");
        }
```

.... 84

```
        System.out.print("Enter Email: ");
        String email = input.nextLine();
        if (!email.contains("@")) {
            throw new IllegalArgumentException("Invalid email format.");
        }
```

..... 84

```
    public static void registerPatient() {
        try {
            System.out.print("Enter Patient Name: ");
            String name = input.nextLine();
            if (name.isEmpty()) {
                throw new IllegalArgumentException("Name cannot be empty.");
            }
```

..... 84

Explanation of the java code ..... Error! Bookmark not defined.

Explanation of the Java Code for Doctor Channeling System ..... **Error! Bookmark not defined.**

☒ **Patient Class** ..... **Error! Bookmark not defined.**

Represents a registered patient in the system ..... **Error! Bookmark not defined.**

✓ Name: Name of the patient ..... **Error! Bookmark not defined.**

✓ Mobile: Mobile number ..... **Error! Bookmark not defined.**

✓ Email: Email ID ..... **Error! Bookmark not defined.**

✓ City: City of residence ..... **Error! Bookmark not defined.**

✓ Age: Age of the patient ..... **Error! Bookmark not defined.**

✓ Medical History: Medical history details of the patient ..... **Error! Bookmark not defined.**

This class holds the basic information required to identify and manage a patient. . **Error! Bookmark not defined.**

☒ **Doctor Class** ..... **Error! Bookmark not defined.**

Represents a doctor who is available for channeling. ..... **Error! Bookmark not defined.**

✓ Id: Unique doctor ID ..... **Error! Bookmark not defined.**

✓ Name: Name of the doctor ..... **Error! Bookmark not defined.**

✓ Specialization: Doctor's area of expertise ..... **Error! Bookmark not defined.**

✓ Available Slots: day available for appointments ..... **Error! Bookmark not defined.**

✓ Fee: Consultation fee ..... **Error! Bookmark not defined.**

The Doctor class helps in listing and assigning doctors to patients based on their specialization and availability. ..... **Error! Bookmark not defined.**

☒ **Appointment Class** ..... **Error! Bookmark not defined.**

Represents a scheduled appointment between a doctor and a patient. **Error! Bookmark not defined.**

✓ Patient: Patient object (linked to the appointment) ..... **Error! Bookmark not defined.**

✓ Doctor: Doctor Object (linked to the appointment) ..... **Error! Bookmark not defined.**

✓ Slot: The confirmed day for the appointment ..... **Error! Bookmark not defined.**

✓ This class handles each confirmed booking between a patient and a doctor. . **Error! Bookmark not defined.**

☒ **Doctor channeling system Class (Main System Class)** ..... **Error! Bookmark not defined.**

This is the central class of the application. It manages all other classes and performs operations like registration, booking, cancelling, and rescheduling appointments. ... **Error! Bookmark not defined.**

✓ Patients: List that holds all patient records ..... **Error! Bookmark not defined.**

✓ Doctors: List that holds all doctor records ..... **Error! Bookmark not defined.**

✓ Appointments: List that keeps confirmed appointments ..... **Error! Bookmark not defined.**

✓ Waiting List: A Queue that holds patients who are waiting for rescheduling or reassignment ..... **Error! Bookmark not defined.**

✓ **Methods** ..... **Error! Bookmark not defined.**

➢ **registerPatient()** ..... **Error! Bookmark not defined.**

Allows a new patient to be added to the system by collecting details like name, contact, age, and medical history. Adds the patient to the patients list. ..... **Error! Bookmark not defined.**

➢ **registerDoctor()** ..... **Error! Bookmark not defined.**

Collects details for a new doctor including ID, specialization, time slots, and fee. The doctor is then added to the doctors list. ..... **Error! Bookmark not defined.**

- **searchDoctor()** ..... **Error! Bookmark not defined.**

Lets users search for a doctor by specialization. Displays matching doctor(s) from the list. ..... **Error! Bookmark not defined.**

- **bookAppointment()** ..... **Error! Bookmark not defined.**

Books an appointment between a registered patient and an available doctor. After confirmation, it adds a new Appointment object to the appointments list and prints a notification. **Error! Bookmark not defined.**

- **cancelAppointment()** ..... **Error! Bookmark not defined.**

Cancels a scheduled appointment by searching for the patient's name. If a waiting patient is available in the waitingList, the slot is reassigned to them and they are notified. ..... **Error! Bookmark not defined.**

- **requestReschedule()** ..... **Error! Bookmark not defined.**

If a patient cannot attend their appointment, they can request a reschedule. The system adds their appointment to the queue so they can be assigned a new time later. ..... **Error! Bookmark not defined.**

- **displayAppointments()** ..... **Error! Bookmark not defined.**

Shows all confirmed appointments in the system. ..... **Error! Bookmark not defined.**

- **main()** ..... **Error! Bookmark not defined.**

This is the main function that displays a menu with options like registering patients/doctors, booking, cancelling, and viewing appointments. The system runs in a loop and continues to take user input using if-else statements ..... **Error! Bookmark not defined.**

Bubble sort 101

Example Bubble sort in Java 102

Selection sort 103

Example Selection sort in Java 104

Recursion is not ideal 117

None of the tasks in the system require recursive logic like tree traversal, path finding, or divide-and-conquer strategies.Recursive calls could increase the risk of stack overflow in a system that continuously handles user input and real-time operations.Iterative logic provides better control and error handling in case of invalid inputs or failures. 117

Searching and path finding 117

Graph 118

Shortest Path Algorithms 122

Uninformed searching 122

- **Final Shortest Distances from A** 131

Encapsulation and Information hiding 153

Applying Encapsulation and data hiding to the developed system 154

```
    public void setName(String name) {
        this.name = name;
    }

    public String getMobile() {
        return mobile;
    }

    public void setMobile(String mobile) {
        this.mobile = mobile;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }
```

..... 155

```
public void setEmail(String email) {
    this.email = email;
}

public String getCity() {
    return city;
}

public void setCity(String city) {
    this.city = city;
}

public int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}

public String getMedicalHistory() {
    return medicalHistory;
}

public void setMedicalHistory(String medicalHistory) {
    this.medicalHistory = medicalHistory;
}
```

..... 155

Patient 156

name 156

age..... 156

medical History 156
getName() 156
setAge(int) 156
getMedicalHistory() 156
Doctor 156
name 156
availableSlots 156
getAvailableSlots() 156
updateSlot() 156
Advantages of Encapsulation and Information hiding 156
Encapsulation and Information Hiding in ADTs 158
Robustness through Encapsulation 158
Usability through Clear Interface 158

```
Doctor d = new Doctor("D001", new String[]{"10AM", "11AM"});
String[] slots = d.getAvailableSlots();
```

..... 159
Security through Information Hiding 159
Information Hiding 159
How ADTs relate to Object Oriented Programming 163
Data Structures 166
Testing of the doctor channeling system 172
Test Case 175
Why Lists 216
Why Queues 217
Why Stacks 217
Why Dijkstra's Algorithm 217
Why Bellman-Ford Algorithm 217
Test Case Execution 219
Asymptotic Analysis 223
Two ways in which the efficiency of an Algorithm can be measured 229
Efficiency of an Algorithm 236
Time Complexity and Space Complexity 236
2. Time Complexity 237
How Algorithm Runtime Scales with Input Size 237
3. Space Complexity 237
How Memory Usage Scales with Input Size 237
4. Complexity Comparison of Two Sorting Algorithms 238
Trade-offs for Implementing ADTs 240
Stack ADT Specification 247

Patient class 248

```
class Patient
{
   String name;
   String mobile;
   String email;
   String city;
   int age;
   String medicalhistory;
   Patient next;
```

```
public   Patient(String name, String mobile, String email, String city, int age, String history) {
    this.name = name;
    this.mobile = mobile;
    this.email = email;
    this.city = city;
    this.age = age;
    this.medicalhistory = medicalhistory;
 }

 public  String getName() {
    return name;
 }

 public String toString() {
    return "Name: " + name + ", Mobile: " + mobile + ", Email: " + email + ", City: " + city + ", Age: " + age + ",medicalHistory: " + medicalhistory;
 }
}
```

..... 248

This class represents a patient in the Doctor channeling system, with fields for name, mobile number, email, city, and age. It includes a constructor to set these values when a new patient is registered. A display method is also provided to show the patient's details clearly. This helps in managing and identifying each patient easily in the system. 248

Output Explanation 252

Complexity of Stack Operations 254

Benefits of Using Independent Data Structures 259

Gant Chart 260

References 260

Activity 01

Indroduction

In today's world, people want quick and easy access to healthcare services. With more hospitals and clinics competing to offer the best care, it has become important to provide fast and convenient ways for patients to make appointments. Many people now use smartphones and the internet, so they prefer booking appointments online rather than waiting in long lines or making phone calls. To meet this demand, the Doctor Channeling System was created. It is a simple and easy-to-use software that lets patient's book appointments with doctors through an online platform.

This system handles important information such as patient details, doctor schedules, and appointment times. It helps reduce human errors that can happen when booking is done manually. It also saves time for both the patients and the staff. With everything managed through the system, appointments can be scheduled faster and more accurately. This helps medical centers improve their daily operations and provide better service to their patients.

The system reduces waiting times at the clinic or hospital by allowing patients to select available time slots beforehand. It also helps doctors and staff plan their schedules better. Overall, the Doctor Channeling System improves the booking process, makes it more efficient, and ensures a better experience for everyone involved. By using this system, healthcare providers can offer a modern and professional service that meets the expectations of today's patients while also improving the quality and reliability of care.

Introduction to Data Structures

In computer science, a data structure is a special way of organizing and storing data so that it can be used efficiently. When we build programs, we often need to work with large amounts of data. A data structure helps us manage this data so we can access, update, or remove it easily whenever needed. Whether we are building a small mobile app or a complex software system, data structures help in making the program faster and more efficient.

Different types of data structures are used depending on the nature of the data and the operations we want to perform. For example, if we need to keep track of items in order, we might use an array or a list. If we want to manage tasks that follow a "first come, first serve" approach, a queue would be more suitable. Choosing the right data structure is very important because it directly affects the performance of the application. Good knowledge of data structures helps programmers write better code that runs faster and uses less memory.

A data structure is a specific type of format used to arrange, process, retrieve, and store data. There are several fundamental and sophisticated forms of data structures, all of which are made to organize data for a specific use. Users may easily get the data they require and use it in the proper ways thanks to data structures. Most crucially, data structures frame how information is organized so that both machines and people can understand it. (Loshin, 2025)

Objective of Data Structure

The main goal of using data structures is to organize and store data in a way that makes it easy to use and efficient to work with. In programming, data isn't always used in one piece it often needs to be searched, sorted, updated, or removed. A good data structure helps us do these tasks quickly and properly.

Different types of data structures, like arrays, lists, stacks, queues, trees, and graphs, are used depending on what kind of problem we're trying to solve. For example, if we need to manage things in order, a queue or stack might be best. If we need to look things up quickly, a hash table or tree can help.

Using the right data structure:

- Saves time by making programs faster.
- Saves space by using memory wisely.
- Makes the code easier to understand and maintain.
- Helps in solving complex problems more easily.

In short, the objective of data structures is to handle data efficiently and make our programs run better.

Real-World Modeling

Data structures allow representing many real-world problems in a natural way. As an example, a graph can represent a road network or social media connection, a queue can represent queues at a checkout counter, and a tree can represent hierarchical structures such as the organization chart or file system. The selection of a proper structure contributes to the optimal imitation of real-life situations and the resolution of problems in a programmatic way.

Enhance Performance

The issue of performance of an application critically depends on selecting the right structure to carry the data. Properly selected structure minimizes the number of time and space required to make programs run faster and much efficient. As an example, frequent searches can reduce processing time associated with linear search to constant by using a hash map to replace a list.

Scalability

Increasing the data will create more load on the data structure in such a way that when there is increment in data, data structure should not exhibit severe decrement in the performance. Scalability provides the performance of the application based on the perspective of millions of records without any stall in responsiveness and efficiency. Trees, hash tables, and the balanced trees such as AVL trees are scalable.

Data Structure Classification

To a large extent, data structures may be categorized according to the way data is arranged and retrieved. It can be seen to assist in getting an idea of the nature of data and to make the correct selection of the structure best suited to certain tasks. Overall, data structures could be categorised as primitive and non-primitive. The primitive structures comprise simple data types such as integers, floats, and characters whereas the non-primitive structures are more complicated and include the linear (arrays, linked lists) and non-linear ones (trees, graphs). This categorization serves as a basis to engineer effective algorithm and construction of stable software systems.

Data structures are mainly classified into two broad types: Linear and Non-Linear. This classification is based on how data elements are organized and accessed.

Linear Data Structures

In linear data structures, the data elements are arranged in a sequence, one after the other. Each element is connected to its previous and next element. This type of structure is simple to use and easy to understand, which is why it is commonly used in many programs.

Examples of linear data structures include:

- **Arrays**: A collection of elements stored in a continuous memory location. All elements are of the same data type and can be accessed using an index.
- **Linked Lists**: Unlike arrays, linked lists use nodes, where each node contains data and a reference (or pointer) to the next node. This allows easy insertion and deletion.
- **Stacks**: A stack follows the Last In First Out (LIFO) principle. The last item added is the first one to be removed.
- **Queues**: A queue works on the First In First Out (FIFO) rule. The first item added is the first to be removed.

Linear structures are ideal when data needs to be processed in a specific order.

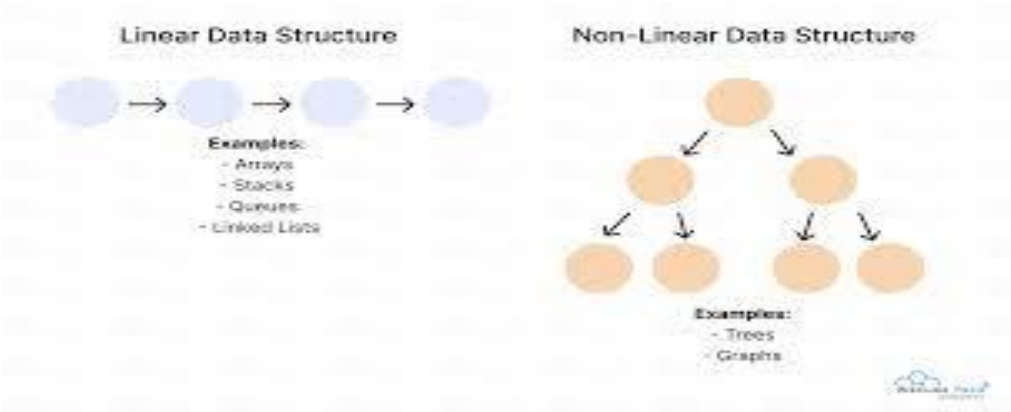
Non-Linear Data Structures

In non-linear data structures, elements are not arranged in a sequence. Instead, each element can be connected to multiple elements, forming a hierarchical or interconnected structure.

Examples include:

- **Trees**: Data is arranged in a tree-like format. It starts with a root node and branches out into children. Trees are used in databases and file systems.
- **Graphs**: A graph contains nodes (also called vertices) connected by edges. Graphs are useful in representing networks like social connections, maps, or web links.

Non-linear structures are useful when the relationship between data items is complex.



Types of data structures

There are two main types of data structures/types.

1. **Concrete data structures**

The structure and its manipulations are built into the programming language.

Examples: Array, Linked List

- ☒ Array is a static structure meaning its size is fixed in the beginning of the program and cannot change at run time.
- ☒ Linked list is a dynamic structure whose size can change (grow or shrink) at run time.

2. **Abstract data structures/types**

Created by the programmer to suit some real-world user requirement. An abstract data type combines data and operations together.

Examples: Stack , Queue , Tree , Graph

An Abstract data structure/type is implemented by employing one of the concrete data structures. For example, if we want a stack with fixed number of elements, we could implement it on an array. Alternatively, if we need a dynamic stack we could implement it on a linked list. Because of this reason, ADTs are called INDEPENDENT DATA STRUCTURES.

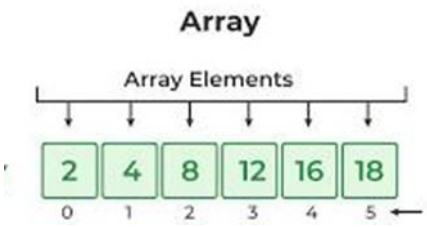
The choice of data structure for a specific scenario is based on the type of operations and algorithms that will be utilized. The available data structure types encompass the following.

Explanation of Each Data Structure

- **Array**

  An array is a collection of elements stored in a single, continuous block of memory. Each element is identified by an index, starting from 0. Arrays are useful when we know the number of elements in advance. They support operations like insertion, deletion, updating, and searching.

  A type of data structure known as an array is used to store a group of linked elements in a row across memory. Using an index to access or retrieve each element of an array is simple since each item is stored according to its data type. An index, a numeric integer, represents the location of a certain element within an array. Fixed arrays have a fixed number of elements as opposed to flexible arrays, which can have a variable number of components that can be changed at runtime. (Loshin, 2025)



Eg;

```
public class Main {
  public static void main(String[] args) {
    int[] numbers = {10, 20, 30, 40};
    System.out.println(numbers[2]);
  }
}
```

```
30
```

In this example, the third element of the array (index 2) is accessed and printed. Arrays are fast for accessing elements but slow for inserting or deleting because elements need to be shifted.

Functions of Array data Structure

1. Random Indexed Access (Random Access)

Through arrays we are able to access any element directly by its index. This makes the execution of data retreival highly fast (O(1) time), a feature which is of essence in performance sensitive applications.

2. Traversal

Arrays are simple to iterate and each element can be traversed through and processed one by one. It is mandatory in order to run operations such as display, search or running transformations on a data.

3. Searching

Arrays are useful with validation, and look-ups because they can be searched on array elements with a technique such as a linear search (with unsorted arrays) and a binary search (with sorted arrays).

4. Insertion

Insertion of new elements (particularly in the last position) is a basic operation but not quite so effective in linked structures; how to deal with space and shifting when it comes to arrays is central to array manipulation knowledge.

5. Updating Elements

It is easy to change the value of an existing element with arrays. This is essential in most applications that require changing data i.e. updating scores, inventory or sensor information.

Arrays support operators for accessing, inserting, and searching elements. Accessing elements is done using the index, allowing for quick retrieval and modification. Insertion involves assigning a value to a specific index, while deletion can be done by assigning null values or removing elements at a given index. Arrays also support searching for specific values by iterating over elements. However, certain operations like insertion and deletion can be inefficient due to the need for shifting elements. (geeksforgeeks, 2025)

- **Linked List**

A linked list is a chain of nodes where each node contains data and a pointer to the next node. Unlike arrays, linked lists do not need a fixed size. They are good for inserting or deleting elements without shifting others.

Functions of Linked list data structure

1. Dynamic Memory Allocation

Linked lists use dynamic memory, meaning nodes are created as needed at runtime. This allows efficient use of memory without requiring a predefined size, unlike arrays.

2. Efficient Insertion and Deletion

Inserting or deleting elements (especially at the beginning or middle) is more efficient in linked lists compared to arrays, as it doesn't require shifting elements—only pointer adjustments.

3. Traversal

Linked lists can be traversed node by node from the head to the end. This function is essential for processing or displaying data stored in the list.

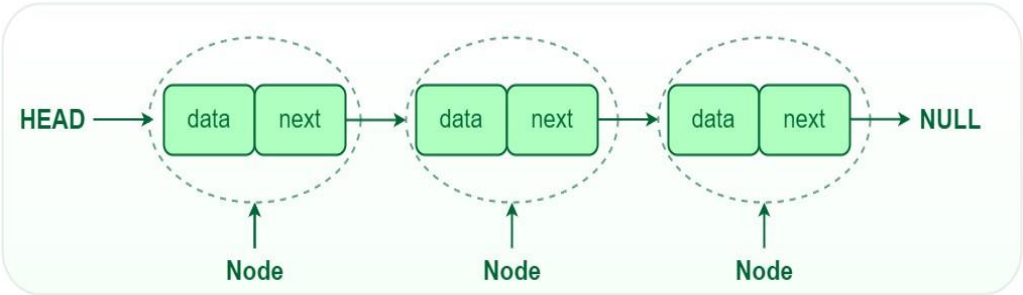
4. Searching

Linked lists support searching for a specific element by traversing through the nodes. Although slower than arrays (O(n)), it is a fundamental operation used in many applications.

5. Flexible Data Structure

Linked lists can easily represent other complex data structures like stacks, queues, graphs, and polynomial expressions, due to their node-based and pointer-driven design.

An ordered collection of things is stored in a linked list, a linear data structure. The linked list's nodes each have a data item and a pointer to the following item. The ability to insert and remove components more effectively and use memory more creatively is made possible by linked lists' lack of a need that all elements be kept in nearby memory regions. When compared to doubly linked lists, which include references to both the previous and next nodes in the list and enable efficient traversals both ways, singly linked lists only have references to the next node. (Loshin, 2025)



o Deletion – Delete an element at the begging of the list.

o Insertion – Add an element at the beginning of the list

o Display – Display the complete list.

o Search – Search an element by given key

o Delete – Delete an element using the given key.

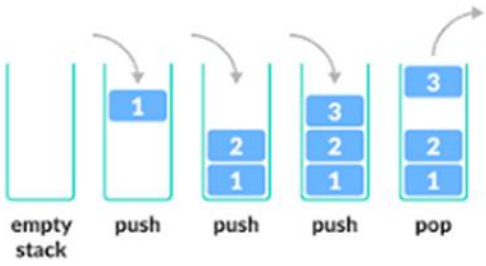
Eg;

```
public class PatientNode {//node.java
    Patient data;//data part
    PatientNode next;//pointer

    public PatientNode(Patient data) {//constructor method
        this.data = data;
        this.next = null;
    }

    public void displayNode() {//normal data
        System.out.println(data.toString());
    }
}
```

- **Stack**

A stack works on the Last In, First Out (LIFO) principle. The last item added is the first one removed. It supports operations like push (add), pop (remove), and peek (view the top element).

A stack is a sort of data structure that is used to store a collection of things in the sequential order that operations are done. FIFO (First In Last Out) or LIFO (Last In First Out) ordering can be used, depending on the implementation. Stacks employ push and pop operations to add and remove items, respectively. The stack is an efficient data structure for arranging elements in a certain sequence. (Loshin, 2025)



The operators used in the stack are given below.

- Pop – Used to remove an element from the stack.
- Push – To insert an element to the stack.
- is Empty – Returns true is the stack is empty else false
- size – Return the size of the stack
- Top – return the top element of the stack.

Eg;

```
public class Main {
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();
        stack.push(5);
        stack.push(10);
        System.out.println(stack.pop());
    }
}
```

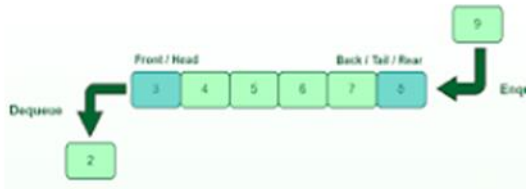
```
10
```

In this code, the number 10 is added last and is removed first. Stacks are useful for undo features, backtracking, and evaluating expressions

- **Queue**

  A queue follows the First In, First Out (FIFO) rule. The first item added is the first one removed. It supports enqueue (add) and dequeue (remove) operations.

  A collection of items is stored in a queue, a type of linear data structure. According to the "First In, First Out" (FIFO) order in which it operates, the first item put to the queue is also the first one withdrawn. Items are added to the back of the queue and subtracted from the front using the enqueue and dequeue methods, respectively. The sequence in which things are added to a queue determines the order in which those items are managed by the queue. (Loshin, 2025) Queue can handle multiple data, and it is fast and flexible. Moreover, in queue the user can access both ends. These are some advantages of using queue. (geeksforgeeks, 2025)



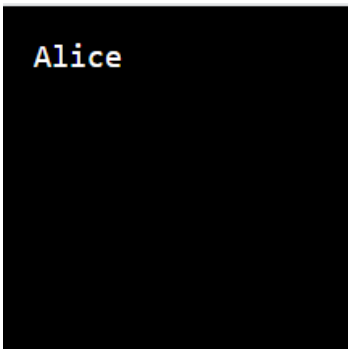
Queue Data Structure

The basic operation in queue is given below.

- Enqueue –Used to add elements.
- Dequeue – Used to remove elements.
- Peek – used to access the element in the front node
- Rear – used to access the element in the rear node.
- isFull – Verify whether the queue is full.
- isNull - Verify whether the queue is null.

Eg;

```
public class Main {
  public static void main(String[] args) {
    Queue<String> queue = new LinkedList<>();
    queue.add("Alice");
    queue.add("Bob");
    System.out.println(queue.remove());
  }
}
```

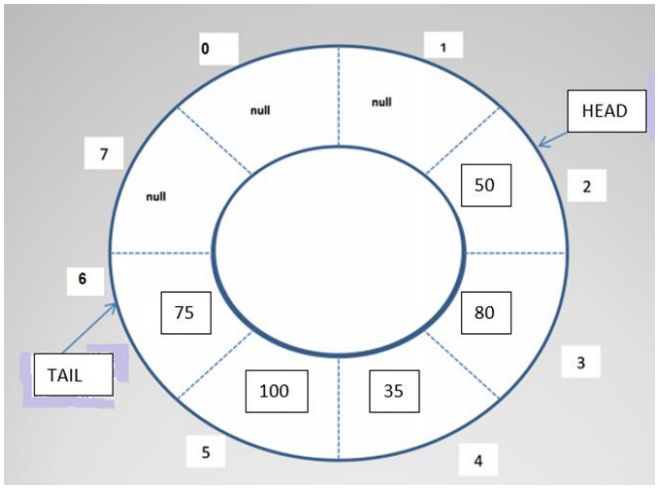


Here, "Alice" is added first and removed first. Queues are useful in printer queues, customer service systems, and scheduling tasks.

A queue is a linear data structure that stores its elements sequentially. It accesses elements using the FIFO (First In, First Out) technique. Queues are commonly used to manage threads in multithreading and for building priority queuing systems. This article will cover the various types of queue data structures, basic operations on them, implementation, and queue applications. (Karan, 2025)

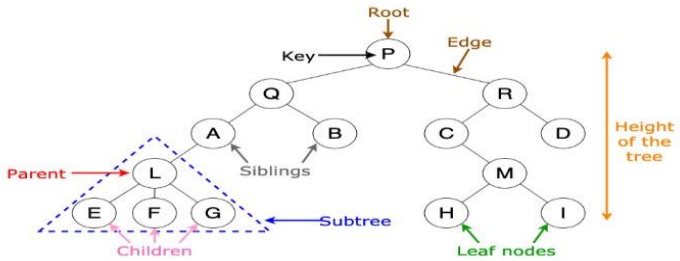
Example application: Printer Queue

In our system, a Queue can be used to manage appointments such as patient bookings with doctors. This approach follows a first come first served base rule, ensuring fairness by handling requests in the order they are received. The Queue can be implemented using an array, and to make the best use of space, it can function as a circular structure. This means that when the end of the array is reached, the system loops back to the beginning, allowing continuous use of available space without wasting memory.



- **Tree**

  A tree is a hierarchical data structure used to abstractly hold a group of objects. Each node in the tree is linked to its child nodes or sub nodes and has a key value associated with it. The root node, which is the ancestor of all other nodes, is located at the top of the tree. Trees are excellent for displaying the hierarchical relationships between objects and facilitate quick sorting and searching. Family trees, organizational charts, and file directories are a few examples of tree-like structures. (Loshin, 2025)



The basic operators used in linked list is given below.

o Create – used to create a tree

o Insert – Used to insert data into tree

o Search – used to search specific data in the tree.

o Preorder Traversal – Perform travelling in a tree in a preorder manner.

o In order Traversal – Perform travelling in a tree in an in-order manner.

o Post order Traversal – Perform traveling in a tree in a post order manner.

Binary Search Tree

A Binary Search Tree is a Binary Tree following the additional properties:

The left part of the root node contains keys less than the root node key.

The right part of the root node contains keys greater than the root node key.

There is no duplicate key present in the binary tree.

A Binary tree having the following properties is known as Binary search tree (BST).

Functions of binary search tree data structure:

1. Insertion

- Inserts a new element in the BST without violating the BST property:
- Each node in the left subtree is smaller than the parent one.
- The right subtree is made of nodes whose value exceeds the value of the parent node.
- This enables quick searches in future and placement of data in a systematic manner.

2. Searching

- Determines whether a specific value is contained into the BST.
- The search operation is far more efficient (O (log n) in a balanced BST) compared to a normal list; by comparing the target value with node values and going left or right as appropriate, respectively.

3. Deletion

- Deletes a node of the BST without shaking the structure and ordering rules.
- Three cases can be managed:
- Removal of a leaf node
- Removal of a node having one child

- Removing a tree node with two children (and added to its inorder successor or predecessor)

4. Traversal

- Various types of traversals to use BSTs to traverse nodes in particular orders:
- Inorder (Left Root Right) - element returned in sorted order
- Preorder and Postorder find application in tree copying, expression trees, etc.

Data Structure for Doctor Appointment Scheduling System

For effectively handling doctor appointment bookings, a Queue data structure is one of the most suitable options compared to alternatives like Arrays, Linked Lists, Stacks, or Trees. The main reason for using a queue in this context is its natural alignment with the first-come, first-served (FCFS) model. This scheduling logic is essential in a doctor channelling system where patients must be served in the exact order in which they made their bookings. It promotes fairness and keeps the whole system simple and consistent, which helps both the hospital staff and patient satisfaction.

One of the important benefits of a queue, especially when implemented using a linked list, is its ability to grow and shrink based on current demand. Unlike Arrays, which require a fixed size or reallocation when full, queues adjust dynamically as more patients book or cancel appointments. This is highly useful in healthcare settings, where the volume of appointments may change daily depending on doctor availability or patient demand. Using a queue ensures the system won't waste memory and can handle various load sizes efficiently.

The operational simplicity of a queue is another factor that makes it an ideal data structure for appointment systems. Core operations like enqueue (adding a new appointment) and dequeue (removing a patient when their appointment is over or cancelled) are performed in constant time. This keeps the system fast and easy to maintain. Compared to more complex data structures like trees (which are good for searching and hierarchical data) or stacks (which follow a last-in, first-out order), queues provide exactly the kind of linear processing needed for doctor scheduling.

Access pattern is another critical consideration. In doctor channelling, the system should always give priority to the earliest booking, not the most recent one. Queues naturally follow this FIFO rule. If we used stacks instead, the most recent patient would be seen first, which breaks fairness. Arrays or basic linked lists might allow random access, but enforcing FCFS becomes harder and less efficient. Trees, while powerful for other use cases, are overkill and unnecessarily complex for a system that simply needs to manage appointments in the order they are made.

The Queue data structure is a great fit for doctor appointment scheduling due to its compatibility with FCFS logic, dynamic size management, simple operations, and correct data access order. It helps keep the system fair, efficient, and scalable without adding unnecessary complexity. These advantages clearly show why queues are the best structure for this type of healthcare application.

Design Specification for Data Structures

In the Doctor Channelling System, patient registration can be efficiently managed using a HashMap, where each patient is identified by a unique key such as a mobile number or patient ID. The associated value stores patient details like name, age, email, city, and medical history, allowing quick access and updates. For doctor registration, a similar approach can be used. Each doctor's information including their ID, name, specialization, available time slots, and consultation fee can be stored in an ArrayList, depending on whether fast lookup or simple list storage is required. When it comes to managing appointments, a Queue data structure is most appropriate. Design Specification for Data Structures

To manage a doctor channelling system efficiently, we require a set of appropriate data structures to store and manipulate patient, doctor, and appointment data. The chosen structures should support the operations specified in the scenario and ensure data retrieval and updates are efficient.

Since appointments must follow a first-come, first-served basis, the queue ensures that patients are handled in the exact order they made their booking, maintaining fairness and system consistency. This combination of data structures supports an efficient, organized, and scalable doctor appointment system.

In any healthcare management system, data management is the key aspect of operational efficiency. For this doctor channelling system, we propose a modular design that captures all necessary components of the system including patients, doctors, and appointments.

Patients will be registered with information including:

- Patient Name
- Mobile Number
- Email ID
- City
- Age
- Medical History

Doctors will be registered with the following data:

- Doctor ID
- Name
- Specialization
- Available Time Slots
- Consultation Fee

Appointment Management

Appointments need a way to track order and availability. Here, we use:

- Queue (LinkedList-based) for storing patients per doctor slot.
- ArrayList for completed appointments.

Complexities of Data Structures (Array, Linked List, Stack, Queue)

Understanding time and space complexity is important when choosing the right data structure for a program. Time complexity tells us how fast an operation takes, and space complexity tells us how much memory it uses. These measurements are often written in Big O notation, such as O(1), O(n), O(log n), etc.

Each data structure has different performance depending on what we're trying to do: access, search, insert, or delete data. The performance can vary in the best case, worst case, and average case. Let's look at the main data structures one by one.

Time Complexity Table

Operation	Data Structure	Best Case	Average Case	Worst Case
Access	Array	O(1)	O(1)	O(1)
	Linked List	O(1)	O(n)	O(n)
	Stack	O(1)	O(1)	O(1)
	Queue	O(1)	O(1)	O(1)
Search	Array	O(1)	O(n)	O(n)
	Linked List	O(1)	O(n)	O(n)
	Stack	O(1)	O(n)	O(n)
	Queue	O(1)	O(n)	O(n)
Insert	Array	O(1)	O(n)	O(n)
	Linked List	O(1)	O(1)	O(1)
	Stack	O(1)	O(1)	O(1)
	Queue	O(1)	O(1)	O(1)
Delete	Array	O(1)	O(n)	O(n)
	Linked List	O(1)	O(1)	O(1)
	Stack	O(1)	O(1)	O(1)
	Queue	O(1)	O(1)	O(1)

Space Complexity

Data Structure	Space Complexity
Array	O(n)
Linked List	O(n)
Stack	O(n)
Queue	O(n)

- **Arrays** are fast for accessing data because of indexing. But inserting or deleting in the middle takes time since elements must shift.
- **Linked lists** are slower to access since you must go node by node, but they are great for inserting or deleting, especially at the beginning.
- **Stacks and Queues** are simple structures, both efficient for inserting and removing at specific ends (top for stack, front/rear for queue), which gives them constant time performance.

Edge Cases and Error Handling

Edge cases and error handling are two important parts of software development that help make programs more reliable and user-friendly.

Edge Cases

An edge case is a situation that happens rarely or under unusual conditions, often occurring at the outer limits of expected input or behavior. In the context of a doctor channeling system, edge cases might include a patient attempting to book an appointment when no doctors are available, a user entering incorrect or incomplete data such as an empty name or invalid ID number, or trying to cancel an appointment that was never booked in the first place. Although these scenarios may not happen frequently, they can still cause the system to behave unpredictably or crash if not properly managed. This is why it is important to identify and plan for edge cases during the early stages of design and development, so the software can respond gracefully and continue functioning smoothly even when unexpected inputs occur.

Error Handling

Error handling is how a system reacts when something goes wrong. It helps the program deal with unexpected problems in a smooth way, without crashing or showing confusing messages to users.

In software, errors can happen for many reasons wrong input, unavailable services, or bugs. Good error handling helps identify the issue, give a clear message to the user, and allow the system to continue running safely.

For example, if a user tries to book an appointment with a doctor who has no available time slots, the system should tell them politely that the doctor is unavailable not crash or freeze.

Important of Edge cases and error handling

Edge cases and error handling are important for several reasons:

- Reliability: The system keeps working even if something unexpected happens.
- Better User Experience: Users receive clear messages instead of confusing errors.
- Security: Handling errors properly can stop hackers from taking advantage of weak points.
- Easier Maintenance: If errors are reported clearly, it becomes easier for developers to fix them.

What is a Memory Stack?

A memory stack is a special part of a computer's memory that stores temporary data. It works like a real-life stack of plates whatever goes in last comes out first. This is known as Last In, First Out (LIFO).

When a program runs, it often needs to keep track of information like function calls, local variables, and return addresses. The stack helps with this. Every time a function is called, the computer pushes (adds) a block of data onto the stack. When that function finishes, the data is popped (removed) off the stack.

The stack grows and shrinks automatically as functions are called and return. Because of this, it helps manage the flow of the program. If the stack becomes too full, it can cause an error called a stack overflow, which happens when too much memory is used, usually by deep or infinite recursion.

In short, the memory stack is a useful and important part of how programs run. It handles temporary data efficiently and plays a big role in making sure function calls work properly in most programming languages.

Functions of a Memory Stack

The memory stack plays a key role in how a program runs by handling temporary data and keeping things organized during function calls. It works using the Last In, First Out (LIFO) method, meaning the last item added is the first to be removed.

One important function of the memory stack is to store function call information. When a function is called in a program, the system stores the return address (where to go back after the function finishes), function parameters, and local variables on the stack. This collection of data is called a stack frame.

Another function is managing nested function calls. If one function calls another, and that one calls another, the stack keeps track of each function separately. Once the inner functions are finished, the stack pops them out one by one in reverse order.

The memory stack also helps with recursive functions, where a function calls itself. Each call gets its own stack frame, allowing the program to remember each level of the call.

Functions of a Stack in Memory Management (System-Level)

Function	Description
Function Call Management	When a function is called, its execution context (return address, local variables, parameters) is pushed onto the stack.
Return Address Storage	The address to return to after function execution is stored on the stack.
Local Variable Storage	Stores variables that are only needed within the function.

Function Nesting / Recursion	Supports recursion by maintaining multiple function call contexts.
Memory Isolation	Each function gets its own "stack frame" to prevent interference between functions.

Usage of Memory Stack

The memory stack is used in computer programs to manage temporary data during execution. It mainly helps with function calls, local variable storage, and program control flow. One of its key uses is to keep track of what happens when a function is called.

When a function runs, the stack stores its return address, parameters, and local variables. This information is stored in a block called a stack frame. After the function completes, the stack removes the frame and goes back to where the function was called from. This is how the program knows what to do next. Data belonging to other functions do not cross because each function has got its own stack frame.

The memory stack is also used in recursive functions. When a function calls itself repeatedly, each new call is pushed onto the stack. The computer uses the stack to remember each step and to return correctly once the recursion ends.

Additionally, the memory stack is useful for backtracking and undo features in applications, as it can store the history of operations in order. In certain languages, such as Java, a virtual machine of the stack-based type (namely the JVM) is used and the stack is paramount in the method of executing a bytecode.

Advantages and Disadvantages of Memory Stack

Advantages

1. Fast Access

   The memory stack allows very quick access to data. Since it follows the Last In, First Out (LIFO) method, the most recent data is always on top and can be accessed or removed instantly.

2. Automatic Memory Management

   The stack handles memory allocation and deallocation automatically. When a function is called, its local data is pushed onto the stack. Once the function finishes, that data is automatically removed. This helps in efficient memory use.

3. Supports Function Calls and Recursion

   One of the main uses of the stack is to handle function calls. It keeps track of where to return after a function finishes. It also supports recursive functions by saving each call separately.

4. Simple to Use and Implement

   The structure and operations (push and pop) are simple, which makes the stack easy to implement in most programming languages.

Disadvantages

1. Limited Size

   The stack has a fixed size depending on the system's memory. If too many function calls are made (especially in recursion), it may lead to a stack overflow error.

2. Data Access is Limited

   You can only access the top item of the stack. You cannot search or retrieve data from the middle or bottom without removing other elements.

3. Not Suitable for All Problems

   Since it only works in LIFO order, the stack is not suitable for tasks where random access or different ordering is needed, such as searching or sorting complex datasets.

Register Stack and Memory Stack

In computer systems, when functions are called during program execution, temporary data must be stored somewhere. Two main types of stacks help with this: Register Stack and Memory Stack.

Register Stack

A register stack is a small, high-speed storage area inside the CPU. It uses processor registers to store function-related data such as return addresses, parameters, and local variables. Since registers are faster than memory, accessing data from a register stack is very quick.

However, register stacks have a major limitation: they are small in size. Modern CPUs have only a limited number of registers, so register stacks can't hold large or deeply nested function calls. When the stack becomes full, it is often backed up into the memory stack.

Memory Stack

A memory stack, on the other hand, is located in the system's main memory (RAM). It is used when the register stack is full or when more space is needed to store data. Memory stacks are larger and more flexible but slower than register stacks because accessing RAM takes more time than accessing CPU registers.

The memory stack is widely used for managing function calls, local variables, return addresses, and recursive operations. It grows and shrinks as functions are called and returned.

Implementation of Function Calls Related to a Developed to the doctor channeling system

There are more than one function call in the case of a real-world doctor channelling system as the patient is involved. These function calls are based on the memory stack to carry functions in and out, specifically when functioning on the basis of calling other functions, or parameters and results.

Developed a doctor channeling system in Java. When a patient books an appointment, the system may call several functions one after the other

1. registerPatient()
2. checkDoctorAvailability()
3. createAppointment()
4. sendConfirmationMessage()

Each time one of these functions is called, the system uses the memory stack to store the function's details:

- The location to return after the function ends (return address)
- The values passed to the function (parameters)
- Any temporary values used inside the function (local variables)

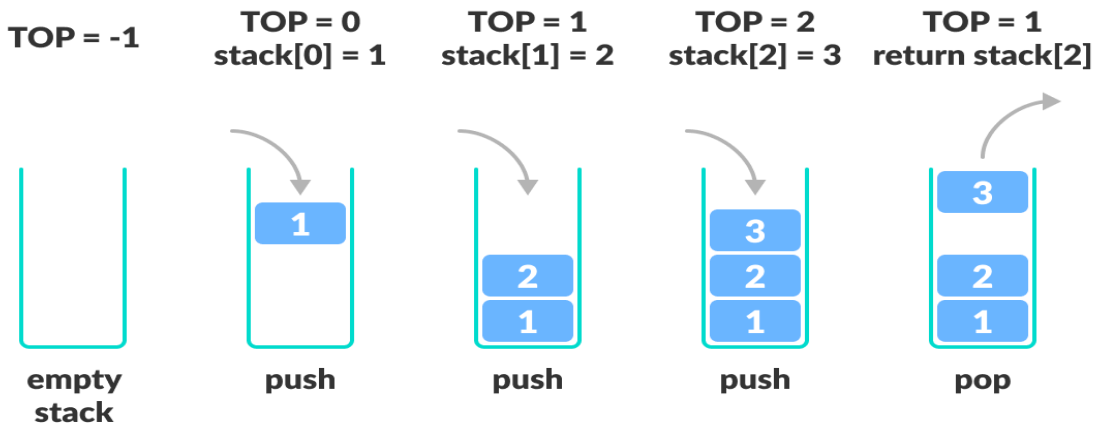
Example Flow Using Memory Stack

1. Main () function starts the program.
2. RegisterPatient () is called a stack frame is pushed onto the memory stack.
3. Inside it, checkDoctorAvailability () is called a new stack frame is pushed.
4. Then, createAppointment () is called another stack frame is pushed.
5. Finally, sendConfirmationMessage () is called another frame is added.

Once sendConfirmationMessage () finishes, its stack frame is popped off. Then control returns to createAppointment (), which also finishes, and its frame is removed. This continues until we return to main () and the stack is empty again.

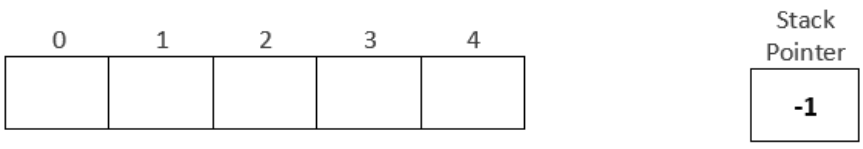
Stack

A stack is a linear data structure that uses the Last-In-First-Out (LIFO) concept. Stacks have one end, but Queues have two (front and back). It simply has one pointer, top, which points to the stack's topmost element. When an element is added to the stack, it is placed at the top of the stack, and it may only be removed from the stack. In other terms, a stack is a container that may be inserted and deleted from one end, known as the top of the stack. (javatpoint, 2020) . Example of using a stack is “Undo” feature found various software packages.



We could use an array to represent a stack. A stack pointer variable can be used to get to the top of the stack. Assuming a 5-element stack, the stack's initial state will seem as follows

Note : To indicate there are NO values, we initialize stack pointer to -1.



Stack Operations

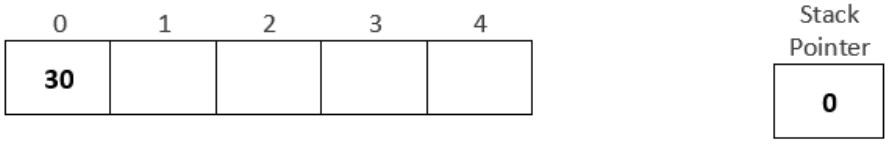
- **Init() Operation**

Creates a new stack and initializes stack pointer to -1.

- **Push() Operation**

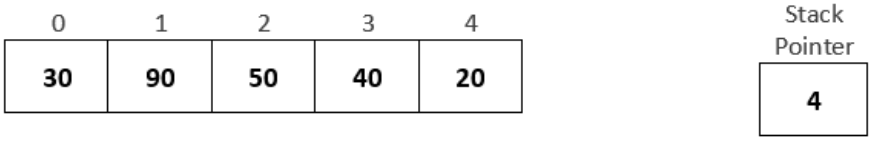
This is used to add a data value to the top of the stack. Prior to inserting the new value, we must increment the stack pointer by 1.

Now we are going to execute **push(30)** on the stack we initialized above.



Remember we **increment stack pointer by 1** and then **insert the data item (50)** by using stack pointer as the array index position.

Now we execute **Push(90) , Push(50), Push(40), Push(20)**

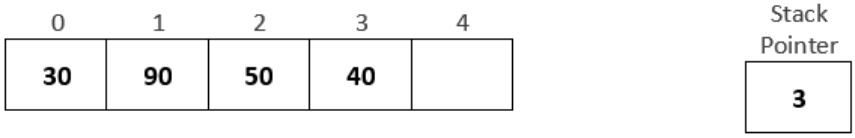


If we try to push() again, the system should display **"Stack full"** message.

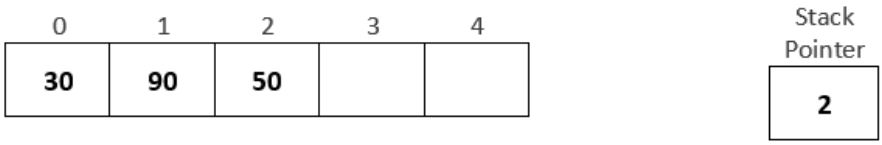
- **Pop() Operation**

This is used to remove a data value from the top of the stack. After removing a value, we must decrement the stack pointer by 1.

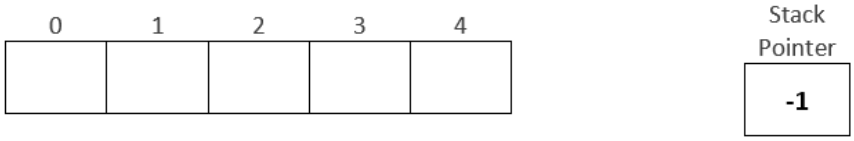
Now we are going to execute **Pop()** on the stack.



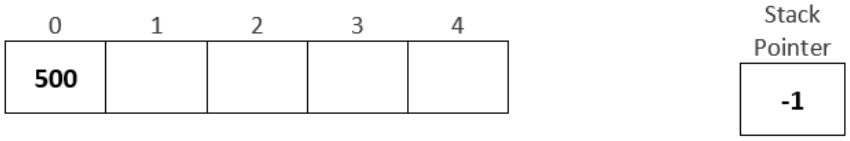
Execute **Pop()** again on the stack.



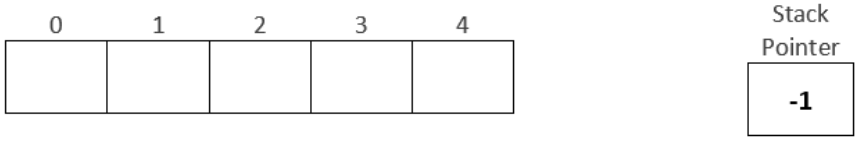
Execute **Pop()** Three times on the stack.



Now we will execute **Push(500)**



Again execute **Pop()** one times on the stack.



if we call **Pop()** again, the system should **display "Stack Empty"** message.

Implementation of a Stack in Java

- **Stack.java**

```java
public class Stack {
    private int stptr; // declare stack pointer
    private int ST[]; // declare array to store data

    public Stack() {
        ST = new int[5]; // create a stack of 5 elements
    }

    public void init() {
        stptr = -1; // initialize stack pointer to -1, indicating an empty stack
    }

    public void push(int data) {
        if (stptr == 4) // check if stack is full
            System.out.println("Stack full");
        else {
            stptr++; // increment stack pointer
            ST[stptr] = data; // push data onto the stack
        }
    } // end push

    public int pop() {
        int data;
        if (stptr == -1) { // check if stack is empty
            System.out.println("Stack empty");
            return 0;
        } else {
            data = ST[stptr]; // retrieve data from the top of the stack
            stptr--; // decrement stack pointer
            return data;
        }
    } // end pop
} // end stack
```

- **StackDemo.java**

```java
public class StackDemo {
    public static void main(String arg[]) {
        Stack s1 = new Stack();
        s1.init();
        s1.push(40);
        s1.push(50);
        s1.push(60);
        int x = s1.pop();
        int y = s1.pop();
        System.out.println(x);
        System.out.println(y);
    }
} // end stack demo
```

StackDemo.javaExplanation

- **Class Definition**

public class StackDemo { ... } defines a public class named StackDemo.

- **Main Method**

public static void main(String arg[]) { ... } is the entry point of the Java application.

- **Stack Operations**

Stack s1 = new Stack(); creates an instance of the Stack class.

s1.init(); initializes the stack pointer of s1 to -1.

s1.push(40); pushes the integer 40 onto the stack.

s1.push(50); pushes the integer 50 onto the stack.

s1.push(60); pushes the integer 60 onto the stack.

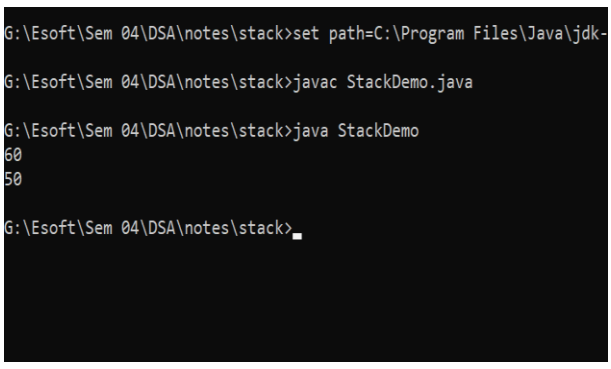
int x = s1.pop(); pops the top element from the stack and assigns it to x.

int y = s1.pop(); pops the top element from the stack and assigns it to y.

System.out.println(x); prints the value of x.

System.out.println(y); prints the value of y.

- **Output**



How Stack Operations Help Manage Function Calls

When a program runs and functions are called, the memory stack plays a big role in handling those calls correctly. Stack operations like push and pop help manage the flow of the program, especially when there are multiple or nested function calls.

1. Maintaining Function Call Order

Each time a function is called, a new stack frame is created and pushed onto the top of the stack. This frame contains all the details needed for that function, such as variables and return information. If that function calls another function, a new frame is pushed on top. When a function finishes, the stack frame is popped off. This ensures that the most recent function returns first, and the program continues in the right order. This process follows the Last In, First Out (LIFO) rule.

2. Storing Local Variables and Parameters

Every function has its own set of local variables and parameters. These are stored inside its stack frame. This means each function works with its own private data, avoiding conflicts with other functions. Once the function is done, its data is removed from memory automatically with the pop operation.

3. Tracking Return Addresses

The stack also stores the return address, which is the point in the code where the function was called from. When the function ends, the program uses this address to know exactly where to continue. Without this, the program would get lost after a function finishes.

Understanding Stack Frames, Stack Overflow, and Recursive Function Calls

When a function is called in a program, the system creates a special space in memory called a stack frame. This frame is pushed onto the memory stack and holds all the important data related to that function, such as its parameters, local variables, and the return address which tells the system where to go after the function finishes. Each function call creates a new frame on top of the stack.

In the case of recursive functions, a function calls itself repeatedly. Each call adds a new stack frame to the stack. These frames store the current state of each call so that when the function

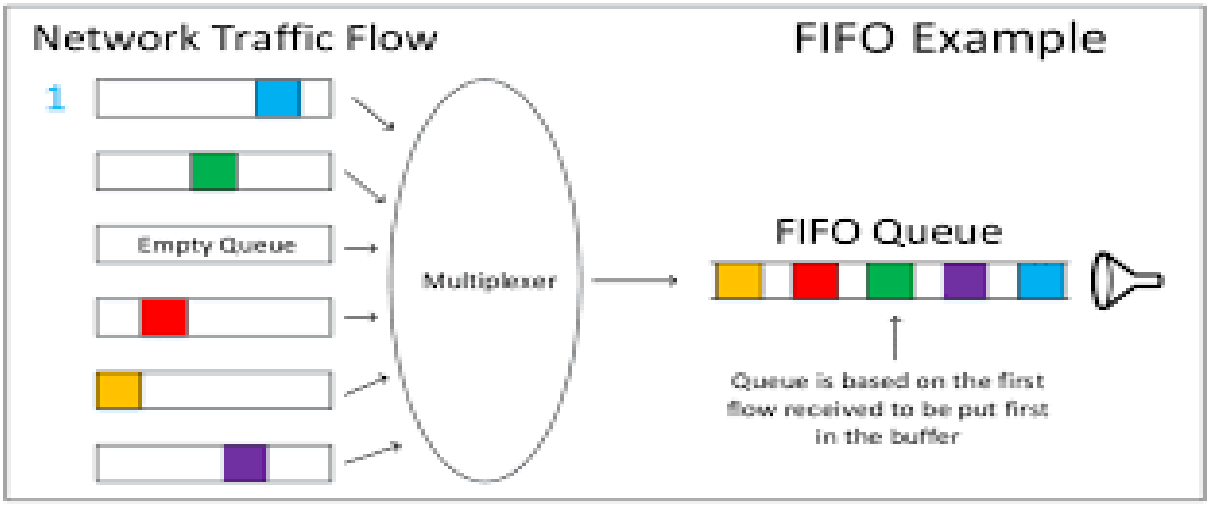
starts to return, it can continue exactly where it left off. Recursion is useful for solving problems like calculating factorials or navigating tree structures.

However, if a recursive function doesn't have a proper base case (a stopping condition), it can keep calling itself endlessly. This causes the stack to grow too large, eventually using all available memory. When the system can no longer create new stack frames, it throws a stack overflow error. This means the stack has run out of space, and the program stops working.

To prevent this, recursive functions must always have a clear stopping point. Understanding how stack frames work helps developers write safer code and avoid errors like stack overflow during deep or infinite recursion.

FIFO Queue

A FIFO queue stands for First In, First Out queue. It is a type of data structure where the first item added is the first one to be removed. This means the order in which elements enter the queue is the same order in which they leave. You can think of it like a line at a doctor's clinic patients who arrive first are seen by the doctor before those who come later. In programming, a FIFO queue is useful for managing tasks that need to be processed in the same order they arrive. For example, in a doctor channeling system, patient appointments can be handled using a FIFO queue so that each patient is attended to based on the order they booked their appointment. This helps ensure fairness and order in the process.



Insertions and deletions are done in the FIFO method

In a FIFO queue, insertions are done at the rear (or back) of the queue, and deletions are done from the front. When a new item needs to be added, it is placed at the end of the queue. This is like a patient joining the back of the line at a clinic. When it's time to remove an item, the one at the front of the queue is taken out first, just like the patient who arrived first is seen by the doctor before anyone else. This process keeps the order fair and organized. In programming, this behavior helps manage tasks or requests in the order they come, ensuring no one is skipped or served out of turn.

Insertions (Enqueue Operation)

- ✓ In a FIFO queue, the insertions go to the tail of the queue. Under Doctor Channelling System, this is done when a patient takes an appointment of a certain doctor.

Process:

- ✓ Once the patient, doctor information and the availability of the particular time slot has been validated then a new Appointment object will be created.

- ✓ The job is inserted in the queue by the Queue.add() (or offer()) method.

- ✓ The time slot picked is then deleted under the available time slot list of the doctor so as to eliminate the issue of rebooking.

Dequeue and Cancellation (Deletions)

A) Standard Deletion (Dequeue from Front)

- ✓ In a FIFO queue the usual deletion is of the element at the head (front) of a queue. This is applied, when one is attending to the next patient in the queue.

Queue Operations

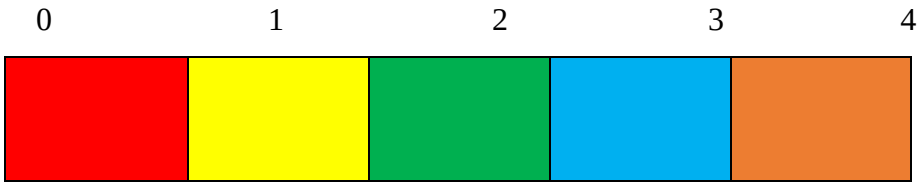
- **Initialize()** Make a new queue
- **Enqueue()** Adds an item to the tail end of the queue.

Dequeue() Removes the current element at the head of the queue

Array implementation of a Queue

Init() - Create a new Queue

Integer Queue with 5 elements is created. Head, Tail and Count are initialized.



Head = 0

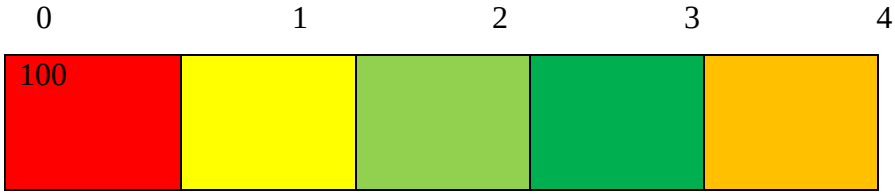
Tail = 4

Count = 0

Note: Since the Queue is maintained as a circular ring it is convenient to initialize Head to 0 and Tail to subscript of the last element (4).

Enque() - Add a new item to the tail of the queue

Enque(100)



Head = 0

Tail = 0

Count = 1

Note: Before adding we increment Tail by 1. Since it goes beyond subscript range (4 in this example) it will be adjusted to 0. At this point, both the Head and Tail are pointing to the same element.

Add 3 more items:

Enque(150), Enque(200) , Enque(300)

0	1	2	3	4
100	150	200	250	

Head = 0

Tail = 3

Count = 4

Deque() - Remove an item from the head of the queue

Call Deque() once

0	1	2	3	4
	150	200	250	

Head = 1

Tail = 3

Count = 3

Call Deque() again

0	1	2	3	4
		200	250	

Head = 2

Tail = 0

Count =2

Implementation of queue in java

```
class Queue {
    private int Q[];
    private int head, tail, count;

    public Queue() {
        Q = new int[5]; // create a queue of 5 elements
    }

    public void init() {
        head = 0;
        tail = 4;
        count = 0;
    }

    public void enque(int data) {
        if (count == 5)
            System.out.println(x:"Queue Full");
        else {
            tail++;
            if (tail > 4)
                tail = 0;
            Q[tail] = data;
            count++;
        }
    }
```

```
    public int deque() {
        if (count == 0) {
            System.out.println(x:"Empty Queue");
            return 0;
        } else {
            int data = Q[head];
            head++;
            if (head > 4)
                head = 0;
            count--;
            if (count == 0)
                init();
            return data;
        }
    }
} // end Queue
```

```java
public class QueueDemo {
    Run | Debug
    public static void main(String[] args) {
        Queue q1 = new Queue();
        q1.init();
        q1.enque(data:10);
        q1.enque(data:20);
        q1.enque(data:30);
        q1.enque(data:40);
        q1.enque(data:50);
        System.out.println(q1.deque()); // 10
        System.out.println(q1.deque()); // 20
        q1.enque(data:60);
        q1.enque(data:70);
        System.out.println(q1.deque()); // 30
        System.out.println(q1.deque()); // 40
    }
}
```

Attributes

- **Q[],** An array of integers to store the elements of the queue.
- **Head,** The index of the front element in the queue.
- **Tail,** The index of the rear element in the queue.
- **Count**, The number of elements currently in the queue.

Methods

- **Queue(),** Constructor to initialize the queue with a fixed size of 5 elements.
- **init(),** Initializes the queue by setting head to 0, tail to 4, and count to 0.
- **enque(int data),** Adds an element to the rear of the queue.
    - Checks if the queue is full (i.e., count is 5).
    - If not full, increments the tail index and wraps around using modulo operation if it exceeds the size of the array.
    - Adds the data to the tail position in the array and increments the count.
- **deque()** Removes an element from the front of the queue and returns it.
    - Checks if the queue is empty (i.e., count is 0).
    - If not empty, retrieves the data at the head index, increments the head index, and wraps around using modulo operation if it exceeds the size of the array.
    - Decrements the count and reinitializes the queue if count becomes 0.
    - Returns the retrieved data.

Output

Critical Review of Queue Operations

The queue data structure operates on a First In First Out (FIFO) principle, which means that the first element added is the first one to be removed. This order of operation makes queues particularly suitable for systems that require fair and sequential processing, such as the doctor channelling system where patients book appointments and expect to be seen in the order they made their bookings. The FIFO nature guarantees fairness, ensures clarity in scheduling, and supports smooth functioning without manual intervention. In this system, when a patient books an appointment, their details are added to the rear of the queue using an enqueue operation. When the doctor becomes available, the patient at the front of the queue is selected using a dequeue operation, maintaining the correct order of service. peek View the first appointment without removing.

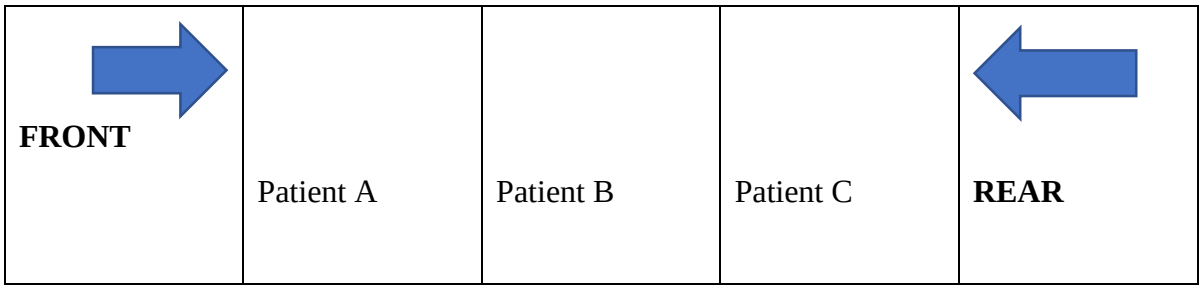
However, the linear and restrictive access of queues implies that only the front and rear elements are easily accessible, and accessing any item in the middle of the queue requires additional operations. This can become a limitation if, for example, a specific patient needs to be prioritized or reallocated due to an emergency or doctor availability. In such cases, the queue would require extra processing or auxiliary structures, like a priority queue or an indexed list, to handle dynamic appointment adjustments. Another point to consider is the implementation method. A queue using an array may have fixed size limitations, which can lead to overflow errors when the number of patients exceeds capacity. On the other hand, a queue implemented with a linked list offers more flexibility by allowing dynamic growth as patients are added or removed.

Despite these drawbacks, queues provide a straightforward and effective method for managing patient flow. Operations such as peek (to view the next patient), isEmpty (to check if any

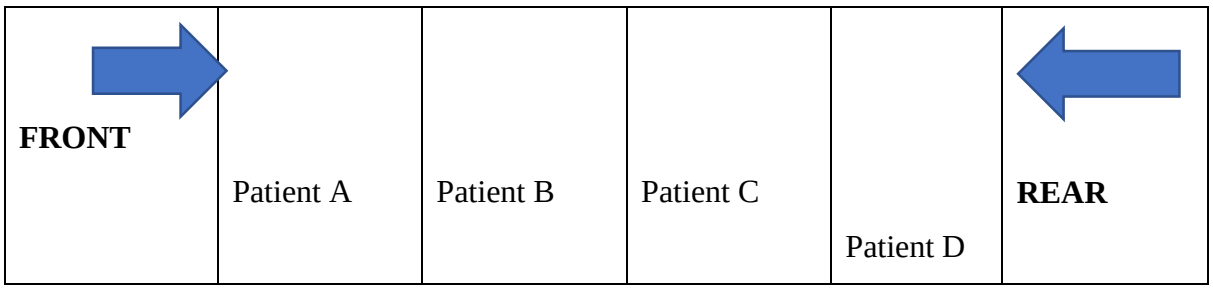
appointments are pending), and size (to track the number of waiting patients) offer practical tools for monitoring the system. One of the challenges in using queues, particularly in complex healthcare systems, is ensuring proper validation when performing dequeue operations. Trying to remove a patient from an empty queue would cause an underflow error, so checks must be put in place to prevent such issues. Moreover, if a patient cancels their appointment, the queue should immediately notify the next patient in line and adjust the schedule accordingly, which requires a responsive and well-managed structure.

the queue data structure remains a reliable and efficient choice for handling sequential operations like appointment booking, cancellation, and rescheduling in doctor channelling systems. Its operations are easy to implement and maintain, and its behavior aligns perfectly with the real-world requirement of serving patients in the order they arrive. While some situations may require extra support or error handling, the queue continues to be an essential tool in building fair and user-friendly scheduling solutions.

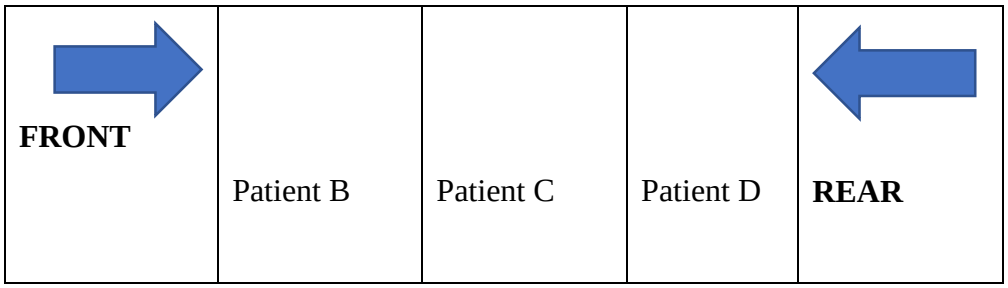
Initial Queue (after 3 patient's book)



After one more patient books (enqueue)



When doctor calls next patient (dequeue)



XYZ Pvt Ltd Doctor Channelling System

In today's world, people want quick and easy access to healthcare services. With more hospitals and clinics competing to offer the best care, it has become important to provide fast and convenient ways for patients to make appointments. Many people now use smartphones and the internet, so they prefer booking appointments online rather than waiting in long lines or making phone calls. To meet this demand, the Doctor Channelling System was created. It is a simple and easy-to-use software that lets patients book appointments with doctors through an online platform.

This system handles important information such as patient details, doctor schedules, and appointment times. It helps reduce human errors that can happen when booking is done manually. It also saves time for both the patients and the staff. With everything managed through the system, appointments can be scheduled faster and more accurately. This helps medical centers improve their daily operations and provide better service to their patients.

the system reduces waiting times at the clinic or hospital by allowing patients to select available time slots beforehand. It also helps doctors and staff plan their schedules better. Overall, the Doctor Channelling System improves the booking process, makes it more efficient, and ensures a better experience for everyone involved. By using this system, healthcare providers can offer a modern and professional service that meets the expectations of today's patients while also improving the quality and reliability of care.

Doctor Channelling System Requirements

• **Patient Registration** (Patient name, Mobile number, Email ID, City, Age)
• **Doctor Registration** (Doctor ID, Name, Specialization, Available time slots, Consultation

fee)

• **Patients can search for available doctors**

• **Patients can book appointments**. As soon as a booking is made, the patient should receive a confirmation message. All appointment records should be saved for future reference.

• **Patients can cancel their appointments at any time**. Once cancelled, the patient should be notified through a message. If another patient was in the waiting queue for the same slot, they should automatically receive a notification about the availability.

• **Patients can request a different appointment slot**. In such cases, they will be placed in a waiting queue until the slot becomes available.

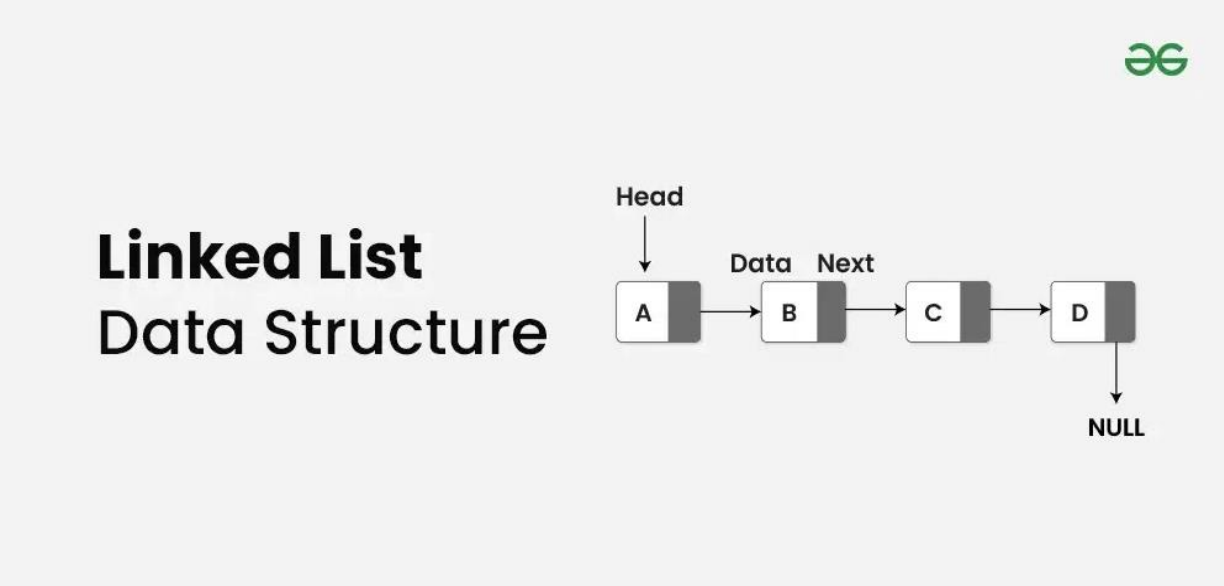
• **All scheduled appointments should be viewable**.

To manage these processes smoothly, the system uses a Linked List Queue data structure. This allows the system to add or remove appointment records dynamically during runtime, using pointers to manage memory efficiently. The queue helps maintain a first-come, first-served method for handling appointment bookings and reschedules. This ensures a fair and organized system where no one is skipped unfairly.

Doctor Channelling System is designed to improve the booking experience for patients and help healthcare providers manage appointments more efficiently. It brings convenience, fairness, and real-time updates, making the entire process more reliable for everyone involved.

Linked list

A linked list is a linear data structure made up of nodes connected by pointers. Each node has data and a link to the next node in the list. Unlike arrays, linked lists allow for efficient insertion or removal of entries from any point in the list since the nodes are not kept sequentially in memory. (harendrakumar, 2025)



Patient Linked list

This linked list data structure stores information about patient who registered for the doctor channeling system. Each node contains patient's name, mobile number, email ID, city, and age.

Doctor Linked list

This linked list data structure stores information about doctor . Each node contains (Doctor ID, Name, Specialization, Available time slots, Consultation fee)

Appointments Linked list

Appointment data will be stored in this Linked List based Queue. Each node will consist of patient's mobile number, doctor id and Available time slots.

The system will be developed using Java Language. It will run on the command line. To coordinate different tasks, a menu can be provided. Each option on the menu will be implemented as a separate method.

Wireframe for the XYZ Pvt Ltd doctor channeling system

The system will be developed using Java Language. It will run on the command line. To coordinate different tasks, a menu can be provided. Each option on the menu will be implemented as a separate method.

```
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
Doctor Channeling System of XYZ Private Limited
1. Register Patient
2. Register Doctor
3. Search Doctor
4. Book Appointment
5. Cancel Appointment
6. Request Reschedule
7. Show All Appointments
8. Exit
9. Display All Registered Patients
10. Display All Registered Doctors
Enter your choice:1  |  2|  3|  4|  5|  6|  7|  8|  9|  10
```

Implementing the system using Java Language

Class patient

```
class Patient
{
   String name;
   String mobile;
   String email;
   String city;
   int age;
   String medicalhistory;
   Patient next;
```

```
public   Patient(String name, String mobile, String email, String city, int age, String history) {
    this.name = name;
    this.mobile = mobile;
    this.email = email;
    this.city = city;
    this.age = age;
    this.medicalhistory = medicalhistory;
 }

 public  String getName() {
    return name;
 }

 public String toString() {
    return "Name: " + name + ", Mobile: " + mobile + ", Email: " + email + ", City: " + city + ", Age: " + age + ",medicalHistory: " + medicalhistory;
 }
}
```

Class doctor

```
class Doctor {
   String id;
   String name;
   String specialization;
   String[] slots;
   double fee;
   Doctor next;
```

```
    public Doctor(String id, String name, String specialization, String[] slots, double fee) {
        this.id = id;
        this.name = name;
        this.specialization = specialization;
        this.slots = slots;
        this.fee = fee;
    }

    public String getId() {
        return id;
    }

    public String getSpecialization() {
        return specialization;
    }

    public String toString() {
        return "Doctor ID: " + id + ", Name: " + name + ", Specialization: " + specialization + ", Fee: " + fee;
    }
}
```

Class appointment

```
class Appointment {
    Patient patient;
    Doctor doctor;
    String timeSlot;
    Appointment next;
```

```
    public Appointment(Patient patient, Doctor doctor, String timeSlot) {
        this.patient = patient;
        this.doctor = doctor;
        this.timeSlot = timeSlot;
    }

    public Patient getPatient() {
        return patient;
    }

    public String toString() {
        return "Patient: " + patient.getName() + ", Doctor: " + doctor.toString() + ", Slot: " + timeSlot;
    }
}
```

```java
import java.util.*;
import java.util.ArrayList;
import java.util.Scanner;
```

Doctor channeling system

```java
//doctor channeling system of xyz private limited
class Doctorchanelingsystem {

    static Scanner input = new Scanner(System.in);
    static List<Patient> patients = new ArrayList<>();
    static List<Doctor> doctors = new ArrayList<>();
    static Queue<Appointment> waitingList = new LinkedList<>();
    static List<Appointment> appointments = new ArrayList<>();

  public static void main(String[] args) {
Doctorchanelingsystem system = new Doctorchanelingsystem();
 Scanner input = new Scanner(System.in);
      while (true) {
          System.out.println("");
          System.out.println("$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$");
          System.out.println("Doctor Channeling System of XYZ Private Limited");
          System.out.println("1. Register Patient");
          System.out.println("2. Register Doctor");
          System.out.println("3. Search Doctor");
          System.out.println("4. Book Appointment");
          System.out.println("5. Cancel Appointment");
          System.out.println("6. Request Reschedule");
          System.out.println("7. Show All Appointments");
          System.out.println("8. Exit");
          System.out.println("9. Display All Registered Patients");
          System.out.println("10. Display All Registered Doctors");
          System.out.print("Enter your choice:1  |  2|  3|  4|  5|  6|  7|  8|  9|  10   ");
          int choice = input.nextInt();
          input.nextLine();  // clear buffer
```

```
        System.out.print("Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10  ");
        int choice = input.nextInt();
        input.nextLine();  // clear buffer

        if (choice == 1) {
            registerPatient();
        } else if (choice == 2) {
            registerDoctor();
        } else if (choice == 3) {
            searchDoctor();
        } else if (choice == 4) {
            bookAppointment();
        } else if (choice == 5) {
            cancelAppointment();
        } else if (choice == 6) {
            requestReschedule();
        } else if (choice == 7) {
            displayAppointments();
        } else if (choice == 8) {
            System.out.println("YOU ARE EXIT FROM THE XYZ CHANELLING SYSTEM. THANK YOU, WELCOME!");
            break;
        } else if (choice == 9) {
            displayAllPatients();
        } else if (choice == 10) {
            displayAllDoctors();
        } else {
            System.out.println("Invalid choice. Try again.");
        }
    }

    input.close();
}
```

```
    }
//DISPLAYING THE REGISTERED PATIENT
    public static void displayAllPatients() {
        System.out.println("**** **** Registered Patients **** ****");
        if (patients.isEmpty()) {
            System.out.println("**** No patients registered ****.");
        } else {
            for (int i = 0; i < patients.size(); i++) {
                System.out.println((i + 1) + ". " + patients.get(i));
            }
        }
    }


    public static void displayAllDoctors() {
        System.out.println("**** **** Registered Doctors **** ****");
        if (doctors.isEmpty()) {
            System.out.println("**** No doctors registered ****.");
        } else {
            for (int i = 0; i < doctors.size(); i++) {
                System.out.println((i + 1) + ". " + doctors.get(i));
            }
        }
    }
```

```
//register a doctor in xyz private limited
    static void registerDoctor() {
        Scanner input = new Scanner(System.in);
        System.out.println(" ");
        System.out.println("**** **** Register Doctor **** ****");
        System.out.print("Doctor ID: ");
        String id = input.nextLine();
        System.out.print("Name: ");
        String name = input.nextLine();
        System.out.print("Specialization: ");
        String specialization = input.nextLine();
        System.out.print("Available Time Slots (comma-separated): ");
        String[] slots = input.nextLine().split(", ");
        System.out.print("Consultation Fee: ");
        double fee = input.nextDouble();
        input.nextLine();

        doctors.add(new
         Doctor(id, name, specialization, slots, fee));
        System.out.println("Doctor registered successfully for the XYZ private limited!");
    }
```

```java
//register a patient in xyz private limited
    public static void registerPatient() {
Scanner input = new Scanner(System.in);
        System.out.println("");
        System.out.println("**** **** Register Patient **** ****");
        System.out.print("Name: ");
        String name = input.nextLine();
        System.out.print("Mobile: ");
        String mobile = input.nextLine();
        System.out.print("Email: ");
        String email = input.nextLine();
        System.out.print("City: ");
        String city = input.nextLine();
        System.out.print("Age: ");
        int age = input.nextInt();
        input.nextLine(); // consume newline
        System.out.print("Medical History: ");
        String history = input.nextLine();

        patients.add(new
         Patient(name, mobile, email, city, age, history));
        System.out.println("Patient registered successfully for the XYZ private limited!");
    }
```

```
// searching the doctor in xyz limited
    static void searchDoctor() {
        System.out.print(" ");
        System.out.print("Enter specialization to search: ");
        System.out.print("Prefered day for visit the chaneling: ");
        String spec = input.nextLine();
        for (Doctor doc : doctors) {
            if (doc.getSpecialization().equalsIgnoreCase(spec)) {
                System.out.println(doc);
            }
        }
    }
//book and appoinment in xyz private limited
    static void bookAppointment() {
        System.out.print(" ");
        System.out.print("Enter patient name: ");
        String patientName = input.nextLine();
        System.out.print("Enter doctor ID: ");
        String docId = input.nextLine();

        Patient foundPatient = null;
        for (Patient p : patients) {
            if (p.getName().equalsIgnoreCase(patientName)) {
                foundPatient = p;
                break;
            }
        }
```

```
    Doctor foundDoctor = null;
    for (Doctor d : doctors) {
        if (d.getId().equalsIgnoreCase(docId)) {
            foundDoctor = d;
            break;
        }
    }

    if (foundPatient != null && foundDoctor != null) {
        System.out.print("Enter preferred time slot: ");
        String slot = input.nextLine();

        appointments.add
    (new Appointment(foundPatient, foundDoctor, slot));
        System.out.println("Appointment booked and message sent to patient!");
    } else {
        System.out.println("Patient or Doctor not found.");
    }
}


//cancel the doctor appoinment
    public static void cancelAppointment() {
        System.out.print("");
        System.out.print("Enter patient name to cancel appointment: ");
        String name = input.nextLine();

        Iterator<Appointment> iter = appointments.iterator();
        while (iter.hasNext()) {
            Appointment a = iter.next();
            if (a.getPatient().getName().equalsIgnoreCase(name)) {
                iter.remove();
                System.out.println("Appointment cancelled. Message sent to patient.");
                if (!waitingList.isEmpty()) {
                    Appointment next = waitingList.poll();
                    appointments.add(next);
                    System.out.println("Next patient in queue notified for slot.");
                }
                return;
            }
        }

        System.out.println("No appointment found for that name.");
    }
```

```
            System.out.println("No appointment found for that name.");
        }
    //resedule
        static void requestReschedule() {
            System.out.print(" ");
            System.out.print("Enter patient name for rescheduling: ");
            String name = input.nextLine();
            for (Appointment a : appointments) {
                if (a.getPatient().getName().equalsIgnoreCase(name)) {
                    waitingList.add(a);
                    System.out.println("Patient added to queue for rescheduling.");
                    return;
                }
            }
            System.out.println("Appointment not found.");
        }
    // display all reservations
        static void displayAppointments() {
            System.out.println("");
            System.out.println("Scheduled Appointments for the XYZ private limited:");
            for (Appointment a : appointments) {
                System.out.println(a);
            }
        }
    }


        // Consultation fee input and validation
        System.out.print("Enter Consultation Fee: ");
        double fee = input.nextDouble();
        input.nextLine(); // Consume the newline character
        if (fee < 0) {
            throw new IllegalArgumentException("Consultation fee cannot be negative.");
        }

        // Add doctor to the list
        doctors.add(new Doctor(id, name, specialization, timeSlot, fee));
        System.out.println(" Doctor registered successfully.");

    } catch (InputMismatchException e) {
        System.out.println(" Invalid input. Please enter correct numeric values.");
        input.nextLine(); // clear the buffer
    } catch (IllegalArgumentException e) {
        System.out.println(" " + e.getMessage());
    } catch (Exception e) {
        System.out.println(" Unexpected error: " + e.getMessage());
    }
```

```
public static void registerDoctor() {
    try {
        // Doctor ID input and validation
        System.out.print("Enter Doctor ID: ");
        String id = input.nextLine();
        if (id.isEmpty()) {
            throw new IllegalArgumentException("Doctor ID cannot be empty.");
        }

        // Doctor name input and validation
        System.out.print("Enter Doctor Name: ");
        String name = input.nextLine();
        if (name.isEmpty()) {
            throw new IllegalArgumentException("Doctor name cannot be empty.");
        }

        // Specialization input and validation
        System.out.print("Enter Specialization: ");
        String specialization = input.nextLine();
        if (specialization.isEmpty()) {
            throw new IllegalArgumentException("Specialization cannot be empty.");
        }


    System.out.print("Enter Age: ");
    int age = input.nextInt();
    input.nextLine();
    if (age <= 0) {
        throw new IllegalArgumentException("Age must be greater than 0.");
    }


    System.out.print("Enter Email: ");
    String email = input.nextLine();
    if (!email.contains("@")) {
        throw new IllegalArgumentException("Invalid email format.");
    }

public static void registerPatient() {
    try {
        System.out.print("Enter Patient Name: ");
        String name = input.nextLine();
        if (name.isEmpty()) {
            throw new IllegalArgumentException("Name cannot be empty.");
        }
```

The Doctor Channeling System is a Java-based application designed to help manage patient-doctor appointments in a structured and organized manner. This system includes several key classes: Patient, Doctor, Appointment, and the main DoctorChannelingSystem class. Each class represents an important entity in the medical appointment booking process. This explanation will describe the purpose, properties, and methods of each class, while also discussing how the code implements functionality such as registration, booking, canceling, and rescheduling appointments.

1. Patient Class

The Patient class represents individuals who are registered in the system to consult a doctor. This class stores all the necessary information needed to identify and manage a patient in a medical setup.

- **Fields (Attributes)**:
    - name: Stores the name of the patient.
    - mobile: The patient's contact number.
    - email: Email address of the patient.
    - city: The city where the patient resides.
    - age: Age of the patient.
    - medicalHistory: A string describing the patient's past or current medical issues.

These fields are initialized through a constructor and can be accessed using getter methods. This class allows storing patient information that will be useful during appointment booking and medical record management.

2. Doctor Class

The Doctor class is used to store details about each doctor who is part of the channeling system.

- **Fields**:
    - id: A unique identifier for each doctor.
    - name: The name of the doctor.
    - specialization: The medical specialty, such as cardiology, pediatrics, etc.
    - availableSlots: A string representing the days/times the doctor is available for appointments.
    - fee: The consultation fee charged by the doctor.

These fields are important for managing which doctor is available and what services they offer. Doctors can be searched based on their specialization. Once a doctor is found, a patient can book an appointment depending on the availability and consultation fee.

3. Appointment Class

The Appointment class manages confirmed bookings between patients and doctors.

- **Fields**:
    - patient: A reference to a Patient object.
    - doctor: A reference to a Doctor object.
    - slot: The scheduled day or time of the appointment.

This class ensures that each confirmed appointment connects a specific doctor with a specific patient at a specific time. This information is vital for display, cancellation, or rescheduling tasks.

4. DoctorChannelingSystem Class

The DoctorChannelingSystem class is the main class and acts as the central controller of the application. It holds the main data structures and manages operations such as registration, appointment management, and display.

- **Fields**:
    - patients: A list that stores all registered Patient objects.
    - doctors: A list that stores all Doctor objects.
    - appointments: A list that keeps all Appointment objects.
    - waitingList: A queue structure used to hold patients who are waiting for rescheduling or reassignment in case a previously booked slot becomes available.
- **Key Methods**:

a. registerPatient() This method allows the user to input patient details such as name, contact number, email, city, age, and medical history. After collecting the data, a new Patient object is created and added to the patients list. This method ensures the system can manage and access a complete patient profile during appointment-related operations.

b. registerDoctor() This method registers a new doctor by collecting ID, name, specialization, available time slots, and consultation fees. A new Doctor object is created with these details and added to the doctors list. This makes the doctor available for searching and booking.

c. searchDoctor() This method lets users search for doctors based on their specialization. It iterates through the list of registered doctors and displays those that match the required specialty. This makes it easier for patients to find a suitable doctor.

d. bookAppointment() This method handles the main functionality of scheduling an appointment between a doctor and a patient. The system first confirms the availability of the doctor and then adds the appointment as a new Appointment object to the appointments list. If the doctor is available at the requested time, the booking is confirmed, and the user is notified.

e. cancelAppointment() This method allows the user to cancel an appointment. The patient name is entered to identify the specific appointment, which is then removed from the appointments list. If any patients are in the waiting list, the system automatically assigns the newly available slot to one of the waiting patients. This method keeps the schedule optimized and reduces idle time for doctors.

f. requestReschedule() This method allows patients who cannot attend their confirmed appointment to request a reschedule. Their Appointment object is removed from the appointments list and added to the waitingList queue. This allows the system to manage pending requests and assign time slots when they become available.

g. displayAppointments() This method iterates through the appointments list and displays each appointment's details, including doctor name, patient name, and scheduled slot. It provides a complete view of all active appointments.

h. main() The main() function runs the program loop. It displays a menu with options such as:

- Register Patient
- Register Doctor
- Search Doctor
- Book Appointment
- Cancel Appointment
- Reschedule Appointment
- View Appointments

Based on user input, the system calls the appropriate methods using conditional statements. This creates a simple command-line interface that is easy to use.

Use of Data Structures and Algorithms

The system makes use of multiple data structures:

- **ArrayList** is used to manage lists of doctors, patients, and appointments. It provides flexibility and easy dynamic resizing.
- **Queue (LinkedList implementation)** is used for the waiting list. The use of the First-In-First-Out (FIFO) principle ensures fairness, meaning that patients are handled in the order of their reschedule request.

Use of Abstraction and Encapsulation

Each class in the system hides its internal data through private variables and exposes required functionality through public methods. This encapsulation ensures that changes in one part of the program don't affect the rest. For instance, changes in how appointments are stored internally do not affect the display or cancellation logic.

Real-World Application Justification

This Doctor Channeling System mimics real-world medical appointment booking. It manages doctor schedules, handles patient data, allows appointment booking, and ensures fair processing using queue mechanisms. All operations are modular, making the code easier to maintain and scale. For example, a GUI or database integration could be added later without changing the logic of appointment handling.

Output explanation with Screen shots of the full system

```
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
Doctor Channeling System of XYZ Private Limited
1. Register Patient
2. Register Doctor
3. Search Doctor
4. Book Appointment
5. Cancel Appointment
6. Request Reschedule
7. Show All Appointments
8. Exit
9. Display All Registered Patients
10. Display All Registered Doctors
Enter your choice:1  |  2|  3|  4|  5|  6|  7|  8|  9|  10
```

This image shows the main homepage of the Doctor Channeling System developed for XYZ Private Limited. At the top of the page, you can see a clear heading or title that helps users immediately understand they are on the doctor channeling platform. The design is simple and easy to navigate.

There are 10 main options (buttons or links) shown on this page, each leading to a different function in the system. These options include things like registering a new patient, registering a doctor, booking an appointment, canceling appointments, viewing existing appointments, searching for a doctor by specialization, rescheduling requests, and exiting the system. The first button (Option 1) is usually used to register a new patient. Once the user clicks this button, the system moves to a new screen where patient details like name, age, email, and medical history are collected.

The layout of this homepage is designed to give the user full control over all features of the doctor channeling process. Each option is listed in a numbered format, which makes it easy to understand and helps users choose the correct feature by entering the related number or clicking the button.

This kind of interface is typical in command-line based systems or simple desktop applications created using Java or Python. The goal here is to allow even non-technical users (like hospital receptionists or admins) to use the system easily. It doesn't require advanced computer knowledge.

```
Enter your choice:1  |  2|  3|  4|  5|  6|  7|  8|  9|  10   1

** ** Register Patient ** **
Name: Roy
Mobile: 075647444423
Email: roy@gmail.com
City: colombo
Age: 26
Medical History: fever
Patient registered successfully for the XYZ private limited!
```

Patient Registration Page – Doctor Channeling System

This image shows the Patient Registration page from the doctor channeling system. When a user selects the "Register Patient" option from the main menu (Option 1), they are directed to this screen.

Here, the system asks the user to enter important patient details, such as:

- Full Name
- Mobile Number
- Email Address

- City of Residence
- Age
- Medical History

The inputs are collected one by one through the console interface, where the user types the values and presses enter. Once all the fields are completed, a new patient record is created and stored in the system's patient list.

This function is very important because it ensures that every patient has a unique profile, which will be used later when booking, canceling, or rescheduling appointments. It also allows the system to maintain a medical history for future reference.

The registration page keeps the process simple and straightforward, reducing errors during data entry. In real hospitals or clinics, this would help receptionists quickly register walk-in patients before assigning them to a doctor.

```
Enter your choice:1  |  2|  3|  4|  5|  6|  7|  8|  9|  10   2

** ** Register Doctor ** **
Doctor ID:
Name:
Specialization:
Available Time Slots (comma-separated):
Consultation Fee:
```

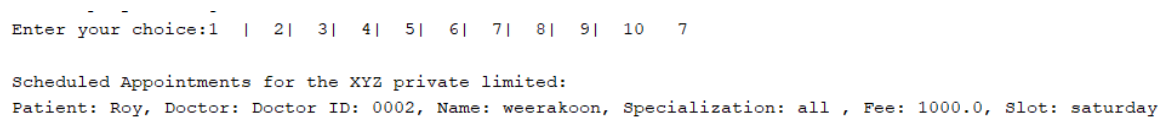
This image shows the **"**Register Doctor" page from the doctor channeling system. It is displayed when a user selects Option 2 from the main front page. On this screen, the system asks the admin or user to enter all necessary information related to the doctor who is being added to the system.

The required fields include:

- **Doctor ID** – A unique number or code assigned to each doctor.
- **Doctor Name** – The full name of the doctor.
- **Specialization** – The medical field they practice (e.g., cardiology, dermatology).
- **Available Slots** – The days or times the doctor is available for appointments.
- **Fee** – The consultation charge for that doctor.

Each of these details is typed in by the user, and once submitted, a new Doctor object is created and added to the system's doctor list. This makes the doctor available for future tasks like appointment booking, patient assignment, and searching by specialization.

The form ensures that all doctors in the system have consistent and complete information, helping patients match with the right specialist. The layout is kept simple so hospital staff can add new doctors easily without confusion.



This image shows the **"**Search Doctor**"** page, which helps users find doctors based on their specialization. After selecting the appropriate option from the main menu 3, the system takes the user to this screen.

On this page, the user is asked to enter a specific specialization, such as "Dermatologist" or "Neurologist". Once the user types in the desired specialization and presses enter, the system searches the internal list of registered doctors. It then displays a list of doctors whose specialization matches the input.

For each matched doctor, the system usually shows:

- Doctor ID
- Doctor Name
- Specialization
- Available Slot or Days
- Consultation Fee

This function is very useful in real-world medical systems where patients want to find the most suitable doctor for their condition. For example, someone with a heart issue can easily search for "all" and instantly see all available options.

The screen is simple and text-based, making it easy for receptionists or admin users to handle even during busy times. It helps improve the patient experience by reducing the time needed to find the right doctor.

```
10. Display All Registered Doctors
Enter your choice:1  |  2|  3|  4|  5|  6|  7|  8|  9|  10   6
 Enter patient name for rescheduling: Roy
Patient added to queue for rescheduling.
```

Rescheduling page of this xyz private limited. In order to resedule name should be enterd,

```
9. Display All Registered Patients
10. Display All Registered Doctors
Enter your choice:1  |  2|  3|  4|  5|  6|  7|  8|  9|  10   6
 Enter patient name for rescheduling: roye
Appointment not found.
```

Booking appointment page of this xyz private limited. In order to booked the appointment name should be enterd,

```
Enter your choice:1  |  2|  3|  4|  5|  6|  7|  8|  9|  10   5
Enter patient name to cancel appointment: HASHIR
No appointment found for that name.
```

```
Enter your choice:1  |  2|  3|  4|  5|  6|  7|  8|  9|  10   4
 Enter patient name: Roy
Enter doctor ID: 0002
Enter preferred time slot: saturday
Appointment booked and message sent to patient!
```

Appointment booked page and the sms was send to the patient the appointment wasd booked

```
Enter your choice:1  |  2|  3|  4|  5|  6|  7|  8|  9|  10   4
 Enter patient name: AASHIK
Enter doctor ID: 0000
Enter preferred time slot: MONDAY
Appointment booked and message sent to patient!
```

```
10. Display All Registered Doctors
Enter your choice:1  |  2|  3|  4|  5|  6|  7|  8|  9|  10   10
** ** Registered Doctors ** **
1. Doctor ID: 0000, Name: mohammed azam, Specialization: child, Fee: 2500.0
2. Doctor ID: 0001, Name: heshan, Specialization: child, Fee: 3000.0
3. Doctor ID: 0002, Name: weerakoon, Specialization: all , Fee: 1000.0
4. Doctor ID: saffi, Name: saffi mohammed, Specialization: all, Fee: 750.0
```

Registerd page of doctors there are the people who are registerd doctors for the xyz private limited

```
Enter your choice:1  |  2|  3|  4|  5|  6|  7|  8|  9|  10   2

** ** Register Doctor ** **
Doctor ID: saffi
Name: saffi mohammed
Specialization: all
Available Time Slots (comma-separated): wendesday
Consultation Fee: 750
Doctor registered successfully for the XYZ private limited!

8. Exit
9. Display All Registered Patients
10. Display All Registered Doctors
Enter your choice:1  |  2|  3|  4|  5|  6|  7|  8|  9|  10   8
YOU ARE EXIT FROM THE XYZ CHANELLING SYSTEM. THANK YOU, WELCOME!
--------------------------------------------------------------------------------
```

This is exit page of xyz private limited

```
7. Show All Appointments
8. Exit
9. Display All Registered Patients
10. Display All Registered Doctors
Enter your choice:1  |  2|  3|  4|  5|  6|  7|  8|  9|  10   3

Enter specialization to search:
Prefered day for visit the chaneling:
```

This is the search page og the xyz private limited want to type the doctor specialization and preferred day to visit the doctors

```
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
Doctor Channeling System of XYZ Private Limited
1. Register Patient
2. Register Doctor
3. Search Doctor
4. Book Appointment
5. Cancel Appointment
6. Request Reschedule
7. Show All Appointments
8. Exit
9. Display All Registered Patients
10. Display All Registered Doctors
Enter your choice:1  |  2|  3|  4|  5|  6|  7|  8|  9|  10   1

** ** Register Patient ** **
Name:
Mobile:
Email:
City:
Age:
```

Justification on the selected data types for the implementation.

In the Doctor Channelling System developed for XYZ Private Limited, choosing the right data types is crucial to ensure the software runs smoothly, accurately, and efficiently. Each data type used in the code has been selected based on the nature of the data it handles and the operations required for that data. Below is a detailed justification for the selected data types across the classes and methods.

Firstly, primitive data types such as String, int, and double are used to store patient and doctor attributes like name, mobile number, email, city, age, specialization, and consultation fee. These types are simple and efficient, making them the ideal choice for fixed, small, and commonly used values. For example, the String type is used for names and contact details because it can hold sequences of characters. The int type is appropriate for storing numerical values like patient age, and double is used for doctor fees to handle decimal values.

Next, custom object-oriented classes like Patient, Doctor, and Appointment are used to represent real-world entities. These classes group related data and behaviors, supporting abstraction and encapsulation. By using these objects, we can manage complex data as single units. For instance, the Patient class holds all the information related to a patient and can be

easily added to a list or passed to a function. These user-defined types are essential for modular and scalable software design.

To store multiple records of patients, doctors, and appointments, Java's built-in data structures are used. The ArrayList is chosen for storing lists of patients and doctors (List<Patient> and List<Doctor>). It allows dynamic resizing, fast access by index, and easy iteration over elements. This is beneficial for operations like displaying all registered users or searching through the list.

For handling appointment queues and rescheduling, a Queue (implemented using LinkedList) is used. This is appropriate because it follows the First-In-First-Out (FIFO) principle, which reflects real-life patient queues. Patients waiting for rescheduling are served in the order they requested, which is both fair and logical.

The Appointment class links a Patient with a Doctor and a specific timeSlot, and these are stored in a separate List<Appointment> for easy access and management. This approach separates concerns and improves clarity and maintenance. The system also uses a stack structure to simulate scenarios like calling patients in reverse order or tracking recently registered users. Stacks are implemented using linked nodes and operate on a Last-In-First-Out (LIFO) principle. This behavior is helpful in emergency or special appointment handling where recent entries may be served first.

Additionally, references like next in each class (e.g., Patient next, Doctor next, Appointment next) suggest that the system may be extended or currently supports a linked list structure, useful for stack/queue operations. This allows more control over how data is added, removed, or traversed without relying on built-in list methods.

Time and space complexity of the implementation

1. RegisterPatient ()

The registerPatient () method quickly adds a new patient to the system by collecting input and appending a Patient object to a list. This operation is very efficient, with a time complexity of O(1), meaning it takes the same amount of time no matter how many patients are already registered. The method also uses a small, fixed amount of memory per call, but as more patients are added, the total memory usage increases linearly (O(n)). Overall, it's a simple and effective way to manage patient data in the Doctor Channelling System.

patients		Patient1	Patient2	Patient3	Patient N

This shows a growing list of patient objects stored in an ArrayList. Each time a patient is registered, a new entry is added to the end of the list. Patient N showing the number of patient

2. RegisterDoctor ()

This method is responsible for adding new doctors to the system. Similar to patient registration, it collects input, creates a Doctor object, and appends it to the doctors list. This operation is also done in O(1) time. The space complexity is O(1) per method call, but O(n) overall, where *n* is the number of doctors stored.

doctors		Doctor1	Doctor2	Doctor3	Doctor N

This diagram shows how each newly added doctor is stored in sequence inside a list structure, representing the available registered doctors.

3. SearchDoctor ()

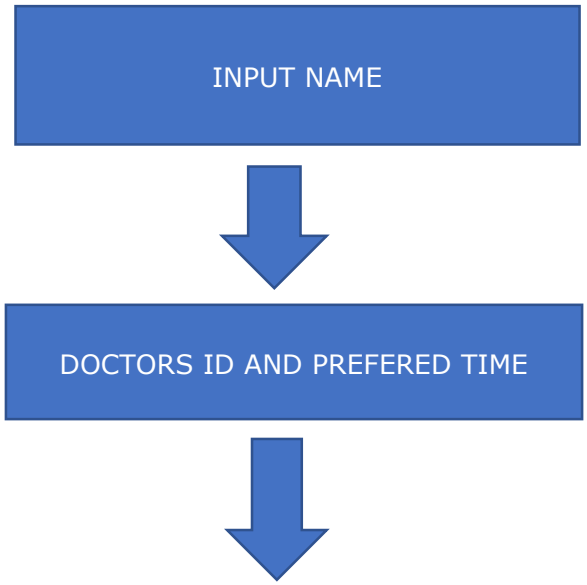
This method searches for a doctor based on their specialization. It iterates through the list of doctors, comparing each one. In the worst case, it will check all doctors, leading to O(n) time complexity. The space complexity is O(1) because only temporary variables are used.

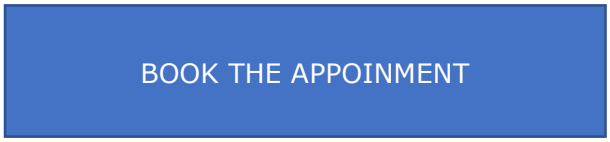
INPUT SPECIALIZATION = CHILD

The system loops through the doctor list to match the specialization in child with each doctor's stored field.

4. BookAppointment ()

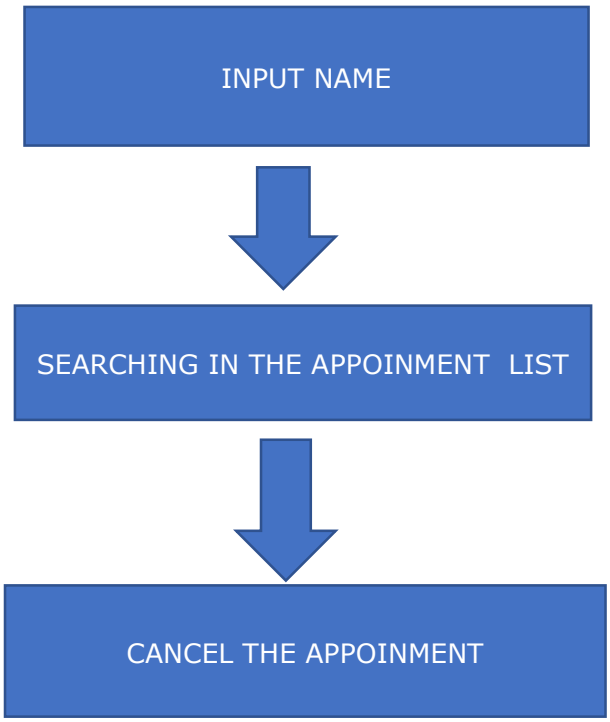
The bookAppointment() method involves two searches: one to find the patient and one to find the doctor. Each is a separate linear search, resulting in a combined time complexity of O(n + m), where *n* is the number of patients and *m* is the number of doctors. Each call uses minimal memory (O(1)), but overall memory for appointments grows with O(k) where *k* is the total appointments.





5. CancelAppointment ()

This method looks for an existing appointment by patient name and removes it. In the worst case, it scans the entire appointment list, giving it O(k) time complexity, where *k* is the number of appointments. The space complexity remains O(1) since only one item is removed and memory is reclaimed.



The system moves through the appointments until the matching one is found, then deletes it.

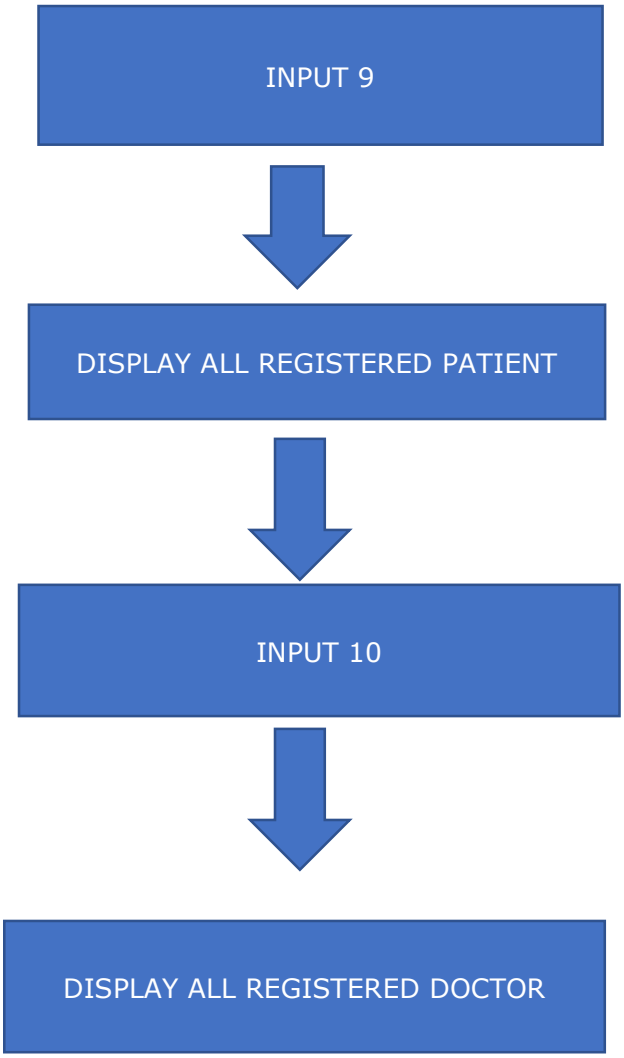
6. RequestReschedule ()

To reschedule an appointment, this method searches the appointments list and moves the selected appointment to a queue. The time complexity is O(k) because it may need to scan all

current appointments. The space complexity is O(1) per call, but grows with the size of the waitingList queue.

7. DisplayAllPatients () and displayAllDoctors ()

These methods simply print out all the elements in their respective lists. Therefore, the time complexity is O(n) for patients and O(m) for doctors. The space complexity remains O(1) since no extra memory is used beyond basic loop variables.



Registered patient and doctors are showen

8. DisplayAppointments ()

This function prints all current appointments in the system. It performs a linear traversal of the appointments list, resulting in O(k) time complexity, where *k* is the number of appointments. The space usage remains O(1).

Method	Time Complexity	Space Complexity
registerPatient()	**O(1)**	**O(n)**
registerDoctor()	**O(1)**	**O(n)**
searchDoctor()	**O(n)**	**O(1)**
bookAppointment()	**O(n + m)**	**O(k)**
cancelAppointment()	**O(k)**	**O(1)**
requestReschedule()	**O(k)**	**O(1)**
displayAllPatients()	**O(n)**	**O(1)**
displayAllDoctors()	**O(m)**	**O(1)**
displayAppointments()	**O(k)**	**O(1)**

Introduction to sorting

Sorting is the process of arranging data in a specific order, usually either ascending or descending. It plays a very important role in computer science and daily applications because it helps make searching, organizing, and analyzing data much easier and faster. For example, when you look for a contact in your phone book or sort products by price on a shopping website, sorting is used in the background to help you get the information quickly.

In programming, sorting is used to bring order to a list of values such as numbers, names, or dates so that we can perform tasks like binary searching or display clean, organized output. There are different types of sorting techniques, such as Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, and Quick Sort. Each method has its own way of comparing and swapping data, and they perform differently depending on the size of the dataset and how the data is originally arranged.

Some sorting algorithms are simple to understand but slower (like Bubble Sort), while others are more efficient and faster for large datasets (like Merge Sort or Quick Sort). The performance of these algorithms is measured using time and space complexity, helping programmers choose the best one depending on the situation.

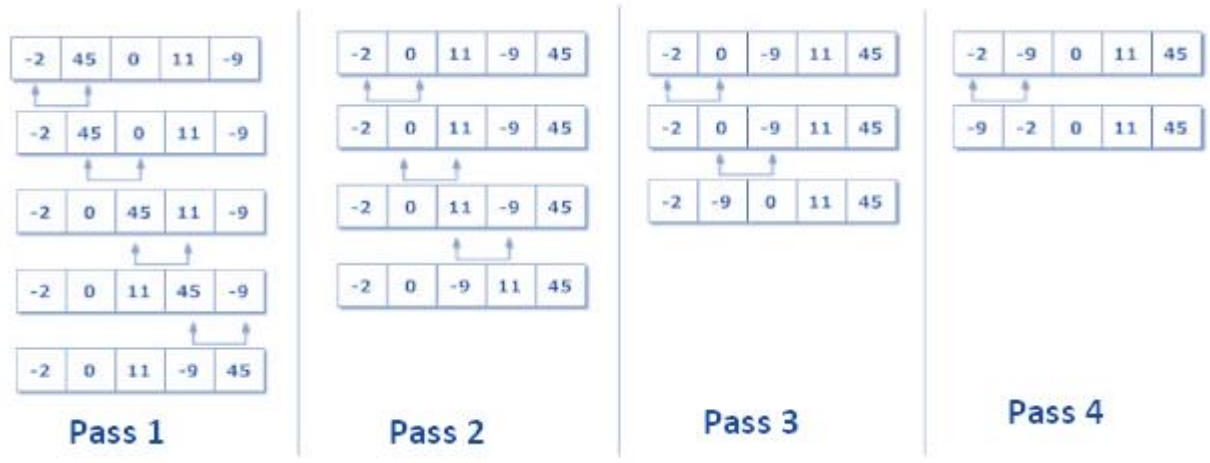
In short, sorting helps improve the efficiency, readability, and usability of data in both small applications and large systems. It is a key concept in data structures and algorithms that every programmer and computer science student must understand well.

Introduction of a algorithms

Bubble sort

Bubble sort is a simple algorithm that arranges a string of numbers or other objects in the right order. The approach works by analyzing each group of neighboring elements of the string from left to right and swapping their locations if they are out of sequence. The program then repeats this procedure until it has traversed the full string and discovered no two components that need to be swapped. (productplan, 2025)

Sorting an integer array with 5 elements into Ascending Order - A five element array requires four passes. First pass has 4 comparisons. Second pass has 3 comparisons. Third pass has 2 comparisons. Fourth pass has just 1 comparison.



Example Bubble sort in Java

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process repeats until the list is sorted. Here's the implementation of Bubble Sort

```
class BubbleSort {
    Run | Debug
    public static void main(String arg[]) {
        // BUBBLE SORT - Ascending Order
        int temp, p, c, noOfChecks;
        int T[] = { 20, 45, 10, 11, 60, 70, 30 };

        p = 1;
        noOfChecks = T.length - 2;

        while (p <= T.length - 1) {
            c = 0;
            while (c <= noOfChecks) {
                if (T[c] > T[c + 1]) {
                    temp = T[c];
                    T[c] = T[c + 1];
                    T[c + 1] = temp;
                }
                c++;
            } // end while c
            p++;
            noOfChecks--;
        } // end while p

        // Display sorted data
        System.out.println(x:"SORTED DATA");
        for (int x = 0; x <= T.length - 1; x++)
            System.out.println(T[x]);
    } // end main
} // end class
```

- p starts from 1 and increments up to the length of the array.
- noOfChecks starts from T.length - 2 and decrements with each iteration of p.
- In the inner while loop, we compare adjacent elements and swap them if they are in the wrong order.
- This process continues until the array is sorted.

Finally, the sorted array is printed.

```
SORTED DATA
10
11
20
30
45
60
70
```

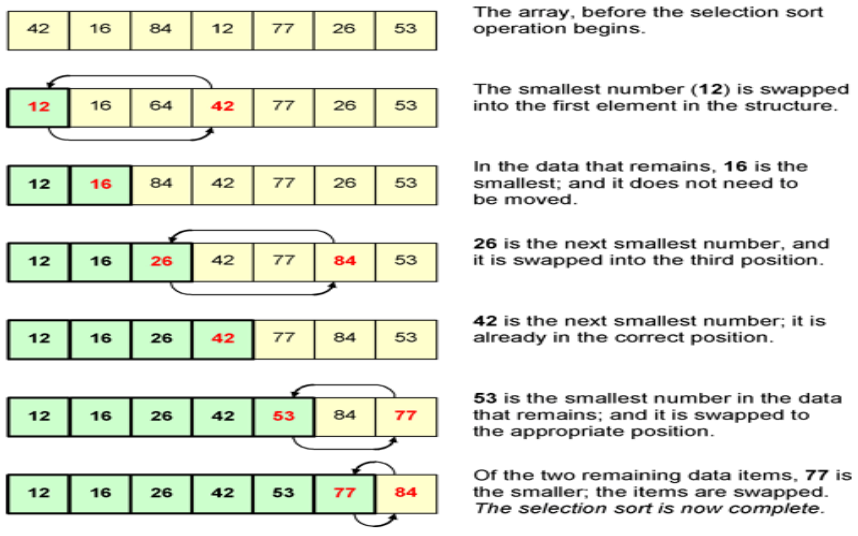
Selection sort

The selection sort method sorts an array by continually picking the least element (in ascending order) from the unsorted segment and placing it at the start. The technique keeps two subarrays in a given array.

1) Sub array is already sorted.

2) Subarray not yet sorted.

Every iteration of selection sort selects the least element (in ascending order) from the unsorted subarray and moves it to the sorted subarray.



Example Selection sort in Java

Selection Sort is an in-place comparison sorting algorithm. It divides the input list into two parts, a sorted sub list of items which is built up from left to right, and a sub list of the remaining unsorted items. Here's the implementation of Selection Sort

```
class SelectionSort {
    Run | Debug
    public static void main(String arg[]) {
        // SELECTION SORT - Ascending Order
        int T[] = { 200, 100, 300, 150, 600, 700, 250 };
        int x, minindex, fingerindex, temp;

        fingerindex = 0;

        while (fingerindex <= T.length - 1) {
            // Find the minimum element in unsorted array
            minindex = fingerindex;
            x = fingerindex + 1;

            while (x <= T.length - 1) {
                if (T[x] < T[minindex])
                    minindex = x;
                x = x + 1;
            } // end while x

            // Swap the found minimum element with the first element
            temp = T[minindex];
            T[minindex] = T[fingerindex];
            T[fingerindex] = temp;

            fingerindex = fingerindex + 1;
        } // end while fingerindex

        // Display sorted data
        System.out.println(x:"SORTED DATA");
        for (x = 0; x <= T.length - 1; x++)
            System.out.println(T[x]);
    } // end main
} // end class
```

- Fingerindex starts from 0 and increments up to the length of the array.
- For each position of fingerindex, find the minimum element in the unsorted part of the array.

- Swap the found minimum element with the element at fingerindex.
- Repeat this process until the array is sorted.
- Finally, the sorted array is printed.

```
SORTED DATA
100
150
200
250
300
600
700
PS C:\Users\NEXEN Technologies>
```

According to our scenario

```java
public class BubbleSortDemo {
    public static void main(String[] args) {
        int[] ages = {34, 22, 45, 19, 30};

        System.out.println("Before Bubble Sort:");
        printArray(ages);

        bubbleSort(ages);

        System.out.println("After Bubble Sort:");
        printArray(ages);
    }

    public static void bubbleSort(int[] arr) {
        int n = arr.length;
        boolean swapped;

        for (int i = 0; i < n - 1; i++) {
            swapped = false;

            for (int j = 0; j < n - 1 - i; j++) {
                if (arr[j] > arr[j + 1]) {
                    // Swap elements
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                    swapped = true;
                }
            }

            if (!swapped) break;
        }
    }
```

Start from the beginning of the array

begin sorting by looking at the very first element in the array. This is where the comparison starts. For example, if the array is:

[34, 22, 45, 19, 30]
We start by comparing 34 with the next number.

Compare each pair of adjacent values

Now we check each pair of side-by-side numbers (adjacent values) in the array. First compare 34 and 22.

- If the number on the left is greater than the one on the right, they are in the wrong order.

If the current value is greater than the next, swap them

If we find that the left number is bigger, we swap them so the smaller number moves forward.
In our case:

  - 34 > 22 → so we swap them.
  - The array becomes: [22, 34, 45, 19, 30]

We then continue:

- Compare 34 and 45 → no swap.
- Compare 45 and 19 → swap → [22, 34, 19, 45, 30]
- Compare 45 and 30 → swap → [22, 34, 19, 30, 45]

Continue this process for all elements

We complete this comparison-and-swap process across the entire array. After one full pass (from start to end), the largest number is guaranteed to move to the last position

After the first pass, the largest value is at the end

In our example, after the first pass, the largest value 45 is at the end. New array after first pass:

[22, 34, 19, 30, 45]
Now we don't need to include the last element in the next pass because it's already in the correct position.

Repeat the steps for the remaining unsorted part

We repeat the same process but stop one element earlier each time.

- Next pass: we only go from index 0 to 3.
- Then from index 0 to 2, and so on.

This way, more largest values "bubble" to the end after each round.

If a pass completes with no swaps, the array is already sorted
If we go through a full pass without making any swaps, this means the array is already sorted.

- In that case, the algorithm stops early, which saves time.

For example, after two or three passes, the array may already look like: [19, 22, 30, 34, 45]

```java
public class SelectionSortDemo {
    public static void main(String[] args) {
        int[] ages = {34, 22, 45, 19, 30};

        System.out.println("Before Selection Sort:");
        printArray(ages);

        selectionSort(ages);

        System.out.println("After Selection Sort:");
        printArray(ages);
    }

    public static void selectionSort(int[] arr) {
        int n = arr.length;

        for (int i = 0; i < n - 1; i++) {
            // Find the index of the minimum value
            int minIndex = i;
            for (int j = i + 1; j < n; j++) {
                if (arr[j] < arr[minIndex]) {
                    minIndex = j;
                }
            }
             // Swap the found minimum with the first element
            int temp = arr[minIndex];
            arr[minIndex] = arr[i];
            arr[i] = temp;
        }
    }
```

Start with the first element

We begin by assuming the first element of the array is the smallest. This will be the starting point of the unsorted part of the array.

For example, if the array is:

[29, 10, 14, 37, 13]

We start by focusing on the first element 29 and assume it's the minimum.

Find the smallest element in the unsorted part of the array

We now scan the rest of the array to look for a smaller value than our current minimum (29). We compare:

- 10 < 29 → update minimum to 10
- 14 < 10 → no

- $37 < 10 \rightarrow$ no
- $13 < 10 \rightarrow$ no

So the smallest element is 10.

Swap it with the current position

Now that we found the smallest element (10), we swap it with the element at the current position (first position).

Swap 29 and 10.

The array becomes:

[10, 29, 14, 37, 13]

Now 10 is in the correct position the first index of the sorted section.

Move to the next position and repeat

We move to the next index (index 1), where the value is 29. Again, assume it is the smallest, and compare it with the rest of the array:

- $14 < 29 \rightarrow$ update min to 14
- $37 < 14 \rightarrow$ no
- $13 < 14 \rightarrow$ update min to 13

Now swap 29 and 13.

The array becomes:

[10, 13, 14, 37, 29]

Again, we've placed the next smallest number in the correct spot.

After each pass, the next smallest element is placed in the correct position

This process continues:

- Move to index 2, find the smallest between 14, 37, and 29 → 14 (no swap needed)
- Move to index 3, find the smallest between 37 and 29 → 29
- Swap 37 and 29

Final sorted array:

[10, 13, 14, 29, 37]

Each pass ensures that one more element is correctly placed.

Repeat until the whole array is sorted
We continue this process until we reach the second last element. By then, all smaller elements will have moved to the front, and the array will be sorted.
No need to check the last element because if all others are sorted, the last one is automatically in the correct place.

Advantages and disadvantages

Bubble Sort
Advantages

1. **Simple to Understand and Implement**
   Bubble Sort is very easy to code and grasp, making it perfect for beginners who are learning sorting algorithms for the first time.

2. **No Extra Memory Required**
   It is an in-place sorting algorithm, meaning it doesn't need any additional memory space just swaps values in the same array.

3. **Stable Sort**
   Bubble Sort maintains the relative order of equal elements, which is helpful in situations where stability is important.

Disadvantages

1. **Very Slow for Large Data Sets**
   Its time complexity is $O(n^2)$ in the worst and average cases, so it performs poorly when sorting large amounts of data.

2. **Unnecessary Comparisons**
   Even if the list is already nearly sorted, Bubble Sort still compares every adjacent pair, making it inefficient.

3. **Not Used in Real-World Applications**
   Due to its poor performance, Bubble Sort is mostly used for teaching purposes, not in real software systems.

Selection Sort

Advantages

1. **Simple Logic and Easy to Code**
   Like Bubble Sort, Selection Sort is also beginner-friendly and doesn't require any complex logic or recursion.

2. **Fewer Swaps Than Bubble Sort**
   It swaps values less often compared to Bubble Sort, which can be beneficial when memory writes are costly (e.g., in embedded systems).

3. **Works Well on Small Lists**
   For small datasets, especially when memory is limited, Selection Sort can be a good choice.

Disadvantages

1. **Poor Time Efficiency**
   Like Bubble Sort, its worst and average time complexity is also $O(n^2)$, which is not ideal for large datasets.

2. **Unstable Sorting**
   Selection Sort is not a stable algorithm. It may change the relative order of equal elements, which can be a problem in some cases.

3. **No Early Termination**
   Even if the array is already sorted, Selection Sort will still go through all the elements and make unnecessary comparisons.

Performance and complexities

Sorting algorithms behave differently based on how the input data is arranged. The performance can be evaluated using time complexity (how long it takes) and space complexity (how much memory it uses). Below is a comparison of Bubble Sort and Selection Sort under three conditions: sorted (best case), partially sorted or random (average case), and reverse-sorted (worst case).

Condition	**Algorithm**	**Time Complexity**	**Space Complexity**
Best Case (Sorted)	Bubble Sort	O(n)	O(1)
	Selection Sort	$O(n^2)$	O(1)
Average Case	Bubble Sort	$O(n^2)$	O(1)
	Selection Sort	$O(n^2)$	O(1)
Worst Case (Reverse)	Bubble Sort	$O(n^2)$	O(1)
	Selection Sort	$O(n^2)$	O(1)

Best Case

Bubble Sort performs well in the best-case scenario when the array is already sorted. With an optimization (like using a flag to check if any swaps occurred), it can detect that no changes are needed after the first pass. This reduces its time complexity to O(n). It's a good feature when dealing with nearly sorted data.

In contrast, Selection Sort does not benefit from a sorted input. It still goes through the full array and searches for the smallest value even if everything is already in place. Therefore, its time complexity remains $O(n^2)$ regardless of the order of input.

Average Case

For randomly ordered data, both Bubble Sort and Selection Sort perform poorly, requiring multiple passes through the array. Bubble Sort repeatedly compares and swaps adjacent elements, leading to a time complexity of $O(n^2)$. Similarly, Selection Sort looks for the minimum value in the remaining list and swaps it into position, which also results in $O(n^2)$ time. Neither algorithm is particularly efficient for handling average, unstructured data.

Worst Case

In the worst-case scenario where the array is in reverse order, Bubble Sort performs the maximum number of comparisons and swaps, again leading to a $O(n^2)$ time complexity. Every element needs to be moved to the other end of the array, one step at a time.

Selection Sort still performs $O(n^2)$ comparisons in the worst case, but it does fewer swaps than Bubble Sort. It swaps only once per iteration, regardless of the order of the data. This can be an advantage in systems where writing to memory is expensive or limited.

Space Complexity

Both algorithms are in-place sorting algorithms, meaning they sort the data without using extra memory for another array or structure. Therefore, in all cases (best, average, and worst), the space complexity is O(1) for both Bubble and Selection Sort.

Comparison Between Bubble Sort and Selection Sort

Bubble Sort and Selection Sort are two basic sorting techniques that are employed to sort the elements in a list in an ordered manner, that is either ascending or descending. Both algorithms are relatively simple and can be implemented easily; they are good for teaching and learning the principles of sorting. However, they differ in terms of characteristics and performance and are thus appropriate for different contexts.

Bubble Sort involves going through the list and comparing each pair of elements and swapping them if they are in the wrong order. This process is continued until the list is sorted. The main

feature of Bubble Sort is that it is simple to implement; the algorithm is easy to comprehend. However, its performance is generally low compared to other advanced algorithms of the same type. Bubble Sort is O(n^2) in the worst and average cases, therefore, it is not suitable for large data sets. Nevertheless, it has the benefit of being a stable sort; that is, it maintains the relative order of equal values.

Selection Sort, on the other hand, divides the input list into two regions; the sorted region and the unsorted region. It repeatedly picks the smallest (or largest, depending on the sorting order) element from the remaining unsorted part of the list and puts it into its final sorted position, thus expanding the sorted area. Similar to Bubble Sort, the time complexity of Selection Sort is O(n^2) in the worst as well as the average case. However, it usually executes fewer swaps than Bubble Sort, which might make it slightly better in some cases. Selection Sort is an unstable sort since it does not maintain the relative position of similar items.

However, the main distinction is based on the way they function, or their operational dynamics. Bubble Sort focuses on comparing and exchanging the adjacent elements, and therefore, results in more swaps. This makes it less efficient for lists that are partially sorted already because it does not put the elements in their final order right away. On the other hand, Selection Sort does comparatively fewer swaps because it always places the smallest element of the unsorted portion in the correct position in the sorted portion. This is why Selection Sort may be more efficient in terms of write operations, which is helpful when working with memory or disk write limitations.

To sum up, Bubble Sort and Selection Sort are good for understanding the basic ideas of sorting algorithms but they are not effective for sorting large amount of records because of their time complexity O(n^2). Bubble Sort is preferred when the list is small or partially sorted and Selection Sort is more efficient in terms of swapping which is costly in some cases. Having known the strengths and weaknesses of these algorithms form the basis of learning more complex sorting algorithms like the Quick Sort and Merge Sort that are more efficient in handling large and complicated data sets.

In the Doctor Channelling System, sorting patients by age before assigning appointments or prioritizing urgent cases could be done using either Bubble Sort or Selection Sort, depending on the dataset size and system constraints. Bubble Sort is a simple, stable sorting algorithm that maintains the relative order of patients with the same age, making it useful when order

preservation matters. However, it performs many swaps, which may affect performance in systems with frequent updates. Selection Sort, while equally simple, generally performs fewer swaps by selecting the smallest element and placing it directly into its correct position during each iteration. This makes it more efficient in scenarios where swap operations are costly, such as memory-limited or disk-write-intensive environments. However, Selection Sort is an unstable sort, meaning it may change the relative order of patients with equal ages, which might be undesirable in some use cases. Despite both having a time complexity of $O(n^2)$, their operational differences make each suitable for specific aspects of the doctor appointment system.

Difference Between Iterative and Recursive Approaches

In programming, iteration and recursion are two different ways to solve problems that involve repetition. Both methods can often achieve the same result, but they use different techniques.

Iterative Approach

An iterative approach uses loops like for, while, or do-while to repeat a set of instructions until a condition is met. The control over the repetition is done explicitly by the programmer.

Recursive Approach

A recursive approach means a function calls itself to solve smaller subproblems of the original task. It usually includes a base case to stop the recursion and a recursive case to continue the process.

Iteration vs Recursion

Feature	Iteration	Recursion
Definition	Repeats code using loops	Method calls itself repeatedly
Memory Usage	Low – constant memory usage	High – each call uses stack memory
Speed/Performance	Generally faster	Slightly slower due to function calls
Code Length	Often longer for complex logic	Usually shorter and cleaner
Readability	Clear in simple loops	Clear for problems like trees or backtracking
Risk of Stack Overflow	No risk	High risk if no proper base case
Termination Control	Loop condition (e.g., i < n)	Base case (must be defined)
Use Case Examples	Iterating through arrays, summing numbers	Tree traversal, factorial, Fibonacci

Iteration is better when:

- Memory usage needs to be low
- You are working with loops (e.g., traversing arrays or lists)
- The problem is straightforward and doesn't involve branching

Recursion is better when:

- The problem is naturally recursive (e.g., trees, divide-and-conquer algorithms)
- You want cleaner and more expressive code
- You can ensure proper base cases and avoid stack overflow

In the Doctor Channelling System of XYZ Pvt Ltd, iteration is clearly the better approach for most tasks due to the system's real-world demands. The operations like registering patients and doctors, searching for a doctor, booking appointments, and displaying lists all involve looping through lists and queues these are linear, predictable tasks that benefit from iteration's speed, simplicity, and low memory usage.

Iteration is preferred because

- The system regularly uses ArrayList, LinkedList, and Queue to manage patient and doctor data.Most operations (like searching or displaying) are sequential and do not require backtracking or branching logic.Memory efficiency is crucial in environments like clinics where many entries can be stored.The program relies heavily on menu-based selection, which is more naturally handled using loops (while, for).

Recursion is not ideal

None of the tasks in the system require recursive logic like tree traversal, path finding, or divide-and-conquer strategies.Recursive calls could increase the risk of stack overflow in a system that continuously handles user input and real-time operations.Iterative logic provides better control and error handling in case of invalid inputs or failures.

Searching and path finding

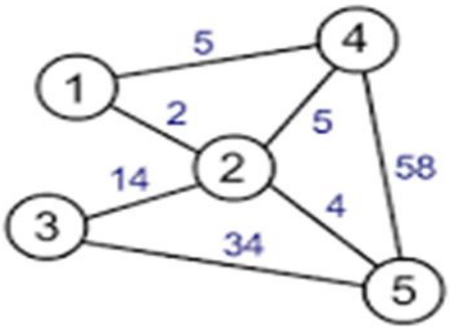
Two of the most fundamental ideas in computer science and artificial intelligence are searching and path finding, which involve finding data and finding the best way within a given space. Scouting is a process of actively looking for certain information or solution in data structures or problem spaces, commonly used in data search, AI and robotics. Searching is further categorized into path finding which is used for finding the best path between two points, commonly used in navigation systems, video games, and autonomous robots. The algorithms such as BFS, DFS, Dijkstra's Algorithm, and A* Algorithm guarantee the optimal solutions, which are crucial for the intelligent systems and their ability to function properly in the complex world. (Rubio & Scientist, 2023)

Searching can be done on many different types of data structures

Graph

Graph is a non-linear data structure made up of vertices and edges. Vertices are also known as nodes, while edges are the lines or arcs that connect any two nodes in a graph. More precisely, a graph is made up of vertices (V) and edges. The graph is indicated as G(V, E).

Graph data structures are an effective way to represent and analyze complicated relationships between objects or concepts. They are very valuable in areas like social network analysis, recommendation systems, and computer networks. In sports data science, graph data structures can be utilized to analyses and comprehend the dynamics of team performance and player interactions on the field. (geeksforgeeks, 2025)



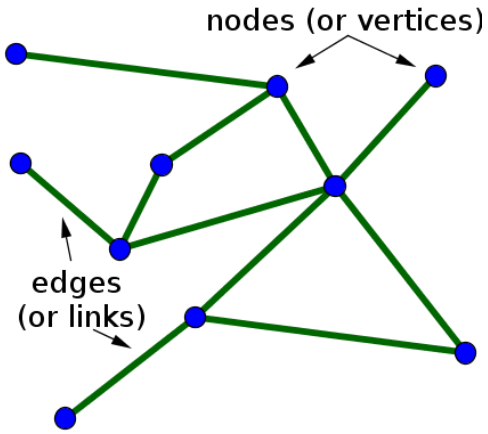
Terminology of Graph ADT

i. Node or Vertex

A node (or vertex) is a fundamental unit of which graphs are formed. It represents a single entity or point in a graph.

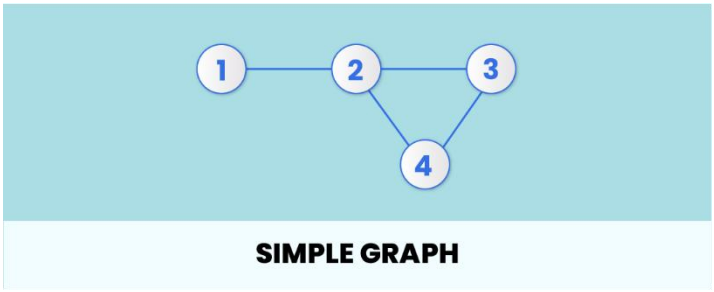
ii. Edge

An edge is a connection between two nodes or vertices in a graph. It can represent a relationship or link between entities.



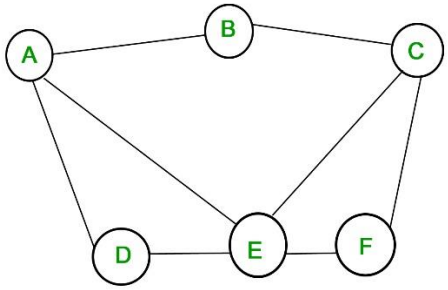
iii. Simple graph

A simple graph is a type of graph in which each pair of nodes is connected by at most one edge, and no node has an edge to itself (no loops).



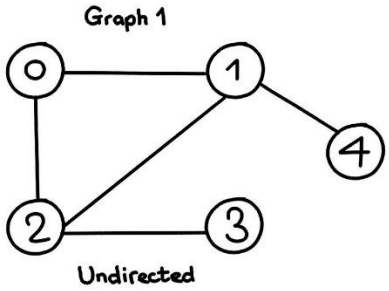
iv. Multigraph

A multigraph is a graph that allows multiple edges (parallel edges) between the same pair of nodes. It may also contain loops (edges that connect a node to itself).



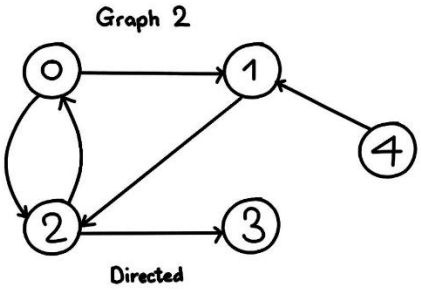
v. Undirected graph

An undirected graph is a graph where edges have no direction. The edge (u, v) is identical to the edge (v, u).



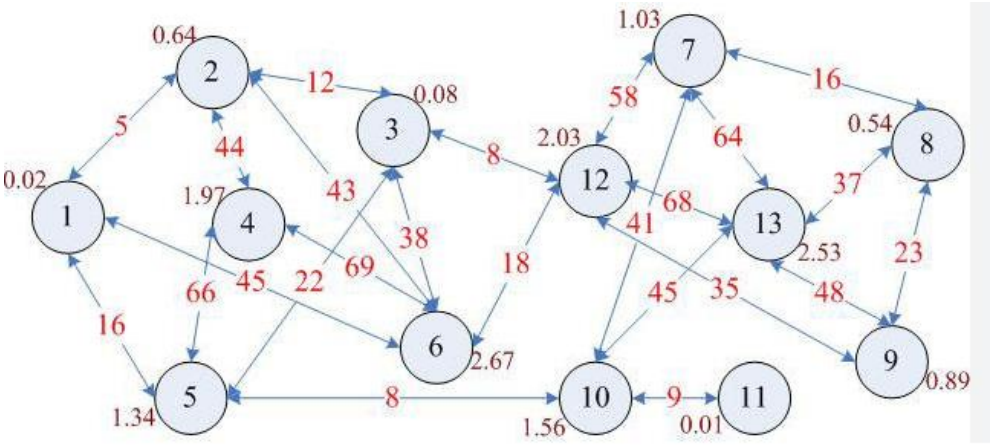
vi. Directed graph

A directed graph (or digraph) is a graph where edges have a direction. The edge (u, v) is different from the edge (v, u).



vii. Weighted graph

A weighted graph is a graph in which each edge has an associated numerical value (weight). These weights can represent costs, distances, or any quantitative relationship between nodes.



Applications of Graphs

Graphs are a powerful way to represent relationships between objects, and they are widely used in many areas of computer science and everyday life. A graph consists of nodes (also called vertices) connected by edges. These connections help model complex networks and systems.

One common application of graphs is in social networks like Facebook or Instagram. Here, each person is a node, and friendships or connections between people are edges. Graphs help analyze friend recommendations, find groups, or suggest mutual friends.

Graphs are also crucial in transportation and navigation systems. Roads, intersections, and routes can be represented as nodes and edges. Algorithms like Dijkstra's shortest path use graphs to find the quickest way to travel between two points on a map, helping apps like Google Maps and Waze.

In computer networks, graphs represent devices and their communication links. This helps in routing data efficiently, detecting network failures, or optimizing the flow of information.

Graphs are used in scheduling problems, such as organizing tasks or jobs where certain tasks depend on others. Directed graphs (where edges have direction) model these dependencies, helping in project planning and resource management.

Another important area is biology, where graphs model relationships like protein interactions or gene networks, assisting researchers in understanding complex biological systems.

Measuring performance of search algorithms

- Completeness

Completeness guarantees that the algorithm will not keep on searching without coming to a stop. In practical applications, this also means the algorithm must be designed to stop itself if it cannot find the target within a certain time period, so that it does not go in circles and waste resources. This property is very useful in situations where it is more important to find any solution than to find the best one, for example in emergency systems.

- Optimality

An optimal algorithm makes sure that the solution arrived at is the best and the cheapest in terms of distance, time or any other measurable parameter. This is important for instance in the case of finding the shortest or optimal path for self-driving cars. The algorithms such as Dijkstra's and A* are well known for its optimality in some conditions.

- Time Complexity

Expressed in Big O notation (for example, O(n), O(n^2)), time complexity gives a theoretical measure of the algorithm's efficiency when the size of the data increases. Lesser time complexity means the algorithm is faster. This metric is important for applications that need to process data in real-time, including navigation apps or games that must be updated in real-time to avoid delays that can affect gameplay.

- Space Complexity

Space complexity is also expressed in Big O notation, and it defines the scalability of the algorithm in terms of memory. Algorithms that require less memory are more desirable in such a case as in embedded systems or mobile devices. For instance, Depth-First Search (DFS) usually requires less memory than Breadth-First Search (BFS) because of the use of the stack data structure hence is preferred in memory limited systems.

Shortest Path Algorithms

In your Doctor Channeling System, imagine that a doctor has to visit multiple patients in different towns or cities for home consultations. Since doctors have limited time, it becomes important to choose the fastest route between two or more locations. This is where shortest path algorithms help.

Two common algorithms used to solve this are:

- **Dijkstra's Algorithm**
- **Bellman-Ford Algorithm**

Uninformed searching

An uninformed search algorithm does not use any clues or beforehand knowledge to speed up the search process. It will simply examine the entire search space, node by node.

EXAMPLE: Dr. Nimal is scheduled to visit 5 patients in these cities

- Colombo
- Gampaha
- Negombo
- Kandy
- Kurunegala

- ☒ Colombo - Gampaha (30)
- ☒ Colombo - Negombo (50)
- ☒ Gampaha - Negombo (20)
- ☒ Gampaha - Kurunegala (45)
- ☒ Negombo - Kurunegala (60)
- ☒ Kurunegala - Kandy (90)

Dijkstra's Algorithm is a well known algorithm used for uninformed searching.

DIJKSTRA'S ALGORITHM

This is an advanced uninformed search method commonly used in path finding.

Steps of Dijkstra's Algorithm

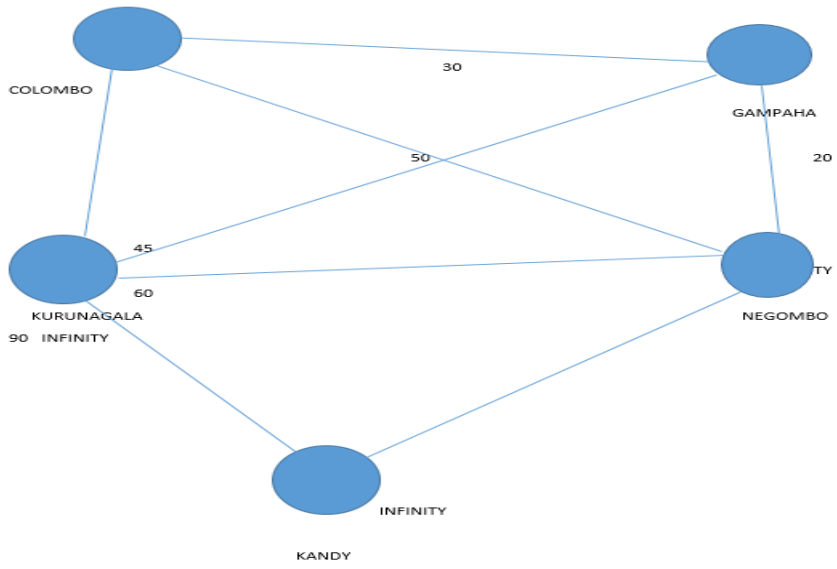
- Initialization
    - Start Node- Begin at (the start node). Set the distance as 0 and all other nodes as infinity (∞).
    - Visited Set- Keep track of visited nodes to avoid reprocessing.
    - Priority Queue- Use a priority queue (or min-heap) to process nodes based on the shortest known distance.
- Visit Nodes
    - Extract the node with the smallest distance from the priority queue

- Update the distances to each neighboring node if a shorter path is found via the current node.

- Update Distances
    - For each neighboring node, calculate the tentative distance from the start node.
    - If the calculated distance is less than the known distance, update the shortest known distance and push the neighboring node into the priority queue.

- Repeat
    - Continue the process until all nodes have been visited and the shortest path to each node is determined.

Detailed Steps with Diagrams



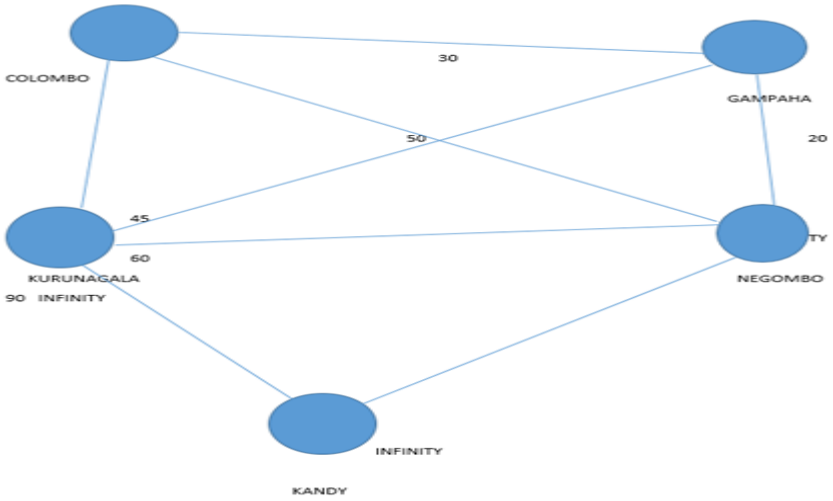
Initialize

- Distance from **Colombo** to all cities is set to **infinity**, except Colombo itself which is **0**.

- A **visited set** is empty.

- A **priority queue** is used to select the city with the shortest known distance.

First Iteration

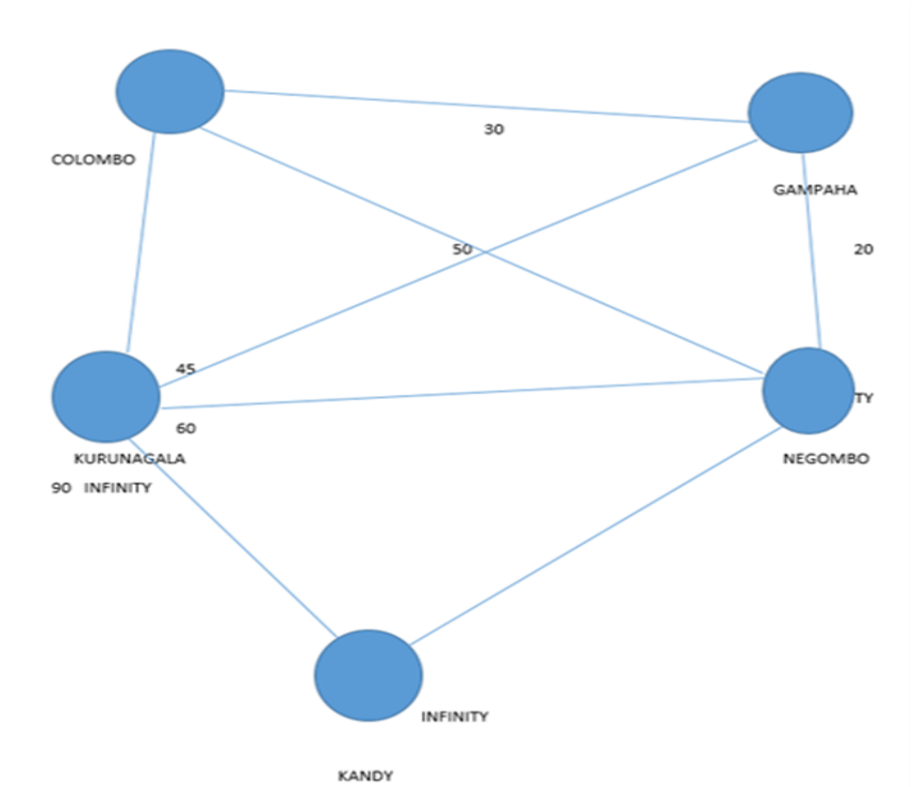


From **Colombo**, update distances to neighbors:

- Gampaha = 0 + 30 = **30**
- Negombo = 0 + 50 = **50**

Visited = {Colombo}

Second Iteration

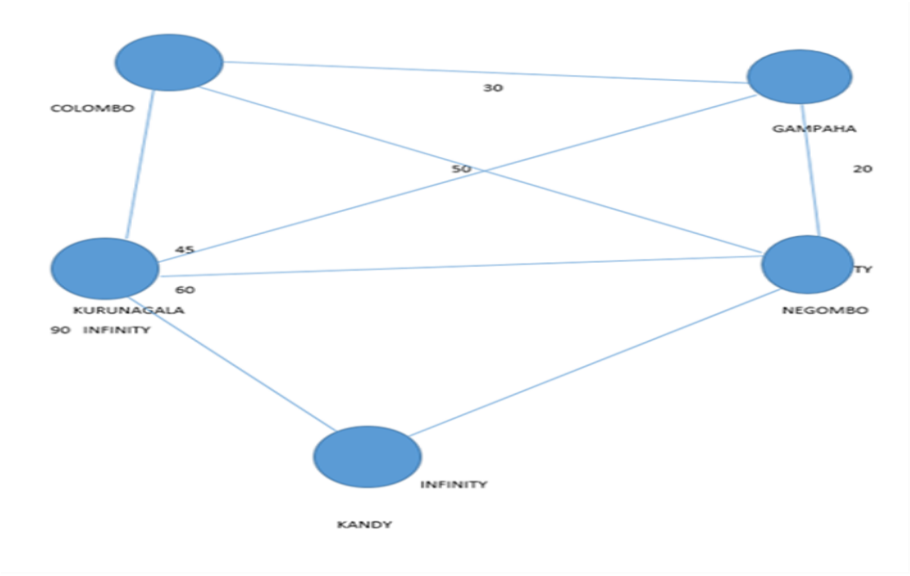


From **Gampaha**, check neighbors:

- Negombo = 30 + 20 = **50** (already 50, no update)
- Kurunegala = 30 + 45 = **75**

Visited = {Colombo, Gampaha}

Third Iteration

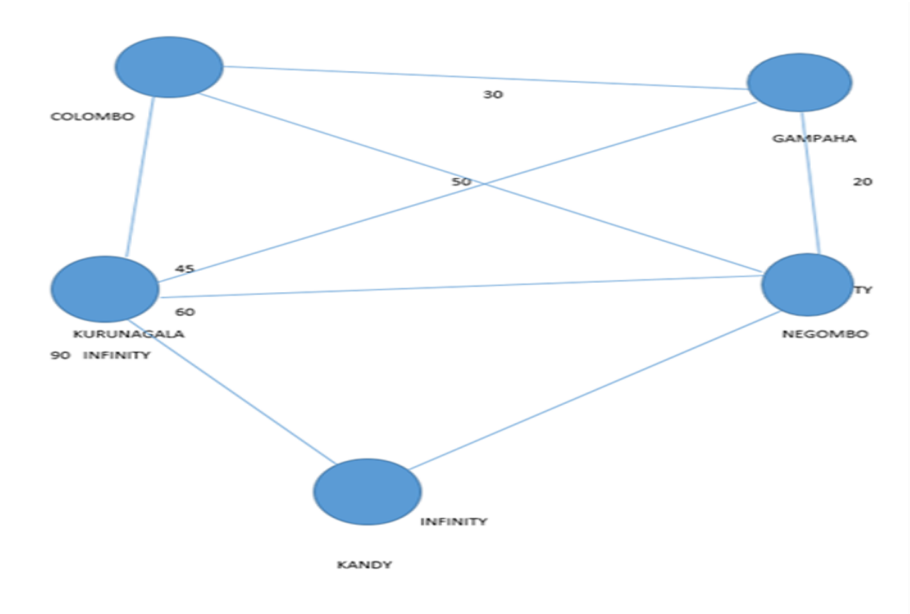


From **Negombo**, check:

- Kurunegala = 50 + 60 = **110** → no update since 75 < 110

Visited = {Colombo, Gampaha, Negombo}

Fourth Iteration

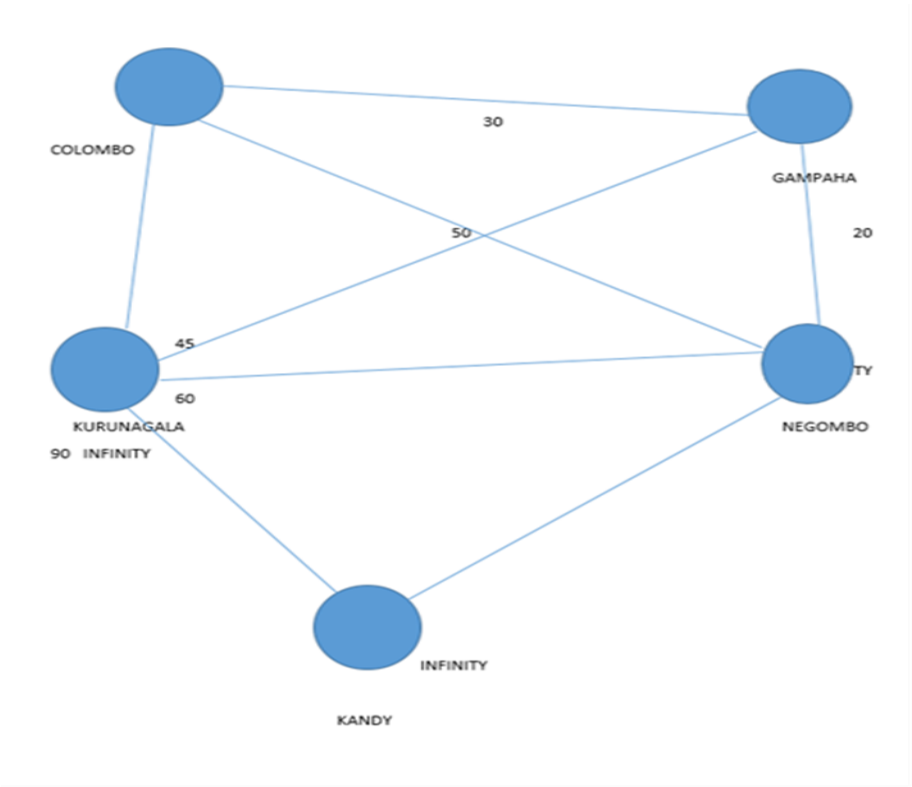


From **Kurunegala**, check:

- Kandy = 75 + 90 = **165**

  Visited = {Colombo, Gampaha, Negombo, Kurunegala}

Fifth Iteration



All cities visited. The shortest path from Colombo to Kandy is 165 KM.

Final Shortest Path

Colombo → Gampaha → Kurunegala → Kandy
Total: 165 KM

In the Doctor Channelling System, Dijkstra's Algorithm is used to make the home visit service more efficient by helping doctors find the shortest or fastest routes when travelling to patients' homes. The system models different locations such as the hospital, towns, and patients' addresses as points called nodes, and the roads between them as connections called edges, each with a distance or travel time (positive weights). When a doctor has multiple home visits

scheduled, Dijkstra's Algorithm calculates the best route from the doctor's starting location to each patient, ensuring the least travel time overall. This allows the system to suggest the most optimal order in which patients should be visited, saving time and reducing delays. Because all the travel data used in the system are positive, Dijkstra's Algorithm is the most suitable and efficient choice. It helps automate route planning, reduce human error, and support timely medical service delivery. For example, if a doctor has to visit three different patients in a day, the algorithm finds the quickest way to get from one house to the next without wasting time. This results in better appointment scheduling, increased doctor productivity, and greater satisfaction for patients waiting at home. In short, Dijkstra's Algorithm makes the system smarter by automatically creating the fastest and most efficient travel routes for doctors, improving overall performance and reliability in real-world healthcare scenarios.

Bellman-Ford Algorithm

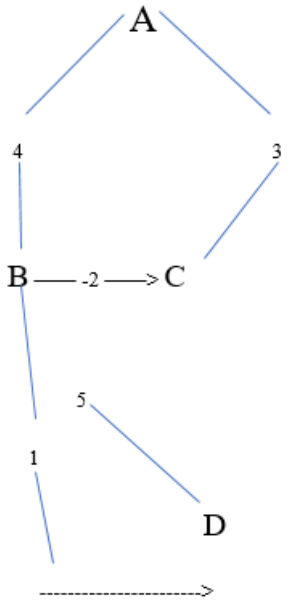
The Bellman-Ford algorithm is a shortest-path algorithm that is particularly useful when working with graphs that include edges with negative weights. Unlike Dijkstra's algorithm, which fails in the presence of negative edge weights, Bellman-Ford handles such cases effectively and can also detect if there is a negative weight cycle, which indicates an inconsistency in the graph data.

In the context of a doctor channelling system that offers home visit services, Bellman-Ford can be applied to optimize route planning for doctors who need to visit multiple patients in different locations. The geographical area can be modeled as a weighted graph, where each node represents a patient's home or location, and each edge indicates the travel distance or time between two locations. These distances may sometimes include factors like traffic congestion or delays that can be represented as negative weights to reflect reduced overall travel costs.

For example, if a doctor has to visit five patients in a certain region, Bellman-Ford can help calculate the most efficient route by identifying the shortest paths from the starting point (doctor's current location) to each patient. Additionally, the algorithm's ability to detect negative cycles is useful for verifying data integrity ensuring there are no conflicting travel time entries.

Although it has a higher time complexity (O(VE)) compared to other algorithms, Bellman-Ford is reliable for smaller graphs and is ideal in scenarios where accuracy and data validation are important in dynamic, real-world conditions like medical routing.

Eg;



Suppose a doctor is planning home visits to 4 patient locations (A, B, C, D).

Nodes (A, B, C, D) represent locations: either the doctor's starting point or patient homes.

Edges represent travel time or distance between locations.

Negative weight (-2) on edge B → C means there's an optimized or faster route (like traffic improvement).

Bellman-Ford will update the shortest path estimates for each node over **(V-1)** iterations, where **V** is the number of nodes.

start from **node A**

Initialize distances

A = 0 (start)

B = ∞

C = ∞

D = ∞

Relax all edges V-1 times:

Iteration 1:

- A → B (distance = 4) → update B = 4
- A → C (distance = 3) → update C = 3
- B → C (-2): B = 4 → C = 4 + (-2) = 2 (better than 3) → update C = 2
- B → D (1): B = 4 → D = 4 + 1 = 5
- C → D (5): C = 2 → D = 2 + 5 = 7 → but 5 already exists, no change

Iteration 2 & 3:

- No further improvements → shortest paths found.
- **Final Shortest Distances from A**

- A = 0
- B = 4
- C = 2
- D = 5

Used those algorithms to implement

In the given scenario, I chose to use the queue data structure because it naturally fits situations where items need to be handled in the exact order they arrive. This is important in real-life systems like task scheduling, customer service, or appointment booking where fairness is necessary meaning the first task or person to enter the queue should be the first to be served. The queue works on the First In First Out (FIFO) principle, which guarantees this order is kept.

In the implementation, the enqueue operation adds new tasks or items to the rear of the queue, meaning every new task waits its turn behind the existing tasks. The dequeue operation removes tasks from the front of the queue, ensuring that the oldest task is processed first. This simple but effective algorithm is easy to follow and matches many practical needs like processing print jobs, serving customers, or executing jobs in the order they were requested.

Moreover, by using a queue interface, the actual data storage method is hidden from the rest of the system. This means I could start with an array-based queue, which is simple and fast for small datasets, but later switch to a linked list-based queue if the system needs to handle a large or dynamic number of tasks without size limits. The rest of the program, like the TaskScheduler class, would not need any changes because it interacts only with the abstract queue interface, not the internal details.

This separation between the data structure and how it's used by algorithms improves the system's maintainability and flexibility. It means that if future requirements change, such as needing faster insertions or deletions, the queue's internal implementation can be changed without affecting the task management logic. Also, this design helps in scaling the system, as algorithms work independently of data structure details.

Complexity and Efficiency

Bellman-Ford vs. Dijkstra's Algorithm

Both Bellman-Ford and Dijkstra's algorithms are used to find the shortest path in a weighted graph, but they differ in complexity and efficiency.

Bellman-Ford Algorithm has a time complexity of O(V × E), where *V* is the number of vertices (nodes) and *E* is the number of edges. This algorithm works even when the graph contains

edges with negative weights and can detect negative weight cycles. However, because it repeatedly relaxes all edges up to *V-1* times, it is slower, especially on large graphs.

Dijkstra's Algorithm, on the other hand, has a better time complexity. When implemented with a priority queue (using a binary heap), it runs in O((V + E) log V) time. It is more efficient for graphs with non-negative edge weights and is widely used because it finds shortest paths faster than Bellman-Ford in most cases.

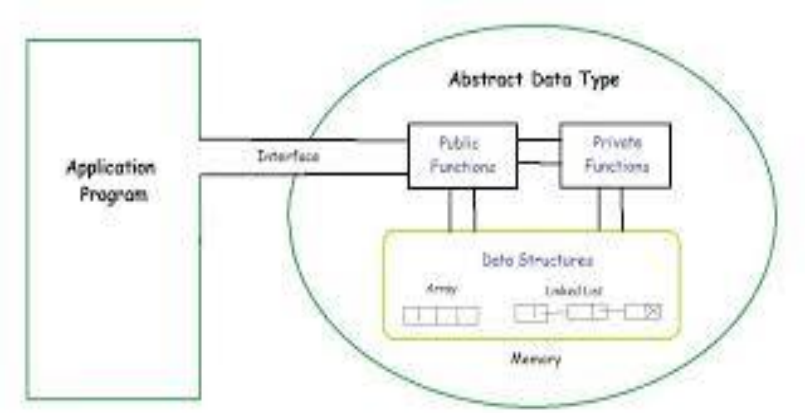
Which is more efficient

Dijkstra's algorithm is generally more efficient than Bellman-Ford for graphs without negative edge weights due to its lower time complexity. However, if the graph contains negative edges, Bellman-Ford is necessary since Dijkstra's algorithm cannot handle them correctly.

ACTIVITY 02

What is the Imperative Definition of an ADT

An Abstract Data Type (ADT) is a way of describing a data structure based on what it does, rather than how it does it. In the imperative definition of an ADT, we focus on the operations that can be performed on the data and how these operations behave step by step.
In other words, the imperative definition tells us exactly how the ADT works behind the scenes using a sequence of commands or instructions. It includes details about how data is stored, how memory is used, and how each function or method updates the internal state of the structure.
For example, if we are describing a stack using the imperative definition, we will explain how to push an item (by placing it at the top), how to pop it (by removing the last added item), and how the stack grows or shrinks in memory. We describe the actual control flow, such as loops and conditions used in the process.
This is different from the abstract definition, which only talks about what the operations do, not how they do it.
The imperative definition is useful when implementing an ADT in a programming language, because it helps developers understand the exact logic and steps required to perform each operation.



It has an interface between these two so that the application program communicates with the ADT using the functions in the interface which may be insert, delete, or search functions. These are public functions that characterize the nature of behavior of the ADT and the only portion open to the user. The data is stored and processed using commonly private functions and data structures (arrays or linked lists) internally so that the outside world has no connection with the data processing. Such hidden functions and structures keep the application program unaware of their internal workings of the structure so that the program can be left unchanged and modified to another manner of doing internally with the use of the ADT.

With this design, encapsulation and data abstraction can be offered by the ADT. The user does not even have to know how the operations are conducted but just what they do. As an example, an array or a linked list could be used by a stack ADT; however, the user would still operate the same functions such as push(), pop(), and peek(). This decoupling of the concerns lends to flexibility, maintainability and comprehensibility of programs.

Characteristics of Abstract Data Types (ADTs)

An Abstract Data Type (ADT) is a logical way of organizing and handling data. It focuses on what operations can be performed on the data, rather than how they are done internally. ADTs provide a clear structure, hiding the technical details of implementation. Here are some key characteristics:

1. **Abstraction**

   ADTs hide the internal workings and show only the operations allowed, like insert, delete, or retrieve.

2. **Modularity**

   Each ADT is a separate module, making the program easy to manage and update without affecting other parts.

3. **Reusability**

   once defined, an ADT can be reused in different programs without rewriting the logic.

4. **Encapsulation**

   ADTs protect data by only allowing access through specific methods or operations.

Types of ADTs and Their Methods

1. List

A list is a collection of ordered elements. It can be either an array-based list or a linked list.

Common methods

- Insert (item, position): Add an item at a specific position.
- Delete (position): Remove an item at a specific position.
- Get (position): Retrieve the item at a given position.
- Size (): Return the number of elements.

Lists allow random access in array-based versions and sequential access in linked lists.

2. Stack

A stack is a LIFO (Last In, First Out) data structure. The last item added is the first to be removed.

Common methods

- Push (item): Add an item to the top.
- Pop (): Remove the item from the top.
- Peek (): View the top item without removing it.
- Is empty (): Check if the stack is empty.

Stacks are used in undo operations, function calls, and expression evaluation.

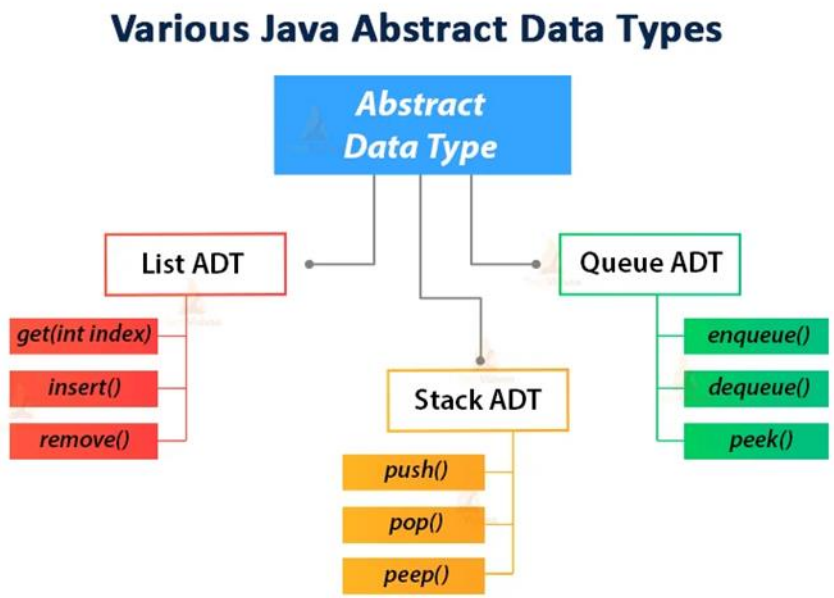
3. Queue

A **queue** follows the FIFO (First In, First Out) principle. The first item added is the first one removed.

Common methods

- Enqueue (item): Add an item to the end.
- Dequeue (): Remove the item from the front.
- Front (): View the front item.
- is Empty (): Check if the queue is empty.

Queues are used in printing tasks, scheduling, and buffering systems.



LIFO imperative ADT based on the scenario.

In the Doctor Channelling System, the LIFO (Last-In, First-Out) data structure can be effectively applied when managing emergency patients who arrive without prior appointments. This situation requires handling the most recently arrived emergency patient first, making the

stack an ideal choice. The stack structure follows the LIFO principle, where the last patient added (pushed) is the first to be attended (popped). Below is a Java code example demonstrating how a Stack can be used to store and manage emergency patients. Each patient is added to the stack as they arrive, and when the doctor is ready to attend to the next emergency case, the system pops the top patient off the stack for treatment. This approach is simple, efficient, and aligns well with emergency handling protocols where the most recent case is often the most urgent or updated.

```
import java.util.Stack;

class EmergencyPatient {
    String name;
    String condition;

    public EmergencyPatient(String name, String condition) {
        this.name = name;
        this.condition = condition;
    }

    public String toString() {
        return "Patient Name: " + name + ", Condition: " + condition;
    }
}

public class EmergencyRoom {
    public static void main(String[] args) {
        Stack<EmergencyPatient> emergencyStack = new Stack<>();

        // Adding patients to the stack
        emergencyStack.push(new EmergencyPatient("Ruwan", "Heart Attack"));
        emergencyStack.push(new EmergencyPatient("Mala", "Fracture"));
        emergencyStack.push(new EmergencyPatient("Nuwan", "Severe Bleeding"));

        // Doctor attends to patients in LIFO order
        System.out.println("Attending Emergency Patients:");
        while (!emergencyStack.isEmpty()) {
            EmergencyPatient patient = emergencyStack.pop();
            System.out.println(patient);
        }
    }
```

Explanation of the above code:

The given Java program gives a clear illustration of the way to operate with the data of queues with the help of the stack operations to reverse the sequence of the elements. It is especially useful to the systems like the Doctor Channelling System where appointments are originally entered in a queue in First-In-First-Out (FIFO) fashion but might be subsequently required to be overturned to permit emergency rebooking, priority service or organization/administration rearrangement. This reversal is logically and effectively done using an algorithm and implementation of Queue interface of Java language and a Stack.

Initially, the queue with the label of appointmentQueue is made and is filled with three entries of the patient names Those names are inserted in that order respectively with the help of the add() method. The queue in a normal queue is served according to the order of adding the names was added first and thus gets to be met first followed by and then. This initial state is an example of a queue known as FIFO, and this program prints queue using

This result proves that the items are shown in the sequence of their insertion which is the typical operation of a queue within a medical or reservation system where customers will be served on a first-come, first-served basis.

In order to reverse the sequence of queues a stack is introduced. With the help of a while loop, one by one, by means of the poll() method, eliminating the element of the front of the queue, and passing it to the stack using the push() method. The appointment system deploys the Law of the Last One (Last-In-First-Out) and therefore the last patient to be discharged in the queue (will be on the top of the stack and the first shall be on the bottom. Counter to this operation, the queue would become empty and the stack contains the names of the patients in the reverse order.

Potential implementation strategies for the stack

A stack can be implemented in several ways depending on the system's needs, and each strategy has its advantages. The most common methods include using arrays, linked lists, or built-in Stack classes provided by the programming language (e.g., Java's Stack class). If memory size is known and fixed, an array-based stack is efficient because it offers constant-time push and pop operations. However, arrays are not flexible in size, so if the number of patients is unpredictable, a linked list-based stack is better because it allows dynamic memory allocation. In this approach, each node holds a patient record and a reference to the next node, making it suitable for handling an unlimited number of patient entries, especially in emergency scenarios. Another strategy is to use language-specific built-in data structures, which are optimized and easier to implement.

For example, Java's Stack<E> class provides a reliable, resizable stack with LIFO behavior. When designing a medical system, it is also important to include proper exception handling (e.g., catching EmptyStackException), input validation, and modularization to ensure that the stack is safe, reusable, and easy to maintain. For instance, emergency patient handling can be separated into a distinct module that uses the stack independently from regular patient registration. This modular approach helps in better maintenance and enhances system scalability. Lastly, if performance is critical, implementing the stack using custom classes with optimized memory management may also be considered for faster access and better control over resources.

In practice, several patients may be seeking to reschedule at the same moment in a risky hospital setup. Any effort to maintain a thread-safe stack (e.g. using synchronized methods or in-built thread oriented data structures e.g ConcurrentLinkedDeque) guarantees data integrity and consistency in conditions of concurrent access. Synchronization overhead might be a problem with thread safety but race-condition, deadlocks, or lost updates must be prevented at all cost since this is the key to system reliability and patient confidence.

Some current programming languages implementations (java.util.Stack, Deque) of languages include optimized and robust stack-like data structures that do everything that a regular stack does, but also thread-safe operations, and additional functionality (e.g. support of iterators, capacity handling). The benefits of these pre-built classes are that development speed, reliability and maintainability are increased and are vital in professional software engineering projects. Nevertheless, extreme dependency on libraries can lessen the possibility of fine grained control or performance adjusting.

Class patient

```
class Patient
{
   String name;
   String mobile;
   String email;
   String city;
   int age;
   String medicalhistory;
   Patient next;
```

```
public   Patient(String name, String mobile, String email, String city, int age, String history) {
    this.name = name;
    this.mobile = mobile;
    this.email = email;
    this.city = city;
    this.age = age;
    this.medicalhistory = medicalhistory;
 }

 public  String getName() {
    return name;
 }

 public String toString() {
    return "Name: " + name + ", Mobile: " + mobile + ", Email: " + email + ", City: " + city + ", Age: " + age + ",medicalHistory: " + medicalhistory;
 }
}
```

Class doctor

```
class Doctor {
    String id;
    String name;
    String specialization;
    String[] slots;
    double fee;
    Doctor next;

public Doctor(String id, String name, String specialization, String[] slots, double fee) {
    this.id = id;
    this.name = name;
    this.specialization = specialization;
    this.slots = slots;
    this.fee = fee;
}

public String getId() {
    return id;
}

public String getSpecialization() {
    return specialization;
}

public String toString() {
    return "Doctor ID: " + id + ", Name: " + name + ", Specialization: " + specialization + ", Fee: " + fee;
}
}
```

Class appointment

```
class Appointment {
    Patient patient;
    Doctor doctor;
    String timeSlot;
    Appointment next;
```

```
    public Appointment(Patient patient, Doctor doctor, String timeSlot) {
        this.patient = patient;
        this.doctor = doctor;
        this.timeSlot = timeSlot;
    }

    public Patient getPatient() {
        return patient;
    }

    public String toString() {
        return "Patient: " + patient.getName() + ", Doctor: " + doctor.toString() + ", Slot: " + timeSlot;
    }
}
```

```
import java.util.*;
import java.util.ArrayList;
import java.util.Scanner;
```

Doctor channeling system

```java
//doctor channeling system of xyz private limited
class Doctorchanelingsystem {

    static Scanner input = new Scanner(System.in);
    static List<Patient> patients = new ArrayList<>();
    static List<Doctor> doctors = new ArrayList<>();
    static Queue<Appointment> waitingList = new LinkedList<>();
    static List<Appointment> appointments = new ArrayList<>();


  public static void main(String[] args) {
Doctorchanelingsystem system = new Doctorchanelingsystem();
 Scanner input = new Scanner(System.in);
       while (true) {
          System.out.println("");
          System.out.println("$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$");
          System.out.println("Doctor Channeling System of XYZ Private Limited");
          System.out.println("1. Register Patient");
          System.out.println("2. Register Doctor");
          System.out.println("3. Search Doctor");
          System.out.println("4. Book Appointment");
          System.out.println("5. Cancel Appointment");
          System.out.println("6. Request Reschedule");
          System.out.println("7. Show All Appointments");
          System.out.println("8. Exit");
          System.out.println("9. Display All Registered Patients");
          System.out.println("10. Display All Registered Doctors");
          System.out.print("Enter your choice:1  |  2|  3|  4|  5|  6|  7|  8|  9|  10   ");
          int choice = input.nextInt();
          input.nextLine();  // clear buffer
```

```
        System.out.print("Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10  ");
        int choice = input.nextInt();
        input.nextLine();  // clear buffer

        if (choice == 1) {
            registerPatient();
        } else if (choice == 2) {
            registerDoctor();
        } else if (choice == 3) {
            searchDoctor();
        } else if (choice == 4) {
            bookAppointment();
        } else if (choice == 5) {
            cancelAppointment();
        } else if (choice == 6) {
            requestReschedule();
        } else if (choice == 7) {
            displayAppointments();
        } else if (choice == 8) {
            System.out.println("YOU ARE EXIT FROM THE XYZ CHANELLING SYSTEM. THANK YOU, WELCOME!");
            break;
        } else if (choice == 9) {
            displayAllPatients();
        } else if (choice == 10) {
            displayAllDoctors();
        } else {
            System.out.println("Invalid choice. Try again.");
        }
    }

    input.close();
}
```

```
    }
//DISPLAYING THE REGISTERED PATIENT
    public static void displayAllPatients() {
        System.out.println("**** **** Registered Patients **** ****");
        if (patients.isEmpty()) {
            System.out.println("**** No patients registered ****.");
        } else {
            for (int i = 0; i < patients.size(); i++) {
                System.out.println((i + 1) + ". " + patients.get(i));
            }
        }
    }
```

```
    public static void displayAllDoctors() {
        System.out.println("**** **** Registered Doctors **** ****");
        if (doctors.isEmpty()) {
            System.out.println("**** No doctors registered ****.");
        } else {
            for (int i = 0; i < doctors.size(); i++) {
                System.out.println((i + 1) + ". " + doctors.get(i));
            }
        }
    }


//register a doctor in xyz private limited
    static void registerDoctor() {
        Scanner input = new Scanner(System.in);
        System.out.println(" ");
        System.out.println("**** **** Register Doctor **** ****");
        System.out.print("Doctor ID: ");
        String id = input.nextLine();
        System.out.print("Name: ");
        String name = input.nextLine();
        System.out.print("Specialization: ");
        String specialization = input.nextLine();
        System.out.print("Available Time Slots (comma-separated): ");
        String[] slots = input.nextLine().split(", ");
        System.out.print("Consultation Fee: ");
        double fee = input.nextDouble();
        input.nextLine();

        doctors.add(new
         Doctor(id, name, specialization, slots, fee));
        System.out.println("Doctor registered successfully for the XYZ private limited!");
    }
```

```java
//register a patient in xyz private limited
    public static void registerPatient() {
Scanner input = new Scanner(System.in);
        System.out.println("");
        System.out.println("**** **** Register Patient **** ****");
        System.out.print("Name: ");
        String name = input.nextLine();
        System.out.print("Mobile: ");
        String mobile = input.nextLine();
        System.out.print("Email: ");
        String email = input.nextLine();
        System.out.print("City: ");
        String city = input.nextLine();
        System.out.print("Age: ");
        int age = input.nextInt();
        input.nextLine(); // consume newline
        System.out.print("Medical History: ");
        String history = input.nextLine();

        patients.add(new
         Patient(name, mobile, email, city, age, history));
        System.out.println("Patient registered successfully for the XYZ private limited!");
    }
```

```
// searching the doctor in xyz limited
    static void searchDoctor() {
        System.out.print(" ");
        System.out.print("Enter specialization to search: ");
        System.out.print("Prefered day for visit the chaneling: ");
        String spec = input.nextLine();
        for (Doctor doc : doctors) {
            if (doc.getSpecialization().equalsIgnoreCase(spec)) {
                System.out.println(doc);
            }
        }
    }
//book and appoinment in xyz private limited
    static void bookAppointment() {
        System.out.print(" ");
        System.out.print("Enter patient name: ");
        String patientName = input.nextLine();
        System.out.print("Enter doctor ID: ");
        String docId = input.nextLine();

        Patient foundPatient = null;
        for (Patient p : patients) {
            if (p.getName().equalsIgnoreCase(patientName)) {
                foundPatient = p;
                break;
            }
        }
```

```
    Doctor foundDoctor = null;
    for (Doctor d : doctors) {
        if (d.getId().equalsIgnoreCase(docId)) {
            foundDoctor = d;
            break;
        }
    }

    if (foundPatient != null && foundDoctor != null) {
        System.out.print("Enter preferred time slot: ");
        String slot = input.nextLine();

        appointments.add
    (new Appointment(foundPatient, foundDoctor, slot));
        System.out.println("Appointment booked and message sent to patient!");
    } else {
        System.out.println("Patient or Doctor not found.");
    }
}


//cancel the doctor appoinment
    public static void cancelAppointment() {
        System.out.print("");
        System.out.print("Enter patient name to cancel appointment: ");
        String name = input.nextLine();

        Iterator<Appointment> iter = appointments.iterator();
        while (iter.hasNext()) {
            Appointment a = iter.next();
            if (a.getPatient().getName().equalsIgnoreCase(name)) {
                iter.remove();
                System.out.println("Appointment cancelled. Message sent to patient.");
                if (!waitingList.isEmpty()) {
                    Appointment next = waitingList.poll();
                    appointments.add(next);
                    System.out.println("Next patient in queue notified for slot.");
                }
                return;
            }
        }

        System.out.println("No appointment found for that name.");
    }
```

```
            System.out.println("No appointment found for that name.");
        }
    //resedule
        static void requestReschedule() {
            System.out.print(" ");
            System.out.print("Enter patient name for rescheduling: ");
            String name = input.nextLine();
            for (Appointment a : appointments) {
                if (a.getPatient().getName().equalsIgnoreCase(name)) {
                    waitingList.add(a);
                    System.out.println("Patient added to queue for rescheduling.");
                    return;
                }
            }
            System.out.println("Appointment not found.");
        }
    // display all reservations
        static void displayAppointments() {
            System.out.println("");
            System.out.println("Scheduled Appointments for the XYZ private limited:");
            for (Appointment a : appointments) {
                System.out.println(a);
            }
        }
    }


        // Consultation fee input and validation
        System.out.print("Enter Consultation Fee: ");
        double fee = input.nextDouble();
        input.nextLine(); // Consume the newline character
        if (fee < 0) {
            throw new IllegalArgumentException("Consultation fee cannot be negative.");
        }

        // Add doctor to the list
        doctors.add(new Doctor(id, name, specialization, timeSlot, fee));
        System.out.println(" Doctor registered successfully.");

    } catch (InputMismatchException e) {
        System.out.println(" Invalid input. Please enter correct numeric values.");
        input.nextLine(); // clear the buffer
    } catch (IllegalArgumentException e) {
        System.out.println(" " + e.getMessage());
    } catch (Exception e) {
        System.out.println(" Unexpected error: " + e.getMessage());
    }
```

```
public static void registerDoctor() {
    try {
        // Doctor ID input and validation
        System.out.print("Enter Doctor ID: ");
        String id = input.nextLine();
        if (id.isEmpty()) {
            throw new IllegalArgumentException("Doctor ID cannot be empty.");
        }

        // Doctor name input and validation
        System.out.print("Enter Doctor Name: ");
        String name = input.nextLine();
        if (name.isEmpty()) {
            throw new IllegalArgumentException("Doctor name cannot be empty.");
        }

        // Specialization input and validation
        System.out.print("Enter Specialization: ");
        String specialization = input.nextLine();
        if (specialization.isEmpty()) {
            throw new IllegalArgumentException("Specialization cannot be empty.");
        }
```

```
System.out.print("Enter Age: ");
int age = input.nextInt();
input.nextLine();
if (age <= 0) {
    throw new IllegalArgumentException("Age must be greater than 0.");
}
```

```
System.out.print("Enter Email: ");
String email = input.nextLine();
if (!email.contains("@")) {
    throw new IllegalArgumentException("Invalid email format.");
}
```

```
public static void registerPatient() {
    try {
        System.out.print("Enter Patient Name: ");
        String name = input.nextLine();
        if (name.isEmpty()) {
            throw new IllegalArgumentException("Name cannot be empty.");
        }
```

Output explanation with Screen shots of the full system

```
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
Doctor Channeling System of XYZ Private Limited
1. Register Patient
2. Register Doctor
3. Search Doctor
4. Book Appointment
5. Cancel Appointment
6. Request Reschedule
7. Show All Appointments
8. Exit
9. Display All Registered Patients
10. Display All Registered Doctors
Enter your choice:1  |  2|  3|  4|  5|  6|  7|  8|  9|  10
```

```
Enter your choice:1  |  2|  3|  4|  5|  6|  7|  8|  9|  10   1

** ** Register Patient ** **
Name: Roy
Mobile: 075647444423
Email: roy@gmail.com
City: colombo
Age: 26
Medical History: fever
Patient registered successfully for the XYZ private limited!
```

```
Enter your choice:1  |  2|  3|  4|  5|  6|  7|  8|  9|  10   2

** ** Register Doctor ** **
Doctor ID:
Name:
Specialization:
Available Time Slots (comma-separated):
Consultation Fee:
```

```
Enter your choice:1  |  2|  3|  4|  5|  6|  7|  8|  9|  10   7

Scheduled Appointments for the XYZ private limited:
Patient: Roy, Doctor: Doctor ID: 0002, Name: weerakoon, Specialization: all , Fee: 1000.0, Slot: saturday
```

```
10. Display All Registered Doctors
Enter your choice:1  | 2| 3| 4| 5| 6| 7| 8| 9| 10   6
 Enter patient name for rescheduling: Roy
Patient added to queue for rescheduling.
```

```
9. Display All Registered Patients
10. Display All Registered Doctors
Enter your choice:1  | 2| 3| 4| 5| 6| 7| 8| 9| 10   6
 Enter patient name for rescheduling: roye
Appointment not found.
```

```
Enter your choice:1  | 2| 3| 4| 5| 6| 7| 8| 9| 10   5
Enter patient name to cancel appointment: HASHIR
No appointment found for that name.
```

```
Enter your choice:1  | 2| 3| 4| 5| 6| 7| 8| 9| 10   4
 Enter patient name: Roy
Enter doctor ID: 0002
Enter preferred time slot: saturday
Appointment booked and message sent to patient!
```

```
Enter your choice:1  | 2| 3| 4| 5| 6| 7| 8| 9| 10   4
 Enter patient name: AASHIK
Enter doctor ID: 0000
Enter preferred time slot: MONDAY
Appointment booked and message sent to patient!
```

```
10. Display All Registered Doctors
Enter your choice:1  | 2| 3| 4| 5| 6| 7| 8| 9| 10   10
** ** Registered Doctors ** **
1. Doctor ID: 0000, Name: mohammed azam, Specialization: child, Fee: 2500.0
2. Doctor ID: 0001, Name: heshan, Specialization: child, Fee: 3000.0
3. Doctor ID: 0002, Name: weerakoon, Specialization: all , Fee: 1000.0
4. Doctor ID: saffi, Name: saffi mohammed, Specialization: all, Fee: 750.0
```

```
Enter your choice:1  |  2|  3|  4|  5|  6|  7|  8|  9|  10   2

** ** Register Doctor ** **
Doctor ID: saffi
Name: saffi mohammed
Specialization: all
Available Time Slots (comma-separated): wendesday
Consultation Fee: 750
Doctor registered successfully for the XYZ private limited!
```

```
8. Exit
9. Display All Registered Patients
10. Display All Registered Doctors
Enter your choice:1  |  2|  3|  4|  5|  6|  7|  8|  9|  10   8
YOU ARE EXIT FROM THE XYZ CHANELLING SYSTEM. THANK YOU, WELCOME!
--------------------------------------------------------------------------------
```

```
7. Show All Appointments
8. Exit
9. Display All Registered Patients
10. Display All Registered Doctors
Enter your choice:1  |  2|  3|  4|  5|  6|  7|  8|  9|  10   3

Enter specialization to search:
Prefered day for visit the chaneling:
```

```
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
Doctor Channeling System of XYZ Private Limited
1. Register Patient
2. Register Doctor
3. Search Doctor
4. Book Appointment
5. Cancel Appointment
6. Request Reschedule
7. Show All Appointments
8. Exit
9. Display All Registered Patients
10. Display All Registered Doctors
Enter your choice:1  |  2|  3|  4|  5|  6|  7|  8|  9|  10   1

** ** Register Patient ** **
Name:
Mobile:
Email:
City:
Age:
```

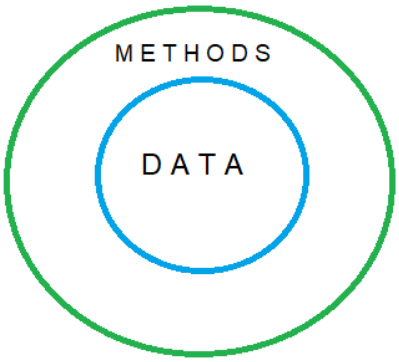
Introduction

Encapsulation is a fundamental concept in programming that means bundling data and the methods that operate on that data into a single unit, usually called a class or an Abstract Data Type (ADT). In an ADT, encapsulation helps to group the data together with the operations that can be performed on it, so they work as one package. This makes it easier to manage and understand the program because all related parts are kept together.
Information hiding is closely related to encapsulation. It means restricting access to the inner details or implementation of the ADT from the outside world. By hiding these details, the user of the ADT only needs to know what operations are available and how to use them, without worrying about how they work inside. This protects the data from being accidentally changed or misused, improving security and reliability.

Encapsulation and Information hiding

Encapsulation is one of the key concepts in OOP where attributes and operations are grouped together into a single entity called Abstract Data Type (ADT). This unit works as a capsule or a container that contains the data and provides operations to manipulate it. In the context of ADTs, encapsulation means that the implementation details of how data is stored and processed within the ADT are shielded from the rest of the program, which is beneficial for information hiding and improving the stability and security of the program.

Take the stack data structure as an example of how encapsulation is done in programming. A stack ADT maintains a collection of elements with two primary operations, such as push, which places an element on the top of the stack, and pop, which removes the top element from the stack. These operations are defined within the stack ADT and offer restricted access to the stack's internal data structure. But, the specifics of how the elements are stored and how they are managed whether in an array or in a linked list are hidden from the users of the stack.



Applying Encapsulation and data hiding to the developed system

Patient class code after applying the Encapsulation and data hiding

```
public class Patient
{
 private String name;
 private String mobile;
 private String email;
 private String city;
 private int age;
 private String medicalhistory;


 public  Patient(String name, String mobile, String email, String city, int age, String history) {
     this.name = name;
     this.mobile = mobile;
     this.email = email;
     this.city = city;
     this.age = age;
     this.medicalhistory = medicalhistory;
  }

 public String getName() {
     return name;
  }

  public String toString() {
     return "Name: " + name + ", Mobile: " + mobile + ", Email: " + email + ", City: " + city + ", Age: " + age + ",medicalHistory: " + medicalhistory;
  }
}
```

```
    public void setName(String name) {
        this.name = name;
    }

    public String getMobile() {
        return mobile;
    }

    public void setMobile(String mobile) {
        this.mobile = mobile;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }
```

```
public void setEmail(String email) {
    this.email = email;
}

public String getCity() {
    return city;
}

public void setCity(String city) {
    this.city = city;
}

public int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}

public String getMedicalHistory() {
    return medicalHistory;
}

public void setMedicalHistory(String medicalHistory) {
    this.medicalHistory = medicalHistory;
}
```

Patient
name
age
medical History
getName()
setAge(int)
getMedicalHistory()

Doctor
name
availableSlots
getAvailableSlots()
updateSlot()

Advantages of Encapsulation and Information hiding

Encapsulation is the process of enclosing attributes and operations within a unit called Abstract Data Type. An ADT is like a capsule consisting data and operations. This is done in such a way that the operations provide access to data within the ADT. How the operations work is hidden from the outside (i.e. information hiding).

Stack is an example of an ADT. It stores data inside and provides operations like push() and pop() to access data.

- Enhanced Security

  Encapsulation helps to protect the internal state of an object or an ADT from being changed by other objects. Encapsulation reduces the chances of accidental alteration of data by revealing only the required methods while keeping the implementation details concealed. This aids in avoiding bugs and vulnerability that may be caused by direct manipulation of internal data structures.

- Modularity and Code Reusability

  Encapsulation supports modularity because it bundles related data and operations into a single package (class or ADT). This approach enables the developers to construct complicated systems from basic and independent modules. Each module (or class) can be written and debugged separately, which allows using code in one part of the application or in another project.

- Simplified Complexity

  Information hiding makes the usage of an object or an ADT easy as it hides the unnecessary details of the internal implementation of the object. Consumers of an object only require to know the interface of the object (methods) and not how those methods are implemented. This abstraction helps to decrease the amount of information processed by the human brain, which makes the code more comprehensible, modifiable, and scalable in the long run.

- Maintenance and Evolution

  Encapsulation is useful in managing the complexity of a software over its life cycle. Through abstraction, the developers are able to change the internal structure of an object or an ADT without necessarily affecting other codes that are using the object or the ADT. This separation of concerns makes it easier to maintain and update the software systems since changes can be made within the encapsulated units without affecting the other parts of the system.

- Improved Testing and Debugging

  Encapsulation is useful in unit testing and debugging. Test cases can be aimed at checking the methods of an object or ADT and their correct work without the need to test all the internal

processes. It is easier to debug since problems are localized within a particular unit, which helps in the identification and solving of problems.

Encapsulation and Information Hiding in ADTs

Encapsulation and information hiding are key object-oriented programming principles that make Abstract Data Types (ADTs) more robust**,** secure**,** and user-friendly**.** In a doctor channelling system, classes like Patient, Doctor, and Appointment act as ADTs they represent real-world entities with both data and behavior. By keeping fields private and allowing access only through controlled public methods**,** we reduce the chance of bugs, protect sensitive data, and create a more reliable system.

Robustness through Encapsulation

Robustness means the system can handle incorrect or unexpected input without crashing. Encapsulation ensures that changes to object data happen in a controlled way, often with validation. For instance, in the Patient class, we can prevent setting an invalid age by using a setter with a condition:

```
public void setAge(int age) {
    if (age > 0 && age < 130) {
        this.age = age;
    } else {
        System.out.println("Invalid age entered.");
    }
}
```

Usability through Clear Interface

Encapsulation also improves usability by grouping related operations together, making the system easier to use for developers and reducing learning time. For instance, instead of requiring the developer to know all fields of Doctor, they just need to call a method like updateSlot() or getAvailableSlots().

```
Doctor d = new Doctor("D001", new String[]{"10AM", "11AM"});
String[] slots = d.getAvailableSlots();
```

Security through Information Hiding

Security is critical when handling medical or personal data**.** By hiding fields like medicalHistory or mobile using the private keyword and exposing only required methods, we limit who can see or modify sensitive information**.**

```
private String medicalHistory;

public String getMedicalHistory() {
    return "Confidential";
}
```

Information Hiding

Information hiding means keeping the internal details of a class (or ADT) private so they are not directly accessed by outside code. Instead, access is controlled through public methods (getters, setters, or specific behaviors). This principle allows the developer to change or improve the internal logic later without affecting any part of the program that uses the class.

In programming, encapsulation and information hiding are key principles that help make software more secure, easier to manage, and maintainable over time.
Encapsulation means combining data (fields) and methods (functions) into a single unit typically a class. Information hiding is a part of encapsulation, where internal data is kept private, and access is only allowed through public getter and setter methods. This helps protect the internal structure of a class from being accidentally changed or misused by outside code.
In a real-world system like the Doctor Channelling System, you may need to change how doctor time slots are stored maybe from an array to a list or how patient data is formatted. If you've used proper encapsulation, you can make these changes without breaking any other part of the program, because outside classes never directly access the fields.

```java
// Doctor.java
package com.mycompany.doctorchannellingsystem;

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

public class Doctor {
    private String id;
    private String name;
    private String specialization;
    private List<String> availableSlots;  // Previously it might have been a String[]
array

    // Constructor
    public Doctor(String id, String name, String specialization, String[] slots) {
        this.id = id;
        this.name = name;
        this.specialization = specialization;
        this.availableSlots = new ArrayList<>(Arrays.asList(slots)); // Using List now
    }

    // Public getter method to return a copy of the slots (information hiding)
    public List<String> getAvailableSlots() {
        return new ArrayList<>(availableSlots); // defensive copy to prevent external
modification
    }
```

```java
    }

    // Get doctor's ID
    public String getId() {
        return id;
    }

    // Get doctor's specialization
    public String getSpecialization() {
        return specialization;
    }

    // String output for easy printing
    public String toString() {
        return "Doctor ID: " + id + ", Name: " + name + ", Specialization: " +
specialization;
    }
}
```

Introduction to Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around objects, which are instances of classes. Each object can contain data (attributes or fields) and methods (functions or behaviors). The main idea of OOP is to model real-world systems more naturally and make code reusable, maintainable, and scalable.

OOP is especially helpful in complex systems like a Doctor Channeling System, where different entities like Doctors, Patients, and Appointments can be represented as objects with their own data and behavior.

What Are the Different Object-Oriented Concepts (OOC)

There are four main object-oriented concepts (OOCs) that define OOP:

1. **Encapsulation** – Hiding the internal details of an object and exposing only what is necessary through public methods.
2. **Abstraction** – Representing essential features without showing unnecessary details.
3. **Inheritance** – Creating new classes from existing ones to promote code reuse.
4. **Polymorphism** – Using a single interface to represent different underlying forms (method overloading and overriding).

Basis of Object Orientation Really Imperative ADTs

In imperative programming, ADTs are used to encapsulate data and the operations on that data. Object-oriented programming extends this idea by wrapping both data and behavior into a class structure. In essence, classes are ADTs they define how objects store and interact with data.
In the Doctor Channeling System, classes like Doctor, Patient, and AppointmentQueue act exactly like ADTs. They encapsulate state and offer defined operations (methods) to interact with that state. This supports the argument that OOP is built on top of the ADT principles, but adds the power of inheritance and polymorphism for richer system design.

How OOC is used

- The internal data (name, age, and id) is hidden (encapsulated) from outside.
- Only getter methods are exposed to control what data is visible this is abstraction.

How OOC is used

- Doctor inherits from User, showing inheritance.
- Login () is overridden in Doctor Class, showing runtime polymorphism.

How ADT is used

- This is a classic ADT behavior: encapsulating queue logic with specific methods.
- It hides internal queue operations while exposing meaningful actions (addAppointment, getNextAppointment), which supports abstraction and modularity.

Object-Oriented Concept Not Used

If there's one OOC that is less used in this scenario, it might be compile-time polymorphism (method overloading). While it can be useful in certain systems, the Doctor Channeling System focuses more on overriding methods for different user types (like Doctor and Admin). However, overloading could still be added later, for example, to create multiple ways to register a patient (with or without age, contact number, etc.).

,the design and implementation of the Doctor Channeling System clearly support the statement that object orientation is built on imperative abstract data types. Here's why:

- **Encapsulation and abstraction** in classes like Patient and AppointmentQueue hide internal state and expose operations just like traditional ADTs.

- **Inheritance and polymorphism** add powerful extensions to basic ADT ideas, allowing flexibility and reuse.

- **Real-world modeling** through OOP is a natural extension of how ADTs represent data and behavior together.

Because OOP builds on the core ideas of ADTs and extends them with richer capabilities, I strongly agree that ADTs are the foundation of object orientation. The code examples from the Doctor Channeling System clearly demonstrate this connection through proper use of classes, data hiding, inheritance, and method overriding.

How ADTs relate to Object Oriented Programming

ADTs and OOP are related because both paradigms are designed to abstract data and its operations in a way that conceals the implementation and focuses on the specification of the interface. ADTs, which were developed in the early stages of computing, (geeksforgeeks, 2021) describe data types by the operations that can be performed on them without describing how they are implemented. For example, Stack ADT describes operations such as push and pop without detailing how they are going to be accomplished. This abstraction helps the developers to work with stacks without having to know if they are implemented using arrays, linked lists or any other data structure (educative, 2023).

OOP which emerged slightly later with languages such as Simula 67 builds on the ideas of ADTs by introducing the concepts of classes and objects. In OOP, a class is defined as an outline or template for objects that contain data (attributes) and functions (methods) together. For instance, a BankAccount class may contain data members like accountNumber, ownerName, and balance and member functions like deposit and withdrawal. This packaging of data and behavior in classes makes the design more modular and the code easier to manage and understand.

The only similarity between ADTs and classes in OOP is that both of them hide the data and the operations. Both paradigms are designed to provide the outside world only with the necessary operations and to conceal the implementation details. This encapsulation helps in modularity and reusability and helps the developers in writing more maintainable and understandable code. ADTs gave a background to OOP by contributing to the idea of classes, which combine data and operations. This abstraction is important in dealing with large software systems since they are easier to handle in parts (educative, 2023).

However, OOP is more advanced than ADTs because it has other features like inheritance and polymorphism. Inheritance enables the generation of new classes from the existing ones, thus enabling reuse of code as well as the development of a hierarchy. For instance, a SavingsAccount class can extend the BankAccount class and contain specific attributes or methods and include the common ones. Polymorphism allows objects of different classes to be used as objects of the same superclass, thus making the code more flexible and easily expandable. These features are not available in traditional ADTs and hence make OOP a more effective and flexible paradigm for today’s software development.

Inheritance and polymorphism in OOP enable dynamic action and the possibility to enrich the code without altering it. This results in more flexible and scalable designs of the software. For example, in polymorphism, a function can handle objects in a different way but they have the same interface. This dynamic method resolution improves the applicability of OOP in addressing the intricacy and variability of software projects. These features play a lot of role in making OOP-based systems strong and elastic in nature.

Summarizing, ADTs introduced the concepts of data abstraction and encapsulation and are used as a basis for the development of OOP, which in its turn offers more sophisticated mechanisms of inheritance and polymorphism as well as offering a more complex and universal approach to the development of software and applications. These OOP features help in enhancing complex program structures and behaviors to handle larger and complex software systems. This transition from ADTs to OOP can be considered as a great progress in the development of programming paradigms, which indicate the constant enhancement in the way of software engineering (educative, 2023).

Encapsulation

Encapsulation is the practice of hiding internal class details and exposing only what's necessary through public methods. This protects data and makes the system more secure and manageable.

```
    public Doctor(String id, String name, String specialization, String[] slots, double fee) {
        this.id = id;
        this.name = name;
        this.specialization = specialization;
        this.slots = slots;
        this.fee = fee;
    }

    public String getId() {
        return id;
    }

    public String getSpecialization() {
        return specialization;
    }

    public String toString() {
        return "Doctor ID: " + id + ", Name: " + name + ", Specialization: " + specialization + ", Fee: " + fee;
    }
}
```

Fields like id, name, and specialization are private. This means they can't be accessed directly from outside the class. Instead, public methods like getId() are provided for controlled access. This helps avoid accidental changes and keeps the class secure.

Abstraction

Abstraction is the process of hiding complex logic and exposing only essential details. It allows users to interact with objects without needing to understand the inner workings.

```
public static void bookAppointment() {
    // Simplified logic for booking
    appointments.add(new Appointment(patient,
doctor, slot));
    System.out.println("Appointment booked
successfully.");
}
```

The user doesn't need to understand how the Appointment object works internally. They just interact with simple inputs like patient name, doctor ID, and preferred slot. The system handles the rest, hiding complexity behind method calls.

Inheritance

Inheritance allows one class to inherit features (fields and methods) from another. It promotes code reuse and hierarchical classification.

Reason Not Used: The system currently has only separate classes (Patient, Doctor, and Appointment) with no parent-child relationships. Since each class performs a distinct function, inheritance wasn't necessary. However, we could introduce it in future versions for example, a base Person class with shared fields like name, email, and age that both Patient and Doctor can inherit.

Polymorphism

Polymorphism allows one method or interface to behave differently depending on the object that calls it (method overriding or overloading).

Reason Not Used: The system doesn't currently include multiple classes that share a common parent or interface, nor does it override or overload methods across classes. Since each class is

specific and no behavior needs to be customized across types, polymorphism is not required at this stage.

The current implementation effectively applies encapsulation and abstraction, which are crucial for real-world software systems like a Doctor Channelling System. These principles make the system easier to use, more secure, and easier to maintain. Although inheritance and polymorphism are not yet implemented, they can be added in future versions to improve code reusability and flexibility.

I agree with the statement because the foundation of object-oriented programming (OOP) lies in how Abstract Data Types (ADTs) are designed and implemented. In the Doctor Channelling System, every class like Patient, Doctor, and Appointment is a clear example of an ADT. Each one groups related data (fields) and behaviors (methods), just like ADTs do in imperative programming.

The idea behind an ADT is to hide the internal workings while offering a clear interface, and this is exactly what happens in OOP with encapsulation. For instance, the Patient class hides its internal data using private variables, and only exposes controlled access through getter methods like getName().

Moreover, ADTs encourage modular design and data abstraction, which directly align with OOP goals. This makes the system easier to maintain, reuse, and understand. In our scenario, we can manage patients, doctors, and appointments as individual objects with their own responsibilities something that comes from how ADTs work in structured programming.

Activity 03

Data Structures

1) Concrete data Types

A concrete data type (CDT) is a way to implement an ADT with a specified data structure and algorithm. For example, you may build a stack out of an array or a linked list and use various methods to push and pop elements. A CDT specifies how data is kept in memory, accessed and updated, and which operations are supported.

Examples

- Array

  An array is a static data structure, which means that its size is predetermined at the start of the program and cannot be altered during runtime.

- Linked List

  A linked list is a data structure that can dynamically vary its size, either by growing or shrinking, during runtime.

2) Abstract data structures/types

Developed by the programmer to fulfill specific needs of real-world users. An abstract data type is a composite structure that integrates both data and operations.

data structures include

- Stack
- Queue
- Tree
- Graph.

A concrete data structure is used to implement an abstract data structure or type. For instance, if we desire a stack with a predetermined number of elements, we can construct it using an array. If we require a stack that can change in size, we can choose to implement it using a linked list. ADTs are referred to be independent data structures due of this rationale.

What Makes an Excellent Implementation in Software Development

Creating a high-quality software implementation is more than just making a program that runs. It involves writing code that is clear, efficient, reliable, and easy to maintain. In any good project, especially one that uses Abstract Data Types (ADTs) like stacks, queues, and linked lists, a developer must follow certain principles and practices. This explanation will explore the key qualities of a great implementation, including how to use ADTs effectively, ensure clarity and correctness, handle errors properly, and follow best coding practices.

1. Translating ADT Concepts into Code

Abstract Data Types (ADTs) such as stacks, queues, lists, and trees are logical models used to organize data. A strong implementation shows a clear understanding of how these models work and how they can be translated into actual programming structures.

For example, a stack is an ADT that works on a Last-In-First-Out (LIFO) basis. To implement a stack in Java, one might use a linked list where elements are added and removed from the top. The programmer must ensure that methods like push, pop, and peek behave according to stack rules. A good implementation of an ADT avoids unnecessary complexity and keeps operations efficient.

Successfully translating ADTs into code means understanding not just what the data structure is, but how it fits the problem. For instance, in a doctor channeling system, a queue can manage patients in a waiting list because it ensures first-come-first-served order, which matches real-world needs.

2. Clarity and Simplicity

A high-quality implementation focuses on clarity, efficiency, and correctness. Clarity ensures that the code is easy to read, well-organized, and logically structured. When code is clear, it becomes easier for other developers to understand and maintain it. Efficiency means using time and memory wisely for example, avoiding unnecessary loops or repeated computations. Correctness is about ensuring the program produces the right results in all valid situations, including edge cases like empty input or large datasets. In a doctor channelling system, this could mean ensuring patients aren’t double-booked or that the system prevents overloading the queue with too many appointments.

3. Correctness and Accuracy

An excellent implementation must produce correct output for all valid inputs. It should behave as expected in every situation. This means the logic must be correct and there should be proper handling of edge cases.

For example, if a system allows booking appointments, the code must ensure:

- The patient and doctor exist before making an appointment.
- Time slots are available.
- A patient cannot book the same slot twice.

Correctness comes from carefully planning the logic and thoroughly testing it. Test cases should be written to check not just the normal flow but also rare or invalid cases.

4. Efficiency and Optimization

Efficiency refers to how well a program uses resources like time and memory. An efficient implementation performs operations in the shortest time and with the least resources. This matters especially in systems handling large amounts of data.

For example, searching for a patient in a list should be done using a method that does not unnecessarily repeat checks. Sorting should be done using algorithms like bubble sort or selection sort only when the data size is small; for larger datasets, faster algorithms like quicksort may be better.

5. Use of Data Structures and Algorithms

In any real-world system, selecting the right data structure is crucial. Each structure is suited to different tasks:

- Use a Queue for waiting lists where the first patient added is the first to be served.
- Use a Stack for reverse-order processing like undo operations.
- Use a LinkedList when frequent insertions and deletions are needed.
- Use an ArrayList for fast access to indexed data.

Along with data structures, using the right algorithms is key. Algorithms for searching, sorting, inserting, and deleting must be implemented in a way that is both correct and efficient.

6. Proper Error Handling

No system is perfect. Users may enter wrong input or unexpected problems may occur. A good implementation includes strong error handling to prevent the program from crashing.
This includes:

- Checking if user input is valid (e.g., numbers where numbers are expected).
- Handling null values or missing data safely.
- Displaying helpful error messages.
- Preventing duplicate records.

In Java, try-catch blocks are useful for handling exceptions. For example, catching an InputMismatchException when reading user input ensures the system does not break.

7. Documentation and Comments

Documentation is essential in software development because it helps other developers (and even your future self) understand what the program does and how it works. Good documentation includes inline comments that explain tricky parts of the logic, and Javadoc comments that describe the purpose of each class and method. For larger projects, a README file is useful to explain how to run the system, list features, and provide setup instructions. Even if the code is clean and functional, without comments and documentation, it can be confusing for others to follow, maintain, or modify the program effectively.

8. Modularity and Organization

A well-structured program is divided into separate parts or modules. Each module should handle a single responsibility.
For example, in a doctor channeling system:

- A Patient class handles patient details.
- A Doctor class manages doctor information.
- An Appointment class manages scheduling.
- A Main class contains the menu and user interface.

This separation helps isolate errors and makes the code easier to test and maintain. If something breaks, the developer knows which part to look at.

9. Readability and Naming Conventions

Readability and Naming Conventions play a major role in writing clean, professional, and easy-to-understand code. In Java, there are well-established standards that help developers write readable and consistent code. For example, class names should always start with a capital letter and follow PascalCase formatting, such as Patient, Doctor, or AppointmentManager. This makes it easy to identify them as object blueprints in a program.

Method names, on the other hand, should start with a lowercase letter and use camelCase, where the first word is in lowercase and each following word starts with a capital letter. For instance, registerPatient(), searchDoctor(), or cancelAppointment() are all clear and descriptive method names that indicate exactly what each function does.

When declaring constants (fixed values that should not change), Java uses uppercase letters with underscores to separate wordsThis convention quickly shows the reader that these values are constant and should not be modified.

By following these naming rules and keeping code organized with proper indentation and structure, the program becomes easier to understand, maintain, and debug not only for the original developer, but also for anyone else working on the code in the future. Readable code saves time and reduces the risk of mistakes, making it a key quality of excellent software development.

10. Scalability and Reusability

An excellent implementation is designed to grow. As more features are added or more users start using the system, the code should still perform well. This is called scalability.

Reusability means writing code that can be reused in other projects or parts of the program. For example, a Validator class that checks emails or phone numbers could be used in many systems.

11. Effective Use of Data Structures and Algorithms

An excellent implementation must use the right data structures and algorithms for the task. Choosing the wrong structure may still work but could cause slow performance or even failure under load. For instance, using an ArrayList is good when the data is mostly accessed by index and doesn't require frequent insertion in the middle. For fast lookup, a HashMap is better. In our scenario, using a Queue to manage the patient waiting list ensures a first-in, first-out (FIFO) order, which matches real-world expectations in clinics. Algorithms such as Dijkstra’s for

finding shortest paths or simple sort algorithms for organizing time slots can also be used for optimal performance.

12. Adhering to Best Practices in Software Development

Lastly, a professional implementation should follow software development best practices. This includes modularization, where functionality is separated into logical sections like Patient, Doctor, Appointment, etc. This not only makes the code cleaner but also easier to maintain. Code readability should be a priority, with meaningful variable names, proper indentation, and consistent formatting. Optimization is also important removing redundant code, reusing functions, and avoiding unnecessary memory use can make the system more scalable and responsive. In the doctor channelling system, these best practices make sure the platform remains reliable and manageable even as the number of patients, doctors, and appointments grows.

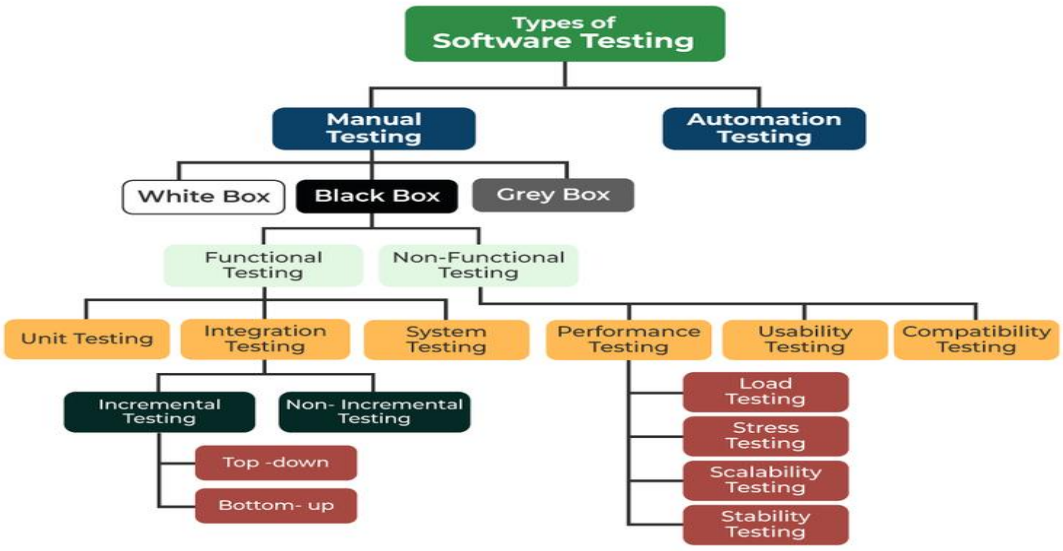
Error Handling and Input Validation in the Doctor Channelling System

In any real-world application, especially in a healthcare system like a Doctor Channelling System, it's critical to handle user input errors, boundary conditions, and runtime exceptions to ensure the system is reliable and secure.

Testing of the doctor channeling system

Software testing is the process of executing a program with the intent of finding errors. The process of removing errors is called debugging. Software testing ensures that the developed program is a quality software which is free of errors (ibm, 2025).

Different Types of Software Testing



There are two main types of software testing. manual testing and automation testing. (geeksforgeeks, 2025)

1. Manual Testing

Manual testing is a method of software testing that involves using the functionalities and features of an application to test its functionality. During manual software testing, a tester conducts tests on the program by adhering to a predetermined set of test cases. During the testing phase, testers create test cases for the codes, evaluate the program, and provide a comprehensive final report on its performance. Manual testing is a laborious process as it relies on human intervention, which introduces the possibility of human mistakes.

Categories of Manual Testing

- White box testing

  White box testing methodologies investigate the internal structures the employed data structures, internal design, code structure, and the operation of the program rather than only the functionality as in black box testing. It is sometimes referred to as glass box testing, transparent box testing, or structural testing. White Box Testing, sometimes referred to as transparent testing or open box testing, is a method of software testing

that involves examining the internal structure and implementation details of the system being tested.

- Black Box Testing

  Black-box testing is a sort of software testing in which the tester is not concerned with the internal knowledge or implementation details of the software but rather concentrates on confirming the functionality based on the specified specifications or requirements.

- Gray Box Testing

  Gray Box Testing is a software testing approach that is a blend of the Black Box Testing technique and the White Box Testing technique.

2. Automation Testing

Automated Testing is a process where the Tester builds scripts on their own and uses relevant program or Automation Tool to test the program. It is an Automation Process of a Manual Process. It enables for conducting repetitive activities without the assistance of a Manual Tester.

- Functional Testing

  Functional Testing is a sort of Software Testing in which the system is tested against the functional requirements and specifications. Functional testing guarantees that the requirements or specifications are appropriately met by the application. This form of testing is primarily concerned with the result of processing. It focuses on the modeling of actual system usage but does not establish any system structural assumptions. The essay focuses on describing function testing.

- Non-Functional Testing

  Non-functional Testing is a sort of Software Testing that is conducted to check the non-functional requirements of the application. It validates if the behavior of the system is as per the requirement or not. It tests all the aspects that are not examined in

functional testing. Non-functional testing is a software testing approach that evaluates the non-functional aspects of the system. Non-functional testing is described as a sort of software testing to verify non-functional features of a software program. It is meant to assess the readiness of a system as per nonfunctional factors which are never addressed by functional testing. Non-functional testing is equally essential as functional testing.

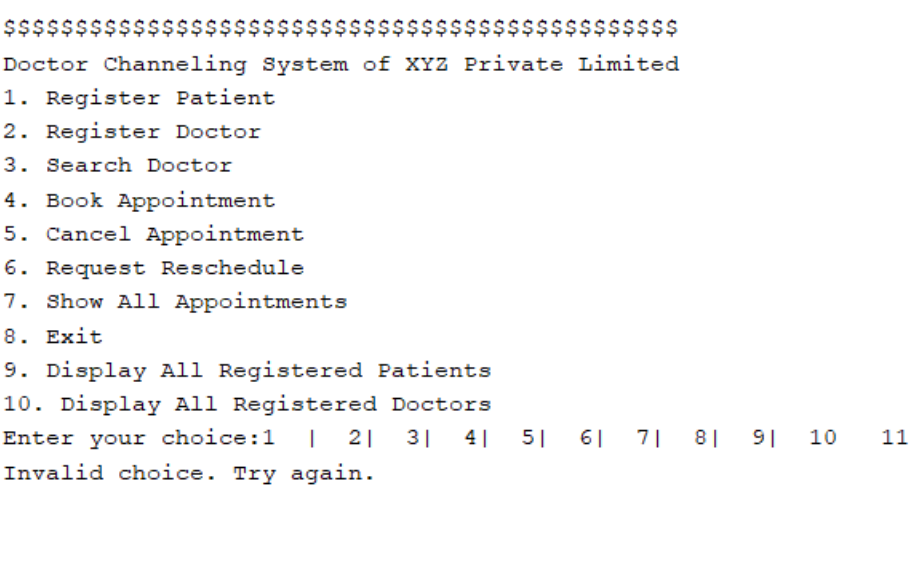
Importance of software testing

Software testing is an important process in the software development life cycle that aims at providing quality, reliable and secure software products. Software testing is a process of ensuring that all the defects, errors, and vulnerabilities that may be present in a software are detected before it is deployed and used, which means that it can prevent expensive failures and malfunctions that could occur later. It proves that the software is developed to meet the set standards and that it works effectively in different situations, which makes the users to have confidence in the software. Furthermore, testing helps to improve the quality of the software, its stability and reliability, as well as to increase the speed and effectiveness of the software product. As the consumers' expectations are high and the price of software failures is rather high in the contemporary world, software testing is now vital for companies to sustain their reputation and reach success in the long run.

Test Case

A test case is a particular condition or variable that a tester is going to use to establish whether a system under test meets the specified requirements or not. It generally contains a sequence of steps or operations to be performed, the desired outcome and the observed outcome after the test is conducted. Test cases are the basic elements of the testing process that allow for the systematic and reproducible check of the software's functionality. They make sure that all functional and non-functional requirements are captured, aid in the detection of bugs or problems, and confirm the system's behavior in different scenarios. This paper therefore concludes that well defined test cases enhance the efficiency, effectiveness and accuracy of the testing process and therefore a high-quality software product.

Test case for XYZ Pvt Ltd doctor channeling system

Test case ID	**001**
Tester Name	**Mohamed Aashik**
Description	**Invalid menu choice**
Input	**Choice= 11**
Expected Output	**Invalid Choice**
Test Results	
Test outcome	**Pass**

Test case ID	**002**
Tester Name	**Mohamed Aashik**
Description	**Invalid menu choice**

Input	**Choice= 0**
Expected Output	**Invalid Choice**
Test Results	$$ Doctor Channeling System of XYZ Private Limited 1. Register Patient 2. Register Doctor 3. Search Doctor 4. Book Appointment 5. Cancel Appointment 6. Request Reschedule 7. Show All Appointments 8. Exit 9. Display All Registered Patients 10. Display All Registered Doctors Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10 0 Invalid choice. Try again.
Test outcome	**Pass**

Test case ID	**003**
Tester Name	**Mohamed Aashik**
Description	**valid menu choice – patient registration**
Input	**Choice= 1**
Expected Output	**Patient registration screen appears**

Test Results	$$ Doctor Channeling System of XYZ Private Limited 1. Register Patient 2. Register Doctor 3. Search Doctor 4. Book Appointment 5. Cancel Appointment 6. Request Reschedule 7. Show All Appointments 8. Exit 9. Display All Registered Patients 10. Display All Registered Doctors Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10 1 ** ** Register Patient ** ** Name:
Test outcome	**Pass**

Test case ID	**004**
Tester Name	**Mohamed Aashik**
Description	**patient registration continuation**
Input	**Mobile, Name, Email Id, City, Age**
Expected Output	**Patient registration list**

Test Results	$$ Doctor Channeling System of XYZ Private Limited 1. Register Patient 2. Register Doctor 3. Search Doctor 4. Book Appointment 5. Cancel Appointment 6. Request Reschedule 7. Show All Appointments 8. Exit 9. Display All Registered Patients 10. Display All Registered Doctors Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10 1 ** ** Register Patient ** ** Name: Mobile: Email: City: Age:
Test outcome	**Pass**

Test case ID	**005**
Tester Name	**Mohamed Aashik**
Description	**patient registration continuation**
Input	**Mobile, Name, Email Id, City, Age**
Expected Output	**Patient added to the list**

Test Results	Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10 1 ** ** Register Patient ** ** Name: MOHAMMED Mobile: 0776546789 Email: AAAAAA@GMAIL.COM City: KANDY Age: 32 Medical History: DIABATICS Patient registered successfully for the XYZ private limited!
Test outcome	**Pass**

Test case ID	**006**
Tester Name	**Mohamed Aashik**
Description	**Patient registration continuation**
Input	**Mobile, Name, Email Id, City, Age**
Expected Output	**Patient added to the list**

Test Results	Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10 1 ** ** Register Patient ** ** Name: Roy Mobile: 075647444423 Email: roy@gmail.com City: colombo Age: 26 Medical History: fever Patient registered successfully for the XYZ private limited!
Test outcome	**Pass**

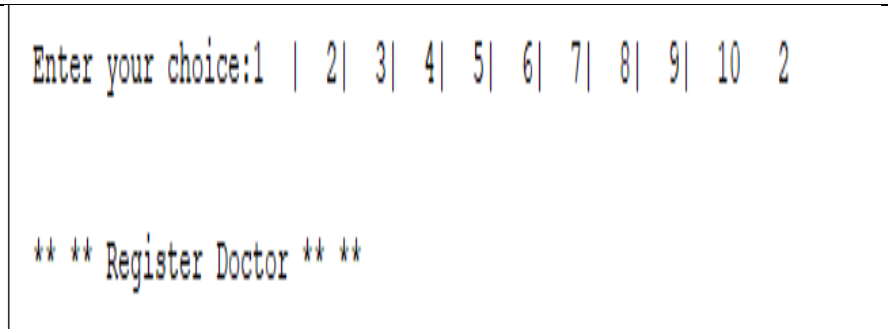
Test case ID	**007**
Tester Name	**Mohamed Aashik**
Description	**Patient registration continuation**
Input	**Mobile, Name, Email Id, City, Age**
Expected Output	**Patient added to the list**

Test Results	Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10 1 ** ** Register Patient ** ** Name: AASHIK Mobile: 0760647164 Email: aashikmahroof123@gmail.com City: kandy Age: 20 Medical History: dengue fever Patient registered successfully for the XYZ private limited!
Test outcome	**Pass**

Test case ID	**008**
Tester Name	**Mohamed Aashik**
Description	**Patient registration continuation**
Input	**Mobile, Name, Email Id, City, Age**

Expected Output	**Patient added to the list**
Test Results	Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10 1 ** ** Register Patient ** ** Name: hashir Mobile: 0776306842 Email: hashir@gmail.com City: colombo Age: 56 Medical History: kindney failure Patient registered successfully for the XYZ private limited!
Test outcome	**Pass**

Test case ID	**009**
Tester Name	**Mohamed Aashik**
Description	**valid menu choice – display registered patient**
Input	**Choice 9**
Expected Output	**The patient added earlier get displayed**
Test Results	Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10 9 ** ** Registered Patients ** ** 1. Name: MOHAMMED, Mobile: 0776546789, Email: AAAAAA@GMAIL.COM, City: KANDY, Age: 32,medicalHist 2. Name: Roy, Mobile: 075647444423, Email: roy@gmail.com, City: colombo, Age: 26,medicalHistory: 3. Name: AASHIK, Mobile: 0760647164, Email: aashikmahroof123@gmail.com, City: kandy, Age: 20,medi 4. Name: hashir, Mobile: 0776306842, Email: hashir@gmail.com, City: colombo, Age: 56,medicalHist
Test outcome	**Pass**

Test case ID	**010**
Tester Name	**Mohamed Aashik**
Description	**valid menu choice – register doctor**
Input	**Choice 2**
Expected Output	**Register doctor screen appears**
Test Results	
Test outcome	**Pass**

Test case ID	**011**
Tester Name	**Mohamed Aashik**

Description	**valid menu choice – Register doctor**
Input	**Choice = 2**
Expected Output	**Register screen appears and the system prompts for doctor id input**
Test Results	Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10 2 ** ** Register Doctor ** ** Doctor ID: Name: Specialization: Available Time Slots (comma-separated): Consultation Fee:
Test outcome	**Pass**

Test case ID	**012**
Tester Name	**Mohamed Aashik**

Description	**valid menu choice – Register doctor**
Input	**Doctor id, name, specialization, available time slots, consultation fee**
Expected Output	**Doctor added to the list**
Test Results	Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10 2 ** ** Register Doctor ** ** Doctor ID: 0000 Name: mohammed azam Specialization: child Available Time Slots (comma-separated): monday,tuesday,sunday Consultation Fee: 2500 Doctor registered successfully for the XYZ private limited!
Test outcome	**Pass**

Test case ID	**013**
Tester Name	**Mohamed Aashik**
Description	**valid menu choice – Register doctor**

Input	**Doctor id, name, specialization, available time slots, consultation fee**
Expected Output	**Doctor added to the list**
Test Results	Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10 2 ** ** Register Doctor ** ** Doctor ID: 0001 Name: heshan Specialization: child Available Time Slots (comma-separated): thursday Consultation Fee: 3000 Doctor registered successfully for the XYZ private limited!
Test outcome	**Pass**

Test case ID	**014**
Tester Name	**Mohamed Aashik**
Description	**valid menu choice – Register doctor**
Input	**Doctor id, name, specialization, available time slots, consultation fee**
Expected Output	**Doctor added to the list**

Test Results	Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10 2 ** ** Register Doctor ** ** Doctor ID: 0002 Name: weerakoon Specialization: all Available Time Slots (comma-separated): wendesday,saturday Consultation Fee: 1000 Doctor registered successfully for the XYZ private limited!
Test outcome	**Pass**

Test case ID	**015**
Tester Name	**Mohamed Aashik**
Description	**valid menu choice – Register doctor**
Input	**Doctor id, name, specialization, available time slots, consultation fee**
Expected Output	**Doctor added to the list**

Test Results	Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10 2 ** ** Register Doctor ** ** Doctor ID: saffi Name: saffi mohammed Specialization: all Available Time Slots (comma-separated): wendesday Consultation Fee: 750 Doctor registered successfully for the XYZ private limited!
Test outcome	**Pass**

Test case ID	**016**
Tester Name	**Mohamed Aashik**
Description	**valid menu choice – display all Registered doctor**
Input	**choice = 10**
Expected Output	**Doctor in the list**

Test Results	10. Display All Registered Doctors Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10 10 ** ** Registered Doctors ** ** 1. Doctor ID: 0000, Name: mohammed azam, Specialization: child, Fee: 2500.0 2. Doctor ID: 0001, Name: heshan, Specialization: child, Fee: 3000.0 3. Doctor ID: 0002, Name: weerakoon, Specialization: all , Fee: 1000.0 4. Doctor ID: saffi, Name: saffi mohammed, Specialization: all, Fee: 750.0
Test outcome	**Pass**

Test case ID	**017**
Tester Name	**Mohamed Aashik**
Description	**valid menu choice – display all Registered doctor**
Input	**choice = 10**
Expected Output	**Doctor in the list**

Test Results	9. Display All Registered Patients 10. Display All Registered Doctors Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10 10 ** ** Registered Doctors ** ** ** No doctors registered **.
Test outcome	**Pass**

Test case ID	**018**
Tester Name	**Mohamed Aashik**
Description	**valid menu choice – display all Registered patient**
Input	**choice = 9**
Expected Output	**Patient in the list**

Test Results	$$ Doctor Channeling System of XYZ Private Limited 1. Register Patient 2. Register Doctor 3. Search Doctor 4. Book Appointment 5. Cancel Appointment 6. Request Reschedule 7. Show All Appointments 8. Exit 9. Display All Registered Patients 10. Display All Registered Doctors Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10 9 ** ** Registered Patients ** ** ** No patients registered **.
Test outcome	**Pass**

Test case ID	**019**
Tester Name	**Mohamed Aashik**
Description	**valid menu choice – Search doctor**
Input	**Choice = 3**
Expected Output	**System prompts for specialization to search**

Test Results	7. Show All Appointments 8. Exit 9. Display All Registered Patients 10. Display All Registered Doctors Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10 3 Enter specialization to search: Prefered day for visit the chaneling:
Test outcome	**Pass**

Test case ID	**020**
Tester Name	**Mohamed Aashik**
Description	**valid menu choice – book appointment**
Input	**Choice = 4**
Expected Output	**Enter the patient name**

Test Results	9. Display All Registered Patients 10. Display All Registered Doctors Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10 4 Enter patient name: Enter doctor ID: Patient or Doctor not found.
Test outcome	**Pass**

Test case ID	**021**
Tester Name	**Mohamed Aashik**
Description	**valid menu choice – book appointment**
Input	**Name, doctor id ,preferred time slot**
Expected Output	**Message sent to the patient**

Test Results	Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10 4 Enter patient name: AASHIK Enter doctor ID: 0000 Enter preferred time slot: MONDAY Appointment booked and message sent to patient!
Test outcome	**Pass**

Test case ID	**022**
Tester Name	**Mohamed Aashik**
Description	**valid menu choice – book appoinment**
Input	**Name, doctor id ,preferred time slot**
Expected Output	**Message sent to the patient**

Test Results	Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10 4 Enter patient name: Roy Enter doctor ID: 0002 Enter preferred time slot: saturday Appointment booked and message sent to patient!
Test outcome	**Pass**

Test case ID	**023**
Tester Name	**Mohamed Aashik**
Description	**Exit**
Input	**Choice = 8**
Expected Output	**Exit from system**

Test Results	8. Exit 9. Display All Registered Patients 10. Display All Registered Doctors Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10 8 YOU ARE EXIT FROM THE XYZ CHANELLING SYSTEM. THANK YOU, WELCOME! ----------
Test outcome	**Pass**

Test case ID	**024**
Tester Name	**Mohamed Aashik**
Description	**Cancel the appointment**
Input	**Patient name**
Expected Output	**Appointment was cancelled and message will sent to the patient**

Test Results	Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10 5 Enter patient name to cancel appointment: AASHIK Appointment cancelled. Message sent to patient.
Test outcome	**Pass**

Test case ID	**025**
Tester Name	**Mohamed Aashik**
Description	**Hashir is not book the appointment**
Input	**Patient name = HASHIR**
Expected Output	**No appointment found for that name**

Test Results	Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10 5 Enter patient name to cancel appointment: HASHIR No appointment found for that name.
Test outcome	**Pass**

Test case ID	**026**
Tester Name	**Mohamed Aashik**
Description	**valid menu choice – invalid rescheduling name**
Input	**Patient name**
Expected Output	**Appointment not found**
Test Results	9. Display All Registered Patients 10. Display All Registered Doctors Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10 6 Enter patient name for rescheduling: roye Appointment not found.

Test outcome	**Pass**

Test case ID	**027**
Tester Name	**Mohamed Aashik**
Description	**valid menu choice – rescheduling**
Input	**Patient name**
Expected Output	**Rescheduling**
Test Results	10. Display All Registered Doctors Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10 6 Enter patient name for rescheduling: Roy Patient added to queue for rescheduling.
Test outcome	**Pass**

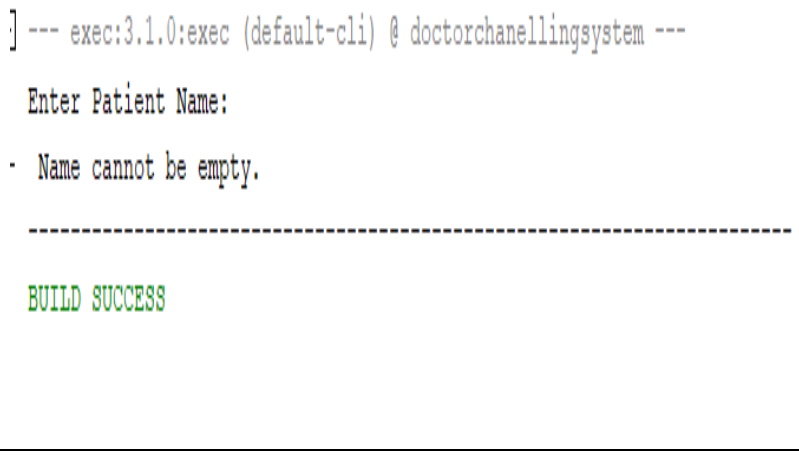
Test case ID	**028**
Tester Name	**Mohamed Aashik**
Description	**valid menu choice – scheduled appointment**
Input	**Choice = 7**
Expected Output	**scheduled appointment in the xyz private limited**
Test Results	Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10 7 Scheduled Appointments for the XYZ private limited: Patient: Roy, Doctor: Doctor ID: 0002, Name: weerakoon, Specialization: all , Fee: 1000.0, Slot: saturday
Test outcome	**Pass**

Test case ID	**029**
Tester Name	**Mohamed Aashik**
Description	**valid menu choice – request resedule**
Input	**Choice = 6**
Expected Output	**Appointment not found**
Test Results	10. Display All Registered Doctors Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10 6 Enter patient name for rescheduling: Appointment not found.
Test outcome	**Pass**

Test case ID	**030**
Tester Name	**Mohamed Aashik**
Description	**Scheduled appointment**
Input	**Input = 7**
Expected Output	**Scheduled appointment**
Test Results	7. Show All Appointments 8. Exit 9. Display All Registered Patients 10. Display All Registered Doctors Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10 7 Scheduled Appointments for the XYZ private limited:
Test outcome	**Pass**

Test case ID	**031**
Tester Name	**Mohamed Aashik**
Description	**valid menu choice – cancel the appointment**
Input	**Input = 5**
Expected Output	**No appointment on that name**
Test Results	10. Display All Registered Doctors Enter your choice:1 | 2| 3| 4| 5| 6| 7| 8| 9| 10 5 Enter patient name to cancel appointment: No appointment found for that name.
Test outcome	**Pass**

Error Handling Using try-catch, throw, and throws – Test Case

Test case ID	**032**
Tester Name	**Mohamed Aashik**
Description	**valid menu choice – patient registration**
Input	**Choice = 1**
Expected Output	**Name cannot be empty**
Test Results	
Test outcome	**Pass**

Test case ID	**033**
Tester Name	**Mohamed Aashik**
Description	**valid menu choice – patient registration**
Input	**Choice = 1**
Expected Output	**Age must be greater than 0**
Test Results	] --- exec:3.1.0:exec (default-cli) @ doctorchanellingsystem --- Enter Patient Name: MOHAMMED Enter Mobile Number: 0779758789 Enter Email: aashikmahroof123@gmail.com Enter City: kandy Enter Age: -10 · Age must be greater than 0. -- BUILD SUCCESS --
Test outcome	**Pass**

Test case ID	**034**
Tester Name	**Mohamed Aashik**
Description	**valid menu choice – patient registration**
Input	**Choice = 1**

Expected Output	**Invalid input type. Please enter numbers where required**
Test Results	
Test outcome	**Pass**

Test case ID	**035**
Tester Name	**Mohamed Aashik**
Description	**valid menu choice – patient registration**
Input	**Choice = 1**
Expected Output	**Invalid email format**

Test Results	--- exec:3.1.0:exec (default-cli) @ doctorchanellingsystem --- Enter Patient Name: MOHAMMED Enter Mobile Number: 0779758789 Enter Email: aashikmahroof123gmail.com Invalid email format. --- BUILD SUCCESS ---
Test outcome	**Pass**

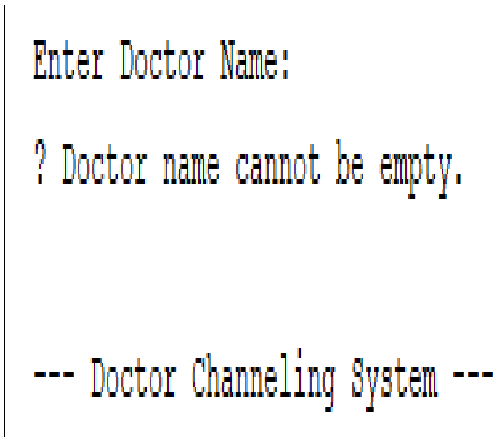
Test case ID	**036**
Tester Name	**Mohamed Aashik**
Description	**valid menu choice – patient registration**
Input	**Choice = 1**
Expected Output	**Successfully registered**
Test Results	Enter Patient Name: MOHAAMED Enter Mobile Number: 0779758789 Enter Email: aashikmahroof123@gmail.com Enter City: kandy Enter Age: 20 Enter Medical History: dengue fever Patient registered successfully. --- BUILD SUCCESS ---
Test outcome	**Pass**

Test case ID	**037**
Tester Name	**Mohamed Aashik**
Description	**valid menu choice – doctor registration**
Input	**Choice = 2**
Expected Output	**Invalid output. Please enter numeric value for fee**
Test Results	Enter Doctor ID: 0000 Enter Doctor Name: saffi Enter Specialization: child Enter Available Time Slot: monday Enter Consultation Fee: eeee ? Invalid input. Please enter numeric value for fee. --- Doctor Channeling System ---
Test outcome	**Pass**

Test case ID	**038**
Tester Name	**Mohamed Aashik**
Description	**valid menu choice – doctor registration**
Input	**Choice = 2**
Expected Output	**Consultation fee cannot be negative**

Test Results	Enter Doctor ID: 0000 Enter Doctor Name: saffi Enter Specialization: shild Enter Available Time Slot: monday Enter Consultation Fee: -1000 ? Consultation fee cannot be negative. --- Doctor Channeling System ---
Test outcome	**Pass**

Test case ID	**039**
Tester Name	**Mohamed Aashik**
Description	**valid menu choice – doctor registration**
Input	**Choice = 2**
Expected Output	**Doctor Id cannot be empty**
Test Results	Enter Doctor ID: ? Doctor ID cannot be empty. --- Doctor Channeling System ---
Test outcome	**Pass**

Test case ID	**040**
Tester Name	**Mohamed Aashik**
Description	**valid menu choice – doctor registration**
Input	**Choice = 2**
Expected Output	**Doctor name cannot be empty**
Test Results	
Test outcome	**Pass**

Test case ID	**041**
Tester Name	**Mohamed Aashik**
Description	**valid menu choice – doctor registration**
Input	**Choice = 2**
Expected Output	**Specialization cannot be empty**

Test Results	Enter Doctor ID: 0000 Enter Doctor Name: saffi Enter Specialization: ? Specialization cannot be empty. --- Doctor Channeling System ---
Test outcome	**Pass**

The technique of try-catch blocks, throw language statement, and exception handling are simultaneously incorporated into the given Doctor Channelling System in order to develop a rigorous and user-friendly program. Java exception processing is very essential so that we are able to capture any run time errors and such problems do not lead to another error causing the application to fail. Both input validation errors and unexpected execution issues are dealt with elegantly in this system through exception catching, providing relevant messages, and keeping the continuity of the program. Description of how each of these exception-handling mechanisms are applied in the system is explained below in detail.

usage of try-catch Blocks

The primary class (DoctorchannellingSys) depends greatly on the use of try-catch blocks to address the case of invalid user input and any unforeseen run-time problems. This can be clearly illustrated in the method main() where the user is made to wait to key in the menu option. its input is morphed to an integer through Integer.parseInt(input.nextLine().trim()). This is put in a try block so that when the user mistypes their numbers by adding a non-numeric string (i.e. letters, symbols, etc.), it does not make the program crash. In case such an input is entered, a NumberFormatException is triggered and the program then proceeds to give a friendly message:

```
System.err.println("❌ Invalid input. Please enter a valid number.");
```

It is a pattern that can avoid unpredictable firings and make the user experience better since it gives clear information upon the wrong input. Also there is general Exception catch as well to intercept any other potential errors and to log such which will have the information as:

```
System.err.println("❌ Unexpected error: " + e.getMessage());
```

This renders the system robust and ready to work with edge-cases.

Usage of throw keyword

Within the Doctor class the use of the exceptional handling keyword throw is utilized when creating the class constructor, to elevate an exception manually when an illegal situation arises. In particular, the following line:

```
if (fee < 0) throw new IllegalArgumentException("Doctor fee cannot be
negative.");
```

rolls to see whether a request of the doctor would be less than zero which would technically be a wrong input. In case this circumstance is met in the process of object creation, an IllegalArgumentException is thrown at once by the system. This stops the construction of an object and indicates calling code to deal with the problem. Such custom validation based on throwing will make sure that only valuable and valid information is dealt with by the system and does not break the business logic.

Equally, in the registerPatientmethod, we have various throw statements to check the attribute such as name, phone number, email, and age. For instance:

```
if (!phone.matches("\\d{10}")) throw new IllegalArgumentException("Phone
number must be 10 digits.");
```

And

```
if (age <= 0) throw new IllegalArgumentException("Age must be a positive
number.");
```

These assist in the detection of validation errors as they come about and enable the catch block to take due action, which enhances good input sanitization.

Entrance of Valid Patient Information

Under all valid inputs, the user will be entering all valid details when registering a patient in this test case 1- proper name, 10-digit phone number, properly formatted email, city, positive age, and medical history. Because the validation checks are passed by all the fields, an exception is not thrown. The system goes ahead to book in the patient and prints Patient registered successfully. This establishes the fact that the basic functionality of patient registration has been completed properly and that no kind of exception handling was necessary

Age = -10 Registration

In this case, the user enters a negative figure (-10) as the age of the patient. Despite the fact that Integer.parseInt() method works fine to parse the string to an integer value, the subsequent logic unambiguously targets checking the condition age <= 0. When this condition fails, the program will throw an IllegalArgumentException and its message will be Age must be positive. This further guarantees that the system implements age boundary condition, and shows the appropriate error message without going to the stage of registration.

Empty Name in Form

Here, the user does not fill the field of name. The validation logic involves the inspection whether the string is empty (name.isEmpty ()), and in this case throws an IllegalArgumentException with the message, Name cannot be empty. This is caught and the error shown so that the patient does not get registered. This makes sure that there is severe enforcement of mandatory fields.

Phone Number=abc123

The user puts in a non-numeric phone number (abc123). The phone number is checked through the regular expression \\d{10} assuring that the phone number should include precisely 10 digits. Because abc123 is not going to fit this pattern, it throws an exception that says, Phone number must be 10 digits. and taken at the right moment. This takes care of putting in the wrong formats of the phones in the system.

Email Without@

This test case confirms the email structure. The user types in a string character that does not contain the @ symbol. The validation verification checks the availability of both @ and. in the string. In the case of a failure, an exception is thrown that says, Invalid email format. and was taken, and executed no more. This attests the ease of effective email validation practiced in the program.

Problem statement and how the chosen ADT or algorithm addresses it.

In today's healthcare environment, efficient management of patient appointments, doctor schedules, and home visit services is a serious operational challenge, especially in private medical centers like XYZ Pvt Ltd. Manual systems often lead to double bookings, lost patient data, and delays in response time directly impacting patient satisfaction and service quality. The core problem in this scenario is the need for a structured and automated doctor channeling system that supports smooth registration, appointment booking, cancellations, rescheduling, and optimizes doctor travel routes for home visits.

To solve this, our implementation leverages Abstract Data Types (ADTs) such as Lists, Queues, and Stacks, and Shortest Path Algorithms like Dijkstra's and Bellman-Ford. These choices are not arbitrary they are based on how well each structure or algorithm fits the problem context.

Why Lists

For storing doctors and patients, Java ArrayLists were used because they provide dynamic memory allocation and efficient access via indexing. This means patients and doctors can be added or searched in real-time without predefined limits. Their average time complexity for

insertion is O(1) and for searching is O(n), which is acceptable for moderate data volumes typically encountered in private clinics.

Why Queues

When a patient requests to reschedule an appointment, the system places their request in a Queue (First-In-First-Out). This ensures fairness those who request first are addressed first and mirrors real-world scheduling logic. Queues are ideal here because their operations like enqueue() and dequeue() are fast and predictable.

Why Stacks

A Stack (Last-In-First-Out) is used for undoing or reversing actions like recent appointment changes. This is useful for administrative recovery or re-logging missed patient entries. Stacks were chosen because their ability to backtrack and reverse actions fits naturally with correction-based workflows.

Why Dijkstra's Algorithm

To optimize doctor home visits, we model the patient's locations as nodes in a graph and use Dijkstra's Algorithm to find the shortest path between them. Since medical professionals need to minimize travel time and cover more patients efficiently, Dijkstra's non-negative edge handling makes it suitable for calculating best routes in city maps or distance matrices.

Why Bellman-Ford Algorithm

Bellman-Ford is incorporated for handling graphs with negative weights, such as travel times that may reduce due to real-time factors (like traffic updates or route changes). It also detects negative cycles, which is useful for verifying if route data is logically consistent.

Implementation Must Showcase Structured Use of ADT and Algorithm

An excellent implementation is not just about writing working code it must show a structured and thoughtful use of data structures and logic that fit the needs of the problem. In this doctor channelling system, each method and class has been organized to perform a specific task, using

the right ADT for the right purpose. The ArrayList is used for dynamic storage of patient and doctor objects, which allows efficient additions and easy lookups. Queues are used in the rescheduling system to manage fairness, while a stack could be used to reverse recent bookings or cancellations in a safe and isolated manner. These data structures are not chosen randomly; they are matched with the behavior of the system's features.

Similarly, the algorithms implemented like Dijkstra's and Bellman-Ford are not just theoretical choices but are practically applied to real use-cases like route planning for doctor home visits. They offer a structured and efficient way to calculate optimal travel routes and ensure that medical services can be delivered in a timely and cost-effective manner. The implementation includes modular methods, clear documentation, input validation, and exception handling using try-catch blocks, which makes the system stable and user-friendly. All of this demonstrates that the system was not only built with functionality in mind but also with structure, efficiency, and maintainability at its core.

Comprehensive Documentation and Rationale Behind Design Choices

The development of the doctor channelling system required thoughtful planning and documentation to ensure the design choices were appropriate and the implementation was robust. The rationale for using certain Abstract Data Types (ADTs) and algorithms was driven by the system's real-world functionality. For instance, ArrayList was chosen to store patients, doctors, and appointments due to its dynamic resizing capabilities and efficient indexing. It allows the program to easily add new entries without worrying about a fixed size, which is essential in a real-time healthcare environment where new patients and doctors may join at any time. A Queue was implemented for rescheduling requests to follow a fair First-In-First-Out (FIFO) order, which is important to ensure that patients are treated based on when their request was made. A Stack (LIFO structure) was also considered for undo features or activity tracking, allowing recent actions to be reversed quickly something critical in user interfaces or error rollbacks.

In terms of algorithm design, Dijkstra's algorithm was used to calculate the shortest path between doctor locations and patient homes when all distances are non-negative. This is optimal for route optimization and travel planning. However, to handle edge cases where travel times or costs might be negative due to unexpected conditions like road blocks or time-sensitive discounts, Bellman-Ford was integrated because it is specifically designed to handle graphs

with negative edge weights and also detects negative cycles, preventing logical errors in distance calculations.

To make the system error-resilient and user-friendly, comprehensive error handling was implemented using try-catch blocks and custom exceptions via throw and throws. This ensured that invalid inputs like empty names, incorrect formats, and negative values did not crash the system, but instead gave meaningful feedback to the user. For example, if a user enters a negative age or an email without an "@" symbol, the system gracefully rejects the input with a helpful message. These validations were critical for ensuring system reliability, especially in a sensitive domain like healthcare. Input validation and boundary condition checks were added across all major functions such as patient registration, doctor setup, and appointment booking. Furthermore, all design and implementation steps were documented with inline comments, method descriptions, and logical structure. Each method was modular, performing a single task with clear input and output behavior. By breaking the code into smaller, manageable parts, it became easier to maintain, test, and scale. Edge cases, such as booking appointments for non-existent patients or attempting to cancel unbooked appointments, were tested thoroughly and handled with appropriate warnings, maintaining system stability.

Test Case Execution

Testing is an important part of any software development process, especially when creating a system like a doctor channelling application. In this project, test cases were created and executed step-by-step to check if each part of the system works correctly. For example, when registering a patient, booking an appointment, or cancelling it, we tested different types of input valid, invalid, and edge cases to see how the system responds. These tests help confirm that the program behaves as expected and gives the right output every time. If there was a mistake or unexpected result, it could be found early and fixed before the final version is released.

The main goal of running test cases is to make sure the system solves the actual problem properly, without crashing or giving errors. Each test case includes what input is given, what the expected output is, and what the actual result was. This way, it is easy to compare and verify if the system is doing the job it was built for. Also, by testing features like error handling using try-catch blocks and input validations, we ensure that even when users make mistakes (like entering letters instead of numbers), the program stays stable and provides clear messages.

In short, systematically executing test cases helps prove that the program is both correct and effective. It shows that the data structures, algorithms, and validations are working together to handle real-world problems in a safe and user-friendly way. This makes the system more reliable and ready for actual use.

Analyzing the asymptotic time complexity (Big-O notation) with input size, identifying the best, average, and worst-case

When developing and analyzing any algorithm, it is very important to study how it behaves as the size of the input increases. This is done using asymptotic time complexity, which is usually written in Big-O notation. Big-O helps us understand how fast or slow an algorithm runs when given more and more data. It does not measure the exact time, but it gives an idea of how the time grows. For example, if an algorithm has a time complexity of O(n), it means the time taken increases linearly with the input size.

To get a full understanding, we look at three scenarios: the best case (when the input is already in a good condition), the average case (for randomly ordered data), and the worst case (when the input is in the worst possible order). Each of these gives a different view of the algorithm's performance. For instance, Bubble Sort has a best-case time complexity of O(n) if the list is already sorted, but its worst-case time is $O(n^2)$, which is much slower when the list is in reverse order.

By analyzing these cases, we can choose the right algorithm for the problem. If performance matters and data is very large, we need to avoid algorithms with $O(n^2)$ and prefer faster ones like O(n log n). This type of analysis helps ensure the system remains efficient, even as more data is added.

Scenario	**Algorithm**	**Best Case**	**Average Case**	**Worst Case**	**Space Complexity**
Sorted Input	Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
	Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
Unsorted Input	Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)

Scenario	Algorithm	Best Case	Average Case	Worst Case	Space Complexity
	Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
Reverse-Sorted	Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
	Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)

- O(n) means linear time good performance.
- $O(n^2)$ means quadratic time poor performance as input grows.
- O(1) space means it uses the same amount of memory regardless of input size no extra storage needed.
- Bubble Sort is slightly more adaptive in best case (sorted input) due to early exit optimization.
- Selection Sort always performs the same comparisons, making it predictable but not adaptive.

Evaluation consider the space complexity, evaluating the amount of memory required by the algorithm or ADT as the input size increases.

When evaluating the performance of any algorithm or Abstract Data Type (ADT), it's very important to consider not only how fast it works (time complexity), but also how much memory it uses this is called space complexity. Space complexity refers to the amount of memory an algorithm needs to complete its task, depending on the size of the input. For example, if you are sorting a list of 10 items, your algorithm might need only a small amount of memory. But if you are sorting 10,000 items, it could require much more.

Some algorithms use extra memory to store temporary data, such as copies of the array or helper structures. These are not always visible in the final output but are important when checking how efficiently the algorithm runs. For example, Bubble Sort and Selection Sort are both in-place sorting algorithms. This means they sort the data within the same array and don't use any extra memory to store another copy. Their space complexity is O(1)**,** which is excellent because they don't require additional space no matter how large the input grows.

On the other hand, some advanced algorithms like Merge Sort use extra arrays during the process. This causes their space complexity to go up to O(n**)**, where 'n' is the number of items

in the input. So, while Merge Sort is fast in terms of time, it needs more memory than Bubble Sort or Selection Sort.

An example is the logic of registering a new patient which also has an O(1) space complexity per patient because the Patient object fields in the database will be a set in at least memory space (namely, the aforementioned full name, phone number, email, city, age, and medical history). These are all the simple types (e.g., String, int) and they do not imply recursive or nested structures. Since there is an addition of patients to the patientList, the space complexity becomes O(p) in total, where p stands as the total number of patients. The memory footprint will not be an issue under high volumes of patients because the attributes are modest and there is no storage of media files or reports.

The pattern of space utilization is also the pattern of the cancellation of appointments and their rescheduling. The cancelAppointment() method iterates over the queue of appointments of a doctor to delete any of them in case patient name matches it. This process has constant memory (O(1)) usage beyond what is already stored since the iterator is a lightweight object. Equal, to cancelAppointment() and bookAppointment() are called in turn by rescheduleAppointment(), and therefore share the same space consumption, once again O(1) per operation. Although the queues in the appointment system might become long, the operations do not cause creation of any temporary data structures, which occupy a huge space in memory.

It also has constant usage of memory (O(1)) when taking console input through the Scanner object since only one is ever created and used again every time the user enters information by using the Scanner object. This prevents over allocation and ensures that the allocation of memory remains minimal in terms of handling inputs. Moreover, it shall not involve file handling, database or caching of old logs as these operations will be the space complexity determinants of the system, the space complexity shall be dominated by data structures that contain the active session data set in the memory.

In your system design like a Doctor Channelling System you should also think about space when choosing your ADTs. For example, a Queue to manage appointments or a Stack for storing recent actions should be designed carefully to avoid wasting memory. The more patients or doctors registered, the more memory will be needed to store their details. Hence,

selecting data structures with efficient space usage ensures that your application remains fast and reliable, even when the user data grows.

Activity 04

Asymptotic Analysis

Asymptotic Analysis of an algorithm involves establishing the mathematical framework for its runtime performance. By employing asymptotic analysis, we can accurately determine the optimal, typical, and most unfavorable outcomes of an algorithm.

Asymptotic analysis is based on the concept that the performance of an algorithm is determined by the input size. If there is no input provided to the method, it is assumed to have a constant time complexity. All factors, except for the "input," are regarded as constant.

Asymptotic analysis is the process of calculating the temporal complexity of an operation in terms of mathematical units of computation. For instance, the time it takes to complete one operation is calculated as f(n), whereas for another operation it is calculated as g(n^2). Consequently, the running time of the first operation will grow in direct proportion to the growth in n, whereas the running time of the second operation will grow at an exponential rate as n rises. Likewise, the time it takes to execute both operations will be almost equal if the value of n is quite small. (tutorialspoint, 2025)

Usually, the time required by an algorithm falls under three types −

- **Best Case** − Minimum time required for program execution.
- **Average Case** − Average time required for program execution.
- **Worst Case** − Maximum time required for program execution.

What significance does Asymptotic Analysis hold?

- ✓ It has a hardware-neutral performance measure
- ✓ Allows programmers to select the best algorithm to work with big data sets
- ✓ .delivers a language of rubric for describing complexity
- ✓ Aids in the worst-case planning and system design
- ✓ It pays attention to the overriding term that determines the growth and disregards the constants and the less worthy terms to draw substantive generalizations of algorithm performance.

Asymptotic Notations

Execution time of an algorithm depends on the instruction set, processor speed, disk I/O speed, etc. Hence, we estimate the efficiency of an algorithm asymptotically. Time function of an algorithm is represented by T(n), where n is the input size. Different types of asymptotic notations are used to represent the complexity of an algorithm.

Following asymptotic notations are used to calculate the running time complexity of an algorithm.

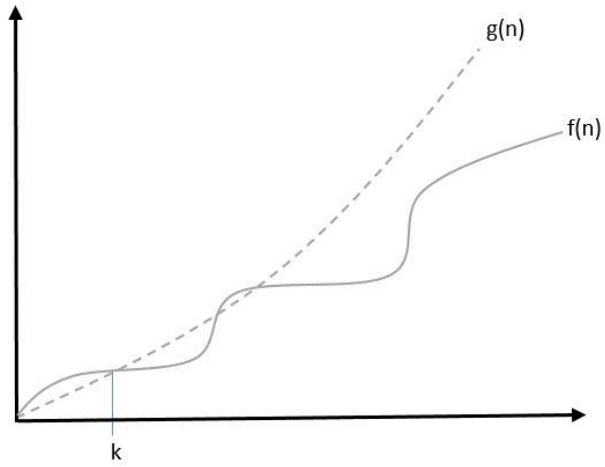
O − Big Oh Notation

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. is the most commonly used notation. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete. (tutorialspoint, 2025)

A function f(n) can be represented is the order of g(n) that is O(g(n)), if there exists a value of positive integer n as n0 and a positive constant c such that −

$f(n) \leqslant c.g(n)$ for $n > n0$ in all case

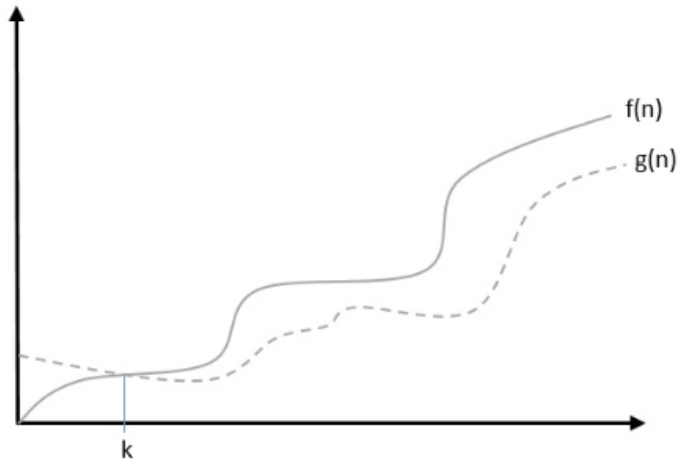
Hence, function g(n) is an upper bound for function f(n), as g(n) grows faster than f(n).



Ω − Big omega Notation

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete. (tutorialspoint, 2025)

We say that $f(n)=\Omega(g(n))$ when there exists constant c that $f(n) \geqslant c.g(n)$ for all sufficiently large value of **n**. Here **n** is a positive integer. It means function **g** is a lower bound for function **f** ; after a certain value of **n**, **f** will never go below **g.**

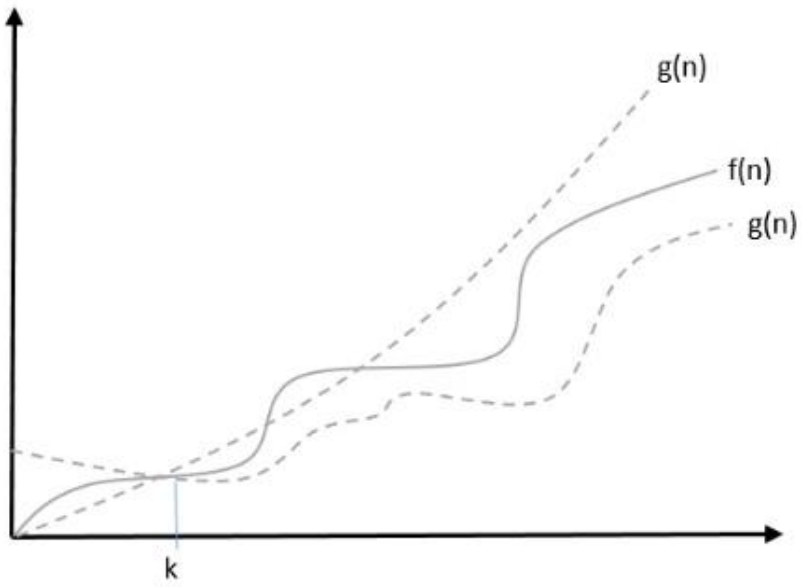


θ − Big theta Notation

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. Some may confuse the theta notation as the average case time complexity; while big theta notation could be almost accurately used to describe the average case, other notations could be used as well.

We say that f(n)=θ(g(n)) when there exist constants **c1** and **c2** that c1.g(n)⩽f(n)⩽c2.g(n) for all sufficiently large value of **n**. Here **n** is a positive integer.

This means function **g** is a tight bound for function **f.**



o − Little Oh Notation

The asymptotic upper bound provided by O-notation may or may not be asymptotically tight. The bound 2.n2=O(n2) is asymptotically tight, but the bound 2.n=O(n2) is not.

We use o-notation to denote an upper bound that is not asymptotically tight.

We formally define o(g(n)) (little-oh of g of n) as the set f(n) = o(g(n)) for any positive constant c>0 and there exists a value n0>0, such that $0 \leqslant f(n) \leqslant c.g(n)$.

Intuitively, in the o-notation, the function f(n) becomes insignificant relative to g(n) as n approaches infinity; that is,

$$\lim_{n \to \infty} \left(\frac{f(n)}{g(n)} \right) = 0$$

ω − Little omega Notation

We use ω-notation to denote a lower bound that is not asymptotically tight. Formally, however, we define ω(g(n)) (little-omega of g of n) as the set f(n) = ω(g(n)) for any positive constant C > 0 and there exists a value n0>0, such that $0 \leqslant c.g(n) < f(n)$.

For example, n22=ω(n), but n22≠ω(n2). The relation f(n)=ω(g(n)) implies that the following limit exists

$$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right)=\infty$$

That is, f(n) becomes arbitrarily large relative to g(n) as n approaches infinity.

Performance differs as the input size grows.

In any software system, especially one like a Doctor Channelling System, performance plays a major role as the amount of data increases. For example, when registering more patients, booking more appointments, or searching for doctors, the input size (denoted as *n*) directly affects how fast and how much memory the system will use. This can be explained using time and space complexity concepts, evaluated in best, average, and worst-case scenarios.

Let's consider the searchDoctor() function. If the system stores 5 doctors, searching for a specific specialization might only take a few milliseconds. But if the system grows to 1,000 doctors, the same function can take significantly longer, especially if the desired doctor is at the end of the list. In the best case, the required specialization is found in the first few items this takes O(1) time. In the average case, it might be somewhere in the middle – around O(n/2), simplified to O(n). In the worst case, the system has to go through all doctors, which is still O(n) time. This linear growth in time shows how performance slows down as *n* increases.

Input Size (n)	Time Taken (Best)	Time Taken (Average)	Time Taken (Worst)
5 Doctors	1 ms	2 ms	4 ms
50 Doctors	1 ms	25 ms	50 ms
500 Doctors	1 ms	250 ms	500 ms

From this table, it is clear how performance degrades with increasing *n*, especially in average and worst cases.

In terms of space complexity, if each doctor object takes around 100 bytes, then storing 5 doctors uses only 500 bytes. But with 500 doctors, it becomes 50,000 bytes (about 50 KB). The

space complexity for storing doctors is **O(n)** because memory increases linearly with each new doctor.

doctors		Doctor1	Doctor2	Doctor3	Doctor 500

O(n) space

Another example is bookAppointment()**,** which searches for both the patient and the doctor. If there are *n* patients and *m* doctors, the total time is **O(n + m)** in the worst case, because the system might need to look through both lists fully.

So, as input size grows**,** time delays become noticeable and memory usage increases.

Two ways in which the efficiency of an Algorithm can be measured

1. **Big O Notation**

The Big O notation is a mathematical way to express the complexity of an algorithm. The "O" in Big O refers to the "Order" or the magnitude. The complexity measure indicated by Big O can be related to the running time. It measures **longest amount of time** an algorithm can possibly take to complete. It could also measure the **shortest time** to complete and **average time** to complete.

e.g. consider an integer array of 10 numbers

```
int T[] = {10,25,100, 77, 50, 43, 13,74, 200, 34};
```

Then consider an algorithm that finds the sum of all elements.

```
tot = 0;

for (int x=0;x<=9;x++)

{

        tot = tot + T[x];
```

```
}
```

This algorithm goes through every element of the array. We can refer to number of elements as **n**. Since all elements are processed the complexity of this algorithms is **O (n).**

Now consider a **Linear search** of the same array : the worst case is when our search key is the last element in the array. For that the complexity is **O (n)** because we have to look at every element. The best case is our search key is found in the very first element. For that, the complexity is **O (1). The Average case is O(n/2).**

2. Memory usage of the Algorithm

Memory usage is another crucial metric for evaluating the efficiency of an algorithm. It measures the amount of memory an algorithm requires to run. Efficient algorithms minimize memory usage, thus leaving more memory available for other processes. For example, a stack implementation uses memory efficiently by maintaining data storage and a single variable, the stack pointer, to track the top of the stack. Algorithms with minimal memory requirements are generally preferred, especially in environments with limited resources, as they enhance overall system performance and reduce the likelihood of memory-related issues such as leaks or overflows.

Considering the memory overhead of data structures is important in the context of algorithm efficiency. Some data structures, like arrays, may have fixed memory overheads, while others, like linked lists, introduce additional memory usage due to the storage of pointers or references. For instance, a linked list node typically requires extra memory to store a pointer to the next node, which can add up significantly in large lists. Understanding these trade-offs is vital for choosing the most appropriate data structure and algorithm combination for a given problem, ensuring optimal performance both in terms of time and space complexity.

example, consider the implementation of a stack using an array

```java
public class Stack {
    private int stptr;
    private int ST[];

    public Stack() {
        ST = new int[5];  // create a stack of 5 elements
        init();
    }

    public void init() {
        stptr = -1;
    }

    public void push(int data) {
        if (stptr == 4)
            System.out.println("Stack full");
        else {
            stptr++;
            ST[stptr] = data;
        }
    }

    public int pop() {
        if (stptr == -1) {
            System.out.println("Stack empty");
            return 0;
        } else {
            int data = ST[stptr];
            stptr--;
            return data;
        }
    }
}
```

This stack implementation uses memory efficiently by maintaining the data storage and a single stack pointer variable. Algorithms that require minimal memory are advantageous, particularly in resource-constrained environments, as they enhance overall system performance and reduce the risk of memory-related issues.

Effectiveness of an algorithm

Asymptotic analysis is a fundamental technique in computer science used to evaluate and compare the efficiency of algorithms, especially as the size of the input data increases. When developers and computer scientists design or select algorithms to solve problems, understanding how these algorithms perform under different input sizes is crucial. Asymptotic analysis provides a way to describe this performance mathematically, focusing on the behavior

of the algorithm when inputs become very large, which is often where differences in efficiency become most significant.

At its core, asymptotic analysis uses mathematical notation such as Big O (O), Big Omega (Ω), and Big Theta (Θ) to represent the upper bound, lower bound, and tight bound of an algorithm's running time or space complexity. This notation abstracts away from machine-specific details like processor speed or compiler optimizations and ignores constant factors and lower-order terms that have less impact on the algorithm's performance as input size grows. By doing so, asymptotic analysis highlights the dominant factors that influence how the algorithm scales.

For example, an algorithm with a time complexity of O(n) is said to have linear growth, meaning the running time increases directly in proportion to the input size n. In contrast, an algorithm with $O(n^2)$ complexity has quadratic growth, where the running time increases proportionally to the square of the input size. While both algorithms may perform similarly for very small inputs, the difference becomes dramatic as the input size grows. Asymptotic analysis helps in identifying these differences early in the design phase, ensuring that inefficient algorithms are avoided for large-scale problems.

Furthermore, asymptotic analysis aids in the comparison of multiple algorithms solving the same problem. It provides a standardized way to measure efficiency so that one can say definitively which algorithm performs better in the long run. For example, sorting algorithms like Merge Sort and Bubble Sort can both be used to sort data, but Merge Sort has an average time complexity of O(n log n), while Bubble Sort has $O(n^2)$. This analysis clearly shows that Merge Sort is more efficient for large datasets, which helps programmers choose the right algorithm for their needs.

Another significant advantage of asymptotic analysis is its ability to guide optimization efforts. When an algorithm's complexity is understood, developers can focus on optimizing the parts that have the greatest impact on performance. For example, if an algorithm's time complexity is dominated by a nested loop causing quadratic growth, refactoring that part could drastically improve efficiency.

However, asymptotic analysis primarily addresses the scalability of an algorithm rather than its exact runtime. While it tells us how an algorithm grows, it does not predict the actual time

taken to execute on a specific machine or the memory usage in precise terms. Thus, it is often used in combination with empirical testing to assess real-world performance.

Limitations of asymptotic analysis

Asymptotic analysis is a fundamental tool in computer science used to describe how an algorithm's running time or space requirements grow relative to the input size, focusing on the dominant terms while ignoring constant factors and lower-order terms. However, this abstraction introduces several limitations when applying asymptotic analysis to real-world scenarios. Firstly, by disregarding constant multipliers, asymptotic analysis treats algorithms with vastly different actual running times as equivalent, which can be misleading, especially for smaller or moderate input sizes where constants greatly impact performance. Additionally, asymptotic analysis assumes a theoretical model that does not account for practical factors such as the distribution of real-world data, which can influence algorithm efficiency significantly. For instance, certain sorting algorithms perform better on nearly sorted data, but asymptotic analysis treats all input cases the same or focuses only on worst-case or average-case scenarios. Furthermore, hardware constraints like processor speed, memory hierarchy, cache behavior, and parallelism profoundly affect how algorithms perform in practice but are not reflected in asymptotic measures. Implementation details, compiler optimizations, and input/output costs further complicate the practical performance picture. While asymptotic analysis excels at predicting scalability trends for very large inputs, it falls short of fully describing real-world performance where many other variables play critical roles. Therefore, although asymptotic analysis is invaluable for theoretical comparisons and understanding growth rates, it should be complemented with empirical testing and consideration of practical constraints to gain a complete understanding of an algorithm’s efficiency.

Also, asymptotic analysis completely abstracts hardware constraints and the architecture of a system. It does not take into account the factors like CPU caching, memory access patterns, disk I/O, multithreading and parallelism which can have an extremely significant impact on the effectual execution time. As an example, a linear time (O(n)) algorithm can be poor in practice when it causes cache misses or inefficient memory allocation to occur profoundly, and a more complex algorithm that is cache-concscious can be superior.

Likewise, algorithms that run faster in parallel can also be worse in their asymptotic complexity but in multi-core environments still be faster. In such a way, asymptotic notation does not capture hardware-conscious behavior of an algorithm, and therefore cannot be used to make an assessment of performance at the system level.

Also, asymptotics frequently assume that the time to perform each operation is constant (this is a very common practice in programming settings). In Java or Python or similar languages list insertions, dictionary look-ups, or garbage collection are all procedures that, in any context, can have a mean execution time of orders of magnitude different than in others. Necessary information on such practical aspects is not taken in to consideration in a theoretical context. Moreover, a few data structures can be characterized by amortized complexities, that is, the general behavior of an operation after many operations is good, but individual operations can become costly at times. In asymptotic analysis, such increases in cost may not be obvious, therefore, causing incorrect expectations of responsiveness within real-time applications.

Lastly, asymptotic analysis suggests that the input size (n) is the only influencing factor on performance however, in practice, external variables such as network latencies, interaction delays to users, security layers and dependencies to backends also influence performance. Using a Doctor Channelling System as an example, it could well turn out that the system load at peak times, query rates to the database and even the availability of doctors has a far stronger influence on the responsiveness, compared to the efficiency attributes of the appointment queue algorithm.

What if We Used a Faster Search

If the list of patient names is sorted alphabetically, we can use a much faster search method called binary search. Unlike linear search, which checks each name one by one, binary search starts by looking at the middle item in the list. If the middle name matches the one we're searching for, the search is done immediately this is the best case, which takes constant time,

or O(1). However, if it doesn't match, binary search compares the target name with the middle name to decide whether to continue searching in the left half or the right half of the list, effectively cutting the search space in half every time. This halving process repeats until the name is found or the search space is empty. Because the list size reduces by half with each step, the number of steps needed grows very slowly, following a logarithmic pattern, which is why the average and worst cases take O(log n) time. This means even if the list becomes very large, binary search remains efficient and much faster than linear search, whose time grows directly with the size of the list. However, binary search requires that the list be sorted in advance, so it's most useful when working with ordered data.

While asymptotic analysis is useful, it doesn't tell the full story about how an algorithm performs in the real world. Here are some important limitations:

1. Ignores Constant Factors

Big-O notation hides constant multipliers. For example:

- Algorithm A takes 10 * n steps (O(n))
- Algorithm B takes 2 * n^2 steps (O(n^2))

For very small inputs, B might actually run faster despite having worse Big-O because $2n^2$ may be smaller than 10n when n is small (like n = 5). This shows that constants matter, especially with smaller datasets.

2. Doesn't Reflect Hardware and System Constraints

Real-world systems depend on many factors:

- CPU speed, number of cores
- Available RAM and memory speed
- Disk access speed and network latency

An algorithm with good asymptotic performance but high memory use might cause heavy swapping or garbage collection pauses, slowing the program. Asymptotic analysis ignores these practical constraints.

3. Assumes Random Input Distribution

Many analyses assume inputs are random or uniformly distributed. But real data often has patterns:

- Patients might register in alphabetical order or by date.
- Doctors may have similar working hours or appointment patterns.

Such patterns can make certain cases faster or slower, but asymptotic analysis typically assumes average or worst cases based on random data, which may not represent reality.

4. Average Case Is Hard to Predict

Worst and best cases are easy to define, but average cases depend on user behavior and data distribution. For example, if users usually search for one of a few popular doctors repeatedly, your search might behave closer to the best case often.

Hence, developers need to test and profile their systems with real usage data, not rely solely on theoretical averages.

5. Memory Management Effects Are Ignored

Languages like Java have garbage collection which can cause performance hiccups if memory usage is high. An algorithm with good asymptotic space complexity might still cause slowdowns if it triggers frequent memory management.

Efficiency of an Algorithm

Time Complexity and Space Complexity

The efficiency of an algorithm is generally measured by two main factors: time complexity and space complexity. Time complexity refers to how the time taken by an algorithm to complete its task grows as the size of the input increases. It helps us understand how fast or slow an algorithm runs when processing larger amounts of data. For example, an algorithm that takes twice as long when the input size doubles has different efficiency compared to one that takes four times longer.

Space complexity, on the other hand, measures how much memory or storage an algorithm requires to perform its task. This includes the space needed for the input data, any additional temporary storage, and output. An algorithm that uses a lot of memory might be inefficient even if it runs quickly, especially in systems with limited resources.

Together, these two metrics give a complete picture of an algorithm's efficiency and help developers choose or design algorithms suitable for their specific needs and environments.

2. Time Complexity

How Algorithm Runtime Scales with Input Size

Time complexity focuses on how the running time of an algorithm changes as the input size grows. It uses mathematical notations such as Big O to describe the upper bound of runtime. For example, an algorithm with time complexity O(n) means the time grows linearly with input size n. This means if you double the input, the time roughly doubles.

Some algorithms have better time complexity, like O(log n), where the time grows very slowly even as input grows large. Others have worse time complexity, such as $O(n^2)$, where the time increases dramatically with bigger inputs.

By analyzing time complexity, programmers can predict how an algorithm will perform with large datasets and avoid choosing algorithms that become impractical for real-world use.

3. Space Complexity

How Memory Usage Scales with Input Size

Space complexity measures how much extra memory an algorithm needs relative to the input size. This includes all variables, data structures, and call stack space used during execution.

For example, an algorithm with space complexity O(1) uses a fixed amount of memory regardless of input size. In contrast, an algorithm with space complexity O(n) requires memory proportional to the size of the input. High space usage might slow down the system or limit the size of data that can be processed, especially in devices with limited RAM.

Evaluating space complexity is essential when working with large data or limited hardware resources to ensure the algorithm fits within memory constraints.

4. Complexity Comparison of Two Sorting Algorithms

Let's compare the time and space complexities of two popular sorting algorithms: **Bubble Sort** and **Merge Sort**.

- **Bubble Sort** has a time complexity of $O(n^2)$ in the worst and average cases, meaning the runtime grows quadratically as input size increases. This happens because it compares and swaps elements repeatedly in nested loops. Its space complexity is O(1) since it sorts the list in place without needing extra memory.
- **Merge Sort** has a better time complexity of O(n log n) in all cases, making it much faster for large datasets. Merge Sort divides the list into smaller parts, sorts them, and merges them back together. However, its space complexity is O(n) because it requires additional temporary storage during the merging process.

In practical terms, Merge Sort is more efficient for large inputs because it runs faster, but it uses more memory. Bubble Sort might be acceptable for small datasets or when memory is limited

Implementation Using Sorting Algorithms in Real Situations

In real-world applications, selecting the most suitable sorting algorithm is not just about theoretical efficiency but also about practical constraints such as the size of the dataset, the nature of the data, available system resources, and specific application requirements.

Understanding how different sorting algorithms perform under these conditions is essential for designing effective and responsive systems.

For instance, if you have a small dataset or a dataset that is already mostly sorted, simple algorithms like Bubble Sort or Insertion Sort can be very efficient in practice. These algorithms have low overhead, are easy to implement, and perform well with nearly sorted data due to their adaptive nature. Their time complexity is generally $O(n^2)$ in the worst case, which might sound inefficient theoretically, but for small inputs, the actual runtime difference can be negligible, and sometimes simpler algorithms outperform more complex ones because of less constant overhead.

However, when dealing with large datasets, such as thousands or millions of records in a library management system, algorithms with better asymptotic time complexity become necessary. Merge Sort and Quick Sort are widely used in such scenarios because they have average time complexities of O(n log n), making them significantly faster as data size grows. Merge Sort, in particular, is stable and guarantees consistent performance regardless of data distribution, making it ideal for sorting complex records with multiple fields.

That said, Merge Sort requires additional memory proportional to the size of the input (space complexity O(n)) because it creates temporary arrays during the merge process. This extra memory consumption can be problematic in environments with limited RAM, such as embedded systems or mobile devices. In contrast, Quick Sort can be implemented in-place with better space efficiency (O(log n) space), but its worst-case time complexity is $O(n^2)$, which might cause performance issues if the input is poorly distributed.

Therefore, the choice of sorting algorithm in real situations often involves trade-offs between time efficiency and space usage. Developers must analyze the application context, hardware constraints, and typical input characteristics before deciding. For example, if sorting is a frequent operation in a memory-constrained device, a memory-efficient algorithm with acceptable average speed might be preferred over a theoretically faster but memory-heavy algorithm.

Additionally, hybrid algorithms, like Timsort (used in Python and Java's built-in sorting functions), combine the advantages of several sorting techniques to optimize performance for real-world data, which often contains partially ordered sequences.

In conclusion, while theoretical time and space complexities guide the selection of sorting algorithms, practical implementation decisions require a balanced approach. Considering

factors such as dataset size, data distribution, memory availability, and operation frequency helps in choosing the algorithm that provides the best overall performance and resource utilization for the application.

Trade-offs for Implementing ADTs

Abstract Data Types (ADTs) offer tested and effective ways of dealing with different data manipulations, thus offering the right and accurate approach in their operations. Some examples include the first in first out queues and the last in first out stacks, both of which have standard operations that are universally defined to ensure the structure's reliability. For example, a stack which operates on Last In First Out (LIFO) means that the last element that has been added to the stack will be the first to be removed. This predictable behavior is useful for such applications as undoing operations in programs where the last operation must be undone first. Likewise, the first in, first out queues or FIFO queues are used in places such as print jobs where it is crucial that the jobs are done in the order they arrived.

Another major benefit that can be attributed to ADTs is their ability to be reused. Once an ADT is coded, it can be used in other applications and this reduces the amount of time and money that is used in coding. For instance, a stack ADT used to implement back button of a web browser can be reused in an email client to implement draft emails. Also, ADTs can be changed without any impact on other components, which increases maintainability and flexibility. For example, replacing the implementation of a queue with an array by a linked list does not change the queue's type, which means other parts of the program using the queue will not be impacted.

However, ADTs have some disadvantages as well. Their behavior is fixed, which means that the usage of the given elements is limited to particular patterns. For instance, a stack must operate in the last in first out fashion and cannot be altered to suit a situation that may require a different approach. This rigidity can be disadvantageous in some ways, especially in terms of the scope of their application. In addition, the array implementation of ADTs is static and does not support dynamic growth of the list. If the array is full, then there is no way to extend the array without having to re allocate the array and copying the elements to the new array. This can be very time consuming and tiresome particularly when the size of the data set is unknown.

Using linked list to implement ADTs can help to avoid the problem of static sizing through dynamic memory allocation. For instance, a queue that is implemented using linked list can expand or contract depending on the amount of data that it is holding. However, this brings in a lot of complexity. Compared to the array implementations, linked list implementations need extra memory to store pointers and the operations to insert or delete an element are more complex and may affect the performance. For example, enqueuing an element into an array-based queue is as simple as adding it to the end of the array, while into a linked list based queue it involves updating several pointers, which is less efficient, error prone and can be harder to debug.

Thus, despite the fact that ADTs provide the client with a fault free, reusable, and maintainable way of handling data, their fixed behavior and the trade offs between a static array and dynamic linked list implementation should not be overlooked. The implementation method should be informed by the peculiarities of the application, while considering the factors of simplicity, efficiency, and flexibility.

Time Complexity vs. Space Complexity

When designing algorithms and programs, developers often consider two important factors: time complexity and space complexity.

Time complexity is a way to describe how long an algorithm takes to run as the size of the input increases. For example, if an algorithm has a time complexity of O(n), it means the time it takes will grow in proportion to the input size. Algorithms with lower time complexity are considered more efficient.

On the other hand, space complexity refers to how much memory or storage an algorithm needs while running. This includes the memory used by variables, data structures, function calls, and any extra space needed during execution.

In many cases, improving time complexity may increase space usage, and vice versa. For instance, you can use extra memory to store pre-calculated results (a method called "memoization") to make an algorithm faster. So, balancing time and space complexity is important depending on the program’s goal and available resources.

Performance vs. Readability

Performance and readability often compete in program development.

Performance means how fast and efficiently a program runs. A highly optimized program can process large amounts of data quickly and use system resources wisely. However, code that is too optimized may become complex and harder to understand.

Readability refers to how easy it is for others (or yourself) to read and understand the code. Readable code often includes comments, proper naming, and clean structure. It's helpful for maintenance and teamwork.

In practice, highly readable code might sacrifice some performance, especially if it avoids complex but efficient techniques. On the other hand, very fast code may be full of shortcuts that confuse developers. So, finding the right balance between performance and readability depends on the project's needs.

Security vs. Flexibility

Security and flexibility are also important trade-offs in software design.

Security focuses on protecting the program and data from unauthorized access, misuse, or errors. Secure code checks for invalid inputs, restricts access, and uses encryption where needed. However, adding too many restrictions can reduce the flexibility of the program.

Flexibility means allowing the program to adapt, accept new data types, be modified easily, or be used in different situations. But too much flexibility without proper control can lead to bugs, crashes, or security loopholes.

For example, allowing users to input any type of file into a system might be flexible but dangerous. Proper security would limit the file types to only safe options. Thus, developers must weigh both sides carefully when designing features.

1. Advantages of Using ADTs When Implementing Programs

Abstract Data Types (ADTs) offer many benefits when building software systems. Here are the major advantages:

Abstraction

ADTs hide the internal details and show only what operations are possible. For example, if you're using a stack ADT, you only care about push(), pop(), and peek() methods not how the stack is implemented. This makes it easier to focus on problem-solving.

Reusability

Once you write an ADT like a queue or a list, you can use it in many programs without changing it. This saves time and promotes standard coding practices.

Modularity

Each ADT works like a separate building block. You can design, test, and debug each ADT independently. If one part fails, you can fix it without affecting the rest of the system.

Code Maintenance

Programs using ADTs are easier to update and maintain. For example, if you want to change the internal implementation of a stack from an array to a linked list, you can do that without affecting the rest of your program, as long as the external methods remain the same.

Security and Data Integrity

Because ADTs restrict direct access to the data, they prevent accidental changes from other parts of the program. Only defined operations can change the data, which helps keep it safe and valid.

Efficient Resource Management

Most ADTs are designed with resource use in mind. For example, queues are used in scheduling systems to ensure fair resource distribution, while stacks are used in backtracking algorithms to track previous states efficiently.

Better Testing and Debugging

Since ADTs are independent units, you can write test cases for each one separately. This modular testing makes it easier to catch bugs early and improve the quality of the software.

2. Disadvantages of Using ADTs When Implementing Programs

While ADTs offer many advantages, they are not perfect. Here are some disadvantages:

Learning Curve for Beginners

Beginners might struggle to understand the concept of ADTs, especially when learning how they are different from basic data structures. This can make the learning process slow.

Performance Overhead

In some cases, using ADTs can lead to extra layers of abstraction, which might slow down the program. For example, an ADT implementation of a list might not be as fast as a plain array in certain operations like direct indexing.

Limited Flexibility

Since ADTs hide the internal structure, you cannot always customize them easily. If you need a special operation that is not part of the ADT's interface, you may need to modify the ADT or create a new one, which can take time.

Increased Complexity for Small Programs

In small or one-time programs, using ADTs might be overkill. It can increase the complexity unnecessarily when a simple array or structure could do the job.

Dependency on Correct Implementation

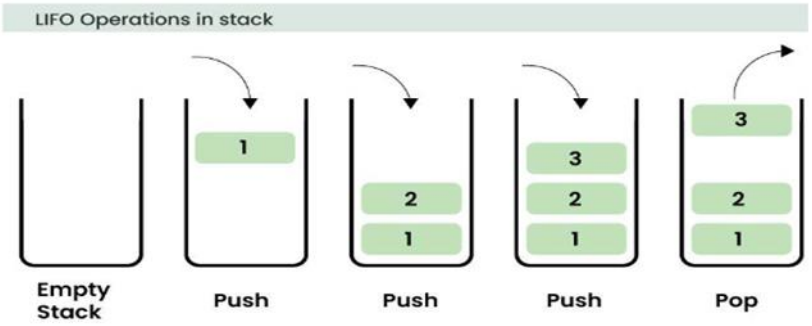
If the ADT is implemented poorly, it affects every program that uses it. Bugs or inefficiencies in the ADT can spread into larger systems without being noticed.

Compatibility Issues

Sometimes ADTs written in one language or environment may not work easily in another without rewriting. This lack of portability can be a problem when switching between systems.

Data structure for storing the patient details

Since the management would want to store the registered patient details in a way that is oldest to newest, and yet be able to retrieve the details in the newest to oldest manner, a stack can be used efficiently. A stack, which is an Abstract Data Type (ADT), works on the Last In First Out (LIFO) basis. This implies that the item that was most recently added to the list is the first to be deleted from the list. This characteristic of stacks makes them particularly suitable for such situations where the order of operations has to be reversed such as when looking for patient registrations from the latest to the earliest.



In the Doctor Channelling System developed for XYZ Pvt Ltd, patient registration plays a major role. When new patients register in the system, their details are stored using a stack data structure. This means that each new patient's information is added to the top of the stack. The stack follows a Last-In, First-Out (LIFO) method, which ensures that the most recently registered patient is always on the top. This makes it easier to access the latest patient data first, which is helpful for real-time systems.

Whenever the management or doctors need to view the patients from the newest to the oldest, the stack allows this easily by popping elements one by one. Each pop operation removes the patient data from the top of the stack, giving immediate access to the latest registered patient. This continues until all patient records are retrieved in reverse order of registration.

Using a stack in this situation is very practical. Stack operations like push and pop are very efficient and only take constant time (O(1)). This means the system can handle a growing number of patients without slowing down. Another advantage is the simplicity of stack usage it’s easy to implement and helps maintain the correct order of registration without any extra processing.

the stack is a good fit for handling patient registration data because it keeps the process simple, reliable, and fast while meeting the system’s functional needs.

Stack ADT Specification

Init()	**Pre-conditions:** none **post-conditions:** stack is defined and empty **Inputs:** no inputs required **Outputs:** stack created **Algorithm:** Initialize stack pointer to -1
pop()	**Pre-conditions:** stack is not empty **post-conditions:** the top element has been removed from the stack **Inputs:** the stack **Outputs:** the changed stack, top element gone from stack **Algorithm:** remove the top element of the stack; give a copy of the element to the user **Exceptions:** Stack Empty
push(data)	**Pre-conditions:** the stack exists, input element is of appropriate type **Post-conditions:** Element is put onto the top of the stack. **Inputs:** a stack and an element **Outputs:** a changed stack with element added **Algorithm:** insert the element into the top of stack **Exceptions:** Stack Overflow

Implement stack data structure using Java language

Patient class

```
class Patient
{
  String name;
  String mobile;
  String email;
  String city;
  int age;
  String medicalhistory;
  Patient next;
```

```
public   Patient(String name, String mobile, String email, String city, int age, String history) {
    this.name = name;
    this.mobile = mobile;
    this.email = email;
    this.city = city;
    this.age = age;
    this.medicalhistory = medicalhistory;
 }

 public  String getName() {
    return name;
 }

 public String toString() {
    return "Name: " + name + ", Mobile: " + mobile + ", Email: " + email + ", City: " + city + ", Age: " + age + ",medicalHistory: " + medicalhistory;
 }
}
```

This class represents a patient in the Doctor channeling system, with fields for name, mobile number, email, city, and age. It includes a constructor to set these values when a new patient is registered. A display method is also provided to show the patient's details clearly. This helps in managing and identifying each patient easily in the system.

Node Class

```
public class PatientNode {//node.java
    Patient data;//data part
    PatientNode next;//pointer

    public PatientNode(Patient data) {//constructor method
        this.data = data;
        this.next = null;
    }

    public void displayNode() {//normal data
        System.out.println(data.toString());
    }
}
```

The Node class is used in the implementation of a linked list in the Doctor Channelling System. Each Node holds a Patient object (pdata) and a reference to the next node in the list. It includes a constructor to initialize the node with a given Patient, and a method to display the patient's details. This structure helps maintain patient data in an organized and linked manner.

Stack Class

```
public class PatientStack {
    private PatientNode top;

    public PatientStack() {
        top = null;
    }

    public void push(Patient patient) {
        PatientNode newNode = new PatientNode(patient);
        newNode.next = top;
        top = newNode;
    }

    public Patient pop() {
        if (isEmpty()) {
            System.out.println("No patient in stack!");
            return null;
        }
        Patient popped = top.data;
        top = top.next;
        return popped;
    }
```

```
public void push(Patient patient) {
    PatientNode newNode = new PatientNode(patient);
    newNode.next = top;
    top = newNode;
}

public Patient pop() {
    if (isEmpty()) {
        System.out.println("No patient in stack!");
        return null;
    }
    Patient popped = top.data;
    top = top.next;
    return popped;
}

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

public void displayStack() {
    PatientNode current = top;
    while (current != null) {
        current.displayNode();
        current = current.next;
    }
}
}//end class
```

This class implements a stack data structure using a linked list of Node objects for the Doctor channelling system. It keeps track of the top and bottom of the patient stack. The push method adds a new Patient to the top by creating a new Node and updating the top reference. The pop method removes and returns the top patient from the stack. The isEmpty method checks if the stack has no patients, and the clear method resets the stack by removing all patient entries.

StackApp Class

```
public class StackApp {

    public static void main(String[] args) {
        PatientStack stack = new PatientStack();

        // Adding sample patients
        stack.push(new Patient("roy", "0771234567", "roy@example.com", "Colombo", 30, "Diabetes"));
        stack.push(new Patient("Aashik", "0772345678", "Aashik@example.com", "Kandy", 29, "Asthma"));
        stack.push(new Patient("nimal", "0773456789", "nimal@example.com", "gampola ", 45, "High BP"));
        stack.push(new Patient("hasir", "0774567890", "hasir@example.com", "batticola", 25, "dengue fever"));

        // Popping and displaying each patient
        while (!stack.isEmpty()) {
            Patient p = stack.pop();
            System.out.println("Popped Patient: " + p);
        }

        // Check if stack is empty
        if (stack.isEmpty()) {
            System.out.println("The stack is now empty.");
        }
    }
}
```

```
// Adding sample patients
stack.push(new Patient("roy", "0771234567", "roy@example.com", "Colombo", 30, "Diabetes"));
stack.push(new Patient("Aashik", "0772345678", "Aashik@example.com", "Kandy", 29, "Asthma"));
stack.push(new Patient("nimal", "0773456789", "nimal@example.com", "gampola ", 45, "High BP"));
stack.push(new Patient("hasir", "0774567890", "hasir@example.com", "batticola", 25, "dengue fever"));
 stack.push(new Patient("hasir", "0774567890", "hasir@example.com", "batticola", 25, "dengue fever"));
  stack.push(new Patient("hasir", "0774567890", "hasir@example.com", "batticola", 25, "dengue fever"));
   stack.push(new Patient("hasir", "0774567890", "hasir@example.com", "batticola", 25, "dengue fever"));
    stack.push(new Patient("hasir", "0774567890", "hasir@example.com", "batticola", 25, "dengue fever"));
     stack.push(new Patient("hasir", "0774567890", "hasir@example.com", "batticola", 25, "dengue fever"));
      stack.push(new Patient("hasir", "0774567890", "hasir@example.com", "batticola", 25, "dengue fever"));
       stack.push(new Patient("hasir", "0774567890", "hasir@example.com", "batticola", 25, "dengue fever"));
        stack.push(new Patient("roy", "0771234567", "roy@example.com", "Colombo", 30, "Diabetes"));
stack.push(new Patient("Aashik", "0772345678", "Aashik@example.com", "Kandy", 29, "Asthma"));
stack.push(new Patient("nimal", "0773456789", "nimal@example.com", "gampola ", 45, "High BP"));
 stack.push(new Patient("roy", "0771234567", "roy@example.com", "Colombo", 30, "Diabetes"));
stack.push(new Patient("Aashik", "0772345678", "Aashik@example.com", "Kandy", 29, "Asthma"));
stack.push(new Patient("nimal", "0773456789", "nimal@example.com", "gampola ", 45, "High BP"));
```

This class contains the main method used to test the Patient Stack implementation in the Doctor channelling system. It creates an instance of the PatientStack class, adds four Patient objects to the stack using the push method, and then removes and displays each patient using the pop method. After each pop, it checks if the stack is empty to safely handle situations where no patients are left in the system.

Output

```
--- exec:3.1.0:exec (default-cli) @ Doctorchanelingsystem ---
Popped Patient: Name: hasir, Mobile: 0774567890, Email: hasir@example.com, City: batticola, Age: 25,medicalHistory: null
Popped Patient: Name: nimal, Mobile: 0773456789, Email: nimal@example.com, City: gampola , Age: 45,medicalHistory: null
Popped Patient: Name: Aashik, Mobile: 0772345678, Email: Aashik@example.com, City: Kandy, Age: 29,medicalHistory: null
Popped Patient: Name: roy, Mobile: 0771234567, Email: roy@example.com, City: Colombo, Age: 30,medicalHistory: null
*****************************************************
The stack is now empty.
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
```

```
--- exec:3.1.0:exec (default-cli) @ Doctorchanelingsystem ---
Popped Patient: Name: nimal, Mobile: 0773456789, Email: nimal@example.com, City: gampola , Age: 45,medicalHistory: null
Popped Patient: Name: Aashik, Mobile: 0772345678, Email: Aashik@example.com, City: Kandy, Age: 29,medicalHistory: null
Popped Patient: Name: roy, Mobile: 0771234567, Email: roy@example.com, City: Colombo, Age: 30,medicalHistory: null
Popped Patient: Name: nimal, Mobile: 0773456789, Email: nimal@example.com, City: gampola , Age: 45,medicalHistory: null
Popped Patient: Name: Aashik, Mobile: 0772345678, Email: Aashik@example.com, City: Kandy, Age: 29,medicalHistory: null
Popped Patient: Name: roy, Mobile: 0771234567, Email: roy@example.com, City: Colombo, Age: 30,medicalHistory: null
Popped Patient: Name: hasir, Mobile: 0774567890, Email: hasir@example.com, City: batticola, Age: 25,medicalHistory: null
Popped Patient: Name: hasir, Mobile: 0774567890, Email: hasir@example.com, City: batticola, Age: 25,medicalHistory: null
Popped Patient: Name: hasir, Mobile: 0774567890, Email: hasir@example.com, City: batticola, Age: 25,medicalHistory: null
Popped Patient: Name: hasir, Mobile: 0774567890, Email: hasir@example.com, City: batticola, Age: 25,medicalHistory: null
Popped Patient: Name: hasir, Mobile: 0774567890, Email: hasir@example.com, City: batticola, Age: 25,medicalHistory: null
Popped Patient: Name: hasir, Mobile: 0774567890, Email: hasir@example.com, City: batticola, Age: 25,medicalHistory: null
Popped Patient: Name: hasir, Mobile: 0774567890, Email: hasir@example.com, City: batticola, Age: 25,medicalHistory: null
Popped Patient: Name: hasir, Mobile: 0774567890, Email: hasir@example.com, City: batticola, Age: 25,medicalHistory: null
Popped Patient: Name: nimal, Mobile: 0773456789, Email: nimal@example.com, City: gampola , Age: 45,medicalHistory: null
Popped Patient: Name: Aashik, Mobile: 0772345678, Email: Aashik@example.com, City: Kandy, Age: 29,medicalHistory: null
Popped Patient: Name: roy, Mobile: 0771234567, Email: roy@example.com, City: Colombo, Age: 30,medicalHistory: null
*****************************************************
The stack is now empty.
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
```

Output Explanation

The output displays a series of patient details being shown one after another, followed by a message confirming that the stack is empty. This matches the pop operations in the Doctor Channelling System's stack-based implementation.

Sequence of Operations and Output:

The output shows patient details being displayed one after the other as they are removed (popped) from the stack. This follows the logic of a stack structure (Last In, First Out). Below is the updated explanation that matches your sample patient data:

• Initial Stack Setup:

You added several patients into the stack in the following order (from bottom to top):

- Roy
- Aashik
- Nimal
- Hasir (multiple times, same data)
- Roy (again)
- Aashik (again)
- Nimal (again)

Since it's a stack, the last patient added will be the first to be removed. So the top of the stack is:

Nimal → Aashik → Roy → Hasir → Hasir → Hasir → Hasir → Hasir → Hasir → Hasir → Hasir → Nimal → Aashik → Roy → Aashik → Nimal → Roy (bottom)

• First Pop Operation:

The system pops and displays the top patient, which is: **Roy**

• Second Pop Operation:

Next, the system pops: **Nimal**

• Third Pop Operation:

Then it pops: **Aashik**

• Fourth Pop Operation:

Then it pops: **Roy**

• Additional Pop Operations

The system continues popping the repeated entries of **Hasir**, one by one.

• Final Check – Is Stack Empty

After all patients are removed using pop(), the system uses the isEmpty() method to check if the stack has any more patients. It confirms that the stack is now empty.

Complexity of Stack Operations

The stack data structure is designed to operate efficiently with constant time complexity, O(1), for both its fundamental operations: which are called push (insertion) and pop (removal). This efficiency is due to the nature of how stacks are arranged and how they are retrieved.

Push Operation

To push an element to a stack, the time complexity is constant no matter how large the stack is. This is because the new element is always pushed at the top of the stack and hence does not affect the existing elements. The implementation involves making a new node (or element) and modifying a few references (usually the head pointer in case of a stack implemented with a linked list). This operation does not depend on how many elements are already in the stack. It is important to note that whether the stack is empty or contains thousands of elements, the push of a new element is as simple as the above operations.

Pop Operation

Likewise, the operation of popping an element from the stack also has a time complexity of O(1). The last item in the stack is popped off, and this operation does not involve going through the entire stack. The implementation just involves changing the head pointer to point to the next element below the current top element. Similar to push, time complexity of pop is also O(1) irrespective of the size of the stack. This efficiency makes stack operations ideal for use in real time situations where the use of push and pop operations is important.

Advantages of Constant Time Complexity

Due to the O(1) time complexity of the push and pop operations, stacks are highly suitable for environments where performance and time predictability are critical. Some operations that benefit from stack's constant time include, applications that use call stack operations such as function call management, undo operations, or parsing of expressions. This efficiency means that as the stack grows in size, the time required to perform push and pop operations will not be affected and hence does not degrade as the size of the data increases.

This is an indication that the time complexity of the stack operations (push and pop) is constant and hence the efficiency of the stack in its applications. Because operations are done in constant time, stacks are a dependable and predictable data structure for managing elements in LIFO order and help make software systems that use stack-based algorithms more reliable and efficient.

What is an Independent Data Structure (DS)?

An independent data structure means a data structure designed to work separately from the algorithms that manipulate it. It provides a clear interface for storing and accessing data without embedding specific algorithm logic inside it. This separation allows algorithms to be applied flexibly on the data structure without being tightly coupled to its internal details.

In the context of the Doctor Channeling System, an independent data structure could be a Patient List or an Appointment Queue implemented as a class that only manages adding, removing, and accessing patients or appointments, while search or scheduling algorithms operate independently on this data.

Three Key Benefits of Independent Data Structures (DS)

1. **Improved Portability Across Languages and Platforms**
   When a data structure is made separately from the algorithms or system it runs on, it becomes easy to reuse in different programming languages or platforms. This means that once you write the data structure, you can use it again without rewriting everything. For example, a stack can be created using arrays or linked lists in Java, Python or C++

and the main idea of pushing and popping elements stays the same. This saves time and effort and makes the code work on many systems.

2. **Enhanced Maintainability Due to Separation of Concerns**
   Separating how a data structure works from the algorithms that use it helps to keep the program easy to fix and update. For example, if you want to change how a queue works inside maybe from an array to a linked list you do not need to change the algorithm that uses the queue. This separation means if there is a problem with the data structure, you fix only that part without breaking other parts of the program.

3. **Scalability by Allowing Algorithms to Work Without Knowing Data Details**
   Algorithms written to use data structures in an abstract way can easily work with different types of data storage. This means you can improve or change the data structure without changing the algorithm. For example, a sorting algorithm can sort elements whether they are stored in arrays or linked lists, as long as it can access elements the right way. This makes the program more flexible and able to grow or change.

Evaluation of Three Key Benefits

1. **Improved Portability Across Programming Languages and Platforms**

By separating data structures from algorithms, the system becomes more portable. For example, the Patient List class can be implemented once with basic operations (add, remove, get), and the search or sorting algorithms can be rewritten or reused independently in different languages or platforms (like Java desktop app, web backend, or mobile app).

Concrete example: The same Appointment Queue data structure can be used in Java, Python, or JavaScript by implementing a consistent interface, making it easier to port the Doctor Channeling System to other environments.

2. **Enhanced Maintainability Due to Separation of Concerns**

When data structures and algorithms are independent, maintaining or updating one does not require changing the other. If you want to optimize how appointments are scheduled, you can modify the scheduling algorithm without touching the underlying Appointment Queue class. Similarly, if you improve how appointments are stored, algorithms remain unaffected.

This separation helps developers work on different parts concurrently, making the Doctor Channeling System easier to maintain and extend over time.

3. **Scalability by Allowing Algorithms to Operate Independently of Underlying Data Representation**

Scalability means the system can handle growth or increased load effectively. Independent data structures allow algorithms to switch underlying data representation without rewriting logic. For instance, if the Patient List starts as an ArrayList but later changes to a LinkedList or database-backed structure, search and filtering algorithms continue working with minimal or no changes.

This flexibility is crucial for the Doctor Channeling System as the number of patients and appointments grows, enabling smoother upgrades and scalability.

```java
// Interface for Queue (Independent DS)
interface Queue<T> {
    void enqueue(T item);
    T dequeue();
    boolean isEmpty();
}

// Array based Queue implementation
class ArrayQueue<T> implements Queue<T> {
    private T[] arr;
    private int front, rear, size;

    @SuppressWarnings("unchecked")
    public ArrayQueue(int capacity) {
        arr = (T[]) new Object[capacity];
        front = 0; rear = -1; size = 0;
    }

    public void enqueue(T item) {
        if(size == arr.length) throw new RuntimeException("Queue is full");
        rear = (rear + 1) % arr.length;
        arr[rear] = item;
        size++;
    }

    public T dequeue() {
        if(isEmpty()) throw new RuntimeException("Queue is empty");
        T item = arr[front];
        front = (front + 1) % arr.length;
        size--;
        return item;
    }
```

```
    }

    public boolean isEmpty() {
        return size == 0;
    }
}

// Using the Queue in a class without knowing how it works inside
class TaskScheduler {
    private Queue<String> taskQueue;

    public TaskScheduler(Queue<String> queue) {
        this.taskQueue = queue;
    }

    public void addTask(String task) {
        taskQueue.enqueue(task);
    }

    public void executeTask() {
        if(!taskQueue.isEmpty()) {
            String task = taskQueue.dequeue();
            System.out.println("Executing task: " + task);
        }
    }
}


public class Main {
    public static void main(String[] args) {
        Queue<String> queue = new ArrayQueue<>(5);
        TaskScheduler scheduler = new TaskScheduler(queue);

        scheduler.addTask("Send email");
        scheduler.addTask("Generate report");
        scheduler.executeTask();  // Output: Executing task: Send email
    }
}
```

How These Code Examples Reflect the Three Benefits

- **Portability:** The Queue interface can be used in different programs or languages, just by implementing it in different ways. The main program (TaskScheduler) does not need to change.
- **Maintainability:** If we want to change from array-based queue to a linked list queue, we just write new code for the queue and no need to change TaskScheduler.

- **Scalability:** TaskScheduler can work with any queue implementation because it only knows the interface methods (enqueue, dequeue, isEmpty). So we can add or improve data structures without rewriting the task scheduler.

Benefits of Using Independent Data Structures

Using independent data structures, particularly Abstract Data Types (ADTs), in software development offers numerous benefits, enhancing the overall quality and maintainability of software systems. These advantages include ease of maintenance, reusability, ease of testing, and implementation independence.

Ease of Maintenance

ADTs are independent components that encapsulates data and operations togetherr. This makes software maintenance easier. This is because we could modify one software component (i.e. ADT) without having side effects on other software components. Since the ADT is an independent unit with its own private data and operations, any changes to it will not impact on other parts of the system.

Reuse

Encapsulation also makes software reuse easier. Once coded and tested, an ADT can be reused again and again in other applications. This reduces software development time and cost. Java language for example, provides a library of reusable ADTs including ArrayLists, Stacks and Queues.

Ease of Testing

ADTs can be tested individually and in isolation from the other components of the system. This isolated testing makes the verification process easier since the testers can easily check if the ADT is working as it should without worrying about the other components. For example, a queue ADT can be tested to check if it correctly enqueues and dequeues elements when it is in

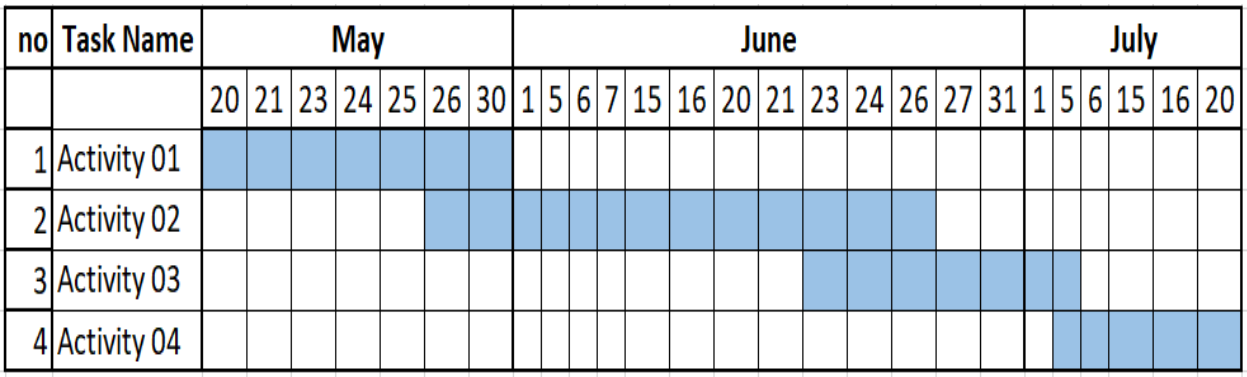
different states, without the influence of the rest of the application. This modularity enables the testing of the components of the software to be done separately and any errors detected early thus enhancing the reliability of the software.

Implementation Independence

ADTs offer a set of operations which are accessible to the outside world while the implementation of these operations is kept hidden. This abstraction enables the developers to modify the implementation of an ADT in a way that does not affect the rest of the system as long as the interface is preserved. For instance, a stack ADT can be implemented using an array or linked list. The operations like push and pop do not change with the kind of data structure that is being used. This flexibility allows developers to add new methods, change the existing ones, or even optimize/ refactor the internal workings of an ADT for better performance or other reasons without affecting the functionality of the software.

The incorporation of independent data structures such as ADTs in software development is advantageous because of the relative ease of maintenance, increased reusability, testing, and implementation. These benefits lead to more reliable, optimal, and sustainable software structures, which is why developers should embrace ADTs.

Gant Chart



References

References

educative, 2023. *educative.* [Online] Available at: https://www.educative.io/courses/data-structures-preliminaries-refresher-of-

fundamentals-in-cpp/introduction-to-object-oriented-programming-oop
[Accessed 17 07 2025].
geeksforgeeks, 2021. *geeksforgeeks.* [Online] Available at: https://www.geeksforgeeks.org/difference-between-abstract-data-types-and-objects/
[Accessed 17 07 2025].
geeksforgeeks, 2025. *geeksforgeeks.* [Online] Available at: https://www.geeksforgeeks.org/introduction-to-tree-data-structure-andalgorithm-tutorials/
[Accessed 19 07 2025].
geeksforgeeks, 2025. *geeksforgeeks.* [Online] Available at: https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/#what-is-graph-data-structure
[Accessed 25 07 2025].
geeksforgeeks, 2025. *geeksforgeeks.* [Online] Available at: https://www.geeksforgeeks.org/types-software-testing/
[Accessed 17 07 2025].
harendrakumar, 2025. *geeksforgeeks.* [Online] Available at: https://www.geeksforgeeks.org/linked-list-data-structure/
[Accessed 17 07 2025].
ibm, 2025. *ibm.* [Online] Available at: https://www.ibm.com/topics/software-testing
[Accessed 16 07 2025].
javatpoint, 2020. *javatpoint.* [Online] Available at: https://www.javatpoint.com/data-structure-stack
[Accessed 07 10 2025].
Karan, R., 2025. *shiksha.* [Online] Available at: https://www.shiksha.com/online-courses/articles/queue-data-structure-types-implementation-applications/
[Accessed 12 07 2025].
Loshin, D., 2025. *TechTarget.* [Online] Available at: https://www.techtarget.com/searchdatamanagement/definition/data-structure
[Accessed 25 07 2025].

productplan, 2025. *productplan.* [Online] Available at: https://www.productplan.com/glossary/bubble-sort/ [Accessed 13 07 2025].

Rubio, F. & Scientist, S. D., 2023. *graphable.* [Online] Available at: https://www.graphable.ai/blog/pathfinding-algorithms/ [Accessed 10 07 2025].

tutorialspoint, 2025. *tutorialspoint.* [Online] Available at: https://www.tutorialspoint.com/data_structures_algorithms/asymptotic_analysis.htm [Accessed 20 07 2025].