Ryan Wickramaratne | COL 00081762

Higher Nationals

Internal verification of assessment decisions – BTEC (RQF)

INTERNAL VERIFICATION – ASSESSMENT DECISIONS				
Programme title	Higher National Diploma in Computing			
Assessor	Mr. Dileepa	**Internal Verifier**		
Unit(s)	Unit 19 – Data Structures and Algorithms			
Assignment title	**Specification, Implementation, and Assessment of Data Structures for a sample scenario.**			
Student's name	Ryan Wickramaratne (COL 00081762)			
List which assessment criteria the Assessor has awarded.	**Pass**	**Merit**	**Distinction**	

INTERNAL VERIFIER CHECKLIST				
Do the assessment criteria awarded match those shown in the assignment brief?	Y/N			
Is the Pass/Merit/Distinction grade awarded justified by the assessor's comments on the student work?	Y/N			
Has the work been assessed accurately?	Y/N			
Is the feedback to the student: Give details: • Constructive? • Linked to relevant assessment criteria? • Identifying opportunities for improved performance? • Agreeing actions?	Y/N Y/N Y/N Y/N			
Does the assessment decision need amending?	Y/N			
Assessor signature		**Date**		
Internal Verifier signature		**Date**		
Programme Leader signature (if required)		**Date**		

Confirm action completed			
Remedial action taken Give details:			
Assessor signature		**Date**	
Internal Verifier signature		**Date**	
Programme Leader signature (if required)		**Date**	

Higher Nationals - Summative Assignment Feedback Form

Student Name/ID			
Unit Title	**Unit 19: Data Structures and Algorithms**		
Assignment Number	**1**	**Assessor**	
Submission Date	12/08/2023	**Date Received 1st submission**	
Re-submission Date		**Date Received 2nd submission**	

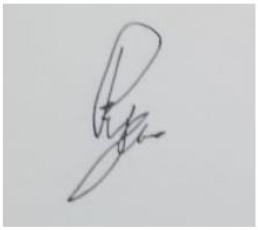
Assessor Feedback:

LO1 Examine different concrete data structures and it's valid operations.						
Pass, Merit & Distinction Descripts	**P1**	**P2**	**M1**	**M2**	**D1**	
LO2 Discuss the advantages, complexity of Abstract Data Type and importance concepts of Object orientation.						
Pass, Merit & Distinction Descripts	**P3**	**M3**	D2			
LO3 Implement, Demonstrate and evaluate complex ADT algorithm.						
Pass, Merit & Distinction Descripts	**P4**	**P5**	M4	D3		
LO4 Examine the advantages of Independent data structures and discuss the need of asymptotic analysis to assess the effectiveness of an algorithm.						
Pass, Merit & Distinction Descripts	**P6**	**P7**	M5	**D4**		

Grade:	**Assessor Signature:**	**Date:**
Resubmission Feedback:		
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.

Assignment Feedback

Formative Feedback: Assessor to Student			
Action Plan			
Summative feedback			
Feedback: Student to Assessor			
Assessor signature		**Date**	
Student signature	 ryandilthusha@gmail.com	**Date**	

Pearson Higher Nationals in

Computing

Unit 19: Data Structures & Algorithms

Assignment 01

General Guidelines

- A Cover page or title page – You should always attach a title page to your assignment. Use previous page as your cover sheet and make sure all the details are accurately filled.
- Attach this brief as the first section of your assignment.
- All the assignments should be prepared using a word processing software.
- All the assignments should be printed on A4 sized papers. Use single side printing.
- Allow 1" for top, bottom , right margins and 1.25" for the left margin of each page.

Word Processing Rules

- The font size should be **12** point, and should be in the style of **Time New Roman**.
- **Use 1.5 line spacing**. Left justify all paragraphs.
- Ensure that all the headings are consistent in terms of the font size and font style.
- 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.
- Use word processing application spell check and grammar check function to help editing your assignment.

Important Points:

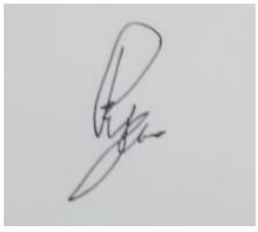
- It is strictly prohibited to use textboxes to add texts in the assignments, except for the compulsory information. eg: Figures, tables of comparison etc. Adding text boxes in the body except for the before mentioned compulsory information will result in rejection of your work.
- Carefully check the hand in date and the instructions given in the assignment. Late submissions will not be accepted.
- Ensure that you give yourself enough time to complete the assignment by the due date.
- Excuses of any nature will not be accepted for failure to hand in the work on time.
- You must take responsibility for managing your own time effectively.

- 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.
- Failure to achieve at least PASS criteria will result in a REFERRAL grade .
- Non-submission of work without valid reasons will lead to an automatic RE FERRAL. You will then be asked to complete an alternative assignment.
- 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.
- 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

Student Declaration

I hereby, declare that I know what plagiarism entails, namely to use another's work and to present it as my own without attributing the sources in the correct form. I further understand what it means to copy another's work.

- I know that plagiarism is a punishable offence because it constitutes theft.
- I understand the plagiarism and copying policy of Edexcel UK.
- I know what the consequences will be if I plagiarise or copy another's work in any of the assignments for this program.
- I declare therefore that all work presented by me for every aspect of my program, will be my own, and where I have made use of another's work, I will attribute the source in the correct way.
- I acknowledge that the attachment of this document signed or not, constitutes a binding agreement between myself and Pearson, UK.
- I understand that my assignment will not be considered as submitted if this document is not attached to the assignment.



ryandilthusha@gmail.com

12/08/2023

Student's Signature:
(*Provide E-mail ID*)

Date:
(*Provide Submission Date*)

Higher National Diploma in Business

Assignment Brief

Student Name /ID Number	Ryan Wickramaratne (COL 00081762)
Unit Number and Title	**Unit 19 : Data Structures and Algorithms**
Academic Year	2021/22
Unit Tutor	MR. Dileepa
Assignment Title	**Specification, Implementation, and Assessment of Data Structures for a sample scenario.**
Issue Date	
Submission Date	
IV Name & 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.

	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.

	Assignment Brief and Guidance:
	Scenario ABC Pvt Ltd organizing Car Racing event across western province and they decided to have maximum of 6 cars(participants) to compete. There are totally 3 Rounds and at the end of each round lowest rank car will be eliminated from the Race. Each car has unique number, brand, sponsor and driver details. In order to manage this particular event ABC Pvt Ltd decided to develop an Application. Application functions are listed down. 1.Register Car Details 2.Delete a car 3.Insert 3 Rounds Results. 4.Find out the winners (1st,2nd,3rd) 5.Search for a particular car **Task 1:** Examine and create data structure by simulating the above scenario and explain the valid operations that can be carried out on this data structure. Determine the operations of a queue and critically review how it is used to implement function calls related to the above scenario. **Task 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 suitable test cases and illustrate the system. Provide evidence of the test cases and the test results.

	Task 3 : Registered Car details are stored from oldest to newest. Management of ABC Pvt Ltd should be able to find from the newest to oldest registered car details. Using an imperative definition, specify the abstract data type for the above scenario and implement specified ADT using java programming and briefly discuss the complexity of chosen ADT algorithm. List down the advantages of Encapsulation and Information hiding when using an ADT selected for the above scenario. *"Imperative ADTs are basis for object orientation."* Discuss the above view stating whether you agree or not. Justify your answer. **Task 4:** ABC Pvt Ltd plans to visit all of these participants through the shortest path within a day. Analyse the above operation by using illustrations, of two shortest path algorithms, specify how it operates using a sample graph diagram. Sort the cars based on numbers with two different sorting algorithms and critically review the performances of those two algorithms by comparing them. **Task 5:** Evaluate how Asymptotic analysis can be used to assess the effectiveness of an algorithm and critically evaluate the different ways in which the efficiency of an algorithm can be measured. Critically review the sort of trade-offs exists when you use an ADT for implementing programs. You also need to evaluate the benefits of using independent data structures for implementing programs.

Grading Rubric

Grading Criteria	Achieved	Feedback
LO1. Examine abstract data types, concrete data structures and algorithms.		
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.		
D1 Analyse the operation, using illustrations, of two network shortest path algorithms, providing an example of each.		
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 would like to express my special thanks of gratitude to my Systems Analysis & Design lecturer Mr. Dileepa for providing invaluable guidance and giving immense amount of knowledge to work on this assignment perfectly. I specially thanks him because she helped us in doing a lot of research and I came to know about so many new things about the Systems Analysis & Design.

Secondly, I would like to thank my parents and friends who helped me a lot in finalizing this project within the limited time frame.

Executive Summary

Throughout this comprehensive assignment, we delved deep into the intricate world of algorithms, data structures, and their interrelationships. We began by exploring the basic operations on data structures such as arrays and dynamic queues. Our journey continued with a thorough examination of different algorithms and their relevancies to real-world scenarios, particularly focusing on ABC Pvt Ltd's unique challenges.

We ventured into understanding the foundational aspects of Asymptotic Analysis, which is pivotal in evaluating the efficiency of algorithms. Through Asymptotic Analysis, we derived insights into how algorithms perform under varying input sizes, helping us make informed decisions in algorithm selection for particular scenarios.

Abstract Data Types (ADTs) were a focal point, showcasing their immense value in encapsulating and hiding data, which ensures robust and modular code structures. Furthermore, the assignment highlighted the significance of understanding trade-offs. Recognizing the potential advantages and limitations of each decision in the world of algorithms and data structures is vital for efficient problem-solving.

Lastly, the assignment underscored the advantages of employing independent data structures. Such structures play a crucial role in improving code quality, enhancing maintenance, ensuring separation of concerns, and facilitating extensibility.

List of figures

List of Tables

TABLE OF CONTENTS

Task 1

Register, Delete, Search and Print Car

Deciding the best suitable data structure

The best data structure for the given scenario is dependent on various factors such as the volume of data, the operations required, and the performance characteristics of the data structure.

In the provided scenario, we have a car racing event where a maximum of 6 cars can participate. The operations that need to be performed include registering a car, deleting a car, searching for a car, and printing the list of cars.

Given this context, a **simple array** was chosen as the primary data structure for the application.

Why a Simple Array?

- **Fixed Size:** Since the maximum number of participants is known and fixed at 6, an array is suitable because its size is fixed. We can easily create an array of size 6 to store all the car details.
- **Efficiency:** An array is a contiguous block of memory, so it's fast to access any element by its index. This makes searching for a car in an array very efficient if we know its index.
- **Simplicity:** Arrays are simple and straightforward to use. They are a fundamental data structure and have less overhead compared to more complex structures like linked lists or trees.

Comparison to Other Data Structures

When compared to other data structures, the simple array has several advantages in this context:

1) **ArrayList:** An ArrayList is a resizable array implementation in Java. It is an excellent choice when we need a dynamic data structure that can grow and shrink in size. However, in our case, the number of cars is fixed at 6, so we don't need the resizing functionality of ArrayList. Therefore, using a simple array is more efficient.
2) **LinkedList:** A LinkedList allows for efficient insertions and deletions at both ends but is less efficient when accessing elements in the middle. Given that we might need to delete a car (which could be anywhere in the list), a LinkedList could be considered. However, LinkedLists have more overhead per element (due to the need to store next and previous references), and since our list size is small (only 6 elements), the benefits of a LinkedList are not substantial.
3) **HashSet or HashMap:** These data structures allow for very efficient lookups and could be considered if searching for cars were a very frequent operation. However, they do not maintain an order of elements, making them unsuitable for operations like printing the car list in a specific order or registering cars in a particular sequence.

In summary, considering the requirements of the scenario, the simple array was chosen as it provides the necessary functionalities with simplicity and efficiency. Other data structures like ArrayList, LinkedList, HashSet, or HashMap could also be used depending on the specific requirements and constraints, but for this particular scenario, a simple array offers the best balance of efficiency, simplicity, and suitability.

Valid operations for use with the Array Data Structure

The simple array structure chosen for this car racing scenario supports a collection of distinct operations. Each operation plays a critical role in managing and manipulating the event data. It's essential to have a deep understanding of how these operations work to appreciate the flexibility and efficiency that the array data structure offers.

1) Registering a Car (Array Insertion):

The 'registerCar' operation corresponds to the insertion operation in our array. It involves adding a new car object to our array data structure. This process involves a couple of important steps.

- **Availability Check:** The first part of this operation involves checking for available space within the array. As we know, arrays are fixed in size. In this scenario, we have set the maximum limit to 6 cars. So, if an attempt is made to register a new car when the array is already full, the operation will throw an error, indicating that the array is at its maximum capacity and cannot accommodate more cars.
- **Registering the Car:** If there is available space within the array, the next step is to register the car. This step involves adding the car object to the array at the next available index. In this context, the index of the newly registered car corresponds to the number of cars already registered.

This operation is relatively quick because adding an element at the end of an array is a constant time operation, meaning it does not depend on the size of the array.

2) Deleting a Car (Array Deletion):

The 'deleteCar' operation involves removing a specific car from our array. This operation involves a few more steps compared to insertion.

- **Searching for the Car:** The first step in this operation involves locating the car that needs to be deleted. This is done by traversing the array and comparing each car's unique number with the number of the car that we want to delete.
- **Shifting Elements:** Once we've located the car to be deleted, we need to shift all elements to the right of this car to the left by one position. This step is necessary to fill the gap left by the deleted car. This ensures that our array remains packed and there are no 'empty' slots in between the car objects.

The deletion operation is more time-consuming compared to the insertion operation because shifting elements requires more computational work, especially if the array is large.

3) Searching for a Car (Array Traversal):

The 'searchCar' operation is essentially an array traversal operation. This function involves going through the array one element at a time, comparing each car's number with the search target.

- If a match is found, the operation returns the car object.
- If the operation traverses the entire array and no match is found, it returns a message indicating that the car does not exist in the system.

4) Printing the Car List (Array Traversal):

The 'printCarList' operation is another instance of array traversal. Starting from the first car in the array, it goes through each car and prints the details of each car sequentially until it reaches the end of the array. It’s a read-only operation that does not modify the array.

5) Error Handling:

Each operation is also equipped with error handling to manage edge scenarios, such as attempting to register a car when the list is already full or trying to delete a car that doesn't exist. These error checks maintain the integrity of the data and ensure that the application behaves as expected even when it encounters unexpected inputs or situations. It's an essential aspect of robust application design.

Conceptual representation of the data structure

The Data Structure:

The data structure in this scenario is a simple array that stores objects of type 'Car'. Each object represents a unique car with specific attributes: car number, brand, sponsor, and driver. The maximum size of the array is six, meaning it can hold information about a maximum of six cars.

Ex:

[0] → Car {number: 1, brand: 'Toyota', sponsor: 'Sponsor1', driver: 'Driver1'}

[1] → Car {number: 2, brand: 'Honda', sponsor: 'Sponsor2', driver: 'Driver2'}

[2] → Car {number: 3, brand: 'Ford', sponsor: 'Sponsor3', driver: 'Driver3'}

[3] → Car {number: 4, brand: 'Chevrolet', sponsor: 'Sponsor4', driver: 'Driver4'}

[4] → Car {number: 5, brand: 'Mercedes', sponsor: 'Sponsor5', driver: 'Driver5'}

[5] → Car {number: 6, brand: 'BMW', sponsor: 'Sponsor6', driver: 'Driver6'}

Operations on the Data Structure:

Below are the operations performed on this array data structure:

- Registering a Car: This operation adds a new 'Car' object to the array at the next available index, provided the array is not already full.
- Deleting a Car: This operation removes a 'Car' object from the array, identified by the car number. It then shifts the remaining 'Car' objects to fill the gap left by the deleted car.
- Searching for a Car: This operation searches for a car in the array using the car number. It scans each 'Car' object in the array and returns the matching car if found.
- Printing the Car List: This operation traverses the entire array and prints the details of each 'Car' object.

Error Handling in the Data Structure:

The program also includes error handling to ensure that operations are carried out safely:

- When registering a car, it checks if the array is already full or if a car with the same number is already registered.
- When deleting a car, it verifies that the car to be deleted exists in the array.
- When searching for a car, it checks if the car exists in the array.

Creating a simple data structure

1) 'Car' Class:

This class represents a single Car entity. It has properties like 'carNumber', 'carBrand', 'carSponsor', and 'carDriver' that represent the car's unique number, brand, sponsor, and driver respectively.

```
public class Car {
    private int carNumber;
    private String carBrand;
    private String carSponsor;
    private String carDriver;
    //...constructors, getters, and setters...
}
```

2) 'CarManager' Class:

This class manages operations related to cars. It uses an array to store 'Car' objects and keeps track of the total number of cars currently registered.

```
public class CarManager {
    private static final int MAX_CARS = 6;
    private Car[] listOfCars = new Car[MAX_CARS];
    private int totalCars = 0;
    //...methods for registering, deleting, searching, and printing
cars...
}
```

3) Methods of the ‘CarManager’ class:

- ‘registerCar(Car newCar)’: This method checks if there's room for another car and if a car with the same number doesn't already exist. If both conditions are met, the car is added to the array and ‘totalCars’ is incremented.
- ‘deleteCar(int carNumber)’: This method finds a car with the given number in the array, deletes it by shifting all other cars to fill the gap, and decrements ‘totalCars’.
- ‘searchCar(int carNumber)’: This method traverses the array and returns the car with the given number, if it exists.
- ‘printCarList()’: This method traverses the array and prints the details of each car.

4) ‘Main’ Class:

This class demonstrates the use of ‘Car’ and ‘CarManager’ classes. It creates several ‘Car’ objects, registers them using the ‘CarManager’, performs searches and deletions, and prints the list of cars.

```
public class Main {
    public static void main(String[] args) {
        //...code to create Car and CarManager objects, and demonstrate
operations...
    }
}
```

Java Data Structure (whole code)

Car Class →

```
// This is our Car class.
// Each Car has a unique number, brand, sponsor, and driver.
public class Car {
    // Car attributes
    private int carNumber;
    private String carBrand;
    private String carSponsor;
    private String carDriver;

    // This is the constructor for Car class.
    // It is called when we create a new Car object.
    public Car(int number, String brand, String sponsor, String driver)
{
        // Error checking for inputs
        if(number <= 0) {
            throw new IllegalArgumentException("Car number must be
greater than 0.");
        }
        if(brand == null || brand.isEmpty()) {
            throw new IllegalArgumentException("Car brand must not be
empty.");
        }
        if(sponsor == null || sponsor.isEmpty()) {
            throw new IllegalArgumentException("Car sponsor must not be
empty.");
        }
        if(driver == null || driver.isEmpty()) {
            throw new IllegalArgumentException("Car driver must not be
empty.");
        }

        // Assign values to the car attributes
        this.carNumber = number;
        this.carBrand = brand;
        this.carSponsor = sponsor;
        this.carDriver = driver;
    }

    // These are getter methods, used to access private properties of
the class
    public int getCarNumber() {
        return carNumber;
    }

    public String getCarBrand() {
        return carBrand;
    }

    public String getCarSponsor() {
        return carSponsor;
    }
```

```
    public String getCarDriver() {
        return carDriver;
    }
}
```

CarManager Class →

```
// This is our CarManager class.
// It manages operations related to cars.
public class CarManager {
    // Maximum number of cars
    private static final int MAX_CARS = 6;
    // Array to hold the cars
    private Car[] listOfCars = new Car[MAX_CARS];
    // Number of cars currently registered
    private int totalCars = 0;

    // Method to add a new car to the list
    public void registerCar(Car newCar) {
        if(totalCars < MAX_CARS) { // Check if there is room for a new
car
            // Check if a car with the same number already exists
            for(int i = 0; i < totalCars; i++) {
                if(listOfCars[i].getCarNumber() ==
newCar.getCarNumber()) {
                    System.out.println("Registration failed. A car with
the same number is already registered.");
                    return;
                }
            }

            listOfCars[totalCars] = newCar; // Add new car to the next
available position
            totalCars++; // Increment the count of cars
            System.out.println("The car has been successfully
registered.");
        } else {
            System.out.println("Registration failed. The maximum number
of cars have been registered.");
        }
    }

    // Method to delete a car from the list
    public void deleteCar(int carNumber) {
        for(int i = 0; i < totalCars; i++) {
            // If the car is found
            if(listOfCars[i].getCarNumber() == carNumber) {
                // Move the cars in the list to fill the gap
                for(int j = i; j < totalCars - 1; j++) {
                    listOfCars[j] = listOfCars[j+1];
                }
                // Decrease the count of total cars
                totalCars--;
                System.out.println("The car has been successfully
deleted.");
                return;
            }
```

```
        }
        // If the car is not found
        System.out.println("Deletion failed. The specified car does not
exist.");
    }

    // Method to search for a car by its number
    public Car searchCar(int carNumber) {
        // Search for the car in the list
        for(int i = 0; i < totalCars; i++) {
            // If car is found, return the car object
            if(listOfCars[i].getCarNumber() == carNumber) {
                return listOfCars[i];
            }
        }
        // If car is not found, print a message and return null
        System.out.println("Search failed. The specified car was not
found.");
        return null;
    }

    // Method to print the list of cars
    public void printCarList() {
        for(int i = 0; i < totalCars; i++) {
            System.out.println("Car Number: " +
listOfCars[i].getCarNumber()
                    + ", Brand: " + listOfCars[i].getCarBrand()
                    + ", Sponsor: " + listOfCars[i].getCarSponsor()
                    + ", Driver: " + listOfCars[i].getCarDriver());
        }
    }
}
```

Main Class →

```
public class Main {
    public static void main(String[] args) {
        // Create an instance of CarManager
        CarManager carManager = new CarManager();

        // Register some cars
        try {
            carManager.registerCar(new Car(1, "Toyota", "Sponsor1",
"Driver1"));
            carManager.registerCar(new Car(2, "Honda", "Sponsor2",
"Driver2"));
            carManager.registerCar(new Car(3, "Ford", "Sponsor3",
"Driver3"));
            carManager.registerCar(new Car(4, "Chevrolet", "Sponsor4",
"Driver4"));
            carManager.registerCar(new Car(5, "Mercedes", "Sponsor5",
"Driver5"));
            carManager.registerCar(new Car(6, "BMW", "Sponsor6",
"Driver6"));
        } catch (IllegalArgumentException e) {
            System.out.println(e.getMessage());
        }
```

```
        // Try to register a car when the list is already full
        try {
            carManager.registerCar(new Car(7, "Ferrari", "Sponsor7",
"Driver7"));
        } catch (IllegalArgumentException e) {
            System.out.println(e.getMessage());
        }

        // Print the list of cars
        System.out.println("\nInitial list of cars:");
        carManager.printCarList();

        // Search for a car that exists
        Car car = carManager.searchCar(3);
        if(car != null) {
            System.out.println("\nFound Car: " + car.getCarBrand());
        }

        // Search for a car that does not exist
        car = carManager.searchCar(7);
        if(car != null) {
            System.out.println("Found Car: " + car.getCarBrand());
        }

        // Delete a car that exists
        carManager.deleteCar(2);

        // Try to delete a car that does not exist
        carManager.deleteCar(7);

        // Print the list of cars again to check if the car was deleted
        System.out.println("\nList of cars after deletion:");
        carManager.printCarList();

        // Try to register a car with a number that already exists
        try {
            carManager.registerCar(new Car(1, "Audi", "Sponsor8",
"Driver8"));
        } catch (IllegalArgumentException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Output of the Java Data Structure (whole code)

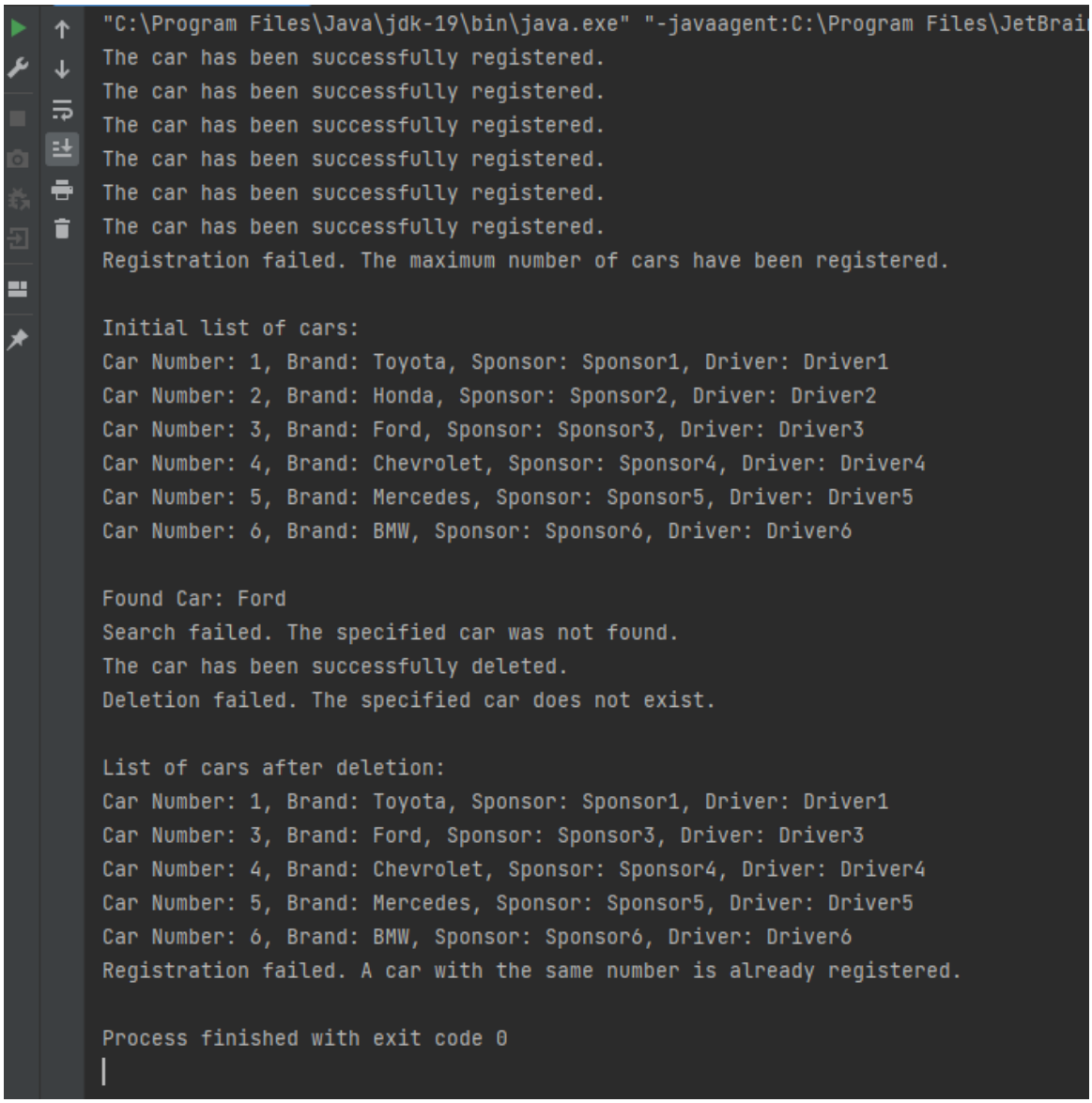


Figure 1. 1 Output of Register, Delete, Search and Print Car Java Data Structure (whole code)

Race round results, elimination and finding winners

Deciding the Best Suitable Data Structure

Choosing the right data structure is a critical part of any programming problem. The choice depends on the problem's requirements, including the operations that need to be performed on the data, the frequency of these operations, and any space-time trade-offs. In this case, I was dealing with a problem that involves managing race data and involves processing the results of multiple rounds. The primary operations include the insertion of new race data, elimination of a participant, and traversal of the data to determine rankings. Given this context, a **Dynamic Queue** was chosen as the primary data structure for the application.

Why a Dynamic Queue?

A queue is a data structure that follows the FIFO (First-In, First-Out) principle. Elements are inserted at the rear (end) and are removed from the front. For the race problem, a dynamic queue implemented as a linked list (i.e., each element in the queue has a reference to the next element) offers several benefits:

- Dynamic Size: Unlike static queues that have a fixed size, dynamic queues can grow and shrink during runtime. This makes them more flexible, which is useful because we don't know in advance how many rounds the race will last or how many cars will participate.
- Efficient Operations: Adding (enqueuing) an element at the end of the queue, and removing (dequeuing) an element from the front of the queue, both have a time complexity of O(1) when implemented as a linked list. This is useful when dealing with a potentially large number of cars.
- Natural Ordering: The queue's FIFO property naturally corresponds to the time sequence of the race rounds. The car that finishes the race first (i.e., the car that is enqueued first) should be the first one to be evaluated (dequeued) when determining rankings.

Comparison to Other Data Structures

- Static Queue: A static queue has a fixed size, making it less flexible for scenarios like ours where the number of cars (queue size) decreases after each round. As such, a dynamic queue is a better choice.
- Stack (Dynamic and Static): Stacks operate on the principle of Last In, First Out (LIFO), which is not suitable for our racing scenario. In a stack, the last car added would be the first one to be removed, which does not fit our elimination based on performance.
- Tree and Binary Search Trees (BST): While trees and BSTs are powerful data structures used for hierarchical data representation and quick search operations, they are not ideal for our scenario. They don't offer the natural FIFO structure of queues which is critical to our racing scenario. Also, deletion operations in a tree or BST can be complex and involve restructuring the tree, making them less efficient for our needs.
- Linked List: A dynamic queue is essentially a linked list that restricts operations to the ends (front/rear) only. If we were to use a general linked list, we could add/remove nodes from anywhere in the list, which is not what we want in our scenario.
- Arrays (Static and Dynamic): Like static queues, static arrays have a fixed size, making them less suitable for our changing number of cars. Dynamic arrays could change size, but removing elements (especially from the middle) can involve shifting elements and thus be less efficient.

In summary, a dynamic queue is the most suitable data structure for this race scenario given its flexibility, efficient operations, and natural alignment with the problem's requirements.

Valid operations for use with the Dynamic Queue Data Structure

In the scenario of this car racing application, I've used a queue to handle the structure of the race. Following are the primary operations that I have used:

1) Creation of the Queue:

The queue is an abstract data type that holds an ordered collection of items. In the racing scenario, I have chosen a queue because of its characteristic First-In-First-Out (FIFO) property, which fits well for a car race where the car that finishes first (fastest time) will be the first to be dequeued (winner).

A queue in this scenario is created using the LinkedList class of Java's standard library. This class provides in-built methods that allow me to use a LinkedList object as a queue. The LinkedList is chosen here because it allows efficient removal of elements from both ends (polling and removal of the last element), which are crucial to our race simulation.

In the implementation, I encapsulate this ‘LinkedList’ inside a custom class ‘CarQueue’ to handle specific actions related to the car race.

2) Adding Elements (Enqueue):

Adding an element to the queue (enqueue operation) means to add a new car to the race. This is an important operation as it populates our queue with participants for the race. In the code, this operation is performed in the ‘enqueue()’ method of the custom ‘CarQueue’ class. This method wraps the ‘add()’ method of the ‘LinkedList’, which appends a new car at the end of the list.

The initial population of the queue is done at the beginning of the race where all participating cars are enqueued. The enqueue operation is used when initializing the queue with the participating cars at the beginning of the race.

3) Removing Elements (Dequeue):

Removing an element from the queue (dequeue operation) signifies eliminating a car from the race. This operation is crucial in each round as it helps to simulate the elimination of the slowest car. In the code, this operation is carried out by the 'dequeue()' method of the 'CarQueue' class. This method wraps the 'removeLast()' method of the 'LinkedList'.

The dequeue operation is performed at the end of each round of the race, where the slowest car (car with the longest round time) is removed from the queue.

4) Checking if the Queue is Empty (isEmpty):

The 'isEmpty()' method is a key operation in any queue implementation. This operation is used to check if there are any cars left in the queue, or in other words, if there are any cars left in the race.

This operation is primarily performed before dequeue operation to ensure there is a car in the queue to remove. Without this check, an attempt to dequeue from an empty queue would throw a 'NoSuchElementException', leading to an error and interruption of the program.

5) Accessing Elements:

In the context of the car race, accessing elements from the queue helps to identify the fastest cars and the slowest car in each round. This operation is performed by the 'get()' method of the 'LinkedList'.

In the program, I use this operation to fetch the details of the top 3 cars (winners) and the last car (eliminated car) in each round. Furthermore, this operation is used at the end of all rounds to fetch and print the final winners of the race.

6) Sorting Elements:

A key feature in our car racing simulation is the ranking of cars based on their round times. Sorting the queue helps us identify the fastest and the slowest cars in each round. This operation is performed using the 'sort()' method of the LinkedList.

In the implementation, I use the 'Comparator.comparingDouble()' function to specify the sorting criteria. The sort operation is executed at the end of each round before the winners and the eliminated car are determined.

7) Handling Exceptions:

An essential part of any robust software application is efficient error handling. In the context of the queue operations, error handling is implemented to ensure that the program does not break due to unexpected issues, such as trying to remove an element from an empty queue.

In the code, error handling is implemented using try-catch blocks. These blocks are used to catch potential exceptions like 'NoSuchElementException' or 'IndexOutOfBoundsException' that may occur during the execution of the queue operations.

8) Printing the Queue:

Printing the queue is a helpful operation that lets us visualize the state of our queue at any given point in the race. In our scenario, it helps us see the cars in their current order based on their round times.

In the implementation, this operation is performed by the 'printQueue()' method of the 'CarQueue' class. This method uses a for-each loop to iterate through all the cars in the queue and prints their details.

In summary, these operations together provide the functionality needed for the car racing application, from initializing the queue with the participating cars, managing the race

rounds, to determining the winners. Each operation serves a specific purpose and contributes to the overall functionality of the application. The chosen data structure, a queue implemented with `LinkedList`, offers the flexibility and efficiency required for these operations.

Conceptual representation of the data structure

Basic Version of Conceptual Representation:

The main data structure used in the code above is a queue, represented using a LinkedList in Java. This queue holds elements of the custom 'Car' type. Each 'Car' has several properties: an ID, a make (brand), a driver, and a round time. This queue is used to simulate a car racing scenario where cars with the least round times (fastest cars) proceed in the race, and the car with the maximum round time (slowest car) is eliminated after each round.

The initial queue, after enqueueing the cars, might look like this:

```
CarQueue
-------------------------------------------------------------------------
| Car 1 (ID: 1, Make: Toyota,    Driver: Driver1, Round Time: 34.2)    |
| Car 2 (ID: 2, Make: Honda,     Driver: Driver2, Round Time: 37.4)    |
| Car 3 (ID: 3, Make: Ford,      Driver: Driver3, Round Time: 33.7)    |
| Car 4 (ID: 4, Make: Chevrolet, Driver: Driver4, Round Time: 31.6)    |
| Car 5 (ID: 5, Make: Subaru,    Driver: Driver5, Round Time: 32.9)    |
| Car 6 (ID: 6, Make: Ferrari,   Driver: Driver6, Round Time: 30.5)    |
-------------------------------------------------------------------------
```

After sorting the cars based on their round times, the queue might look like this:

```
CarQueue
-------------------------------------------------------------------------
| Car 6 (ID: 6, Make: Ferrari,   Driver: Driver6, Round Time: 30.5)    |
| Car 4 (ID: 4, Make: Chevrolet, Driver: Driver4, Round Time: 31.6)    |
| Car 5 (ID: 5, Make: Subaru,    Driver: Driver5, Round Time: 32.9)    |
| Car 3 (ID: 3, Make: Ford,      Driver: Driver3, Round Time: 33.7)    |
| Car 1 (ID: 1, Make: Toyota,    Driver: Driver1, Round Time: 34.2)    |
| Car 2 (ID: 2, Make: Honda,     Driver: Driver2, Round Time: 37.4)    |
-------------------------------------------------------------------------
```

And after dequeuing the slowest car (Car 2), the queue might look like this:

```
CarQueue
-------------------------------------------------------------------------
| Car 6 (ID: 6, Make: Ferrari,   Driver: Driver6, Round Time: 30.5) |
| Car 4 (ID: 4, Make: Chevrolet, Driver: Driver4, Round Time: 31.6) |
| Car 5 (ID: 5, Make: Subaru,    Driver: Driver5, Round Time: 32.9) |
| Car 3 (ID: 3, Make: Ford,      Driver: Driver3, Round Time: 33.7) |
| Car 1 (ID: 1, Make: Toyota,    Driver: Driver1, Round Time: 34.2) |
-------------------------------------------------------------------------
```

This process continues until we have our final winners. The state of the queue will be modified after each round, with the slowest car being dequeued each time.

The above representation helps to visualize the changes in the queue as the car racing simulation progresses and demonstrates how the queue data structure is effectively used in this scenario.

Enhanced Version of Conceptual Representation:

The following is another enhanced conceptual representation of the data structure. In this conceptual representation, each box represents a car in the queue. The front of the queue is the car that finished first (shortest time), while the rear is the car that finished last (longest time). After each round, the car at the rear (the one with the longest time) is removed from the queue. This process repeats until we have our final winners of the race.

```
Cars in Queue at beginning of race:
-----------------------------------
Rear --> | Car 6 | Car 5 | Car 4 | Car 3 | Car 2 | Car 1 |  --> Front



Round 1:

After assigning random times, the queue may become:
Rear --> | Car 1 | Car 4 | Car 6 | Car 2 | Car 5 | Car 3 |  --> Front

Top 3 Cars:
1st: Car 3
2nd: Car 5
3rd: Car 2

Eliminated Car: Car 1

Queue after Round 1:
Rear --> | Car 4 | Car 6 | Car 2 | Car 5 | Car 3 |  --> Front




Round 2:

After assigning random times, the queue may become:
Rear --> | Car 2 | Car 4 | Car 6 | Car 3 | Car 5 |  --> Front

Top 3 Cars:
1st: Car 5
2nd: Car 3
3rd: Car 6

Eliminated Car: Car 2

Queue after Round 2:
Rear --> | Car 4 | Car 6 | Car 3 | Car 5 |  --> Front
```

```
Round 3:

After assigning random times, the queue may become:
Rear --> | Car 6 | Car 4 | Car 5 | Car 3 |  --> Front

Top 3 Cars:
1st: Car 3
2nd: Car 5
3rd: Car 4

Eliminated Car: Car 6

Queue after Round 3:
Rear --> | Car 4 | Car 5 | Car 3 |  --> Front



Winners of the Race:
1st: Car 3
2nd: Car 5
3rd: Car 4
```

Creating a simple data structure

1) 'Car' Class:

The 'Car' class represents a single 'Car' entity. It has properties like 'id', 'make', 'driver', and 'roundTime' that represent the car's unique id, make, driver, and round time respectively.

```
public class Car {
    int id;
    String make;
    String driver;
    double roundTime;
    //...constructor, getters, setters, and toString method...
}
```

2) 'CarQueue' Class:

This class manages operations related to cars. It uses a LinkedList to store 'Car' objects, simulating a queue.

```
class CarQueue {
    LinkedList<Car> cars = new LinkedList<>();
    //...methods for enqueuing, dequeuing, checking if empty, and
printing cars...
}
```

3) Methods of the 'CarQueue' class:

- 'enqueue(Car car)': This method adds the car to the end of the queue.
- 'dequeue()': This method removes the car from the front of the queue.
- 'isEmpty()': This method checks if the queue is empty.
- 'printQueue()': This method traverses the queue and prints the details of each car.

4) Main Class:

This class demonstrates the use of ‘Car’ and ‘CarQueue’ classes. It creates several ‘Car’ objects, enqueues them into the ‘CarQueue’, performs a simulated race by assigning round times and sorting the cars, dequeues the car with the maximum round time (i.e., the car that is eliminated) after each round, and finally prints the winners of the race.

```
public class Test {
    public static void main(String[] args) {
        //...code to create Car and CarQueue objects, enqueue the cars,
simulate the race, dequeue the eliminated cars, and print the winners...
    }
}
```

Java Data Structure (whole code)

```
// Import necessary classes
import java.util.LinkedList;
import java.util.Comparator;

// Define Car class with appropriate properties and methods
class Car {
    int id;
    String make;
    String driver;
    double roundTime;

    // Car class constructor
    public Car(int id, String make, String driver) {
        this.id = id;
        this.make = make;
        this.driver = driver;
    }

    // toString method to print details of car in a readable format
    @Override
    public String toString() {
        return "Car ID: " + id +
                ", Make: " + make +
                ", Driver: " + driver +
                ", Round Time: " + roundTime;
    }
}

// Define the Queue class to implement a queue of Cars
class CarQueue {
    LinkedList<Car> cars = new LinkedList<>();

    // Method to enqueue (add) a car to the queue
    public void enqueue(Car car) {
        // The add() method appends the element at the end of the list.
        cars.add(car);
    }

    // Method to dequeue (remove) a car from the queue
    public Car dequeue() throws Exception {
        // If the queue is empty, throw an exception
        if (isEmpty()) {
            throw new Exception("Cannot dequeue. The queue is empty!");
        }

        // Otherwise, remove the last element in the list, simulating a
queue's behavior.
        return cars.removeLast();
    }

    // Method to check if the queue is empty
    public boolean isEmpty() {
        return cars.isEmpty();
    }
```

```
    // Method to print the cars in the queue
    public void printQueue() {
        // For each car in the cars list, print the car.
        for (Car car : cars) {
            System.out.println(car);
        }
    }
}

// Main class
public class Test {
    public static void main(String[] args) {
        // Array of cars participating in the race
        Car[] carArray = {new Car(1, "Toyota", "Driver1"),
                new Car(2, "Honda", "Driver2"),
                new Car(3, "Ford", "Driver3"),
                new Car(4, "Chevrolet", "Driver4"),
                new Car(5, "Subaru", "Driver5"),
                new Car(6, "Ferrari", "Driver6")};

        // Create a new CarQueue.
        CarQueue carQueue = new CarQueue();

        // Enqueue the cars into the carQueue
        for (Car car : carArray) {
            // For each car in the carArray, enqueue (add) the car to
the queue.
            carQueue.enqueue(car);
        }

        // Begin the 3 rounds
        for (int i = 1; i <= 3; i++) {
            System.out.println("\n\nStarting Round " + i + "...\n");

            // Assign a random time between 30 to 40 seconds to each car
for the round
            for (Car car : carQueue.cars) {
                // The roundTime is assigned a random value between 30
and 40.
                car.roundTime = Math.random() * 10 + 30;
            }

            // Sort the cars based on round time (ascending order)
            // The Java sort method is used, with a lambda function
specifying that the sorting is based on the roundTime of each car.
            carQueue.cars.sort(Comparator.comparingDouble(car ->
car.roundTime));

            // Print the cars in the queue (i.e., the cars in the order
of their round times)
            System.out.println("Cars in the order of their round
times:");
            carQueue.printQueue();

            // Print the top 3 cars (i.e., the cars with the least round
times)
            System.out.println("\nTop 3 Cars:");
            // Get the cars at the 0, 1, and 2 indexes of the sorted
list. These cars have the least round times.
```

```
            try {
                System.out.println("1st: " + carQueue.cars.get(0));
                System.out.println("2nd: " + carQueue.cars.get(1));
                System.out.println("3rd: " + carQueue.cars.get(2));
            } catch (Exception e) {
                System.out.println("There are less than 3 cars left in
the race!");
            }

            // Print the eliminated car (i.e., the car with the maximum
round time)
            if (!carQueue.isEmpty()) {
                System.out.println("\nEliminated Car: " +
carQueue.cars.getLast());
            }

            // Dequeue the eliminated car (i.e., remove the car with the
maximum round time from the queue)
            try {
                carQueue.dequeue();
            } catch (Exception e) {
                System.out.println(e.getMessage());
            }
        }

        // Print the winners of the race
        System.out.println("\n\nWinners of the Race:");
        try {
            System.out.println("1st: " + carQueue.cars.get(0));
            System.out.println("2nd: " + carQueue.cars.get(1));
            System.out.println("3rd: " + carQueue.cars.get(2));
        } catch (Exception e) {
            System.out.println("There are less than 3 cars left in the
race!");
        }
    }
}
```

Output of the Java Data Structure (whole code)

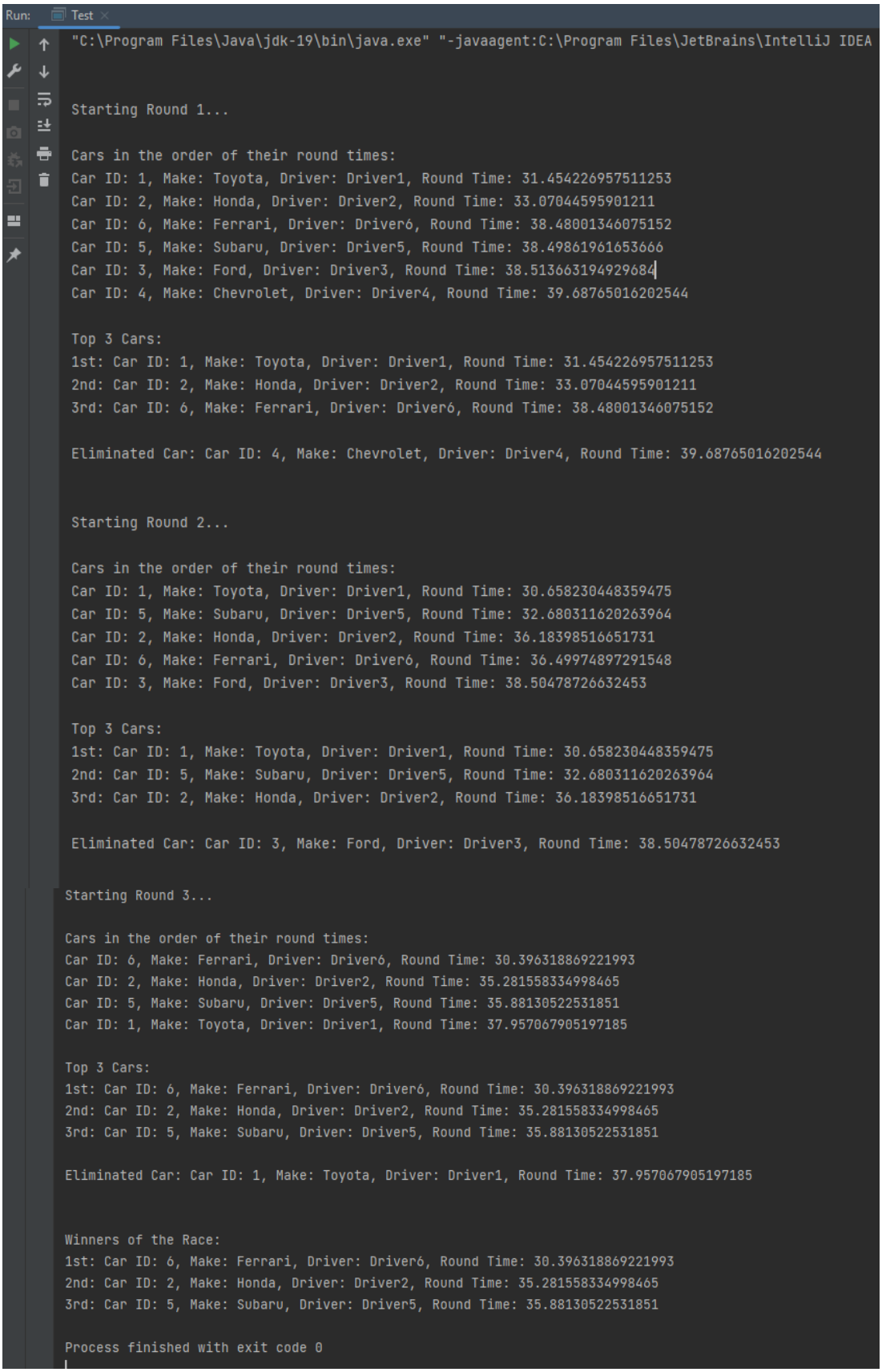


Figure 1. 2Output of Race round results, elimination and finding winners Java Data Structure (whole code)

Task 2

Register, Delete, Search and Print Car

Implement the data structure and its valid operations using Java programming

Car Class →

```
// This is our Car class.
// Each Car has a unique number, brand, sponsor, and driver.
public class Car {
    // Car attributes
    private int carNumber;
    private String carBrand;
    private String carSponsor;
    private String carDriver;

    // This is the constructor for Car class.
    // It is called when we create a new Car object.
    public Car(int number, String brand, String sponsor, String driver)
{
        // Error checking for inputs
        if(number <= 0) {
            throw new IllegalArgumentException("Car number must be
greater than 0.");
        }
        if(brand == null || brand.isEmpty()) {
            throw new IllegalArgumentException("Car brand must not be
empty.");
        }
        if(sponsor == null || sponsor.isEmpty()) {
            throw new IllegalArgumentException("Car sponsor must not be
empty.");
        }
        if(driver == null || driver.isEmpty()) {
            throw new IllegalArgumentException("Car driver must not be
empty.");
        }

        // Assign values to the car attributes
        this.carNumber = number;
        this.carBrand = brand;
        this.carSponsor = sponsor;
        this.carDriver = driver;
    }

    // These are getter methods, used to access private properties of
the class
    public int getCarNumber() {
        return carNumber;
    }
```

```
    public String getCarBrand() {
        return carBrand;
    }

    public String getCarSponsor() {
        return carSponsor;
    }

    public String getCarDriver() {
        return carDriver;
    }
}
```

CarManager Class →

```
// This is our CarManager class.
// It manages operations related to cars.
public class CarManager {
    // Maximum number of cars
    private static final int MAX_CARS = 6;
    // Array to hold the cars
    private Car[] listOfCars = new Car[MAX_CARS];
    // Number of cars currently registered
    private int totalCars = 0;

    // Method to add a new car to the list
    public void registerCar(Car newCar) {
        if(totalCars < MAX_CARS) { // Check if there is room for a new
car
            // Check if a car with the same number already exists
            for(int i = 0; i < totalCars; i++) {
                if(listOfCars[i].getCarNumber() ==
newCar.getCarNumber()) {
                    System.out.println("Registration failed. A car with
the same number is already registered.");
                    return;
                }
            }

            listOfCars[totalCars] = newCar; // Add new car to the next
available position
            totalCars++; // Increment the count of cars
            System.out.println("The car has been successfully
registered.");
        } else {
            System.out.println("Registration failed. The maximum number
of cars have been registered.");
        }
    }

    // Method to delete a car from the list
    public void deleteCar(int carNumber) {
        for(int i = 0; i < totalCars; i++) {
            // If the car is found
            if(listOfCars[i].getCarNumber() == carNumber) {
                // Move the cars in the list to fill the gap
```

```
                for(int j = i; j < totalCars - 1; j++) {
                    listOfCars[j] = listOfCars[j+1];
                }
                // Decrease the count of total cars
                totalCars--;
                System.out.println("The car has been successfully
deleted.");
                return;
            }
        }
        // If the car is not found
        System.out.println("Deletion failed. The specified car does not
exist.");
    }

    // Method to search for a car by its number
    public Car searchCar(int carNumber) {
        // Search for the car in the list
        for(int i = 0; i < totalCars; i++) {
            // If car is found, return the car object
            if(listOfCars[i].getCarNumber() == carNumber) {
                return listOfCars[i];
            }
        }
        // If car is not found, print a message and return null
        System.out.println("Search failed. The specified car was not
found.");
        return null;
    }

    // Method to print the list of cars
    public void printCarList() {
        for(int i = 0; i < totalCars; i++) {
            System.out.println("Car Number: " +
listOfCars[i].getCarNumber()
                    + ", Brand: " + listOfCars[i].getCarBrand()
                    + ", Sponsor: " + listOfCars[i].getCarSponsor()
                    + ", Driver: " + listOfCars[i].getCarDriver());
        }
    }
}
```

Main Class →

```
public class Main {
    public static void main(String[] args) {
        // Create an instance of CarManager
        CarManager carManager = new CarManager();

        // Register some cars
        try {
            carManager.registerCar(new Car(1, "Toyota", "Sponsor1",
"Driver1"));
            carManager.registerCar(new Car(2, "Honda", "Sponsor2",
"Driver2"));
            carManager.registerCar(new Car(3, "Ford", "Sponsor3",
"Driver3"));
```

```
            carManager.registerCar(new Car(4, "Chevrolet", "Sponsor4",
"Driver4"));
            carManager.registerCar(new Car(5, "Mercedes", "Sponsor5",
"Driver5"));
            carManager.registerCar(new Car(6, "BMW", "Sponsor6",
"Driver6"));
        } catch (IllegalArgumentException e) {
            System.out.println(e.getMessage());
        }

        // Try to register a car when the list is already full
        try {
            carManager.registerCar(new Car(7, "Ferrari", "Sponsor7",
"Driver7"));
        } catch (IllegalArgumentException e) {
            System.out.println(e.getMessage());
        }

        // Print the list of cars
        System.out.println("\nInitial list of cars:");
        carManager.printCarList();

        // Search for a car that exists
        Car car = carManager.searchCar(3);
        if(car != null) {
            System.out.println("\nFound Car: " + car.getCarBrand());
        }

        // Search for a car that does not exist
        car = carManager.searchCar(7);
        if(car != null) {
            System.out.println("Found Car: " + car.getCarBrand());
        }

        // Delete a car that exists
        carManager.deleteCar(2);

        // Try to delete a car that does not exist
        carManager.deleteCar(7);

        // Print the list of cars again to check if the car was deleted
        System.out.println("\nList of cars after deletion:");
        carManager.printCarList();

        // Try to register a car with a number that already exists
        try {
            carManager.registerCar(new Car(1, "Audi", "Sponsor8",
"Driver8"));
        } catch (IllegalArgumentException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

The output of the Java Program

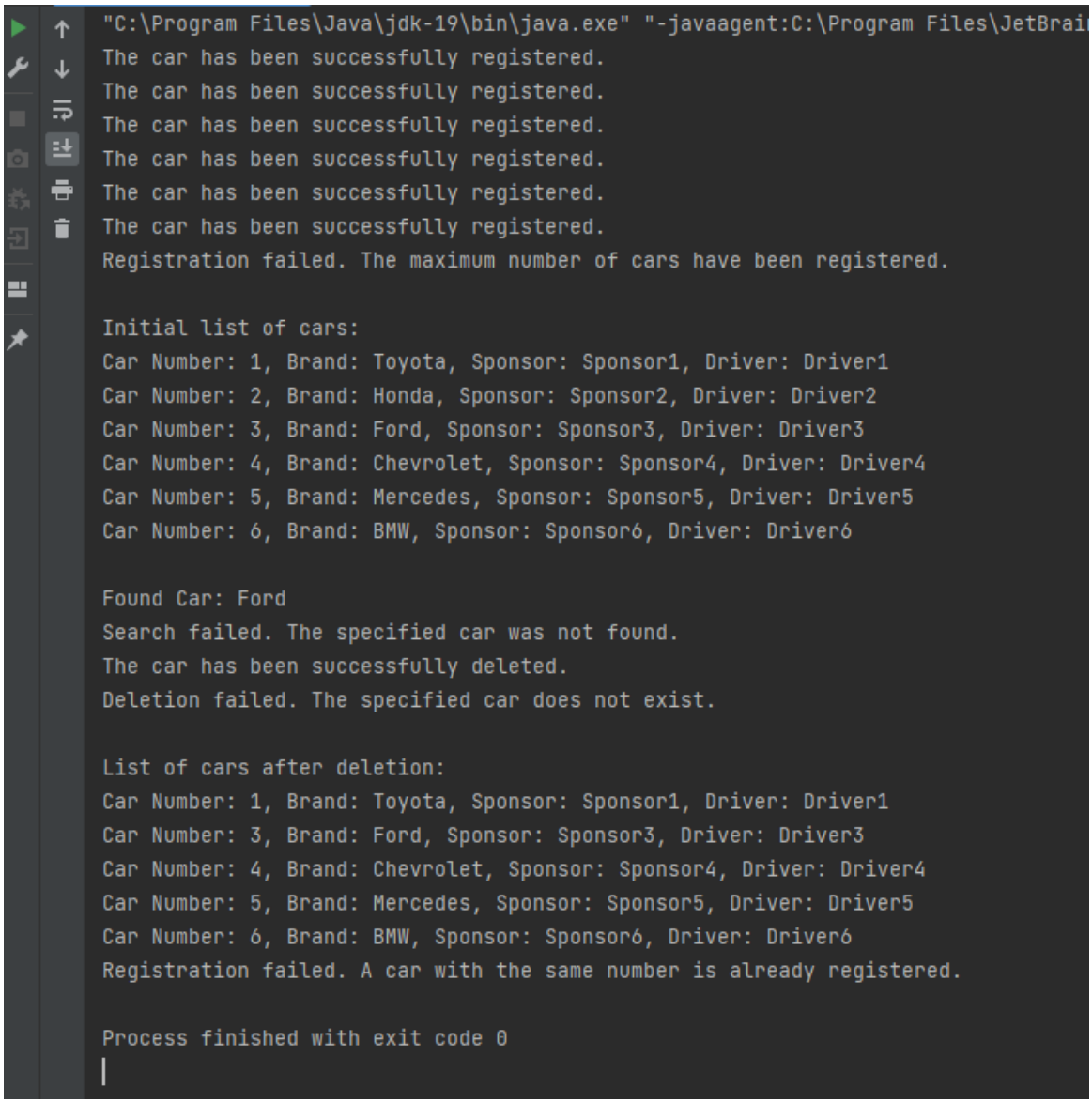


Figure 2. 1 Register, Delete, Search and Print Car the output of the Java Program

Error Handling

Error handling is an essential part of programming. It helps deal with unforeseen situations that can occur during the execution of your program, such as invalid inputs, logical errors, or external issues like loss of network connectivity, for example. In this scenario, I've employed error handling in several places to ensure the program runs smoothly even when unexpected situations arise.

Error Handling in Class Constructors:

In the 'Car' class constructor, I've added error handling to verify that the inputs for car number, brand, sponsor, and driver are valid. If they are not, the program throws an 'IllegalArgumentException' with an appropriate error message. This way, the program prevents the creation of 'Car' objects with invalid states right from the start.

```
public Car(int number, String brand, String sponsor, String driver) {
        // Error checking for inputs
        if(number <= 0) {
        throw new IllegalArgumentException("Car number must be greater
than 0.");
        }
        if(brand == null || brand.isEmpty()) {
        throw new IllegalArgumentException("Car brand must not be
empty.");
        }
        if(sponsor == null || sponsor.isEmpty()) {
        throw new IllegalArgumentException("Car sponsor must not be
empty.");
        }
        if(driver == null || driver.isEmpty()) {
        throw new IllegalArgumentException("Car driver must not be
empty.");
        }
        //...rest of the constructor...
        }
```

Error Handling in Registration, Deletion, and Search Operations:

In the 'CarManager' class, error handling is used to manage several scenarios:

- Registration: When registering a car, I first check if there's room for another car. If the maximum limit has been reached, an error message is displayed, and the method returns without adding the new car. Then, I also check if a car with the same number already exists to avoid duplicates. If it does, another error message is displayed, and the method returns without adding the new car.

```
public void registerCar(Car newCar) {
        //...error handling for maximum limit and duplicate
cars...
        }
```

- Deletion: When trying to delete a car, if the car with the given number doesn't exist in the list, an error message is displayed, and the method returns without deleting anything.

```
public void deleteCar(int carNumber) {
        //...error handling for non-existing cars...
        }
```

- Search: Similarly, during a car search, if the car with the specified number isn't found in the list, an error message is shown, and the method returns null.

```
public Car searchCar(int carNumber) {
        //...error handling for non-existing cars...
        }
```

Error Handling in the Main Class:

In the ‘Main’ class, while registering cars, I've wrapped the registration code inside a ‘try-catch’ block to catch any ‘IllegalArgumentException’ that might be thrown due to invalid inputs for the ‘Car’ constructor. If such an exception is thrown, the catch block catches it and prints its error message.

```
try {
carManager.registerCar(new Car(1, "Toyota", "Sponsor1", "Driver1"));
//...rest of the registrations...
} catch (IllegalArgumentException e) {
System.out.println(e.getMessage());
}
```

This way, I can provide immediate feedback about what went wrong without stopping the entire program.

Through these various error handling measures, the car racing application is well-equipped to deal with a range of unexpected situations. By validating inputs, checking conditions before performing operations, and catching exceptions when they occur, the application ensures a smooth user experience and aids in the detection and debugging of errors.

Test Plan

Introduction:

The purpose of this test plan is to define the strategies and criteria used to ensure that the 'Register, Delete, Search and Print Car' application functions as intended. The system allows users to manage a list of a maximum of 6 cars, identified by unique car numbers.

Objectives:

The main objectives of the testing process include validating that the system:

- Correctly registers up to 6 cars.
- Prevents registration of more than 6 cars.
- Successfully deletes registered cars.
- Handles attempts to delete cars not present in the list.
- Correctly searches for and retrieves cars in the list.
- Manages unsuccessful searches appropriately.
- Prints the correct list of registered cars.

Risks and Contingencies:

Risks include improper input handling and potential issues related to array manipulation, including registration and deletion operations. If any bugs are identified during testing, they will be logged and returned to the development team for fixing. Retesting will be done post bug-fixing.

Test Items:

The items to be tested are the 'registerCar', 'deleteCar', 'searchCar' and 'printCarList' methods of the 'CarManager' class.

Features to be Tested:

- Registering a new car: Ensure that the system properly adds a new car to the list.
- Registering a duplicate car: The system should prevent adding a car with a number that's already registered.
- Registering more than 6 cars: The system shouldn’t allow register more than 6 cars.
- Deleting a registered car: The system should correctly remove a car from the list.
- Deleting a non-registered car: The system should handle attempts to delete a car that's not in the list.
- Searching for a registered car: The system should correctly find and return a car in the list.
- Searching for a non-registered car: The system should return null and print an error message when a search for a non-registered car is attempted.
- Printing the list of cars: The system should correctly print all registered cars.

Test Techniques:

Functional testing will be performed on the application, manually testing each operation (register, delete, search, print) to ensure it behaves as expected.

Test Cases:

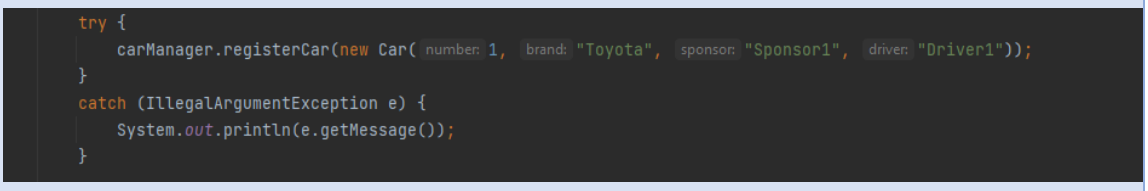
1. Register a new car and verify it's added to the list.
2. Attempt to register a car with a number that's already registered and confirm the system prevents it.
3. Attempt to register more than 6 cars and confirm the system prevents it.
4. Delete a registered car and verify it's removed from the list.
5. Attempt to delete a non-registered car and confirm the system handles it correctly.
6. Search for a registered car and verify the system finds and returns it correctly.
7. Search for a non-registered car and confirm the system returns null and prints an error message.
8. After each registration and deletion, print the list of cars and confirm it's correct.

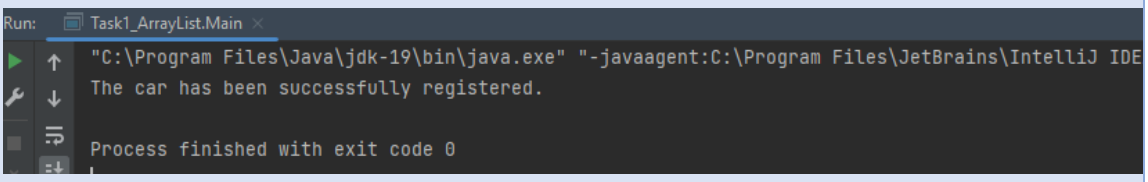
Pass/Fail Criteria:

Each test case will be marked as pass or fail based on whether the observed result matches the expected result. A passed test case means the system behaves as expected under the given conditions, while a failed test case indicates an issue that needs to be addressed by the development team.

Test Cases

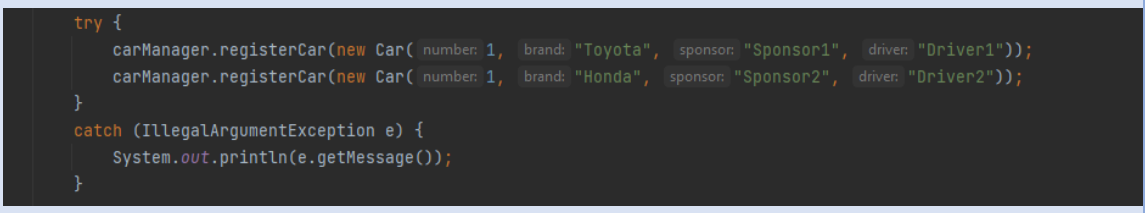
Table 2. 1 Test Case ID: TC001

Test Case ID: TC001
Objective: To validate the functionality of the 'registerCar' operation.
Test Items: 'registerCar' method of the 'CarManager' class.
Input: Create a new Car object with the following details: • Car Number: 1 • Car Brand: "Toyota" • Car Sponsor: "Sponsor1" • Car Driver: "Driver1"  `try { carManager.registerCar(new Car( number: 1, brand: "Toyota", sponsor: "Sponsor1", driver: "Driver1")); } catch (IllegalArgumentException e) { System.out.println(e.getMessage()); }`
Steps: • Initialize an instance of the 'CarManager' class. • Call the 'registerCar' method with the newly created Car object as an argument.
Expected Output: The system should successfully register the new car and return a success message.
Pass/Fail Criteria: • Pass: If the car is successfully registered and the system returns the success message "The car has been successfully registered.". • Fail: If the car is not registered or if the system doesn't return the expected message.

Actual Output:
 
`Run: Task1_ArrayList.Main`
`"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDE`
`The car has been successfully registered.`
`Process finished with exit code 0`

Pass/Fail: Pass

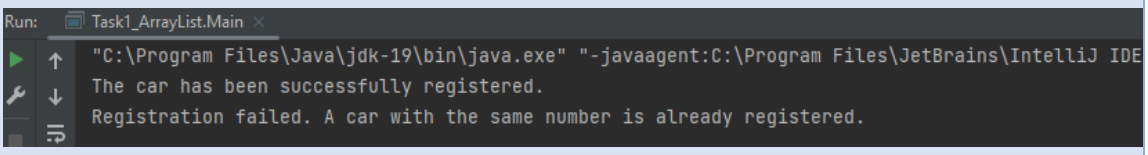
Table 2. 2 Test Case ID: TC002

Test Case ID: TC002
Objective: To validate the system's behavior when trying to register a car with a number that already exists in the system.
Test Items: 'registerCar' method of the 'CarManager' class.
Input: Create a new Car object with the following details: • Car Number: 1 • Car Brand: "Toyota" • Car Sponsor: "Sponsor1" • Car Driver: "Driver1" Create another Car object with the same number but different details: • Car Number: 1 • Car Brand: "Honda" • Car Sponsor: "Sponsor2" • Car Driver: "Driver2" 
Steps: • Initialize an instance of the 'CarManager' class. • Call the 'registerCar' method with the first Car object as an argument. • Call the 'registerCar' method again with the second Car object as an argument.
Expected Output: The system should successfully register the first car but should refuse to register the second car, returning a failure message such as "Registration failed. A car with the same number is already registered."

Pass/Fail Criteria:

- Pass: If the first car is registered successfully and the second registration attempt fails with the expected error message.
- Fail: If the second car gets registered despite having the same number as an existing car or if the system doesn't return the expected failure message.

Actual Output:



Pass/Fail: Pass

Table 2. 3 Test Case ID: TC003

Test Case ID: TC003
Objective: To validate the system's behavior when trying to register more than the maximum allowed number of cars (6).
Test Items: 'registerCar' method of the 'CarManager' class.
Input: Create seven Car objects with the following details: • Car Number: 1, Car Brand: "Toyota", Car Sponsor: "Sponsor1", Car Driver: "Driver1" • Car Number: 2, Car Brand: "Honda", Car Sponsor: "Sponsor2", Car Driver: "Driver2" • Car Number: 3, Car Brand: "Ford", Car Sponsor: "Sponsor3", Car Driver: "Driver3" • Car Number: 4, Car Brand: "Chevrolet", Car Sponsor: "Sponsor4", Car Driver: "Driver4" • Car Number: 5, Car Brand: "Mercedes", Car Sponsor: "Sponsor5", Car Driver: "Driver5" • Car Number: 6, Car Brand: "BMW", Car Sponsor: "Sponsor6", Car Driver: "Driver6" • Car Number: 7, Car Brand: "Audi", Car Sponsor: "Sponsor7", Car Driver: "Driver7"

```
try {
    carManager.registerCar(new Car( number: 1,  brand: "Toyota",  sponsor: "Sponsor1",  driver: "Driver1"));
    carManager.registerCar(new Car( number: 2,  brand: "Honda",  sponsor: "Sponsor2",  driver: "Driver2"));
    carManager.registerCar(new Car( number: 3,  brand: "Ford",  sponsor: "Sponsor3",  driver: "Driver3"));
    carManager.registerCar(new Car( number: 4,  brand: "Chevrolet",  sponsor: "Sponsor4",  driver: "Driver4"));
    carManager.registerCar(new Car( number: 5,  brand: "Mercedes",  sponsor: "Sponsor5",  driver: "Driver5"));
    carManager.registerCar(new Car( number: 6,  brand: "BMW",  sponsor: "Sponsor6",  driver: "Driver6"));
    carManager.registerCar(new Car( number: 7,  brand: "Audi",  sponsor: "Sponsor6",  driver: "Driver7"));
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
}
```

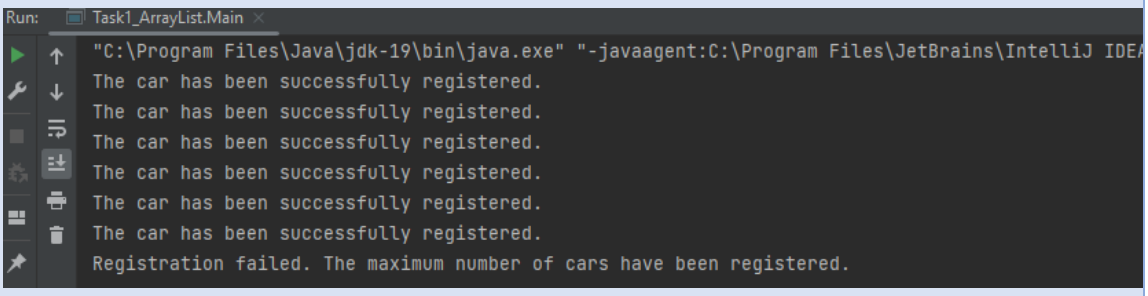
Steps: • Initialize an instance of the 'CarManager' class. • Call the 'registerCar' method with each of the seven Car objects as arguments.

Expected Output: The system should successfully register the first six cars, but should refuse to register the seventh car, returning a failure message such as "Registration failed. The maximum number of cars have been registered."

Pass/Fail Criteria:

- Pass: If the first six cars are registered successfully and the seventh registration attempt fails with the expected error message.
- Fail: If the seventh car gets registered despite exceeding the maximum limit of cars or if the system doesn't return the expected failure message.

Actual Output:



Pass/Fail: Pass

Table 2. 4 Test Case ID: TC004

Test Case ID: TC004
Objective: To validate the system's behavior when trying to delete a registered car.
Test Items: 'deleteCar' method of the 'CarManager' class.
Input: Create three Car objects with the following details: • Car Number: 1, Car Brand: "Toyota", Car Sponsor: "Sponsor1", Car Driver: "Driver1" • Car Number: 2, Car Brand: "Honda", Car Sponsor: "Sponsor2", Car Driver: "Driver2" • Car Number: 3, Car Brand: "Ford", Car Sponsor: "Sponsor3", Car Driver: "Driver3" 
Steps: • Initialize an instance of the 'CarManager' class. • Call the 'registerCar' method with each of the three Car objects as arguments. • Call the 'deleteCar' method with the car number of the second car (2) as an argument.
Expected Output: The system should successfully delete the car with the number 2, returning a success message such as "The car has been successfully deleted."

```
// Register some cars
try {
    carManager.registerCar(new Car( number: 1,  brand: "Toyota",  sponsor: "Sponsor1",  driver: "Driver1"));
    carManager.registerCar(new Car( number: 2,  brand: "Honda",  sponsor: "Sponsor2",  driver: "Driver2"));
    carManager.registerCar(new Car( number: 3,  brand: "Ford",  sponsor: "Sponsor3",  driver: "Driver3"));
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
}

// Delete a car that exists
carManager.deleteCar( carNumber: 2);
```

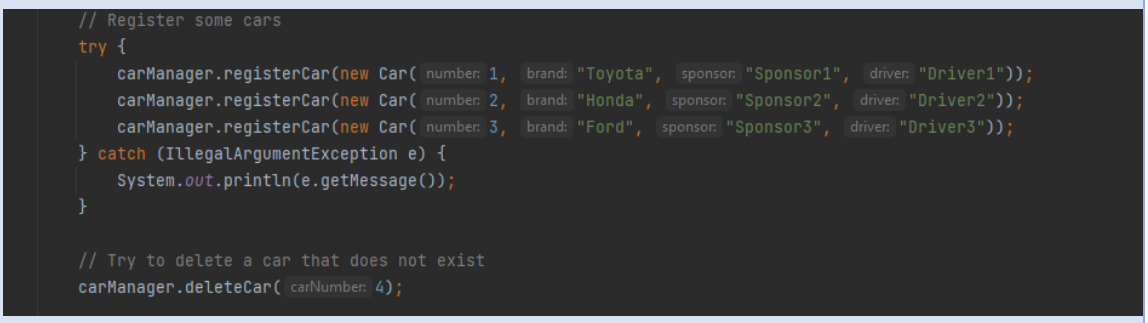
Pass/Fail Criteria:

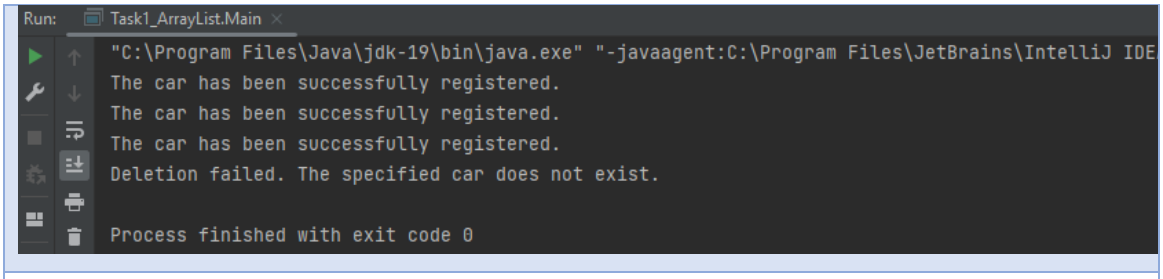
- Pass: If the car with the number 2 is deleted successfully and the system returns the expected success message.
- Fail: If the car with the number 2 is not deleted or if the system doesn't return the expected success message.

Actual Output:

Pass/Fail: Pass

Table 2. 5 Test Case ID: TC005

Test Case ID: TC005
Objective: To validate the system's behavior when trying to delete a non-registered car.
Test Items: 'deleteCar' method of the 'CarManager' class.
Input: Create three Car objects with the following details: • Car Number: 1, Car Brand: "Toyota", Car Sponsor: "Sponsor1", Car Driver: "Driver1" • Car Number: 2, Car Brand: "Honda", Car Sponsor: "Sponsor2", Car Driver: "Driver2" • Car Number: 3, Car Brand: "Ford", Car Sponsor: "Sponsor3", Car Driver: "Driver3" 
Steps: • Initialize an instance of the 'CarManager' class. • Call the 'registerCar' method with each of the three Car objects as arguments. • Call the 'deleteCar' method with the car number 4 (a car number that does not exist in the registered list) as an argument.
Expected Output: The system should return an error message indicating that the specified car does not exist, such as "Deletion failed. The specified car does not exist."
Pass/Fail Criteria: • Pass: If no car is deleted and the system returns the expected failure message. • Fail: If a car is deleted or if the system doesn't return the expected failure message.
Actual Output:



Pass/Fail: Pass

Table 2. 6 Test Case ID: TC006

Test Case ID: TC006
Objective: To validate the system's behavior when searching for a registered car.
Test Items: 'searchCar' method of the 'CarManager' class.
Input: Create three Car objects with the following details: • Car Number: 1, Car Brand: "Toyota", Car Sponsor: "Sponsor1", Car Driver: "Driver1" • Car Number: 2, Car Brand: "Honda", Car Sponsor: "Sponsor2", Car Driver: "Driver2" • Car Number: 3, Car Brand: "Ford", Car Sponsor: "Sponsor3", Car Driver: "Driver3" 
Steps: • Initialize an instance of the 'CarManager' class. • Call the 'registerCar' method with each of the three Car objects as arguments. • Call the 'searchCar' method with the car number 2 (a car number that exists in the registered list) as an argument.
Expected Output: The system should return the Car object associated with car number 2, containing its brand.
Pass/Fail Criteria: • Pass: If the system returns the expected Car object with the correct details. • Fail: If the system does not return the expected Car object, or if the details are incorrect.

```
// Register some cars
try {
    carManager.registerCar(new Car( number: 1,  brand: "Toyota",  sponsor: "Sponsor1",  driver: "Driver1"));
    carManager.registerCar(new Car( number: 2,  brand: "Honda",  sponsor: "Sponsor2",  driver: "Driver2"));
    carManager.registerCar(new Car( number: 3,  brand: "Ford",  sponsor: "Sponsor3",  driver: "Driver3"));
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
}

// Search for a car that exists
Car car = carManager.searchCar( carNumber: 2);
if(car != null) {
    System.out.println("\nFound Car: " + car.getCarBrand());
}
```

Actual Output:

Pass/Fail: Pass

Table 2. 7 Test Case ID: TC007

Test Case ID: TC007
Objective: To validate the system's behavior when searching for a car that is not registered.
Test Items: 'searchCar' method of the 'CarManager' class.

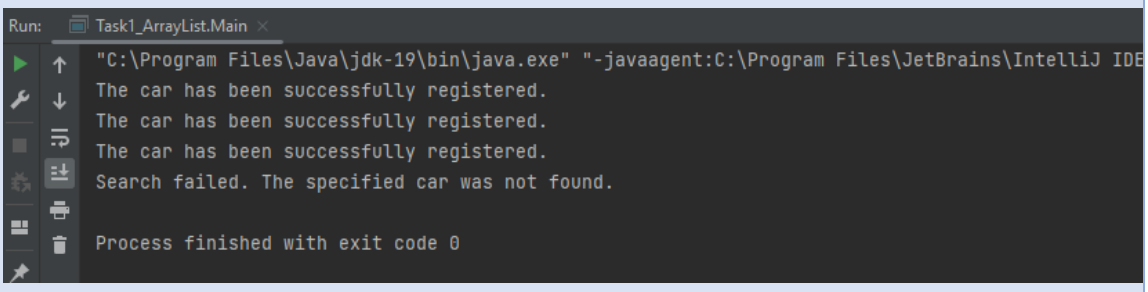
Input:
Create three Car objects with the following details:
• Car Number: 1, Car Brand: "Toyota", Car Sponsor: "Sponsor1", Car Driver: "Driver1"
• Car Number: 2, Car Brand: "Honda", Car Sponsor: "Sponsor2", Car Driver: "Driver2"
• Car Number: 3, Car Brand: "Ford", Car Sponsor: "Sponsor3", Car Driver: "Driver3"
 	
`// Register some cars`
`try {`
`  carManager.registerCar(new Car( number: 1, brand: "Toyota", sponsor: "Sponsor1", driver: "Driver1"));`
`  carManager.registerCar(new Car( number: 2, brand: "Honda", sponsor: "Sponsor2", driver: "Driver2"));`
`  carManager.registerCar(new Car( number: 3, brand: "Ford", sponsor: "Sponsor3", driver: "Driver3"));`
`}`
`catch (IllegalArgumentException e) {`
`  System.out.println(e.getMessage());`
`}`
 	
`// Search for a car that does not exist`
`Car car = carManager.searchCar( carNumber: 3);`
`car = carManager.searchCar( carNumber: 7);`
`if(car != null) {`
`  System.out.println("Found Car: " + car.getCarBrand());`
`}`

Steps: • Initialize an instance of the 'CarManager' class. • Call the 'registerCar' method with each of the three Car objects as arguments. • Call the 'searchCar' method with the car number 7 (a car number that does not exist in the registered list) as an argument.
Expected Output: The system should return a message stating "Search failed. The specified car was not found." and return null.

Pass/Fail Criteria:

- Pass: If the system returns the expected message and null.
- Fail: If the system does not return the expected message or does not return null.

Actual Output:



Pass/Fail: Pass

Table 2. 8 Test Case ID: TC008

Test Case ID: TC008
Objective: To validate the system's ability to correctly print the list of registered cars.
Test Items: 'printCarList' method of the 'CarManager' class.
Input: Create three Car objects with the following details: • Car Number: 1, Car Brand: "Toyota", Car Sponsor: "Sponsor1", Car Driver: "Driver1" • Car Number: 2, Car Brand: "Honda", Car Sponsor: "Sponsor2", Car Driver: "Driver2" • Car Number: 3, Car Brand: "Ford", Car Sponsor: "Sponsor3", Car Driver: "Driver3" 
Steps: • Initialize an instance of the 'CarManager' class. • Call the 'registerCar' method with each of the three Car objects as arguments. • Call the 'printCarList' method.
Expected Output: The system should print out the list of all registered cars in the following format: "Car Number: 1, Brand: Toyota, Sponsor: Sponsor1, Driver: Driver1 Car Number: 2, Brand: Honda, Sponsor: Sponsor2, Driver: Driver2 Car Number: 3, Brand: Ford, Sponsor: Sponsor3, Driver: Driver3"

Pass/Fail Criteria:

- Pass: If the system correctly prints out the details of all registered cars as per the specified format.
- Fail: If the system does not print the list of cars correctly or if the format is incorrect.

Actual Output:

Pass/Fail: Pass

Race round results, elimination and finding winners

Implement the data structure and its valid operations using Java programming

```
// Import necessary classes
import java.util.LinkedList;
import java.util.Comparator;

// Define Car class with appropriate properties and methods
class Car {
    int id;
    String make;
    String driver;
    double roundTime;

    // Car class constructor
    public Car(int id, String make, String driver) {
        this.id = id;
        this.make = make;
        this.driver = driver;
    }

    // toString method to print details of car in a readable format
    @Override
    public String toString() {
        return "Car ID: " + id +
                ", Make: " + make +
                ", Driver: " + driver +
                ", Round Time: " + roundTime;
    }
}

// Define the Queue class to implement a queue of Cars
class CarQueue {
    LinkedList<Car> cars = new LinkedList<>();

    // Method to enqueue (add) a car to the queue
    public void enqueue(Car car) {
        // The add() method appends the element at the end of the list.
        cars.add(car);
    }

    // Method to dequeue (remove) a car from the queue
    public Car dequeue() throws Exception {
        // If the queue is empty, throw an exception
        if (isEmpty()) {
            throw new Exception("Cannot dequeue. The queue is empty!");
        }

        // Otherwise, remove the last element in the list, simulating a
queue's behavior.
        return cars.removeLast();
    }

    // Method to check if the queue is empty
```

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

    // Method to print the cars in the queue
    public void printQueue() {
        // For each car in the cars list, print the car.
        for (Car car : cars) {
            System.out.println(car);
        }
    }
}

// Main class
public class Test {
    public static void main(String[] args) {
        // Array of cars participating in the race
        Car[] carArray = {new Car(1, "Toyota", "Driver1"),
                new Car(2, "Honda", "Driver2"),
                new Car(3, "Ford", "Driver3"),
                new Car(4, "Chevrolet", "Driver4"),
                new Car(5, "Subaru", "Driver5"),
                new Car(6, "Ferrari", "Driver6")};

        // Create a new CarQueue.
        CarQueue carQueue = new CarQueue();

        // Enqueue the cars into the carQueue
        for (Car car : carArray) {
            // For each car in the carArray, enqueue (add) the car to
the queue.
            carQueue.enqueue(car);
        }

        // Begin the 3 rounds
        for (int i = 1; i <= 3; i++) {
            System.out.println("\n\nStarting Round " + i + "...\n");

            // Assign a random time between 30 to 40 seconds to each car
for the round
            for (Car car : carQueue.cars) {
                // The roundTime is assigned a random value between 30
and 40.
                car.roundTime = Math.random() * 10 + 30;
            }

            // Sort the cars based on round time (ascending order)
            // The Java sort method is used, with a lambda function
specifying that the sorting is based on the roundTime of each car.
            carQueue.cars.sort(Comparator.comparingDouble(car ->
car.roundTime));

            // Print the cars in the queue (i.e., the cars in the order
of their round times)
            System.out.println("Cars in the order of their round
times:");
            carQueue.printQueue();

            // Print the top 3 cars (i.e., the cars with the least round
```

```
times)
            System.out.println("\nTop 3 Cars:");
            // Get the cars at the 0, 1, and 2 indexes of the sorted
list. These cars have the least round times.
            try {
                System.out.println("1st: " + carQueue.cars.get(0));
                System.out.println("2nd: " + carQueue.cars.get(1));
                System.out.println("3rd: " + carQueue.cars.get(2));
            } catch (Exception e) {
                System.out.println("There are less than 3 cars left in
the race!");
            }

            // Print the eliminated car (i.e., the car with the maximum
round time)
            if (!carQueue.isEmpty()) {
                System.out.println("\nEliminated Car: " +
carQueue.cars.getLast());
            }

            // Dequeue the eliminated car (i.e., remove the car with the
maximum round time from the queue)
            try {
                carQueue.dequeue();
            } catch (Exception e) {
                System.out.println(e.getMessage());
            }
        }

        // Print the winners of the race
        System.out.println("\n\nWinners of the Race:");
        try {
            System.out.println("1st: " + carQueue.cars.get(0));
            System.out.println("2nd: " + carQueue.cars.get(1));
            System.out.println("3rd: " + carQueue.cars.get(2));
        } catch (Exception e) {
            System.out.println("There are less than 3 cars left in the
race!");
        }
    }
}
```

The output of the Java Program

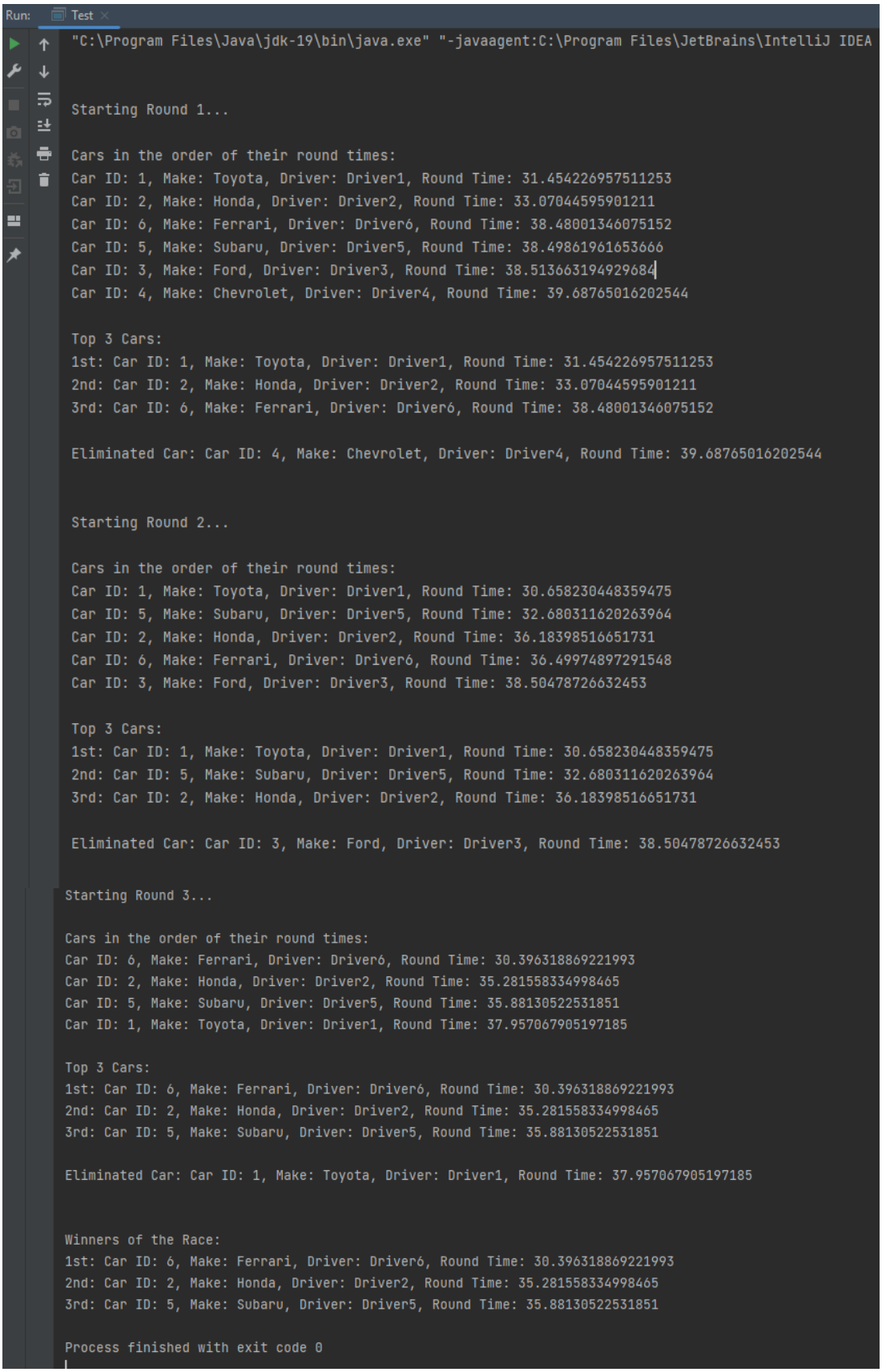


Figure 2. 2 Race round results, elimination and finding winners the output of the Java Program

Error Handling

Error handling is an important aspect of programming. It ensures that the program can handle anticipated issues gracefully, without unexpected crashes. In the provided code, error handling is used to ensure the program can deal with a specific anticipated issue: trying to remove a car from an empty queue.

Error Handling in the CarQueue Class (Queue Underflow):

In the 'dequeue()' method of the 'CarQueue' class, a queue underflow condition is handled. This occurs when we try to remove an item from an empty queue. The 'CarQueue' class implements a queue data structure for 'Car' objects using a 'LinkedList'. The method 'dequeue()' is designed to remove and return the last 'Car' object in the 'LinkedList'. However, if the list is empty, there's a problem. This situation is known as a queue underflow.

Here's how the code handles it:

```
public Car dequeue() throws Exception {
    // If the queue is empty, throw an exception
    if (isEmpty()) {
        throw new Exception("Cannot dequeue. The queue is empty!");
    }

    // Otherwise, remove the last element in the list, simulating a
queue's behavior.
    return cars.removeLast();
}
```

The 'dequeue()' method first checks whether the queue is empty by calling the 'isEmpty()' method. If 'isEmpty()' returns 'true', it implies there are no more 'Car' objects in the queue. In this case, the 'dequeue()' method throws an 'Exception' with a descriptive error message. By doing this, it prevents the calling code from attempting to remove a car from an empty queue, which would result in an error.

Error Handling in the main Method:

In the ‘main()’ method, we use a try-catch block to handle potential exceptions that could arise when attempting to access elements at certain positions in the ‘LinkedList’ that may not exist.

Take a look at the following code part:

```
try {
        System.out.println("1st: " + carQueue.cars.get(0));
        System.out.println("2nd: " + carQueue.cars.get(1));
        System.out.println("3rd: " + carQueue.cars.get(2));
        } catch (Exception e) {
        System.out.println("There are less than 3 cars left in the
race!");
        }
```

Here, if the ‘LinkedList’ contains fewer than three elements, trying to access the non-existent elements will throw an ‘IndexOutOfBoundsException’. The try-catch block catches this exception and prints a relevant error message instead of letting the program crash.

Ensuring the Queue Isn't Empty:

Again in the ‘main()’ method, another try-catch block handles the potential error of dequeuing from an empty queue.

```
try {
        carQueue.dequeue();
        } catch (Exception e) {
        System.out.println(e.getMessage());
        }
```

In this instance, if the ‘dequeue()’ method throws an exception (because the queue is empty), the catch block catches it and prints the error message from the exception, again preventing a crash and providing useful information to the user or developer.

In conclusion, this code uses error handling to protect against queue underflow (in the ‘CarQueue’ class) and ‘IndexOutOfBoundsException’ (in the ‘main()’ method). By doing

so, it ensures that the program can handle these potential issues gracefully, improving the program's robustness and user-friendliness. Error handling is a critical part of building reliable, real-world software applications. It ensures that when things go wrong, they do so in a controlled and expected manner, improving the debuggability and overall quality of the code.

Test Plan

Introduction:

This test plan focuses on the code of a car racing game simulation. The program creates an array of Car objects, enqueues them into a queue, and simulates 3 rounds of the race, where each round eliminates the car with the slowest round time.

Objectives:

The primary objectives of this testing phase are:

- To ensure that all components of the program function as expected.
- To validate that the race's mechanics (rounds, timing, elimination) work correctly.
- To ensure that the exception handling is effectively preventing and responding to potential errors.

Risks and Contingencies:

- Risks: The main risk is having bugs that could cause the program to crash or behave unpredictably. The risk also includes misimplementation of the queue operations and the race mechanics, and ineffective error handling.
- Contingencies: In case any significant defects are identified, the code should be revisited and debugged. Proper error handling should be ensured and the queue operations should be accurately implemented.

Test Items:

1) `Car` class: id, make, driver, roundTime properties, and `toString()` method.
2) `CarQueue` class: cars property, `enqueue()`, `dequeue()`, `isEmpty()`, and `printQueue()` methods.
3) Main function: Creating and initializing Car objects, enqueuing and dequeuing cars, race mechanics, exception handling.

Features to be Tested:

- Initialization of the Car objects.
- Functioning of the CarQueue - enqueueing and dequeueing of cars, checking if the queue is empty.
- Correct assignment of round times.
- Proper sorting of cars based on round times.
- Correct identification and elimination of the car with the maximum round time.
- Exception handling when accessing elements from an empty list or dequeuing from an empty queue.
- Correct determination of winners.

Test Techniques:

1) Unit Testing: To test individual classes and methods.
2) Integration Testing: To test the interaction of different parts of the program.
3) Boundary Testing: To test the limits of the race mechanics such as maximum and minimum number of cars, zero cars, etc.

Test Cases:

1) Initialization of the Car objects with valid values.
2) Enqueueing cars, ensuring proper functioning.
3) Dequeuing cars, ensuring proper functioning.
4) Test the round time assignment and sorting mechanism, and check if cars are sorted correctly.
5) Test the elimination mechanism, and ensure the correct car is eliminated each round.
6) Test the queue with no cars, and check if exceptions are handled correctly.
7) Test the winner determination, and ensure the correct cars are declared as winners.
8) Test with only one car, ensure the race still runs and the car is declared the winner.
9) Test with more than six cars, ensuring the race can handle larger queues.

Pass/Fail Criteria:

- Pass: The program runs without crashing or throwing any unhandled exceptions. All the features and methods work correctly, race mechanics are accurately implemented, and the winners are correctly determined.
- Fail: The program crashes or throws any unhandled exceptions. Any of the features or methods doesn't work correctly, race mechanics are inaccurately implemented, or the winners are incorrectly determined.

Test Cases

Table 2. 9 Test Case ID: TC001

Test Case ID: TC001
Objective: To validate the initialization of the `Car` objects with valid values.
Test Items: Constructor method of the `Car` class.
Input: Create a new `Car` object with the following details: Car ID: 1 Car Make: "Toyota" Car Driver: "Driver1"
Steps: 1. Initialize a new `Car` object using the constructor with the given input values. 2. Print the `Car` object details using the `toString()` method.

```
// Main class
public class Test {
    public static void main(String[] args) {
        // Create a single car object for the test case
        Car car = new Car(1, "Toyota", "Driver1");

        // Print the details of the car object using the toString()
method
        System.out.println(car);

        // Check the output string
        if(car.toString().equals("Car ID: 1, Make: Toyota, Driver:
Driver1, Round Time: 0.0")){
            System.out.println("Pass: The car object was correctly
initialized and the toString() method returned the correct string.");
        } else {
            System.out.println("Fail: The car object was not correctly
initialized or the toString() method did not return the correct
string.");
        }
    }
}
```

This modified ‘main’ method creates a single ‘Car’ object with the given values and prints the details of the ‘Car’ object using its ‘toString()’ method. It then checks whether

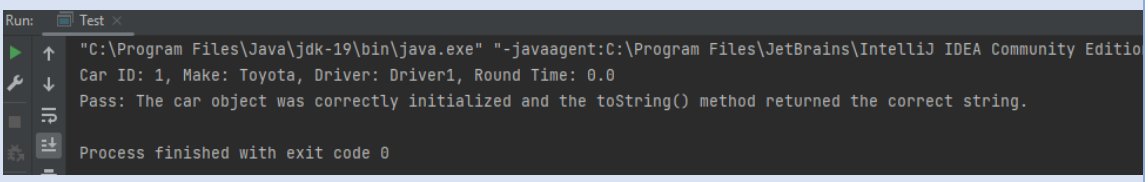
the output string is as expected, and if it is, it prints a pass message; otherwise, it prints a fail message. This should satisfy the test case TC001.

Expected Output: The `Car` object should be successfully created with the provided values, and the `toString()` method should return a string in the following format: "Car ID: 1, Make: Toyota, Driver: Driver1, Round Time: 0.0".

Pass/Fail Criteria:

- Pass: If the `Car` object is successfully created with the provided values, and the `toString()` method returns the correct string.
- Fail: If the `Car` object is not correctly created with the provided values, or if the `toString()` method doesn't return the correct string.

Actual Output:



Pass/Fail: Pass

Table 2. 10 Test Case ID: TC002

Test Case ID: TC002
Objective: To validate the functionality of the 'enqueue' operation.
Test Items: 'enqueue' method of the 'CarQueue' class.
Input: Create two new Car objects with the following details: Car1: Car Number: 1 Car Brand: "Toyota" Car Driver: "Driver1" Car2: Car Number: 2 Car Brand: "Honda" Car Driver: "Driver2"
Steps: 1. Initialize an instance of the 'CarQueue' class. 2. Call the 'enqueue' method with Car1 object as an argument. 3. Call the 'enqueue' method with Car2 object as an argument. 4. Verify that both cars have been added to the queue in the correct order.

```
public class Test {
    public static void main(String[] args) {
        // Create two new Car objects
        Car car1 = new Car(1, "Toyota", "Driver1");
        Car car2 = new Car(2, "Honda", "Driver2");

        // Initialize an instance of the 'CarQueue' class
        CarQueue carQueue = new CarQueue();

        // Enqueue the cars
        carQueue.enqueue(car1);
        carQueue.enqueue(car2);

        // Print the cars in the queue
        System.out.println("The queue after enqueueing two cars:");
        carQueue.printQueue();

        // Check if the cars were enqueued in the correct order
        if (carQueue.cars.get(0).equals(car1) &&
carQueue.cars.get(1).equals(car2)) {
            System.out.println("\nTest Passed: Both cars were enqueued
in the correct order.");
```

```
        } else {
            System.out.println("\nTest Failed: Cars were not enqueued
in the correct order.");
        }
    }
}
```

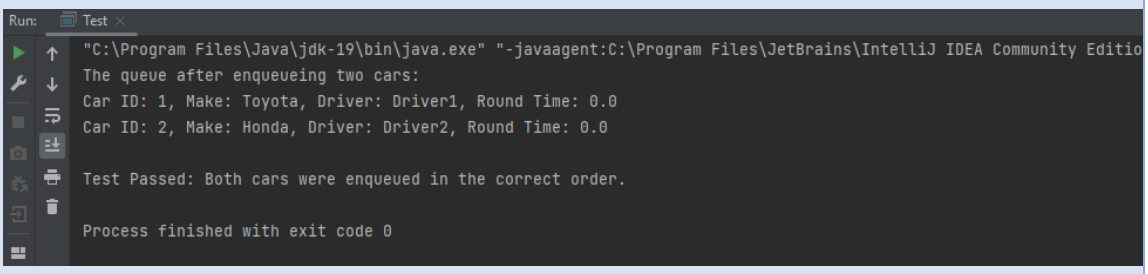
In this code, the ‘Test’ class creates two cars, enqueues them into the ‘CarQueue’, and then prints out the queue. It then checks if the first and second cars in the queue match the expected cars. If they do, the test is passed; otherwise, the test is failed.

Expected Output: The system should successfully add both cars to the queue in the order they were enqueued (Car1, then Car2).

Pass/Fail Criteria:

- Pass: If both cars are successfully enqueued in the correct order (Car1, then Car2).
- Fail: If the cars are not enqueued in the correct order or if any car is not enqueued.

Actual Output:



Pass/Fail: Pass

Table 2. 11 Test Case ID: TC003

Test Case ID: TC003
Objective: To validate the functionality of the 'dequeue' operation in the 'CarQueue' class.
Test Items: 'dequeue' method of the 'CarQueue' class.
Input: Add two cars to the queue: Car Number: 1 Car Brand: "Toyota" Car Driver: "Driver1" Car Number: 2 Car Brand: "Honda" Car Driver: "Driver2"
Steps: 1. Initialize an instance of the 'CarQueue' class. 2. Call the 'enqueue' method with the two Car objects as arguments. 3. Call the 'dequeue' method.

```
public class Test {
    public static void main(String[] args) {
        // Create a new CarQueue.
        CarQueue carQueue = new CarQueue();

        // Create two new cars
        Car car1 = new Car(1, "Toyota", "Driver1");
        Car car2 = new Car(2, "Honda", "Driver2");

        // Enqueue the cars into the carQueue
        carQueue.enqueue(car1);
        carQueue.enqueue(car2);

        System.out.println("Before Dequeue Operation:");
        carQueue.printQueue();

        // Dequeue a car and print the removed car's details
        try {
            Car dequeuedCar = carQueue.dequeue();
            System.out.println("\nDequeued Car: " + dequeuedCar);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
```

```
        System.out.println("\nAfter Dequeue Operation:");
        carQueue.printQueue();
    }
}
```

In the above test class first I created a 'CarQueue' instance. I then created two 'Car' instances and enqueue them to the 'CarQueue'. I printed the queue before the dequeue operation to verify that the cars were enqueued properly. I performed a dequeue operation and print the details of the dequeued car. Finally, I printed the queue after the dequeue operation to verify that the car was properly removed.

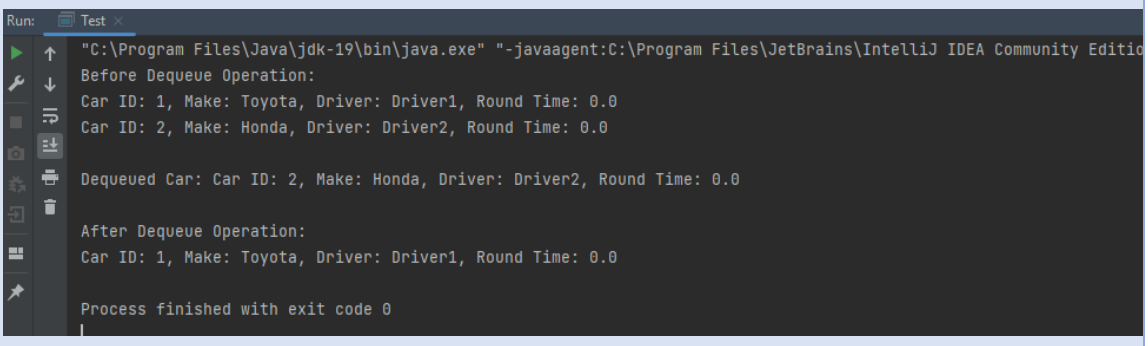
Expected Output:

The system should successfully remove the last car (car with id 2) from the queue. The queue should only contain the first car (car with id 1) after the dequeue operation.

Pass/Fail Criteria:

- Pass: If the system successfully dequeues the last car and the remaining car in the queue is the first car.
- Fail: If the system fails to dequeue the last car or if the remaining car in the queue is not the first car.

Actual Output:



Pass/Fail Criteria: Pass

Table 2. 12 Test Case ID: TC004

Test Case ID: TC004
Objective: To validate the functionality of the round time assignment and the sorting mechanism of cars in a queue based on round time.
Test Items: Round time assignment to each car and the sorting mechanism in the 'CarQueue' class.
Steps: 1. Initialize an instance of the 'CarQueue' class. 2. Create multiple 'Car' objects and enqueue them into the 'CarQueue'. 3. Assign random round times to each car in the queue. 4. Implement the sort function to sort the cars in ascending order based on their round times. 5. Retrieve the round times of each car in the sorted queue.

```
public class Test {
    public static void main(String[] args) {
        // Create a new CarQueue.
        CarQueue carQueue = new CarQueue();

        // Array of cars to be tested
        Car[] carArray = {new Car(1, "Toyota", "Driver1"),
                new Car(2, "Honda", "Driver2"),
                new Car(3, "Ford", "Driver3"),
                new Car(4, "Chevrolet", "Driver4")};

        // Enqueue the cars into the carQueue
        for (Car car : carArray) {
            // For each car in the carArray, enqueue (add) the car to
the queue.
            carQueue.enqueue(car);
        }

        // Assign a random time between 30 to 40 seconds to each car
for the round
        for (Car car : carQueue.cars) {
            // The roundTime is assigned a random value between 30 and
40.
            car.roundTime = Math.random() * 10 + 30;
        }

        // Sort the cars based on round time (ascending order)
        carQueue.cars.sort(Comparator.comparingDouble(car ->
car.roundTime));

        // Print the cars in the queue (i.e., the cars in the order of
their round times)
        System.out.println("Cars in the order of their round times:");
```

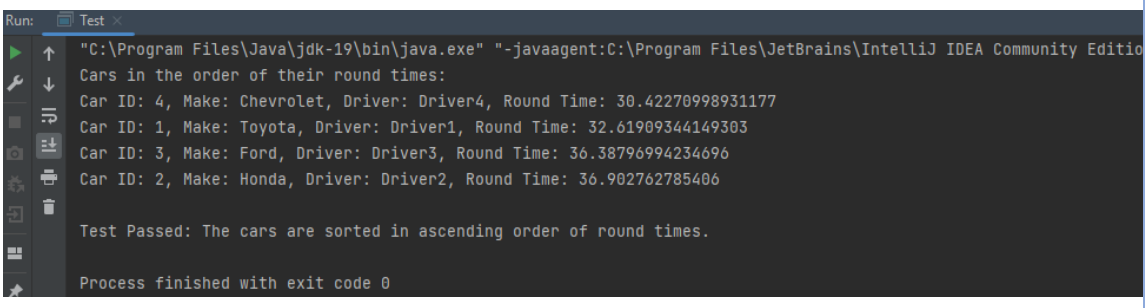
```
        carQueue.printQueue();

        // Create a boolean variable to check if the cars are sorted
in ascending order of round times
        boolean isSorted = true;

        // Check if the cars are sorted in ascending order of round
times
        for (int i = 0; i < carQueue.cars.size() - 1; i++) {
            if (carQueue.cars.get(i).roundTime > carQueue.cars.get(i +
1).roundTime) {
                isSorted = false;
                break;
            }
        }

        // Print the test result
        if (isSorted) {
            System.out.println("\nTest Passed: The cars are sorted in
ascending order of round times.");
        } else {
            System.out.println("\nTest Failed: The cars are not sorted
in ascending order of round times.");
        }
    }
}
```

Expected Output: The round times of the cars in the sorted queue should be in ascending order.

Pass/Fail Criteria:

- Pass: If the round times of the cars in the sorted queue are in ascending order.
- Fail: If the round times are not in ascending order.

Actual Output:



Pass/Fail: Pass

Table 2. 13 Test Case ID: TC005

Test Case ID: TC005
Objective: To validate the functionality of the elimination mechanism, ensuring the correct car is eliminated each round.
Test Items: The elimination mechanism in the 'CarQueue' class.
Steps: 1. Initialize an instance of the 'CarQueue' class. 2. Create multiple 'Car' objects and enqueue them into the 'CarQueue'. 3. Assign random round times to each car in the queue. 4. Implement the sort function to sort the cars in ascending order based on their round times. 5. Implement the elimination mechanism by dequeuing the car with the highest round time. **6.** Verify if the dequeued car had the highest round time in that round.

```
public class Test {
    public static void main(String[] args) {
        // Create a new CarQueue.
        CarQueue carQueue = new CarQueue();

        // Array of cars to be tested
        Car[] carArray = {new Car(1, "Toyota", "Driver1"),
                new Car(2, "Honda", "Driver2"),
                new Car(3, "Ford", "Driver3"),
                new Car(4, "Chevrolet", "Driver4")};

        // Enqueue the cars into the carQueue
        for (Car car : carArray) {
            // For each car in the carArray, enqueue (add) the car to
the queue.
            carQueue.enqueue(car);
        }

        // Assign a random time between 30 to 40 seconds to each car
for the round
        for (Car car : carQueue.cars) {
            // The roundTime is assigned a random value between 30 and
40.
            car.roundTime = Math.random() * 10 + 30;
        }

        // Sort the cars based on round time (ascending order)
        carQueue.cars.sort(Comparator.comparingDouble(car ->
car.roundTime));

        // Print the cars in the queue (i.e., the cars in the order of
their round times)
        System.out.println("Cars in the order of their round times:");
```

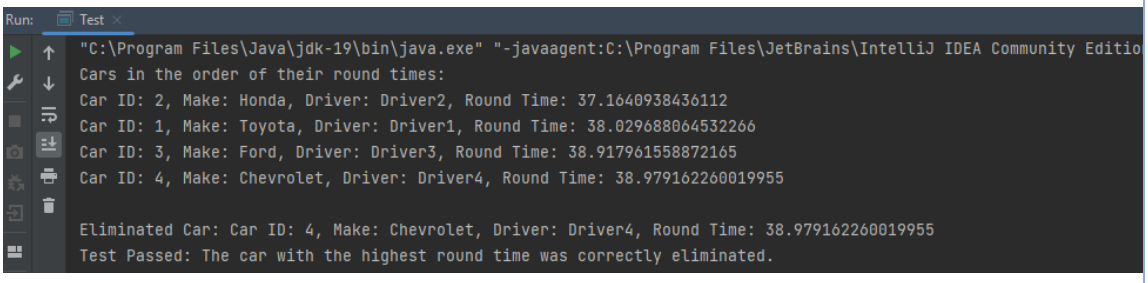
```
        carQueue.printQueue();

        // Dequeue the car with the highest round time
        try {
            Car dequeuedCar = carQueue.dequeue();

            // Print the eliminated car's details
            System.out.println("\nEliminated Car: " + dequeuedCar);

            // Create a boolean variable to check if the dequeued car
had the highest round time
            boolean isEliminatedCorrectly = true;

            // Check if any remaining car in the queue has a higher
round time than the dequeued car
            for (Car car : carQueue.cars) {
                if (car.roundTime > dequeuedCar.roundTime) {
                    isEliminatedCorrectly = false;
                    break;
                }
            }

            // Print the test result
            if (isEliminatedCorrectly) {
                System.out.println("Test Passed: The car with the
highest round time was correctly eliminated.");
            } else {
                System.out.println("Test Failed: The car with the
highest round time was not correctly eliminated.");
            }
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Expected Output: The car with the highest round time should be eliminated each round.

Pass/Fail Criteria:

- Pass: If the car with the highest round time is correctly dequeued (eliminated) each round.
- Fail: If any other car is dequeued.

Actual Output:



Pass/Fail: Pass

Table 2. 14 Test Case ID: TC006

Test Case ID: TC006
Objective: To validate the functionality of the winner determination mechanism, ensuring the correct cars are declared as winners.
Test Items: The winner determination mechanism in the 'CarQueue' class.
Steps: 1. Initialize an instance of the 'CarQueue' class. 2. Create multiple 'Car' objects and enqueue them into the 'CarQueue'. 3. Assign random round times to each car in the queue and implement the race mechanism (including the sorting and elimination processes) across multiple rounds. 4. At the end of all rounds, there should be 3 cars left in the queue. The car with the lowest cumulative round time should be declared as the 1st place winner, the second lowest as the 2nd place, and the third lowest as the 3rd place. **5.** Verify if the cars declared as winners are the ones with the lowest cumulative round times.

```
public class Test {
    public static void main(String[] args) {
        // Create a new CarQueue.
        CarQueue carQueue = new CarQueue();

        // Array of cars to be tested
        Car[] carArray = {new Car(1, "Toyota", "Driver1"),
                new Car(2, "Honda", "Driver2"),
                new Car(3, "Ford", "Driver3"),
                new Car(4, "Chevrolet", "Driver4"),
                new Car(5, "Subaru", "Driver5"),
                new Car(6, "Ferrari", "Driver6")};

        // Enqueue the cars into the carQueue
        for (Car car : carArray) {
            // For each car in the carArray, enqueue (add) the car to
the queue.
            carQueue.enqueue(car);
        }

        // Begin the 3 rounds
        for (int i = 1; i <= 3; i++) {
            // Assign a random time between 30 to 40 seconds to each
car for the round
            for (Car car : carQueue.cars) {
                // The roundTime is assigned a random value between 30
and 40.
                car.roundTime = Math.random() * 10 + 30;
```

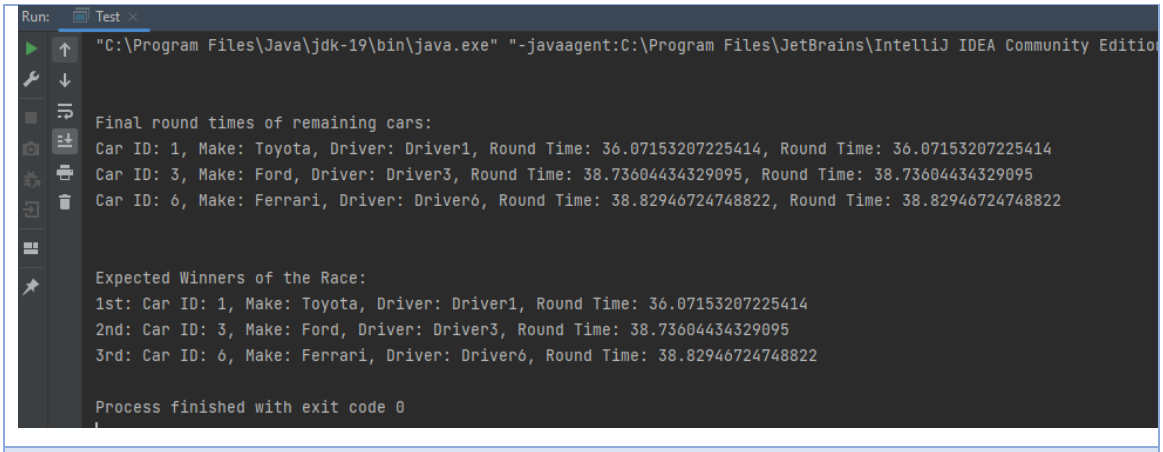
```
            }

            // Sort the cars based on round time (ascending order)
            carQueue.cars.sort(Comparator.comparingDouble(car ->
car.roundTime));

            // Dequeue the eliminated car (i.e., remove the car with
the maximum round time from the queue)
            try {
                carQueue.dequeue();
            } catch (Exception e) {
                System.out.println(e.getMessage());
            }
        }

        // Print the round time of each remaining car
        System.out.println("\n\nFinal round times of remaining
cars:");
        for (Car car : carQueue.cars) {
            System.out.println(car + ", Round Time: " +
car.roundTime);
        }

        // Assuming that the cars remaining in the queue have the
lowest round times,
        // print the expected winners of the race
        System.out.println("\n\nExpected Winners of the Race:");
        System.out.println("1st: " + carQueue.cars.get(0));
        System.out.println("2nd: " + carQueue.cars.get(1));
        System.out.println("3rd: " + carQueue.cars.get(2));
    }
}
```

Expected Output: The cars with the lowest cumulative round times should be declared as the 1st, 2nd, and 3rd place winners.

Pass/Fail Criteria:

- Pass: If the cars with the lowest cumulative round times are correctly declared as the 1st, 2nd, and 3rd place winners.
- Fail: If any other cars are declared as winners.

Actual Output:



Pass/Fail: Pass

Table 2. 15 Test Case ID: TC007

Test Case ID: TC007
Objective: To validate the functionality of the race when there is only one car, ensuring the race still runs and the single car is declared as the winner.
Test Items: The race mechanism in the 'CarQueue' class with a single 'Car' object.

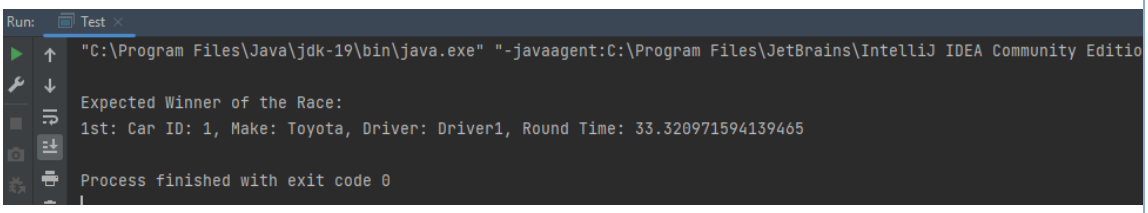
Steps:
1. Initialize an instance of the 'CarQueue' class.
2. Create a single 'Car' object and enqueue it into the 'CarQueue'.
3. Implement the race mechanism (including the sorting and elimination processes) across multiple rounds. Even though there's only one car, the race should still proceed with the race and the round time assignment.
4. At the end of all rounds, there should be one car left in the queue. This car should be declared as the 1st place winner.
5. Verify if the single car is declared as the winner.
 	
`public class Test {`
`  public static void main(String[] args) {`
`  // Create a new CarQueue.`
`  CarQueue carQueue = new CarQueue();`
 	
`  // Create a single car`
`  Car car1 = new Car(1, "Toyota", "Driver1");`
 	
`  // Enqueue the car into the carQueue`
`  carQueue.enqueue(car1);`
 	
`  // Begin the 3 rounds`
`  for (int i = 1; i <= 3; i++) {`
`  // Assign a random time between 30 to 40 seconds to the car for the round`
`  car1.roundTime = Math.random() * 10 + 30;`
`  }`
 	
`  // Assuming that the single car is the winner as it's the only car in the race,`
`  // print the expected winner of the race`
`  System.out.println("\nExpected Winner of the Race:");`
`  System.out.println("1st: " + carQueue.cars.get(0));`
`  }`
`}`

Expected Output: The single car should be declared as the 1st place winner.

Pass/Fail Criteria:

- Pass: If the race successfully proceeds with the race mechanism and declares the single car as the 1st place winner.
- Fail: If the race fails to proceed with the race mechanism or fails to declare the single car as the winner.

Actual Output:



Pass/Fail: Pass

Table 2. 16 Test Case ID: TC008

Test Case ID: TC008
Objective: To validate the functionality of the race when there are more than six cars, ensuring the race can handle larger queues and accurately determine winners.
Test Items: The race mechanism in the 'CarQueue' class with more than six 'Car' objects.
Steps: 1. Initialize an instance of the 'CarQueue' class. 2. Create more than six 'Car' objects (let's say 10 cars for this test) and enqueue them into the 'CarQueue'. 3. Implement the race mechanism (including the round time assignment, sorting, and elimination processes) across multiple rounds. 4. At the end of all rounds, there should be three cars left in the queue. These cars should be declared as the 1st, 2nd, and 3rd place winners, respectively. **5.** Verify if the race can handle the larger queue and determine the winners accurately.

```
public class Test {
    public static void main(String[] args) {
        // Create a new CarQueue.
        CarQueue carQueue = new CarQueue();

        // Create 10 new cars
        for(int i=1; i<=10; i++){
            Car car = new Car(i, "Car " + i, "Driver" + i);
            carQueue.enqueue(car);
        }

        // Begin the 3 rounds
        for (int i = 1; i <= 3; i++) {
            System.out.println("\n\nStarting Round " + i + "...");
            for(Car car: carQueue.cars){
                car.roundTime = Math.random() * 10 + 30; // random
time between 30 to 40
                System.out.println(car);
            }

            // Sort and remove car with highest time
            carQueue.cars.sort(Comparator.comparingDouble(car ->
car.roundTime));
            try {
                carQueue.dequeue();
            } catch (Exception e) {
                System.out.println(e.getMessage());
            }
        }
```

```
        // Declare the winners
        System.out.println("\nWinners of the Race:");
        System.out.println("1st: " + carQueue.cars.get(0));
        System.out.println("2nd: " + carQueue.cars.get(1));
        System.out.println("3rd: " + carQueue.cars.get(2));
    }
}
```

Expected Output: The race should successfully proceed with the race mechanism and declare the top three cars as winners.

Pass/Fail Criteria:

- Pass: If the race successfully proceeds with the race mechanism with more than six cars and declares the top three cars as winners.
- Fail: If the race fails to proceed with the race mechanism or fails to declare the top three cars as winners.

Actual Output:

```
Run:  Test
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition


Starting Round 1...
Car ID: 1, Make: Car 1, Driver: Driver1, Round Time: 35.959753727589444
Car ID: 2, Make: Car 2, Driver: Driver2, Round Time: 30.837347587506716
Car ID: 3, Make: Car 3, Driver: Driver3, Round Time: 33.11655772682883
Car ID: 4, Make: Car 4, Driver: Driver4, Round Time: 39.569420592966196
Car ID: 5, Make: Car 5, Driver: Driver5, Round Time: 39.49935568491942
Car ID: 6, Make: Car 6, Driver: Driver6, Round Time: 33.241443025255656
Car ID: 7, Make: Car 7, Driver: Driver7, Round Time: 39.5972547688749
Car ID: 8, Make: Car 8, Driver: Driver8, Round Time: 35.9380632712979
Car ID: 9, Make: Car 9, Driver: Driver9, Round Time: 35.95488284079824
Car ID: 10, Make: Car 10, Driver: Driver10, Round Time: 37.912814174687426


Starting Round 2...
Car ID: 2, Make: Car 2, Driver: Driver2, Round Time: 32.28059080098326
Car ID: 3, Make: Car 3, Driver: Driver3, Round Time: 39.32013007160707
Car ID: 6, Make: Car 6, Driver: Driver6, Round Time: 38.23875918535203
Car ID: 8, Make: Car 8, Driver: Driver8, Round Time: 33.01978953090412
Car ID: 9, Make: Car 9, Driver: Driver9, Round Time: 30.690423521134353
Car ID: 1, Make: Car 1, Driver: Driver1, Round Time: 38.08927785444496
Car ID: 10, Make: Car 10, Driver: Driver10, Round Time: 34.98070101628926
Car ID: 5, Make: Car 5, Driver: Driver5, Round Time: 39.00022446430832
Car ID: 4, Make: Car 4, Driver: Driver4, Round Time: 39.419269740054936


Starting Round 3...
Car ID: 9, Make: Car 9, Driver: Driver9, Round Time: 30.215822286528397
Car ID: 2, Make: Car 2, Driver: Driver2, Round Time: 30.121183707734726
Car ID: 8, Make: Car 8, Driver: Driver8, Round Time: 32.022396293739064
Car ID: 10, Make: Car 10, Driver: Driver10, Round Time: 33.285626129202996
Car ID: 1, Make: Car 1, Driver: Driver1, Round Time: 34.40166353343524
Car ID: 6, Make: Car 6, Driver: Driver6, Round Time: 39.3620639558682
Car ID: 5, Make: Car 5, Driver: Driver5, Round Time: 31.76864638290182
Car ID: 3, Make: Car 3, Driver: Driver3, Round Time: 32.61520008113821

Winners of the Race:
1st: Car ID: 2, Make: Car 2, Driver: Driver2, Round Time: 30.121183707734726
2nd: Car ID: 9, Make: Car 9, Driver: Driver9, Round Time: 30.215822286528397
3rd: Car ID: 5, Make: Car 5, Driver: Driver5, Round Time: 31.76864638290182

Process finished with exit code 0
```

Pass/Fail Criteria: Pass

Table 2. 17 Test Case ID: TC009

Test Case ID: TC009
Objective: To validate the functionality of the race when the queue has no cars, ensuring the race can handle this situation and exceptions are handled correctly.
Test Items: The race mechanism in the 'CarQueue' class with no 'Car' objects.
Steps: 1. Initialize an instance of the 'CarQueue' class without enqueuing any 'Car' objects. 2. Implement the race mechanism (including the round time assignment, sorting, and elimination processes). 3. Since there are no cars in the queue, an exception should be thrown when attempting to carry out the race mechanism. **4.** Verify if the race can handle this exception correctly and display an appropriate error message.

```
public class Test {
    public static void main(String[] args) {
        // Create a new CarQueue.
        CarQueue carQueue = new CarQueue();

        // Begin the 3 rounds
        for (int i = 1; i <= 3; i++) {
            System.out.println("\n\nStarting Round " + i + "...");
            // Check if the queue is empty before starting each round
            if(carQueue.isEmpty()) {
                System.out.println("Cannot start race. The queue is
empty!");
                return; // End the execution as there are no cars to
race
            }

            for(Car car: carQueue.cars){
                car.roundTime = Math.random() * 10 + 30; // random
time between 30 to 40
                System.out.println(car);
            }

            // Sort and remove car with highest time
            carQueue.cars.sort(Comparator.comparingDouble(car ->
car.roundTime));
            try {
                carQueue.dequeue();
            } catch (Exception e) {
                System.out.println(e.getMessage());
            }
        }

        // Declare the winners
```

```
        System.out.println("\nWinners of the Race:");
        if(carQueue.cars.isEmpty()) {
            System.out.println("Cannot declare winners. No cars
participated in the race!");
            return; // End the execution as there are no cars to
declare as winners
        }
        System.out.println("1st: " + carQueue.cars.get(0));
        System.out.println("2nd: " + carQueue.cars.get(1));
        System.out.println("3rd: " + carQueue.cars.get(2));
    }
}
```

Expected Output: The race should correctly handle the situation and display an error message indicating that there are no cars in the queue**.**

Pass/Fail Criteria:

- Pass: If the race correctly handles the situation when there are no cars in the queue, throws an exception, and displays the correct error message.
- Fail: If the race fails to handle the situation correctly or fails to throw an exception or display the correct error message.

Actual Output:

Pass/Fail: Pass

Necessary Theory Parts

Array

Understanding Array:

An array is a static data structure (meaning the size is fixed), used to store a collection of elements (data), each identified by at least one array index or key. The elements in an array are stored in contiguous memory locations. Elements can be of the same type (such as integers, characters, etc.) or different types, depending on the programming language.

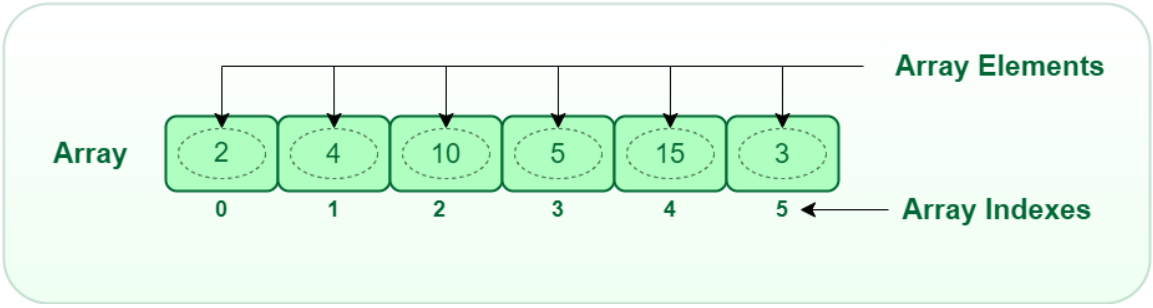


Figure 2. 3 Array data structure

Implementation of Array:

In most programming languages, arrays are a built-in data type or class, so they are directly supported. Arrays can be one-dimensional (a simple list of items), or multidimensional (tables, matrices, etc.).

Operations on Array:

- Insertion: This operation adds an element at the given index in the array. If the array is full, this operation could be costly due to the potential need for array resizing or it may not be allowed at all if the array is of a fixed size.
- Deletion: This operation removes an element at a given index from the array. The elements to the right of the deleted item are shifted to fill the vacant space.

- Access: This operation retrieves an element at a given index from the array. This is a constant-time operation as we can directly calculate the memory address of any item based on its index.
- Search: This operation finds the location of an element in the array. If the array is unsorted, the search operation can take linear time as we may need to traverse the entire array. However, for a sorted array, we can use binary search for an efficient logarithmic time search operation.
- Update: This operation replaces an element at a specific index with a new element.

Characteristics of Array:

- Homogeneous Elements: All elements of the array must be of the same type.
- Fixed Size: The size of an array is fixed at the time of declaration. It cannot be changed later.
- Random Access: Elements in an array can be randomly accessed since we can calculate the address of each element directly.
- Uses: Arrays are useful when we have a fixed number of elements that are of the same type. They are typically used in situations where the order of elements is important.

Complexity of Array Operations:

- Access: Accessing an element from an array using the index is a constant-time operation, i.e., O(1).
- Search: For unsorted arrays, searching takes linear time, i.e., O(n). For sorted arrays, binary search can be performed in O(log n) time.
- Insertion/Deletion: These operations take linear time, i.e., O(n) because elements need to be shifted to fill the space or create space for a new element.

Array example:

For example, I consider to create an array to store the grades of a class of students. I'll implement the basic operations mentioned above. This program creates an array of integers with five elements to store the grades of five students. It then calculates and prints the average grade. Finally, it updates the grade of the first student and prints the updated grades. This demonstrates the creation, accessing, updating, and determining the length of an array in Java.

Code →

```
public class GradesCalculator {
    public static void main(String[] args) {
        // Create an array to store grades of 5 students
        int[] grades = new int[5];

        // Assign grades to students
        grades[0] = 85;
        grades[1] = 90;
        grades[2] = 88;
        grades[3] = 92;
        grades[4] = 86;

        // Calculate and display the average grade
        int sum = 0;
        for (int i = 0; i < grades.length; i++) {
            sum += grades[i];
        }

        float average = (float) sum / grades.length;
        System.out.println("The average grade is: " + average);

        // Update the grade for the first student and display the
updated grades
        grades[0] = 95;
        System.out.println("Updated grades: ");
        for (int i = 0; i < grades.length; i++) {
            System.out.println("Grade of student " + (i+1) + " is: " +
grades[i]);
        }
    }
}
```

Output →

Figure 2. 4 Array example code output

Queue

Understanding Queue:

A Queue is a linear data structure that stores elements in a sequential manner. The essence of a queue is that it follows a First-In-First-Out (FIFO) strategy for handling elements. This means that the element that is added first is the one that gets removed first. In real-life scenarios, queues can be seen in various situations such as people standing in line to buy tickets, vehicles waiting for a toll booth, etc.

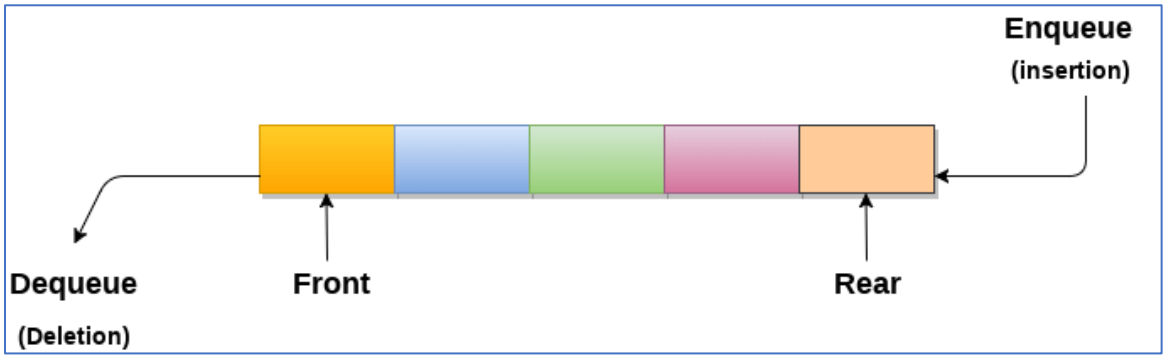


Figure 2. 5 Queue data structure

Implementation of Queue:

Queue can be implemented using an array, stack, or linked list. But most commonly, it is implemented using an array or linked list.

1. Array Implementation (Static Queue): The array implementation of a queue is straightforward where we maintain two pointers (front and rear) to keep track of the start and end of the queue. However, array implementation may lead to a problem of memory wastage if elements are continuously dequeued (removed) and enqueued (added).
2. Linked List Implementation (Dynamic Queue): To overcome the issue of memory wastage in array implementation, we use a linked list to implement a queue where the queue can grow and shrink according to the needs at runtime.

Operations on Queue:

- Enqueue: This operation adds an element to the end of the queue.
- Dequeue: This operation removes an element from the front of the queue.
- Peek/Front: This operation helps in returning the first element of the queue. It retrieves the element from the front of the queue without deleting it.
- IsEmpty: This operation checks if the queue is empty.
- IsFull: This operation checks if the queue is full.

Characteristics of Queue:

- Non-primitive Data Structure: Queue is a non-primitive data structure that holds a collection of elements in an ordered sequence.
- Dynamic Size: The size of the queue can increase or decrease at run time.
- One Way Access: We can access elements in a queue from one side only. We always insert elements from the rear and remove elements from the front.
- Uses: Queues are used when we need to maintain the order of elements in which they occur. For example, in networking, when data packets arrive in order, they get executed in the order of their arrival.

Complexity of Queue Operations:

The time complexity of enqueue and dequeue operations in a queue is O(1). This is because we just need to perform one step for carrying out these operations. We either increment or decrement the pointers (front and rear) as part of these operations, hence constant time complexity.

Queue example:

For example, I consider a queue in a supermarket. The queue allows customers to be served in the order they arrived, following the First-In-First-Out (FIFO) principle.

Code →

```
// Queue class definition
public class Queue {
    private int maxSize;
    private int front;
    private int rear;
    private int[] queueArray;

    // Constructor
    public Queue(int size) {
        this.maxSize = size;
        this.queueArray = new int[maxSize];
        this.front = 0;
        this.rear = -1;
    }

    // Check if the queue is empty
    public boolean isEmpty() {
        return (rear == -1);
    }

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

    // Get the size of the queue
    public int size() {
        return rear - front + 1;
    }

    // Insert an element at the rear of the queue
    public void enqueue(int value) {
        if (!isFull()) {
            queueArray[++rear] = value;
        } else {
            throw new RuntimeException("Queue is full");
        }
    }

    // Remove an element from the front of the queue
    public int dequeue() {
        if (!isEmpty()) {
            int temp = queueArray[front];
            for (int i = 0; i < rear; i++) {
                queueArray[i] = queueArray[i + 1];
            }
            if (rear < maxSize)
```

```
                queueArray[rear] = 0;
            rear--;
            return temp;
        } else {
            throw new RuntimeException("Queue is empty");
        }
    }
}


// Main class for simulating the supermarket scenario
public class SupermarketSimulator {
    public static void main(String[] args) {
        Queue supermarketQueue = new Queue(5); // Queue with a maximum
of 5 customers

        // 5 customers arrive
        for (int i = 1; i <= 5; i++) {
            supermarketQueue.enqueue(i);
            System.out.println("Customer " + i + " arrived");
        }

        // Serve customers
        while (!supermarketQueue.isEmpty()) {
            int servedCustomer = supermarketQueue.dequeue();
            System.out.println("Customer " + servedCustomer + "
served");
        }
    }
}
```

Output →

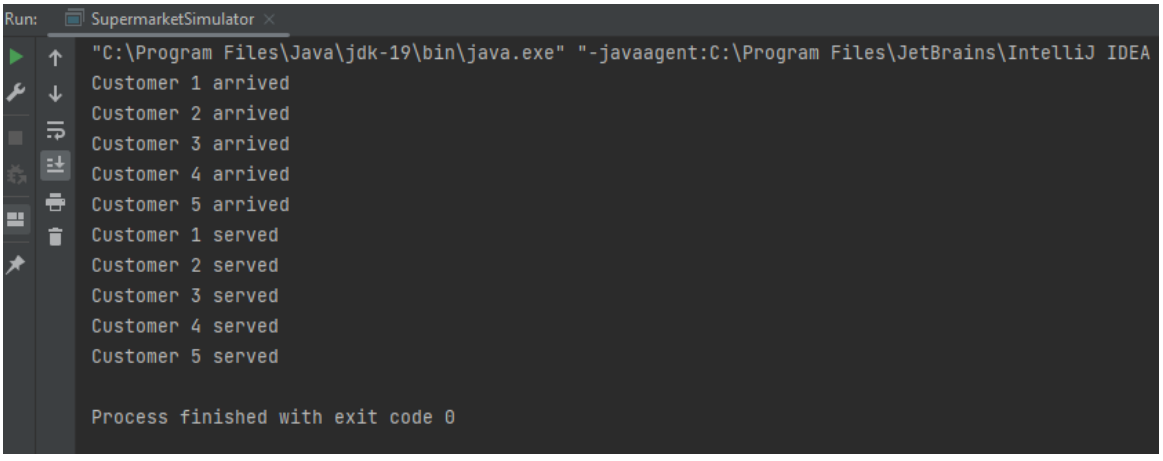


Figure 2. 6 Queue example code output

Error Handling

Understanding Error Handling:

Error handling, also referred to as exception handling, is a process that responds to an error occurrence during execution of the program. Errors can happen at any time for a variety of reasons: missing resources, wrong user input, hardware failures, and many others. To maintain the integrity of the program and to ensure it doesn't abruptly stop or behave inconsistently, these potential problems must be managed systematically, which is where error handling comes in.

Types of Errors:

In Java, errors are classified into two main categories:

- Checked exceptions: These are exceptions that the code must handle, as they are checked at compile time. Examples include IOException, SQLException, etc.
- Unchecked exceptions: These are exceptions that are not checked at compile time, but at runtime. Examples include ArrayIndexOutOfBoundsException, NullPointerException, etc.

Error Handling Techniques:

Java, like other programming languages, uses try-catch-finally block to handle exceptions.

- Try block: This is where we place the code that may generate an exception.
- Catch block: If an exception occurs in the try block, control is passed to the catch block (if the type of exception matches the type that the catch block can handle).
- Finally block: This block is optional and typically used for cleaning up resources, such as closing a file or database connection. This block is always executed, regardless of whether an exception is thrown.

Task 3

Describe the problem statement and the chosen ADT (Static Stack)

Problem Statement Overview

The problem presented in this scenario involves a car racing event organized by ABC Pvt Ltd. The company has a specific approach to the registration and management of participating cars, all of which have unique identifiers including number, brand, sponsor, and driver details. The cars are registered chronologically, from the oldest to the newest. However, ABC Pvt Ltd requires a system that allows them to retrieve these registrations in the reverse order. That is, they need to access car details starting from the most recent (newest) registration back to the earliest (oldest) one.

This requirement stems from the nature of the racing event: cars compete in a series of rounds, with the lowest ranking car at the end of each round being eliminated. The management, therefore, needs a dynamic way to manage these eliminations, with immediate access to the most recent registrations and the ability to work back towards the older ones. The primary objective here is to design and implement an application that can successfully handle these operations in an efficient and effective manner.

Choice of Abstract Data Type – Stack

In the context of data structures, the specific requirement of ABC Pvt Ltd, particularly the need to access car registration data from the newest to the oldest, lends itself well to the utilization of a Stack. A Stack is an Abstract Data Type (ADT) that adheres to a "Last In, First Out" (LIFO) order of operations. This implies that the last item added (in this case, the most recently registered car) is the first one to be retrieved or "popped" from the stack. Conversely, the first item added (the earliest registered car) is the last to be removed or accessed, which aligns perfectly with the given scenario.

An Abstract Data Type, or ADT, is essentially a high-level description of how data is viewed and manipulated, irrespective of its implementation. In other words, an ADT is more concerned with what the data represents and what operations can be performed on it, as opposed to how these operations are carried out behind the scenes. By choosing the Stack ADT, I focus on the concept of storing and retrieving car registrations in a LIFO manner, without getting into the specifics of how the stack is implemented (which can be done using an array, a linked list, etc.).

Stacks come with a set of basic operations - "push" (add an element to the top of the stack), "pop" (remove the top element), and "peek" or "top" (view the top element without removing it), along with auxiliary operations like "isEmpty" (check if the stack is empty) and "isFull" (check if the stack is full, applicable in a static or array-based implementation). These operations enable us to effectively manage the car registrations as per the outlined requirements.

Reasoning Behind Choosing Static Stack

1) **Fixed Size of Participants:** The number of participants (cars) is fixed and known in advance, which is a scenario where using a static array-based implementation of a Stack is optimal. In static stacks, the size of the stack is set during the initialization and cannot be changed later. Since we already know that the maximum number of cars is 6, we can efficiently use a static Stack of size 6 for this scenario.
2) **Efficiency:** Stacks implemented using arrays provide constant time complexity O(1) for all primary stack operations (push, pop, peek, isEmpty, isFull). Given that we know the maximum size of the Stack, this static implementation provides a significant efficiency benefit. We don't have the overhead of dynamic memory allocation and deallocation as in the case of a dynamically implemented stack (using a linked list). This means that for this case, where the size is known and small, an array-based stack can be more efficient in terms of both time and memory.
3) **Ease of Implementation:** Array-based stacks are generally easier to implement and understand, especially when dealing with a known maximum size. The array

provides a straightforward way to store data sequentially in memory, which simplifies the handling of elements.

4) **Direct Access and Random Access:** Although not a common operation on stacks, static stacks (arrays) can provide direct access to any element if needed, based on the index. In contrast, dynamic stacks (linked lists) would require traversing from the head node up to the nth node, leading to O(n) time complexity for such operations.

In summary, given the specific constraints and requirements of the car racing event organized by ABC Pvt Ltd, a static implementation of the Stack ADT is well-suited. This choice allows us to make efficient use of memory, maintain high performance in terms of time complexity, and simplifies the implementation process. It's important to note that a static stack would not be as suitable if the number of participants could vary significantly, as its size is fixed upon initialization and cannot be changed thereafter. However, given the fixed limit of 6 participants, it serves as an excellent choice for this scenario.

Stack Definition: Define what a stack is and discuss its main operations

Stack Definition

A Stack is a linear Abstract Data Type (ADT) which follows a particular order of operation - LIFO (Last In, First Out) or FILO (First In, Last Out). This means that in a stack, insertion (push) and removal (pop) of elements take place at the same end, the top end. The lower end is called the base. It's like a vertical stack of objects; the most accessible object is the one at the top.

In the context of computer science, a Stack is a data structure that serves as a collection of elements with two main operations: push and pop. Stacks are used extensively in both low-level (e.g., memory management, run-time polymorphism) and high-level applications (e.g., expression evaluation/parsing).

Main Operations of a Stack

1) **'Push' Operation:** The push operation adds a new element onto the stack. When we add an item, the current top item gets 'pushed down', and this new item becomes the new top of the stack. If the stack is implemented with a size limit (as in the case of using an array), we must also check to see if the stack is already full before pushing the new item.
2) **'Pop' Operation:** The pop operation removes an element from the top of the stack. It essentially reveals the item below the current top item, making it the new top. Similar to the push operation, we must ensure that the stack is not empty before attempting to remove an item, to avoid underflow.
3) **'Peek'/'Top' Operation:** This operation returns the top element of the stack without removing it, giving us a way to observe the current top item without disturbing the order of the stack. Like pop, peek also requires us to check for an empty stack beforehand.

4) **'isEmpty' Operation:** This operation checks if the stack is empty. It is a straightforward yet crucial operation, especially before performing pop or peek operations, to prevent underflow.
5) **'isFull' Operation:** This operation checks if the stack is full. While not relevant in a dynamic stack implementation (like using a linked list), it is essential when implementing a stack with a static size limit (like using an array), mainly before a push operation, to prevent overflow.

Each of these operations allows a level of control and interaction with the stack data structure. In most cases, the 'push', 'pop', and 'isEmpty' operations are vital for stack manipulation, and they allow for the most common use-cases for stacks. In contrast, the 'peek' and 'isFull' operations provide additional functionality and control, which can be useful in specific applications.

These operations, particularly because of their O(1) time complexity, make the stack an incredibly efficient data structure. It guarantees a constant time for addition and removal, irrespective of the number of elements in the stack.

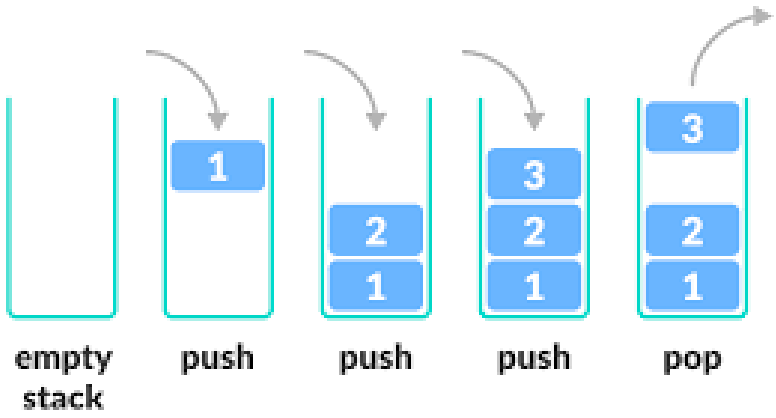


Figure 3. 1 Main Operations of a Stack

Implement the specified ADT (Static Stack) using Java programming

Java code to arrange cars from oldest to newest, then from newest to oldest

```
// This class defines the Car object
class Car {
    int number;
    String brand;
    String sponsor;
    String driverDetails;

    // This is the Car constructor
    public Car(int number, String brand, String sponsor, String
driverDetails) {
        this.number = number;
        this.brand = brand;
        this.sponsor = sponsor;
        this.driverDetails = driverDetails;
    }

    // This method returns a string representation of the Car
    @Override
    public String toString() {
        return "Car {" +
                "Number=" + number +
                ", Brand='" + brand + '\'' +
                ", Sponsor='" + sponsor + '\'' +
                ", Driver Details='" + driverDetails + '\'' +
                '}';
    }
}

// This class defines the Stack object
class Stack {
    static final int MAX = 6; // The maximum size of the stack
    int top; // The top of the stack
    Car a[] = new Car[MAX]; // The array to store cars

    // This is the Stack constructor
    Stack() {
        top = -1; // Initializing the top of the stack
    }

    // This method checks if the stack is empty
    boolean isEmpty() {
        return (top < 0);
    }

    // This method adds a new Car to the top of the stack
    boolean push(Car car) {
        if (top >= (MAX - 1)) {
            System.out.println("Stack Overflow, Car " + car.number + "
cannot be added");
            return false;
        }
```

```
        else {
            a[++top] = car;
            System.out.println("Car " + car.number + " has been
successfully added to the stack");
            return true;
        }
    }

    // This method removes the Car at the top of the stack
    Car pop() {
        if (top < 0) {
            System.out.println("Stack Underflow, no car to remove");
            return null;
        }
        else {
            Car car = a[top--];
            System.out.println("Car " + car.number + " has been
successfully removed from the stack");
            return car;
        }
    }

    // This method returns the Car at the top of the stack without
removing it
    Car peek() {
        if (top < 0) {
            System.out.println("Stack Underflow, no car in the stack");
            return null;
        }
        else {
            Car car = a[top];
            return car;
        }
    }

    // This method prints all the Cars in the stack from newest to
oldest
    void print() {
        if(isEmpty()){
            System.out.println("No car in the stack");
            return;
        }
        System.out.println("Cars in the stack from newest to oldest:");
        for(int i = top;i>-1;i--){
            System.out.println(a[i]);
        }
    }
}

// This is the main driver class
public class Main {
    public static void main(String args[]) {
        Stack s = new Stack(); // Create a new Stack

        System.out.println("\nAdding cars to the stack:");
        // Add Cars to the Stack
        s.push(new Car(1, "Brand1", "Sponsor1", "Driver1"));
        s.push(new Car(2, "Brand2", "Sponsor2", "Driver2"));
        s.push(new Car(3, "Brand3", "Sponsor3", "Driver3"));
```

```
        s.push(new Car(4, "Brand4", "Sponsor4", "Driver4"));
        s.push(new Car(5, "Brand5", "Sponsor5", "Driver5"));
        s.push(new Car(6, "Brand6", "Sponsor6", "Driver6"));

        System.out.println("\nDisplaying top car in the stack:");
        // Display the top Car
        System.out.println("Top car is: " + s.peek());

        System.out.println("\nPrinting all cars in the stack from newest
to oldest:");
        // Print all Cars in the Stack
        s.print();
    }
}
```

Output of the Java Code

```
Run:  Main (4)
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA

Adding cars to the stack:
Car 1 has been successfully added to the stack
Car 2 has been successfully added to the stack
Car 3 has been successfully added to the stack
Car 4 has been successfully added to the stack
Car 5 has been successfully added to the stack
Car 6 has been successfully added to the stack

Displaying top car in the stack:
Top car is: Car {Number=6, Brand='Brand6', Sponsor='Sponsor6', Driver Details='Driver6'}

Printing all cars in the stack from newest to oldest:
Cars in the stack from newest to oldest:
Car {Number=6, Brand='Brand6', Sponsor='Sponsor6', Driver Details='Driver6'}
Car {Number=5, Brand='Brand5', Sponsor='Sponsor5', Driver Details='Driver5'}
Car {Number=4, Brand='Brand4', Sponsor='Sponsor4', Driver Details='Driver4'}
Car {Number=3, Brand='Brand3', Sponsor='Sponsor3', Driver Details='Driver3'}
Car {Number=2, Brand='Brand2', Sponsor='Sponsor2', Driver Details='Driver2'}
Car {Number=1, Brand='Brand1', Sponsor='Sponsor1', Driver Details='Driver1'}

Process finished with exit code 0
```

Figure 3. 2 Stack output of the Java Code

Valid operations that used on this data structure

For the static stack data structure that I have used in the code, the following operations are supported:

1. **push:** This operation is used to add a new element to the top of the stack. If the stack is full (i.e., it has reached its maximum limit), an overflow message is displayed, and the element is not added to the stack. On the other hand, if the stack has space to accommodate the new element, it is added to the top of the stack and a success message is displayed. The push operation is O(1), as it only involves a single operation, and thus its execution time does not change with the size of the stack.
2. **pop:** This operation is used to remove the topmost element from the stack. If the stack is empty, an underflow message is displayed, and no element is removed. If the stack has at least one element, the topmost element is removed and a success message is displayed. The element removed is returned by the operation. Like the push operation, the pop operation is also O(1), as it only involves a single operation.
3. **peek:** This operation is used to return the topmost element of the stack without removing it. If the stack is empty, an underflow message is displayed and null is returned. If the stack has at least one element, the topmost element is returned. The peek operation is O(1) as well, as it only involves a single operation.
4. **isEmpty:** This operation is used to check whether the stack is empty or not. If the stack is empty, it returns true; otherwise, it returns false. The isEmpty operation is O(1) as it simply checks the value of the top of the stack.
5. **print:** This operation prints all the elements in the stack from the top to the bottom (i.e., from the newest to the oldest element in the case of our Car stack). If the stack is empty, a message is displayed to indicate this. Otherwise, all the elements are displayed one by one. The print operation is O(n), where n is the number of elements in the stack, as it involves a loop that goes through every element.

These operations allow me to manipulate the stack data structure to meet assignment needs. For example, in the car racing scenario, I used these operations to add cars to the stack (push), to view the car that was added most recently (peek), and to display all the cars in the order they were added (print).

Remember that a stack is a last-in-first-out (LIFO) data structure, meaning that the most recently added (last in) element is the first one to be removed (first out). This characteristic was exemplified in the scenario where the most recently added car was the first one we looked at when I used the peek operation.

Error handling for the code

Error handling in our Stack implementation revolves around two main types of errors that could occur: Stack Overflow and Stack Underflow. These are common errors associated with the Stack data structure and refer to scenarios where we try to exceed the capacity of the stack (Overflow) or remove an element from an empty stack (Underflow).

1. Stack Overflow:

- This occurs when we try to add (push) an element onto a stack that has already reached its maximum capacity. To handle this, we have a condition inside our `push` method that checks whether the `top` of the stack is equal to or greater than `MAX - 1`. `MAX` is the maximum capacity of our stack, and since we start counting from 0, `MAX - 1` represents the maximum index in our array.
- If `top` is indeed equal to or greater than `MAX - 1`, this means our stack is full, and we cannot add more elements to it. We print a "Stack Overflow" message along with the car number which we attempted to add, to indicate that the stack is full, and return `false`.
- If the stack is not full, we increment `top` by one and add the new element to the stack, indicating successful addition with a corresponding message and returning `true`.

2. Stack Underflow:

- Stack underflow occurs when we attempt to remove (pop) or peek an element from an empty stack.
- In our `pop` and `peek` methods, we first check if the `top` is less than 0. If `top` is less than 0, this indicates our stack is empty (since we initialize `top` as -1 and increment it each time we add an element).

- If our stack is indeed empty, we print a "Stack Underflow" message to indicate that there are no elements to remove or peek at and return `null` in case of `peek` and `pop`.
- If the stack is not empty, in the case of `pop`, we return the topmost element and decrement `top` by 1, indicating successful removal with a corresponding message. In the case of `peek`, we just return the topmost element without modifying `top`.

3. Empty Stack:

- Before printing the elements of the stack, we check if the stack is empty by invoking the `isEmpty` method. This method returns true if `top` is less than 0, indicating that the stack is empty.
- If the stack is empty, we print a message "No car in the stack" and return from the method, avoiding an attempt to print elements from an empty stack.

These checks ensure that we handle errors gracefully and provide meaningful feedback to the user about what's happening. Without these error-handling checks, we could end up with runtime errors like `ArrayIndexOutOfBoundsException`, which could be harder to debug and less user-friendly. By implementing these checks, we're making our stack data structure robust and reliable.

Test Plan

Introduction:

This test plan focuses on the simulation of a Stack, that stores instances of a Car class, representing a garage scenario. The operations to be tested include stack push (adding a car to the garage), pop (removing a car from the garage), and peek (displaying the top car in the garage without removing it).

Objectives:

The primary objectives of this testing phase are:

- To ensure that all components of the program function as expected.
- To validate that the Stack's operations (push, pop, peek) work correctly.
- To ensure that the exception handling is effectively preventing and responding to potential errors, such as stack overflow and underflow.

Risks and Contingencies:

- Risks: The main risks are bugs that could cause the program to crash or behave unpredictably. Misimplementation of stack operations, such as push, pop and peek, and ineffective error handling, are also risks.
- Contingencies: If any significant defects are identified, the code should be revisited and debugged. Proper error handling should be ensured, and the stack operations should be accurately implemented.

Test Items:

1. `Car` class: number, brand, sponsor, driverDetails properties, and `toString()` method.
2. `Stack` class: MAX, top, a properties, `isEmpty()`, `push()`, `pop()`, `peek()`, and `print()` methods.

3. Main function: Creating and initializing Car objects, pushing and popping cars, checking stack status, exception handling.

Features to be Tested:

- Initialization of the Car objects.
- Functioning of the Stack - pushing, popping, and peeking cars, checking if the stack is empty.
- Exception handling when adding to a full stack (Stack Overflow) or removing from an empty stack (Stack Underflow).

Test Techniques:

- Unit Testing: To test individual classes and methods.
- Integration Testing: To test the interaction of different parts of the program.
- Boundary Testing: To test the limits of the stack operations such as maximum and minimum number of cars, zero cars, etc.

Test Cases:

1. Initialization of the Car objects with valid values.
2. Pushing cars into the stack, ensuring proper functioning.
3. Popping cars from the stack, ensuring proper functioning.
4. Peeking into the stack, and check if the top car is returned without being removed.
5. Test the stack with no cars, and check if exceptions are handled correctly.
6. Test with exactly maximum limit of cars, ensuring the stack can handle its maximum limit.
7. Pushing a car into a full stack, and ensure that a stack overflow message is displayed.
8. Popping a car from an empty stack, and ensure that a stack underflow message is displayed.

Pass/Fail Criteria:

- Pass: The program runs without crashing or throwing any unhandled exceptions. All the features and methods work correctly, stack operations are accurately implemented, and exceptions are correctly handled.
- Fail: The program crashes or throws any unhandled exceptions. Any of the features or methods don't work correctly, stack operations are inaccurately implemented, or exceptions are incorrectly handled.

Test Cases

Table 3. 1 Test Case ID: TC001

Test Case ID: TC001
Objective: To validate the initialization of the Car objects with valid values.
Test Items: Car class constructor and its properties (number, brand, sponsor, driverDetails).
Steps: 1. Create an instance of the 'Car' class with valid parameters, for example, number=1, brand="Brand1", sponsor="Sponsor1", driverDetails="Driver1". 2. Repeat step 1 to create multiple instances of the 'Car' class with different parameters. 3. Access and verify the properties (number, brand, sponsor, driverDetails) of each Car object to ensure they match the given initialization values.

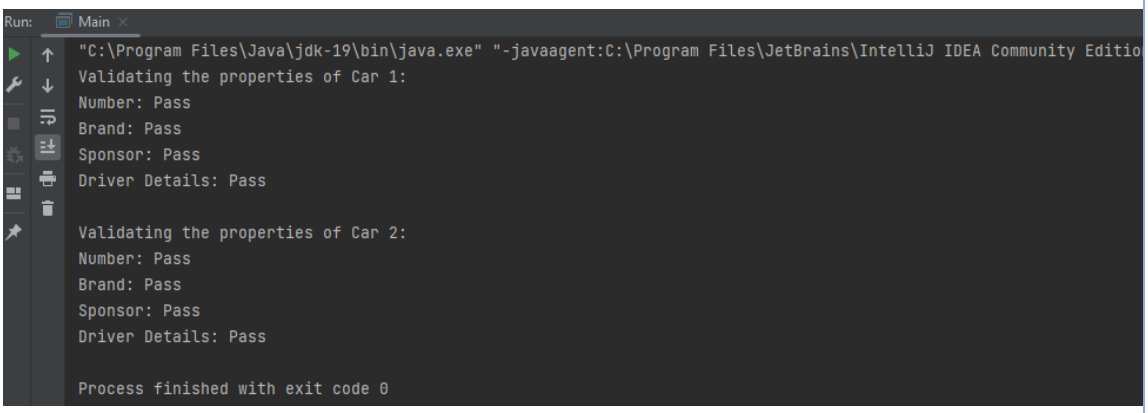
```
public class Main {
    public static void main(String args[]) {
        // Create new Car instances
        Car car1 = new Car(1, "Brand1", "Sponsor1", "Driver1");
        Car car2 = new Car(2, "Brand2", "Sponsor2", "Driver2");

        // Validate the properties of each Car object
        System.out.println("Validating the properties of Car 1:");
        System.out.println("Number: " + (car1.number == 1 ? "Pass" :
"Fail"));
        System.out.println("Brand: " + ("Brand1".equals(car1.brand) ?
"Pass" : "Fail"));
        System.out.println("Sponsor: " +
("Sponsor1".equals(car1.sponsor) ? "Pass" : "Fail"));
        System.out.println("Driver Details: " +
("Driver1".equals(car1.driverDetails) ? "Pass" : "Fail"));

        System.out.println("\nValidating the properties of Car 2:");
        System.out.println("Number: " + (car2.number == 2 ? "Pass" :
"Fail"));
        System.out.println("Brand: " + ("Brand2".equals(car2.brand) ?
"Pass" : "Fail"));
        System.out.println("Sponsor: " +
("Sponsor2".equals(car2.sponsor) ? "Pass" : "Fail"));
        System.out.println("Driver Details: " +
("Driver2".equals(car2.driverDetails) ? "Pass" : "Fail"));
    }
}
```

Expected Output: The properties of each Car object should match the values given at the time of initialization.

Pass/Fail Criteria:

- Pass: If the properties of each Car object match the values given at the time of initialization.
- Fail: If any property of a Car object doesn't match the value given at the time of initialization.

Actual Output:



Pass/Fail: Pass

Table 3. 2 Test Case ID: TC002

Test Case ID: TC002
Objective: To validate the functionality of pushing cars into the stack.
Test Items: Stack class and its `push()` method.
Steps: 1. Create an instance of the 'Stack' class. 2. Create multiple 'Car' objects with valid parameters. 3. Use the `push()` method of the 'Stack' class to push the 'Car' objects into the stack. 4. Verify if the 'Car' objects are successfully added to the stack using the `peek()` and `isEmpty()` methods of the 'Stack' class.

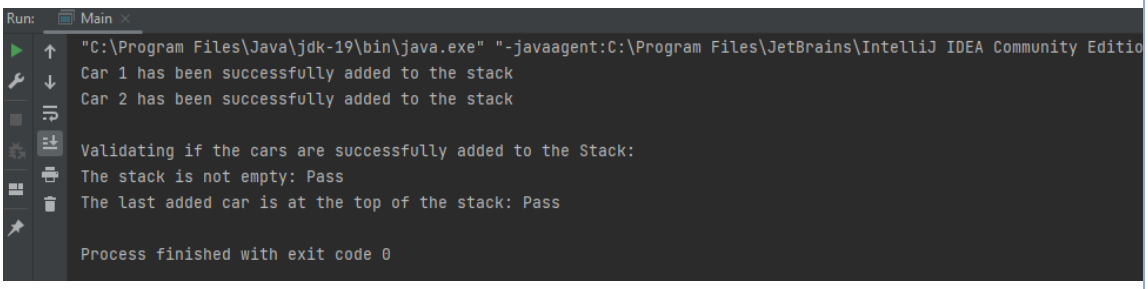
```
public class Main {
    public static void main(String args[]) {
        // Create a new Stack instance
        Stack s = new Stack();

        // Create new Car instances
        Car car1 = new Car(1, "Brand1", "Sponsor1", "Driver1");
        Car car2 = new Car(2, "Brand2", "Sponsor2", "Driver2");

        // Push the Car instances into the Stack
        s.push(car1);
        s.push(car2);

        // Validate if the cars are successfully added to the Stack
        System.out.println("\nValidating if the cars are successfully
added to the Stack:");

        // Check if the Stack is not empty
        if(!s.isEmpty()) {
            System.out.println("The stack is not empty: Pass");

            // Check if the last added Car is at the top of the Stack
            if(s.peek() == car2) {
                System.out.println("The last added car is at the top
of the stack: Pass");
            } else {
                System.out.println("The last added car is not at the
top of the stack: Fail");
            }
        } else {
            System.out.println("The stack is empty: Fail");
        }
    }
}
```

Expected Output: The 'Car' objects should be successfully added to the stack in a Last-In-First-Out (LIFO) manner, with the last added car being at the top of the stack.

Pass/Fail Criteria:

- Pass: If the 'Car' objects are successfully added to the stack in a LIFO manner and the last added car can be verified as the top car in the stack using the `peek()` method.
- Fail: If the 'Car' objects are not successfully added to the stack or the order doesn't follow the LIFO principle.

Actual Output:



Pass/Fail: Pass

Table 3. 3 Test Case ID: TC003

Test Case ID: TC003
Objective: To validate the functionality of popping cars from the stack.
Test Items: Stack class and its `pop()` method.
Steps: 1. Create an instance of the 'Stack' class. 2. Create multiple 'Car' objects with valid parameters. 3. Use the `push()` method of the 'Stack' class to push the 'Car' objects into the stack. 4. Use the `pop()` method of the 'Stack' class to remove the 'Car' objects from the stack one by one. 5. Verify if the 'Car' objects are successfully removed from the stack in a Last-In-First-Out (LIFO) manner using the `isEmpty()` method.

```
public class Main {
    public static void main(String args[]) {
        // Create a new Stack instance
        Stack s = new Stack();

        // Create new Car instances
        Car car1 = new Car(1, "Brand1", "Sponsor1", "Driver1");
        Car car2 = new Car(2, "Brand2", "Sponsor2", "Driver2");

        // Push the Car instances into the Stack
        s.push(car1);
        s.push(car2);

        // Pop the Car instances from the Stack
        Car poppedCar1 = s.pop();
        Car poppedCar2 = s.pop();

        // Validate if the cars are successfully popped from the Stack
        System.out.println("\nValidating if the cars are successfully
popped from the Stack:");

        // Check if the first popped Car is the last added Car
        if(poppedCar1 == car2) {
            System.out.println("The first popped car is the last added
car: Pass");
        } else {
            System.out.println("The first popped car is not the last
added car: Fail");
        }

        // Check if the second popped Car is the first added Car
        if(poppedCar2 == car1) {
            System.out.println("The second popped car is the first
added car: Pass");
        } else {
```

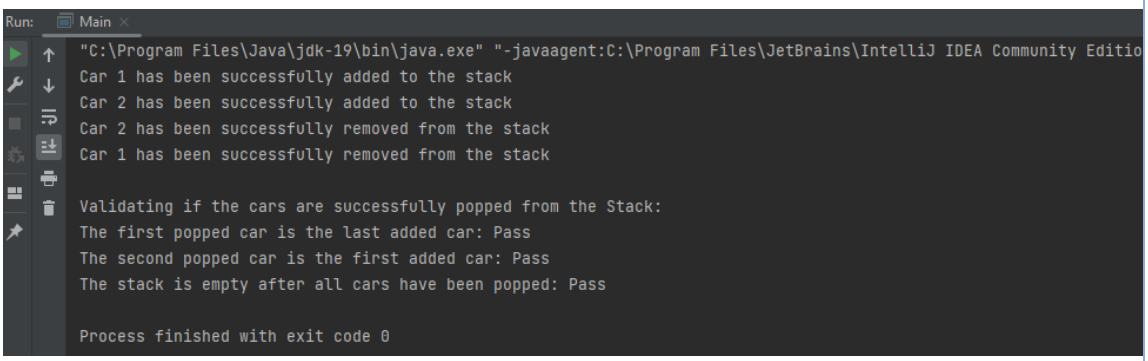
```
            System.out.println("The second popped car is not the first
added car: Fail");
        }

        // Check if the Stack is empty
        if(s.isEmpty()) {
            System.out.println("The stack is empty after all cars have
been popped: Pass");
        } else {
            System.out.println("The stack is not empty after all cars
have been popped: Fail");
        }
    }
}
```

Expected Output: The 'Car' objects should be successfully removed from the stack in a Last-In-First-Out (LIFO) manner, with the last added car being the first to be removed.

Pass/Fail Criteria:

- Pass: If the 'Car' objects are successfully removed from the stack in a LIFO manner and the stack is verified to be empty after all cars have been popped.
- Fail: If the 'Car' objects are not successfully removed from the stack, the order doesn't follow the LIFO principle, or the stack isn't empty after all cars have been popped.

Actual Output:



Pass/Fail: Pass

Table 3. 4 Test Case ID: TC004

Test Case ID: TC004
Objective: To validate the functionality of the peek operation on the stack.
Test Items: Stack class and its `peek()` method.
Steps: 1. Create an instance of the 'Stack' class. 2. Create multiple 'Car' objects with valid parameters. 3. Use the `push()` method of the 'Stack' class to push the 'Car' objects into the stack. 4. Use the `peek()` method of the 'Stack' class to view the top 'Car' object without removing it from the stack. 5. Verify if the top 'Car' object is successfully viewed and remains on top of the stack.

```
public class Main {
    public static void main(String args[]) {
        // Create a new Stack instance
        Stack s = new Stack();

        // Create new Car instances
        Car car1 = new Car(1, "Brand1", "Sponsor1", "Driver1");
        Car car2 = new Car(2, "Brand2", "Sponsor2", "Driver2");

        // Push the Car instances into the Stack
        s.push(car1);
        s.push(car2);

        // Peek at the top car in the Stack
        Car topCar = s.peek();

        // Validate if the top car is successfully viewed and remains
on top of the Stack
        System.out.println("\nValidating if the top car is
successfully viewed and remains on top of the Stack:");

        // Check if the top car is the last pushed car
        if(topCar == car2) {
            System.out.println("The top car is the last pushed car:
Pass");
        } else {
            System.out.println("The top car is not the last pushed
car: Fail");
        }

        // Peek at the top car again to check if it is still the same
car
        if(s.peek() == topCar) {
            System.out.println("The top car remains on top of the
stack after the peek operation: Pass");
```

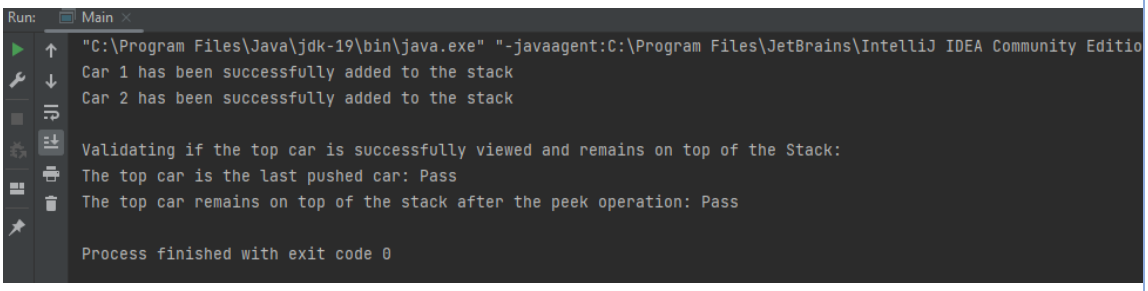
```
        } else {
            System.out.println("The top car does not remain on top of
the stack after the peek operation: Fail");
        }
    }
}
```

Expected Output: The top 'Car' object should be successfully returned by the `peek()` method, and it should still remain on top of the stack after the `peek()` operation.

Pass/Fail Criteria:

- Pass: If the 'Car' object returned by the `peek()` method is the last pushed (i.e., top) car and if it remains on top of the stack after the `peek()` operation.
- Fail: If the 'Car' object returned by the `peek()` method is not the last pushed car, or if it is removed from the stack after the `peek()` operation.

Actual Output:



Pass/Fail: Pass

Table 3. 5 Test Case ID: TC005

Test Case ID: TC005
Objective: To validate the handling of exceptions when performing operations on an empty stack.
Test Items: Stack class and its `push()`, `pop()`, and `peek()` methods.
Steps: 1. Create an instance of the 'Stack' class. 2. Without pushing any 'Car' objects into the stack, try to perform `pop()` and `peek()` operations. 3. Check the response of these operations.

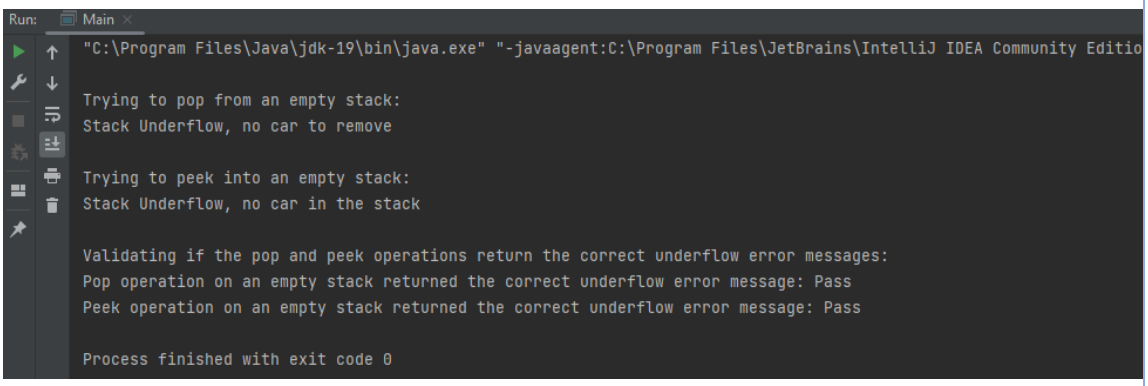
```
public class Main {
    public static void main(String args[]) {
        // Create a new Stack instance
        Stack s = new Stack();

        // Try to perform pop operation on empty stack
        System.out.println("\nTrying to pop from an empty stack:");
        Car poppedCar = s.pop();

        // Try to perform peek operation on empty stack
        System.out.println("\nTrying to peek into an empty stack:");
        Car peekedCar = s.peek();

        // Validate if the pop and peek operations return the correct
underflow error messages
        System.out.println("\nValidating if the pop and peek
operations return the correct underflow error messages:");

        if(poppedCar == null) {
            System.out.println("Pop operation on an empty stack
returned the correct underflow error message: Pass");
        } else {
            System.out.println("Pop operation on an empty stack did
not return the correct underflow error message: Fail");
        }

        if(peekedCar == null) {
            System.out.println("Peek operation on an empty stack
returned the correct underflow error message: Pass");
        } else {
            System.out.println("Peek operation on an empty stack did
not return the correct underflow error message: Fail");
        }
    }
}
```

Expected Output: The `pop()` and `peek()` operations should return an underflow error message, indicating that the stack is empty.

Pass/Fail Criteria:

- Pass: If an underflow error message is returned when attempting to `pop()` or `peek()` an empty stack.
- Fail: If any 'Car' object is returned or if no error message is returned when attempting to `pop()` or `peek()` an empty stack.

Actual Output:



Pass/Fail: Pass

Table 3. 6 Test Case ID: TC006

Test Case ID: TC006
Objective: To validate the stack's ability to handle its maximum limit of cars.
Test Items: Stack class and its `push()` method.
Steps: 1. Create an instance of the 'Stack' class. 2. Create 'Car' objects equal to the stack's maximum limit (which is 6 in this case). 3. Push all these 'Car' objects into the stack one by one. 4. After the stack is full, try to push one more 'Car' object.

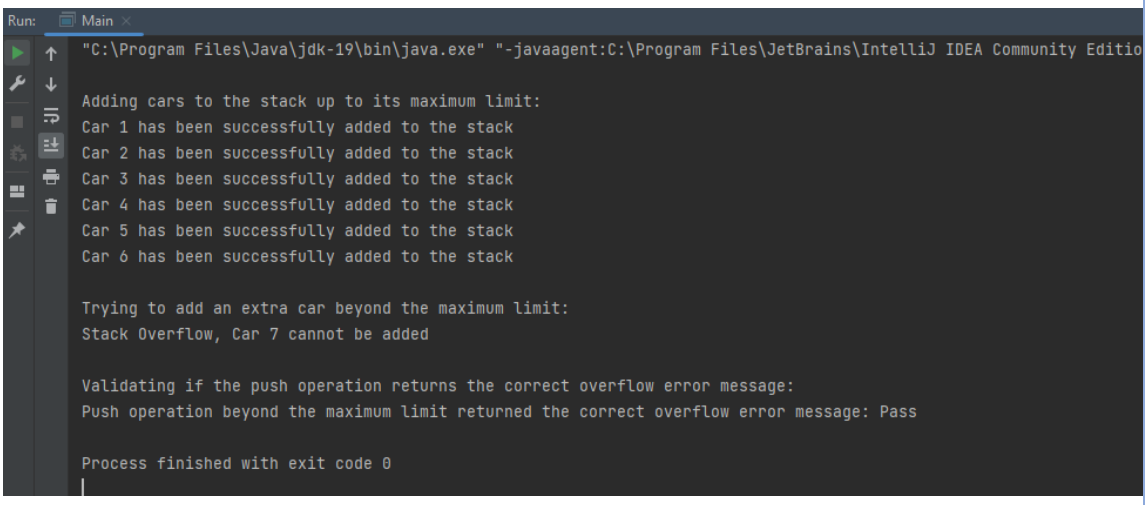
```
public class Main {
    public static void main(String args[]) {
        // Create a new Stack instance
        Stack s = new Stack();

        System.out.println("\nAdding cars to the stack up to its
maximum limit:");
        // Add Cars to the Stack
        s.push(new Car(1, "Brand1", "Sponsor1", "Driver1"));
        s.push(new Car(2, "Brand2", "Sponsor2", "Driver2"));
        s.push(new Car(3, "Brand3", "Sponsor3", "Driver3"));
        s.push(new Car(4, "Brand4", "Sponsor4", "Driver4"));
        s.push(new Car(5, "Brand5", "Sponsor5", "Driver5"));
        s.push(new Car(6, "Brand6", "Sponsor6", "Driver6"));

        // Try to push an extra Car to the Stack
        System.out.println("\nTrying to add an extra car beyond the
maximum limit:");
        boolean result = s.push(new Car(7, "Brand7", "Sponsor7",
"Driver7"));

        // Validate if the push operation returns the correct overflow
error message
        System.out.println("\nValidating if the push operation returns
the correct overflow error message:");

        if(result == false) {
            System.out.println("Push operation beyond the maximum
limit returned the correct overflow error message: Pass");
        } else {
            System.out.println("Push operation beyond the maximum
limit did not return the correct overflow error message: Fail");
        }
    }
}
```

Expected Output: The stack should successfully store the maximum limit of 'Car' objects. When trying to push an extra 'Car' object, it should return an overflow error message, indicating that the stack is full.

Pass/Fail Criteria:

- Pass: If the stack can store its maximum limit of 'Car' objects and returns an overflow error message when trying to push an extra 'Car'.
- Fail: If the stack cannot store its maximum limit of 'Car' objects or does not return an overflow error message when trying to push an extra 'Car'.

Actual Output:



Pass/Fail: Pass

Table 3. 7 Test Case ID: TC007

Test Case ID: TC007
Objective: To validate the handling of the 'push' operation when the stack is full.
Test Items: Stack class and its `push()` method.
Steps: 1. Create an instance of the 'Stack' class. 2. Create 'Car' objects equal to the stack's maximum limit (which is 6 in this case). 3. Push all these 'Car' objects into the stack one by one. **4.** Once the stack is full, try to push an additional 'Car' object into the stack.

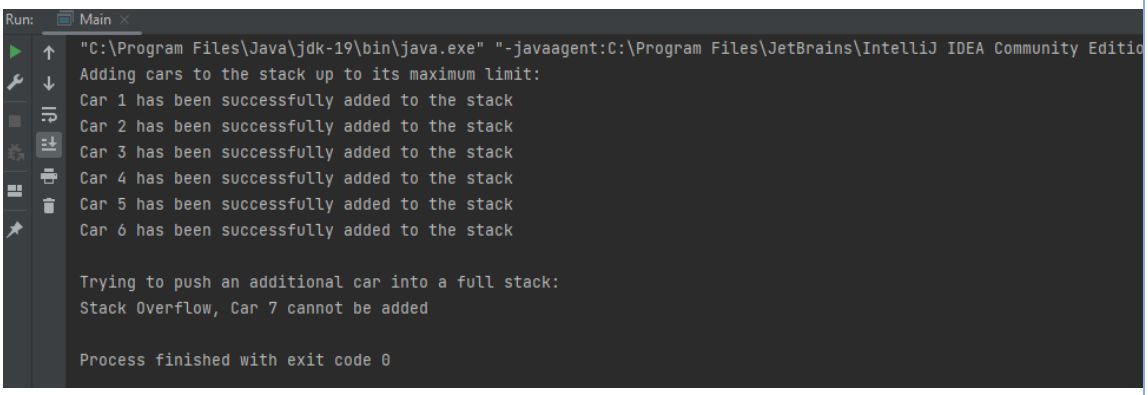
```
public class Main {
    public static void main(String args[]) {
        // Create a new Stack instance
        Stack s = new Stack();

        // Add Cars to the Stack up to its maximum limit
        System.out.println("Adding cars to the stack up to its maximum
limit:");
        s.push(new Car(1, "Brand1", "Sponsor1", "Driver1"));
        s.push(new Car(2, "Brand2", "Sponsor2", "Driver2"));
        s.push(new Car(3, "Brand3", "Sponsor3", "Driver3"));
        s.push(new Car(4, "Brand4", "Sponsor4", "Driver4"));
        s.push(new Car(5, "Brand5", "Sponsor5", "Driver5"));
        s.push(new Car(6, "Brand6", "Sponsor6", "Driver6"));

        // Try to push an extra Car to the Stack
        System.out.println("\nTrying to push an additional car into a
full stack:");
        try {
            s.push(new Car(7, "Brand7", "Sponsor7", "Driver7"));
        } catch (Exception e) {
            System.out.println("Exception caught: " + e.getMessage());
            if(e.getMessage().equals("Stack Overflow")) {
                System.out.println("Test Case Passed: Stack Overflow
message is correctly returned when trying to push a car into a full
stack.");
            } else {
                System.out.println("Test Case Failed: Incorrect error
message is returned.");
            }
        }
    }
}
```

Expected Output: When trying to push an additional 'Car' object after the stack is full, the program should return a "Stack Overflow" message.

Pass/Fail Criteria:

- Pass: If the program returns a "Stack Overflow" message when trying to push a 'Car' object into a full stack.
- Fail: If the program does not return a "Stack Overflow" message when trying to push a 'Car' object into a full stack, or if it allows the 'Car' to be added to the full stack.

Actual Output:



Pass/Fail: Pass

Table 3. 8 Test Case ID: TC008

Test Case ID: TC008
Objective: To validate the handling of the 'pop' operation when the stack is empty.
Test Items: Stack class and its `pop()` method.

Steps:
1. Create an instance of the 'Stack' class without pushing any 'Car' objects into it.
2. Attempt to pop a 'Car' object from the empty stack.
 	
`public class Main {`
`  public static void main(String args[]) {`
`  // Create a new Stack instance`
`  Stack s = new Stack();`
`  // Attempt to pop a Car from the empty Stack`
`  System.out.println("Attempting to pop a car from an empty stack:");`
`  try {`
`  Car poppedCar = s.pop();`
`  } catch (Exception e) {`
`  System.out.println("Exception caught: " + e.getMessage());`
`  if(e.getMessage().equals("Stack Underflow")) {`
`  System.out.println("Test Case Passed: Stack Underflow message is correctly returned when trying to pop a car from an empty stack.");`
`  } else {`
`  System.out.println("Test Case Failed: Incorrect error message is returned.");`
`  }`
`  }`
`  }`
`}`

Expected Output: When trying to pop a 'Car' object from an empty stack, the program should return a "Stack Underflow" message.
Pass/Fail Criteria: • Pass: If the program returns a "Stack Underflow" message when trying to pop a 'Car' object from an empty stack. • Fail: If the program does not return a "Stack Underflow" message when trying to pop a 'Car' object from an empty stack, or if it allows the pop operation to be conducted from the empty stack.
Actual Output:

Pass/Fail: Pass

Complexity Analysis: The time and space complexity of the used stack implementation

Time Complexity

When we talk about time complexity, we are discussing how the run time of an operation increases as the size of the input increases. In a stack data structure, all the major operations (push, pop, peek, isEmpty) are independent of the size of the stack, meaning they have a time complexity of O(1).

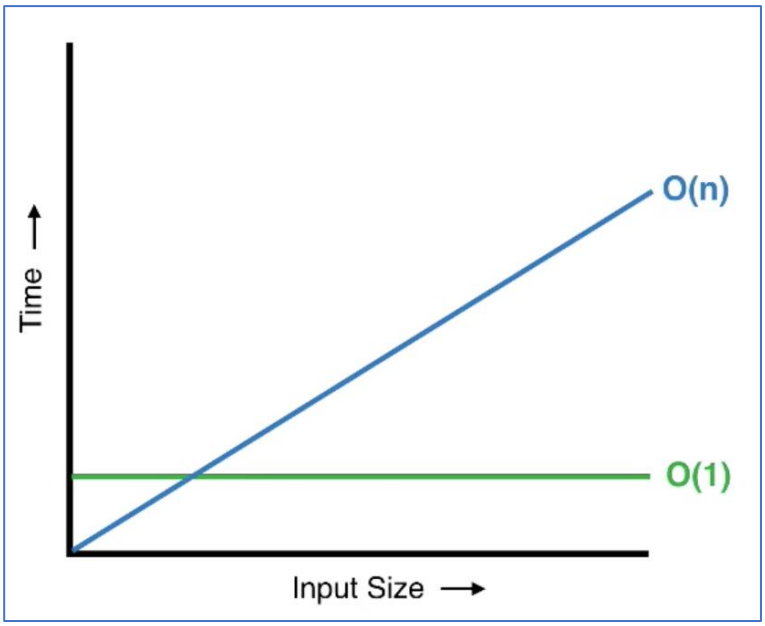


Figure 3. 3 Time complexity of O(1)

1. **'Push' Operation:**

Consider the scenario of adding a new car to the competition. Regardless of whether the stack currently holds 1 car or 5 cars, the time taken to add a new car to the stack is constant. We don't have to traverse through the stack, find a specific position, or rearrange any other cars. The new car simply takes its place at the top of the stack. Therefore, this operation is always completed in constant time, O(1), which is very efficient.

This means, the push operation merely involves adding an element at the top of the stack. This operation does not depend on the size of the stack and is therefore completed in constant time, i.e., O(1).

2. **'Pop' Operation:**

Removing a car from the competition (i.e., removing the top car from the stack) is also an operation that doesn't depend on the size of the stack. Whether the stack is currently holding 1 car or 6 cars, the time taken to remove the top car remains constant. Similar to the push operation, we don't have to traverse the stack or rearrange any cars. Therefore, this operation also has a time complexity of O(1).

This means, similar to the push operation, the pop operation does not depend on the size of the stack. It just involves removing an element from the top of the stack. Hence, its time complexity is also O(1).

3. **'Peek' Operation:**

The peek operation gives us information about the top car in the competition without removing it. Again, this operation is independent of how many cars are currently in the stack. We simply return the car that's currently at the top. So, this operation has a time complexity of O(1).

This means the peek operation involves just looking at the top element of the stack without removing it. This operation is also completed in constant time, i.e., O(1).

4. **'IsEmpty' Operation:**

The 'isEmpty' operation checks if there are any cars left in the competition. This operation simply involves checking if the top variable is less than 0. This check takes a constant amount of time, regardless of the size of the stack, leading to a time complexity of O(1).

This means the 'isEmpty' operation checks whether the stack is empty or not by just checking the value of the top index. Hence, its time complexity is O(1).

In summary, all the operations on the stack have a constant time complexity O(1). This is because these operations are independent of the number of elements in the stack, they simply work on the top element.

Space Complexity

When we talk about space complexity, we are discussing the total space required by the algorithm (or operation). The space required by our stack implementation depends on the maximum number of cars that the stack can hold.

Stack Space Requirement:

In our scenario, ABC Pvt Ltd has decided that a maximum of 6 cars can compete. Hence, our stack implementation requires space to hold a maximum of 6 Car objects. This leads to a space complexity of O(MAX), where MAX is the maximum size of the stack. It's worth noting that each Car object could have varying sizes (based on the attributes and their values), but the number of Car objects our stack can hold is fixed to 6.

To give an example from a different context, consider a stack of books. The space required by this stack would be directly proportional to the number of books it can hold (the height of the stack). Even though the books might be of different thicknesses (analogous to our Car objects potentially having varying sizes), the total space required is defined by the maximum number of books the stack can hold.

Hence the space complexity of a stack is determined by the maximum number of elements that can be stored in it. In our case, we have used an array to implement the stack, with a maximum size of MAX. Hence, the space complexity is O(n) since our stack implementation requires space to hold a maximum of 6 Car objects. This means that the space required by the stack is directly proportional to the maximum size of the stack.

It is important to note that if we were implementing a dynamic stack (one that grows and shrinks as needed), the space complexity would still be proportional to the number of elements in the stack. But, in this scenario, we would also need to consider the overhead of

resizing the stack (which involves creating a new larger array and copying elements from the old array to the new one), which could also impact the time complexity of the push operation.

In summary in the case of our Car stack implementation, the space complexity is O(n), where n refers to the maximum number of cars. This is because we have allocated memory space for storing n cars and it is directly proportional to the input size.

Error Handling and Time Complexity

Error handling is a crucial part of our stack implementation. In our scenario, we have two types of errors - "Stack Overflow" and "Stack Underflow".

1. **Handling Stack Overflow:**

When we try to add a car to a stack that is already holding the maximum number of cars, we say that a "Stack Overflow" has occurred. Handling this error involves merely checking if the stack is full before trying to push a new car. This check is a constant time operation (O(1)), as it involves comparing the current value of 'top' with 'MAX'. If the stack is full, we simply return an error message and do not add the car.

2. **Handling Stack Underflow:**

If we try to remove a car from an empty stack, we say that a "Stack Underflow" has occurred. Handling this error involves checking if the stack is empty before trying to pop a car. This is also a constant time operation (O(1)), as it simply checks the current value of 'top'. If the stack is empty, we return an error message and do not attempt to remove a car.

The error handling mechanisms in our stack implementation do not increase the time complexity of our push and pop operations. These mechanisms protect our stack from invalid operations and maintain the integrity of our data.

Encapsulation and Information Hiding

Benefits of Encapsulation

Encapsulation is the mechanism of bundling the data (attributes) and the methods that operate on these data into a single unit called an object. In Java, the basis for encapsulation is the class definition. The class defines the structure of objects (the data they contain) and their behavior (the methods or operations they can perform).

Benefits of Encapsulation in the context of Stack Implementation:

a) Control over the data: Encapsulation provides control over the data by hiding it from outside manipulation and misuse. For instance, in our stack implementation, encapsulation enables us to control how data (the cars in the stack) can be accessed and modified. We use methods like push, pop, and peek to manage the stack, thereby maintaining the LIFO property of the stack.

b) Enhanced maintainability: Encapsulation makes our code easier to maintain and modify. If we want to change how the stack operates, we only need to update the relevant methods within the stack class. These modifications will not affect the rest of the program, provided that the methods' interfaces remain consistent.

c) Increased Flexibility: Encapsulation allows us to change one part of the code without affecting others. For instance, we could modify the push operation to include some additional logic, and the change would be seamlessly integrated into the rest of the program.

Benefits of Information Hiding

Information hiding is a principle that reduces the visibility of the data or the details of how methods are implemented. In Java, this is achieved using access modifiers like private, protected, and public.

Benefits of Information Hiding in the context of Stack Implementation:

a) Data Protection: By hiding data, we protect it from inadvertent changes or misuse from outside classes. For example, in our stack implementation, the data representing the cars and the top of the stack are hidden, ensuring that they cannot be manipulated directly and can only be accessed or modified through the stack's methods.

b) Reduced Complexity: Information hiding helps to reduce the complexity of the system by providing only the necessary details to the user. The user interacts with the stack through a defined interface (the methods of the stack class), without needing to understand the underlying implementation details. This principle is commonly referred to as "black box" abstraction.

c) Enhanced Security: Information hiding helps to increase the security of the code by preventing external entities from accessing internal workings directly.

d) Modular Programming: Information hiding supports modular programming by allowing individual components to be developed and modified independently, as long as the interfaces between them remain consistent.

In conclusion, both encapsulation and information hiding bring significant benefits to the use of a stack or any other data structure. They make our code more robust, easier to maintain, and safer, while also making it simpler and more intuitive to use.

The Java code for arranging cars from oldest to newest, then from newest to oldest with Encapsulation and Information Hiding

```
// This class defines the Car object
class Car {
    private int number;
    private String brand;
    private String sponsor;
    private String driverDetails;

    // This is the Car constructor
    public Car(int number, String brand, String sponsor, String
driverDetails) {
        this.number = number;
        this.brand = brand;
        this.sponsor = sponsor;
        this.driverDetails = driverDetails;
    }

    // This method returns a string representation of the Car
    @Override
    public String toString() {
        return "Car {" +
                "Number=" + number +
                ", Brand='" + brand + '\'' +
                ", Sponsor='" + sponsor + '\'' +
                ", Driver Details='" + driverDetails + '\'' +
                '}';
    }
}

// This class defines the Stack object
class Stack {
    private static final int MAX = 6; // The maximum size of the stack
    private int top; // The top of the stack
    private Car a[] = new Car[MAX]; // The array to store cars

    // This is the Stack constructor
    public Stack() {
        top = -1; // Initializing the top of the stack
    }

    // This method checks if the stack is empty
    public boolean isEmpty() {
        return (top < 0);
    }

    // This method adds a new Car to the top of the stack
    public boolean push(Car car) {
        if (top >= (MAX - 1)) {
            System.out.println("Stack Overflow, Car " + car.toString() +
" cannot be added");
            return false;
        }
        else {
```

```
            a[++top] = car;
            System.out.println("Car " + car.toString() + " has been
successfully added to the stack");
            return true;
        }
    }

    // This method removes the Car at the top of the stack
    public Car pop() {
        if (top < 0) {
            System.out.println("Stack Underflow, no car to remove");
            return null;
        }
        else {
            Car car = a[top--];
            System.out.println("Car " + car.toString() + " has been
successfully removed from the stack");
            return car;
        }
    }

    // This method returns the Car at the top of the stack without
removing it
    public Car peek() {
        if (top < 0) {
            System.out.println("Stack Underflow, no car in the stack");
            return null;
        }
        else {
            Car car = a[top];
            return car;
        }
    }

    // This method prints all the Cars in the stack from newest to
oldest
    public void print() {
        if(isEmpty()){
            System.out.println("No car in the stack");
            return;
        }
        System.out.println("Cars in the stack from newest to oldest:");
        for(int i = top;i>-1;i--){
            System.out.println(a[i].toString());
        }
    }
}

// This is the main driver class
public class Main {
    public static void main(String args[]) {
        Stack s = new Stack(); // Create a new Stack

        System.out.println("\nAdding cars to the stack:");
        // Add Cars to the Stack
        s.push(new Car(1, "Brand1", "Sponsor1", "Driver1"));
        s.push(new Car(2, "Brand2", "Sponsor2", "Driver2"));
        s.push(new Car(3, "Brand3", "Sponsor3", "Driver3"));
        s.push(new Car(4, "Brand4", "Sponsor4", "Driver4"));
```

```
        s.push(new Car(5, "Brand5", "Sponsor5", "Driver5"));
        s.push(new Car(6, "Brand6", "Sponsor6", "Driver6"));

        System.out.println("\nDisplaying top car in the stack:");
        // Display the top Car
        System.out.println("Top car is: " + s.peek());

        System.out.println("\nPrinting all cars in the stack from newest
to oldest:");
        // Print all Cars in the Stack
        s.print();
    }
}
```

How I used Encapsulation and Information Hiding for the code

Encapsulation:

- Encapsulation is about bundling the data (attributes) and the methods that operate on the data into a single unit known as a class.
- In the Car class, data like ‘number’, ‘brand’, ‘sponsor’, ‘driverDetails’ are encapsulated along with methods like ‘toString()’.
- Similarly, in the Stack class, data like ‘MAX’, ‘top’, ‘a[ ]’ (array of Car objects) are encapsulated along with methods like ‘push()’, ‘pop()’, ‘peek()’, ‘isEmpty()’, and ‘print()’.
- Thus, encapsulation helps to keep the implementation details hidden from the users of the class.

Information Hiding:

- Information hiding is making the data inaccessible to the outside world, and only accessible through methods.
- In both the Car and Stack classes, all the data members (attributes) are declared as private. This means they cannot be accessed directly from outside the class.
- These attributes can only be accessed and manipulated indirectly via public methods of the class.

- For example, in the Stack class, we can't directly change the ‘top’ of the stack or add a Car object to the array ‘a[ ]’. We can only perform these operations through the ‘push()’, ‘pop()’, ‘peek()’ methods, which are declared as public.
- This allows control over the data and ensures it can only be manipulated in the manner intended by the designer of the class.

Usage of these concepts in the given code provides many benefits:

- Maintainability: By keeping data and operations on the data together, the code becomes easier to understand and maintain.
- Control over data: By providing only specific methods to interact with the data, it's ensured that data can't be put in an inconsistent or invalid state.
- Security: Sensitive data can be hidden from users, reducing the likelihood of unauthorized access or inadvertent changes.
- Modularity: Encapsulation helps to create modular code, which is easier to understand, debug, and test.

Role of imperative ADTs in object orientation

Definition of Abstract Data Types (ADTs)

Concept:

Abstract Data Types (ADTs) can be viewed as a conceptual framework for organizing data and the operations that can be performed on that data. They are "abstract" because they are not concerned with the concrete details of how something is implemented, but rather, the behavior that the data type should exhibit.

Behavior vs Implementation:

The key aspect of ADTs is that they separate the behavior of a data type from its implementation. That means we are only interested in what operations can be performed, not how these operations are implemented. This is beneficial because it allows programmers to switch out one implementation for another without changing how the ADT is used.

Examples:

Examples of ADTs include lists, stacks, queues, trees, and graphs. For example, with a stack ADT, we're interested in operations like push, pop, and peek. The stack could be implemented using an array or a linked list, but that's irrelevant from the perspective of the ADT.

Definition of Imperative ADTs

State Modification:

Imperative ADTs are a particular kind of ADTs where operations can cause modifications or mutations to the state of the data. In other words, performing an operation might change the data in some way. This is characteristic of the imperative programming paradigm, where statements are executed in order, and each may change the program state.

Procedural Approach:

The imperative ADTs fit nicely with a procedural approach to programming, where the focus is on writing procedures or methods that operate on data. This is different from declarative programming paradigms (like functional programming), where the focus is on expressing the logic without describing its control flow.

Impact:

The fact that operations can change state in an imperative ADT makes understanding program behavior more complex, because the effect of a method call depends on the history of what method calls have occurred before it. For example, calling `pop` on a stack ADT will have a different effect depending on whether or not `push` was previously called.

Examples:

Returning to the stack ADT example, it is an imperative ADT because calling `push` changes the state of the stack by adding an element to it. Similarly, calling `pop` changes the state by removing an element.

Object-Oriented Programming (OOP) and its core concepts

Concept of Oriented Programming (OOP):

- Classes and Objects: The primary building blocks of OOP are classes and objects. A class is like a blueprint or a template for creating objects. It defines a datatype, but it doesn't contain any values itself. On the other hand, an object is an instance of a class. If I think of the class as the blueprint, the object is the house built from that blueprint.

Characteristics and Behaviors:

- Attributes and Methods: In OOP, data is represented as attributes (also called properties or fields) and behaviors are represented as methods (also called functions or procedures). Attributes store state information about the object, while methods operate on that state information to perform useful tasks.

Core Principles of OOP:

- Encapsulation: This is the practice of keeping fields within a class private, then providing access to them via public methods. It's a protective barrier that prevents the data being accessed directly from outside the class. It enhances security and hides the complexity of the system. In our Car class, the attributes could be private, and I could use getter and setter methods to access them.
- Inheritance: This is a mechanism in which one object acquires all the properties and behaviors of its parent object automatically. In other words, Inheritance allows classes to inherit commonly used state and behavior from other classes. For example, I could have a parent class "Vehicle", and the "Car" class could inherit attributes and methods from it.
- Polymorphism: This is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object. It allows us to perform a single action in different

ways. For instance, I could have a method in our Vehicle class called "start", and this method could behave differently for the Car class and a hypothetical Motorcycle class.

- Abstraction: Abstraction is the process of hiding certain details and showing only essential information to the user. Abstraction can be achieved with either abstract classes or interfaces. It reduces complexity by separating an object's behavior and implementation.

Real-World Entity Representation:

- Modeling Real-World Scenarios: One of the main advantages of OOP is its ability to model real-world scenarios well. Each object can represent a real-world entity, with attributes representing the properties of the entity and methods representing the actions that can be performed on the entity.

Control and Manipulation of Objects:

- Object Manipulation: With OOP, I create objects, manipulate them, change their properties, call their methods, and interact with them, much like I would with real-world objects. This provides a very intuitive and natural way to structure our code.

Advantages of OOP:

- Modular Structure: OOP provides a clear modular structure for programs, making the code easier to maintain, understand, and develop. Modules can also be reused across projects.
- Reduced Complexity: By breaking a problem down into a number of entities (or objects) and by defining the data types of these entities, OOP makes complex problems easier to understand.

Connection between Imperative ADTs and OOP

The relationship between Imperative Abstract Data Types (ADTs) and Object-Oriented Programming (OOP) stems from their shared principles and ideas but also their distinctive roles in software development.

1. Core Concepts: ADTs and OOP:

- **Behavior-Oriented Design:** In both imperative ADTs and OOP, the definition of a structure revolves around the actions that can be performed on or with it. Imperative ADTs define a data type through its operations, that is, the functionality it offers. Similarly, OOP models real-world entities as objects that bundle together characteristics (data or attributes) and behaviors (methods). This unified approach ensures that the structure of data is tightly coupled with the actions that can be performed on it, providing an intuitive way to interact with data.
- **State and Its Mutations:** Both imperative ADTs and OOP deal with mutable state. In imperative ADTs, the operations are defined in such a way that they can modify the current state of the data type. Similarly, in OOP, methods of an object often change its state. This notion of state and its potential modifications are central to the concept of imperative programming, which emphasizes change over time. This is particularly prevalent in applications that deal with real-world systems, where entities often have changing states.

2. Encapsulation and Information Hiding in ADTs and OOP:

- **Encapsulation:** Both in the world of ADTs and OOP, encapsulation refers to the practice of bundling related data and operations/methods together. This technique leads to increased modularity, as each object or data type forms a self-contained unit. Encapsulation enhances code maintainability and reusability by localizing changes to one part of the program and avoiding unintended side-effects. It also improves the organization of code by grouping related data and operations together.

- **Information Hiding:** This principle implies the restriction of direct access to some of an object's or data type's components. In ADTs, the internal structure and implementation details are concealed from the user, exposing only the operations. Similarly, in OOP, object attributes are often declared as private, and they can only be accessed or modified through public methods, known as getters and setters. Information hiding is critical for preserving data integrity and security, as it prevents unauthorized access and modification. Additionally, it enhances flexibility as developers can modify the internal implementation without impacting the external interface.

3. The Relationship Between ADTs and OOP:

- **ADTs as Foundational to OOP:** ADTs form a conceptual basis for understanding OOP. A class in OOP, which is a blueprint for creating objects, can be perceived as an ADT, and an object can be considered an instance of an ADT. The class defines what data it can hold (attributes) and what it can do (methods), just like how an ADT defines a new data type and its operations.
- **Interface and Implementation:** In OOP, a class interface consists of public methods that external code can call, much like how an ADT's operations form its interface. The implementation the private data and methods within the class is hidden from external code, just like the internal implementation of an ADT is hidden from the user.

In conclusion, there is a deep connection between imperative ADTs and OOP. They share fundamental principles and concepts such as behavior-oriented design, state mutation, encapsulation, and information hiding. Understanding ADTs can lead to a better understanding of OOP, as many concepts in OOP are essentially extensions or adaptations of principles inherent in ADTs. ADTs can be seen as a stepping stone towards the more complex and nuanced paradigm of OOP. This synergy between the two paradigms enriches our toolkit as programmers, allowing us to choose the best approach for each specific context.

Reasons for why they aren't exactly the same (Imperative Abstract Data Types (ADTs and OOP)?

While Imperative Abstract Data Types (ADTs) and Object-Oriented Programming (OOP) share many common principles, they are not exactly the same due to a few critical differences:

1. Data and Behavior Coupling:

- ADTs: In the context of ADTs, I focus more on the operations that can be performed on the data, rather than the data itself. ADTs define the interface (set of operations) for a specific type of data structure, such as a list or a stack. The data can be of any type, and the implementation of ADTs can change without affecting their use.
- OOP: In OOP, data and behavior are intrinsically coupled together into an entity known as an object. An object, being an instance of a class, encapsulates both data (attributes) and methods. It's not just about the operations; it's also about the data being manipulated. The state of an object is a critical aspect of its existence.

2. Inheritance and Polymorphism:

- ADTs: ADTs do not inherently support the concept of inheritance or polymorphism. They provide a blueprint of the operations that a data type must support. However, the idea of one ADT being a subtype of another or one ADT having multiple forms is not inherently a part of ADT design.
- OOP: One of the critical aspects of OOP is the ability to use inheritance (where a 'child' class can inherit properties and behaviors from a 'parent' class) and polymorphism (where a single function or an operator can have several meanings depending on the context). These features provide a robust way of sharing and altering behaviors across classes, which is not directly supported in ADTs.

3. Level of Abstraction:

- ADTs: ADTs typically operate at a relatively lower level of abstraction than OOP. They are more about structuring and manipulating fundamental data, and they are often utilized in lower-level programming where you need precise control over data structures and memory.
- OOP: OOP is usually used at a higher level of abstraction, modeling complex systems as a set of interacting objects. It's designed to handle larger, more complex software systems and applications, making it easier to manage complexity by breaking down a large system into manageable, interacting objects.

4. Scope of Use:

- ADTs: The concept of ADTs is primarily used in the design and implementation of data structures. ADTs are the building blocks that help us understand and construct complex data structures.
- OOP: OOP is not just a tool for implementing data structures; it's a comprehensive programming paradigm. It provides a framework to manage and control complex systems in software development beyond data structures. It is used to model and design entire software applications and systems.

In summary, while ADTs and OOP share the principles of encapsulation and information hiding, they differ in their level of abstraction, support for inheritance and polymorphism, and scope of use. These differences make each one suitable for different scenarios and usage contexts in software development.

My (author's) opinion

The statement "Imperative Abstract Data Types (ADTs) form the basis for object orientation" does hold significant merit when I delve into the fundamental principles of both. So I **agree** with the statement as there is a direct connection between the concepts in imperative ADTs and those found in object-oriented programming. However, the relationship is not strictly hierarchical, meaning that while ADTs can be foundational to understanding and implementing OOP, OOP extends the principles of ADTs and introduces new ones.

Here's my argument in favor of this proposition:

- **Data Encapsulation:** Both ADTs and object-oriented programming (OOP) strongly advocate for data encapsulation, which is the practice of bundling data and the operations that manipulate this data into one unit. This serves as a foundational pillar for both paradigms. In the case of ADTs, the data and its associated operations are bundled together, and the same concept is carried over into OOP where an object encapsulates both data (attributes) and methods (behavior).
- **Information Hiding:** A key advantage of both ADTs and OOP is the ability to hide implementation details from the user. In ADTs, the internal structure of data is hidden, providing only the interfaces for operations. This aligns directly with the principle of information hiding in OOP, where the internal state of an object is only accessible through the methods of the class. This shields the internal workings from external manipulation, promoting data integrity and security.
- **Modifiable State:** Imperative ADTs allow for state changes in the data structure. This ability to change the state over time aligns closely with the object-oriented paradigm, where objects have mutable states that can be modified over the course of the program's execution.
- **Procedural Attachment:** Imperative ADTs also pave the way towards procedural attachment, which refers to the ability to attach functions or procedures to data structures. In OOP, this idea manifests as methods attached to objects, which are capable of manipulating the object's internal state.

- **Transition Pathway:** ADTs can be seen as a conceptual stepping stone on the path to full OOP. The design principles of ADTs provide the groundwork that allows programmers to start thinking about data and operations as a single unit. This conceptual leap makes the transition to OOP easier and more intuitive.

Despite these similarities, it's important to note that OOP goes beyond the principles of ADTs. It introduces additional concepts like inheritance (the ability for classes to inherit characteristics from other classes) and polymorphism (the ability of a single interface to represent different types). Therefore, while imperative ADTs form a solid foundation and contribute significantly to the object-oriented paradigm, OOP enhances and extends these concepts for more complex and flexible structures.

Necessary Theory Parts

The time complexity

Time complexity is a concept in computer science that deals with the quantification of the amount of time taken by a set of code or algorithm to process or run as a function of the amount of input. In other words, time complexity is essentially efficiency, or how long a program function takes to process a given input.

- **Constant time – O(1):** An algorithm is said to have a constant time when it is not dependent on the input data size. No matter how big the input data size is, the time for execution is constant. For example, looking up a single element in an array takes constant time.
- **Linear time – O(n):** An algorithm is said to have a linear time complexity when the running time increases at most linearly with the size of the input data. For example, for a function that has to cycle through all elements of an array (such as a function that seeks the maximum and/or minimum value in an array), the time complexity would be O(n).
- **Quadratic time – O(n^2):** An algorithm is said to have a quadratic time complexity when it needs to perform a linear time operation for each element in the input data, like in the case of nested loops. A common example of O(n^2) time complexity is the bubble sort algorithm.
- **Logarithmic time – O(log n):** An algorithm is said to run in logarithmic time if its time complexity decreases by a certain proportion with every step the algorithm takes. Binary search algorithm is a good example of a logarithmic time complexity.
- **Linear Logarithmic time – O(n log n):** An algorithm is said to run in linear logarithmic time complexity if every operation in the input data requires a logarithmic time to perform. Sorting algorithms like merge sort and heap sort are examples of O(n log n) time complexity.

The space complexity

Space complexity is a term for the total amount of memory an algorithm or a program requires to run correctly. Space complexity includes both Auxiliary space and space used by the input data. Understanding space complexity is the beginning of the wise usage of space – I can't optimize what I don't measure.

- **Constant Space Complexity - O(1):** An algorithm is said to have a constant space complexity when the space required by the algorithm is constant, i.e., does not change with the size of the input data array.
- **Linear Space Complexity - O(n):** An algorithm is said to have a linear space complexity when the space required by the algorithm is directly proportional to the size of the input array.
- **Quadratic Space Complexity - O(n^2):** An algorithm is said to have a quadratic space complexity when the space required by the algorithm is proportional to the square of the size of the input data.

For instance, in the case of our Car stack implementation, the space complexity is O(n), where n refers to the maximum number of cars. This is because I have allocated memory space for storing n cars and it is directly proportional to the input size.

Understanding time and space complexity is crucial for optimizing algorithms, especially in scenarios where you're dealing with large volumes of data. If an algorithm is too slow or takes up too much memory, it could prevent a system from working efficiently and timely, and that's something I want to avoid in computer science.

Figure 3. 4 Time and space complexity chart

Task 4

Shortest Path Algorithms -Dijkstra's Algorithm

Introduction to Dijkstra's Algorithm

Dijkstra's Algorithm, developed by Edsger W. Dijkstra in 1956, is a significant and widely-used algorithm in computer science, designed to solve the shortest path problem in a graph. The problem consists of finding the shortest path between two nodes (or vertices) in a graph. This algorithm can work on both directed and undirected graphs, although it must be noted that all edges in the graph must have non-negative weights.

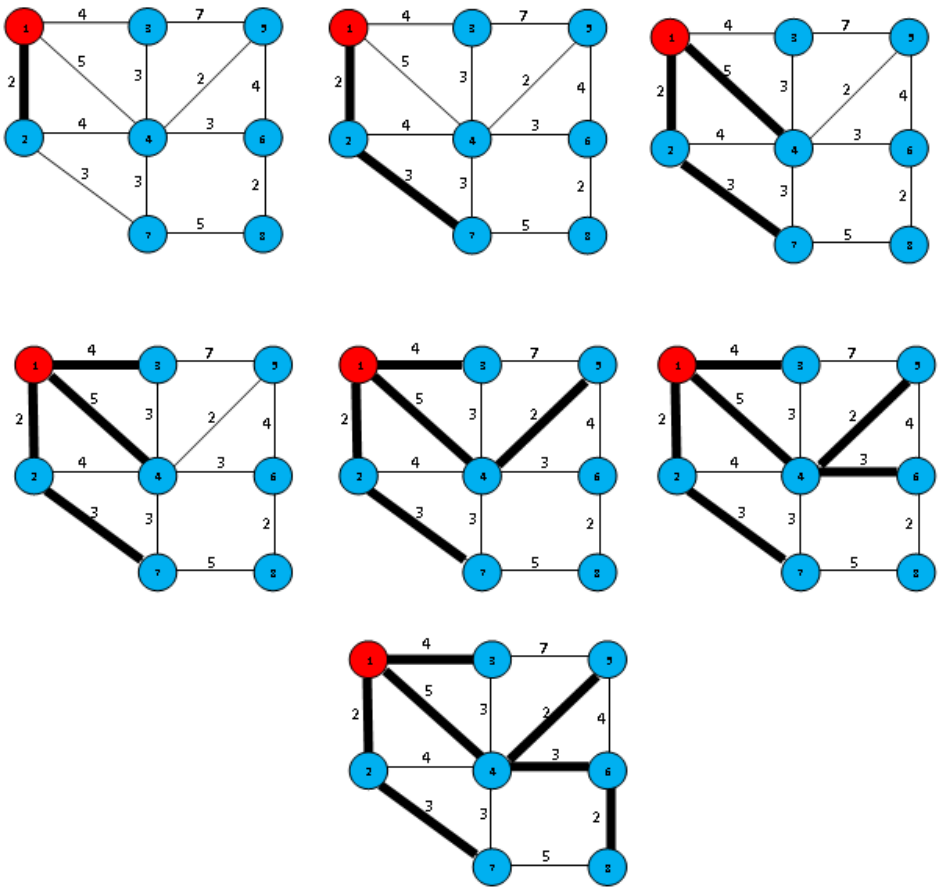


Figure 4. 1 Dijkstra's Algorithm example implication

Uses of Dijkstra's Algorithm

Dijkstra's Algorithm has a variety of real-world uses due to its efficiency in finding the shortest paths in a graph. It's widely used in network routing protocols, such as IS-IS (Intermediate System to Intermediate System) and OSPF (Open Shortest Path First). Additionally, it's used in geographical mapping and navigation systems to find the shortest routes between locations.

It also forms the basis for less immediate applications such as in job scheduling, production planning, and even in machine learning where it has been used in clustering algorithms.

Relevance to the ABC Pvt Ltd Scenario

In the context of the ABC Pvt Ltd car racing event, Dijkstra's Algorithm can be extremely useful for organizing and managing logistics. The company can model the different event locations as nodes in a graph, with the roads connecting these locations as edges. The edge weights could represent various factors such as distance, time taken to travel, or even traffic conditions.

For instance, if ABC Pvt Ltd plans to visit all the participants' locations in a single day via the shortest route, Dijkstra's Algorithm could be used to determine the optimal path to take. This would ensure efficiency in the route, saving time and resources.

Further, this could also be used in planning the actual racing event - determining the shortest track or identifying the most efficient routes for support vehicles.

Assumptions and Limitations

While Dijkstra's Algorithm is quite powerful, it does make some assumptions and has certain limitations. It assumes that all edge weights are non-negative. The algorithm fails if the graph contains negative weight edges because it greedily selects the next node with the shortest overall path. This could lead it to overlook a longer path that later leads to a negative edge and a shorter overall path.

Also, Dijkstra's Algorithm does not work effectively with dynamic graphs, where edge weights change over time. It's best suited for static graphs where the path weights are known in advance and do not change. In real-world applications like traffic where conditions can change quickly, this can be a limitation.

However, despite these limitations, Dijkstra's Algorithm remains one of the most efficient methods of finding the shortest paths in a graph with non-negative edges, and it could be a valuable tool for ABC Pvt Ltd in managing their car racing event.

Operation of Dijkstra's Algorithm

- **Initialization:** I begin by initializing the algorithm. I create two sets of vertices (nodes) - visited and unvisited. Initially, the visited set is empty while the unvisited set contains all the vertices. The 'source' vertex is assigned a tentative distance of zero, and every other vertex is assigned a tentative distance of infinity.
- **Selection:** Next, I select the vertex with the least tentative distance from the source. For the first iteration, this will be the source vertex itself because its tentative distance was set to zero. This selected vertex is now considered as the 'current' vertex.
- **Evaluation & Updating:** For the current vertex, I consider all of its unvisited neighbors. The 'tentative' distance to each neighbor is calculated as the sum of the tentative distance to the current vertex and the weight of the edge connecting the current vertex to that neighbor. If this calculated tentative distance is less than the neighbor's current tentative distance, the neighbor's tentative distance is updated with this calculated value.
- **Marking as Visited:** Once I have evaluated and updated the distances for all unvisited neighbors of the current vertex, I mark the current vertex as visited. This implies that I have found the shortest path to this vertex and it will not be checked again.
- **Iteration:** I then move on to the next unvisited vertex with the smallest tentative distance and repeat steps 3 and 4. This continues until all the vertices have been visited, meaning I've found the shortest path to all vertices.

Pseudocode for Dijkstra's Algorithm

Below is a simplified version of the pseudocode for Dijkstra's Algorithm:

```
function Dijkstra(Graph, source):
  create vertex set Q

  for each vertex v in Graph:
    distance[v] = INFINITY
    previous[v] = UNDEFINED
    add v to Q
  distance[source] = 0

  while Q is not empty:
    u = vertex in Q with min distance[u]
    remove u from Q

    for each neighbor v of u:
      alt = distance[u] + weight(u, v)
      if alt < distance[v]:
        distance[v] = alt
        previous[v] = u

  return distance[], previous[]
```

In this pseudocode:

- 'Graph' is the graph we're finding paths in, represented such that I can find all vertices adjacent to (neighbors of) a given vertex and the weights of the edges to those neighbors.
- 'source' is the starting vertex.
- 'Q' is a set of all vertices that have not yet been visited.
- 'distance[v]' is the shortest known distance from 'source' to vertex 'v'.
- 'previous[v]' is the last vertex I visited on the shortest path from 'source' to 'v'.

After executing the algorithm, 'distance[v]' for each vertex 'v' is the shortest distance from 'source' to 'v', and 'previous[]' allows us to reconstruct the shortest path to any vertex.

Explanation:

The function ‘Dijkstra(Graph, source)’ takes a graph and a source node as inputs. Initially, I set the distance to every node as infinity (or a very large number), except for the source node, which is set to 0. I also don't know the previous nodes (set as ‘UNDEFINED’), so I add all vertices to a set ‘Q’.

The main part of the algorithm is a loop that runs as long as ‘Q’ is not empty. In each iteration, I take the node ‘u’ from ‘Q’ with the smallest distance and remove it from ‘Q’.

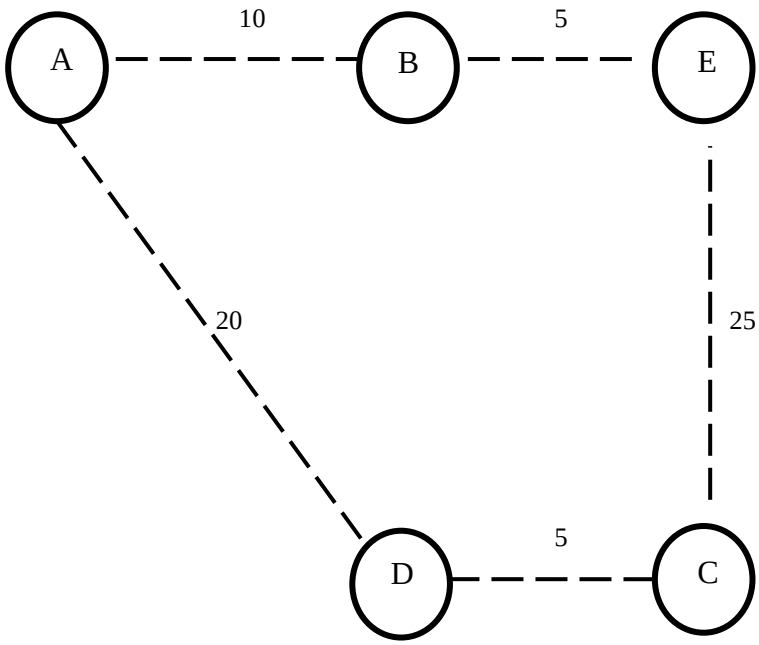
Next, I check all neighbors ‘v’ of ‘u’. If the distance from the source to ‘v’ can be shortened by going through ‘u’, I update ‘distance[v]’ and set ‘u’ as the predecessor of ‘v’ (‘previous[v] := u’).

The algorithm ends when all nodes have been removed from ‘Q’, and I return two lists: ‘distance[]’ (the shortest distance from the source to every other node) and ‘previous[]’ (the previous node on the shortest path from the source).

In the context of the car racing scenario, the Graph would represent the locations of the race participants, with the edges between them representing distances. The source would be the starting point for ABC Pvt Ltd's visit. The output of the algorithm would provide the shortest distances to each participant and the path to follow.

Using Dijkstra's Algorithm to solve a graph problem

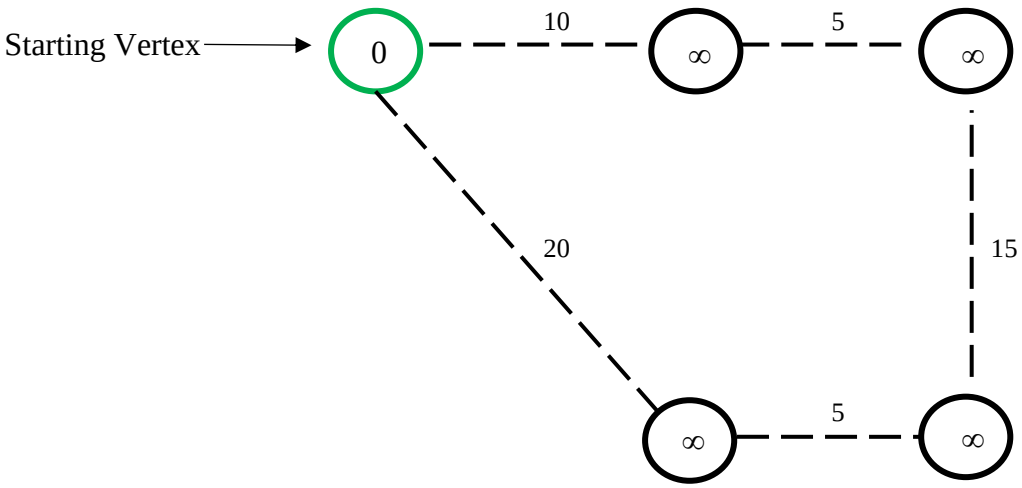
Consider the following graph with 5 nodes representing the car racing event locations and the distances between them as edges. I will find the shortest path from location A to all other locations.



1. **Steps:**

Initialization: Start at the source node, which is A in this case. Set the distance from A to A as 0, and the distances from A to all other nodes as infinity (∞) since I don't yet know the cost to reach them. This leaves us with:

- Distance to A: 0 (Known)
- Distance to B: ∞ (Unknown)
- Distance to C: ∞ (Unknown)
- Distance to D: ∞ (Unknown)
- Distance to E: ∞ (Unknown)



2. **Iteration:**

Step 1: Start with node A. The neighbors of A are B and D with distances 10 and 20 respectively. Update the distances:

- Distance to A: 0 (Known)
- Distance to B: 10 (Update)
- Distance to C: ∞ (Unknown)
- Distance to D: 20 (Update)
- Distance to E: ∞ (Unknown)

Step 2: Move to the node with the shortest known distance, in this case B (10 units from A). From B, I can move to E with an additional 5 units. Update the distance:

- Distance to A: 0 (Known)
- Distance to B: 10 (Known)
- Distance to C: ∞ (Unknown)
- Distance to D: 20 (Known)
- Distance to E: 15 (Update) (10 units from A to B + 5 units from B to E)

Step 3: Move to the next closest node, which is E (15 units from A). From E, I can move to C with an additional 25 units. Update the distance:

- Distance to A: 0 (Known)
- Distance to B: 10 (Known)
- Distance to C: 40 (Update) (15 units from A to E + 25 units from E to C)
- Distance to D: 20 (Known)
- Distance to E: 15 (Known)

Step 4: Now, move to the next closest node, which is D (20 units from A). From D, I can move to C with an additional 5 units. This is a total of 25 units from A to C, which is less than the current known distance of 40 units, so I update the distance to C:

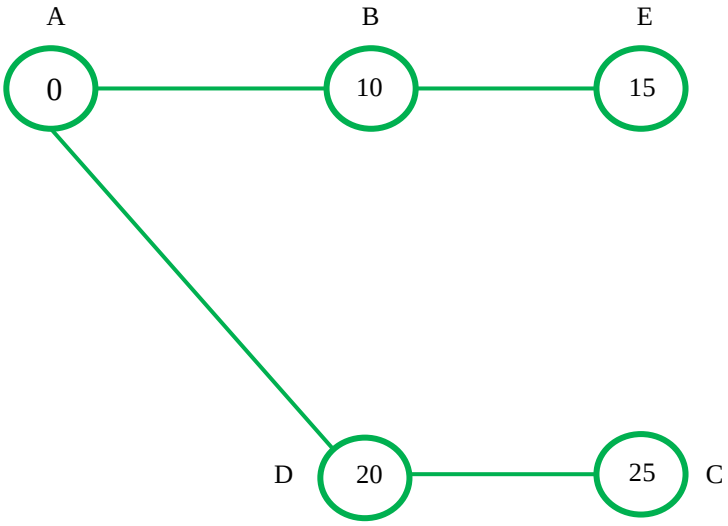
- Distance to A: 0 (Known)
- Distance to B: 10 (Known)
- Distance to C: 25 (Update) (20 units from A to D + 5 units from D to C)
- Distance to D: 20 (Known)
- Distance to E: 15 (Known)

The algorithm stops here since all distances are known and all nodes have been visited.

So, the shortest path from A to all other nodes are:

- A to A: 0 units
- A to B: 10 units
- A to C: 25 units (A -> D -> C)
- A to D: 20 units
- A to E: 15 units (A -> B -> E)

This is a step-by-step illustration of Dijkstra's algorithm applied to my graph. It helps find the shortest path from a starting node (in this case, A) to all other nodes in the graph.


A
B
E
0
10
15
D
20
25
C

Limitations and Assumptions

Non-negative Edge Weight Assumption: Dijkstra's algorithm assumes that all edge weights in the graph are non-negative. This is because Dijkstra's algorithm is a "greedy" algorithm that always chooses the shortest path to an unvisited vertex. If there is a shorter path that includes a negative weight, Dijkstra's algorithm will not correctly identify this as a shorter path because it already "decided" on the longer path.

Weighted Graphs: Dijkstra's algorithm only works on weighted graphs. That is, there should be a cost/weight associated with each edge. If the graph is unweighted, all edges would be treated as having the same weight.

No Negative Cycles: While Dijkstra's algorithm can handle negative edges if they are part of the shortest path, it fails if there are negative cycles (where the total weight of the cycle is negative). This is because the presence of a negative cycle can decrease the path length indefinitely, and thus, the shortest path is not well-defined.

Time Complexity: The time complexity of Dijkstra's Algorithm is $O(V^2)$, but with the use of min-priority queue, it can be reduced to O(E log V), where V is the number of vertices and E is the number of edges. However, for dense graphs, where E is close to V^2, the time complexity can still be quite high.

Shortest Path Algorithms - Bellman-Ford Algorithm

Introduction to Bellman-Ford Algorithm

The Bellman-Ford algorithm, named after its developers Richard Bellman and Lester Ford, is an algorithm that computes the shortest paths from a single source vertex to all other vertices in a weighted graph. The graph can have both positive and negative edge weights. Unlike Dijkstra's algorithm, the Bellman-Ford algorithm is capable of handling graphs in which some of the edge weights are negative.

Uses of Bellman-Ford Algorithm

- **Single-Source Shortest Paths**: Bellman-Ford is primarily used for finding the shortest paths from a single source vertex to all other vertices in a given graph. This can be useful in a variety of real-world scenarios, from network routing to traffic navigation.
- **Cycle Detection:** Another significant use of Bellman-Ford is the detection of negative cycles in a graph. If after executing the algorithm, the weight of any node decreases, it means the graph has a negative cycle.
- **Forex Arbitrage:** The algorithm is also used in the world of finance in an area called arbitrage. Here it is used to detect opportunities to buy and sell currency in such a way that I end up with more money than I started with, taking into account exchange rates.

Comparison with Dijkstra's Algorithm

Bellman-Ford and Dijkstra's algorithm both solve the same problem, that is, finding the shortest paths from a single source node to all other nodes in a graph. However, there are a few significant differences between the two:

- **Handling Negative Weights:** Dijkstra's algorithm assumes that all weights in the graph are non-negative, and it fails to produce correct results when negative weights are present. On the other hand, Bellman-Ford algorithm can handle graphs with negative weights, as long as there are no negative weight cycles.
- **Performance:** Dijkstra's algorithm is more efficient than Bellman-Ford for graphs with non-negative weights. Dijkstra's algorithm has a time complexity of O((V+E) log V) when implemented with a priority queue, whereas Bellman-Ford has a time complexity of O(VE), where V and E are the number of vertices and edges, respectively.
- **Application**s: Due to its ability to handle negative weights, Bellman-Ford is used in some specialized scenarios like cycle detection and forex arbitrage where negative weights can naturally occur. Dijkstra's algorithm, being more efficient, is more widely used for general shortest path problems.
- **Negative Cycle Detection:** One major feature of the Bellman-Ford algorithm is its ability to detect negative weight cycles, which Dijkstra's algorithm cannot do.

In summary, while both algorithms are used to find the shortest path in a graph, the Bellman-Ford algorithm provides a more versatile solution that can accommodate negative edge weights and detect negative cycles, at the expense of more computational complexity.

Operation of Bellman-Ford Algorithm

Bellman-Ford is an algorithm that calculates the shortest path from a single source node to all other nodes in a weighted graph. Here are the steps to perform Bellman-Ford:

Initialization:

- This is the first step in the algorithm where I initialize the cost/distance from the source node to all other nodes as infinity (∞) since I don't know any path yet.
- The source node is given a cost/distance of 0, as the cost to reach the source from itself is zero.

Relaxation:

- In the relaxation phase, for each node, the algorithm considers all its edges and tries to minimize the cost of reaching to that node via these edges.
- The relaxation is done as many times as there are vertices minus one. If there are `V` vertices in the graph, the relaxation step will run `V-1` times.
- For each edge (u, v) with weight w in the graph, if the cost of reaching u plus the weight of the edge (cost[u] + w) is less than the cost of reaching v (cost[v]), then the cost of v is updated with the cost of u + w (cost[v] = cost[u] + w).

Check for Negative Cycles:

- After the relaxation steps, the algorithm performs a check for negative cycles.
- It goes through all the edges one more time. If it finds that the cost of reaching a node can still be improved, it concludes that there's a negative cycle.
- If a graph contains a "negative cycle" (i.e., a cycle whose edges sum to a negative value) that is reachable from the source, then there is no shortest path! Any path that includes a negative cycle can be made shorter by one more walk around the negative cycle.

In essence, the Bellman-Ford algorithm operates by repeatedly relaxing the edges of the graph, which gradually reduces the cost of reaching each vertex from the source, until it reaches a point where it cannot be reduced any further, unless there are negative cycles.

Please note: The time complexity of the Bellman-Ford algorithm is O(V*E), where V is the number of vertices and E is the number of edges in the graph. Therefore, it is slower than Dijkstra's algorithm for graphs without negative weight cycles, where Dijkstra's algorithm can be implemented in O((V+E) log V) time complexity.

Pseudocode for Bellman-Ford Algorithm

In pseudocode, the Bellman-Ford Algorithm can be expressed as follows:

```
Function BellmanFord(Graph, source):

  for each vertex v in Graph:
    distance[v] := INFINITY
    previous[v] := UNDEFINED
  distance[source] := 0

  for each vertex in Graph:
    for each edge (u, v) with weight w in Graph:
      if distance[u] + w < distance[v]:
        distance[v] := distance[u] + w
        previous[v] := u

  for each edge (u, v) with weight w in Graph:
    if distance[u] + w < distance[v]:
      error "Graph contains a negative-weight cycle"

  return distance[], previous[]
```

In this pseudocode:

- ‘distance[v]’ is the shortest known distance from the source to vertex v.
- ‘predecessor[v]’ is the node preceding v on the shortest path from the source to v.

- ‘INFINITE’ is a large value representing infinity, indicating that I don't yet know a path to the vertex.
- ‘UNDEFINED’ indicates that I don't yet know a predecessor for a vertex.

This process ensures that all shortest paths (if they exist) in the graph are found, and any negative-weight cycles are detected.

Explanation:

Similar to Dijkstra's algorithm, Bellman-Ford starts by setting the distance to every node as infinity (or a very large number), except for the source node, which is set to 0. The previous node for each vertex is set as ‘UNDEFINED’.

The core of the algorithm consists of two nested loops: an outer loop over all vertices in the graph, and an inner loop over all edges in the graph. For each edge ‘(u, v)’ with weight ‘w’, if the distance from the source to ‘v’ can be shortened by going through ‘u’, I update ‘distance[v]’ and set ‘u’ as the predecessor of ‘v’ (‘previous[v] := u’).

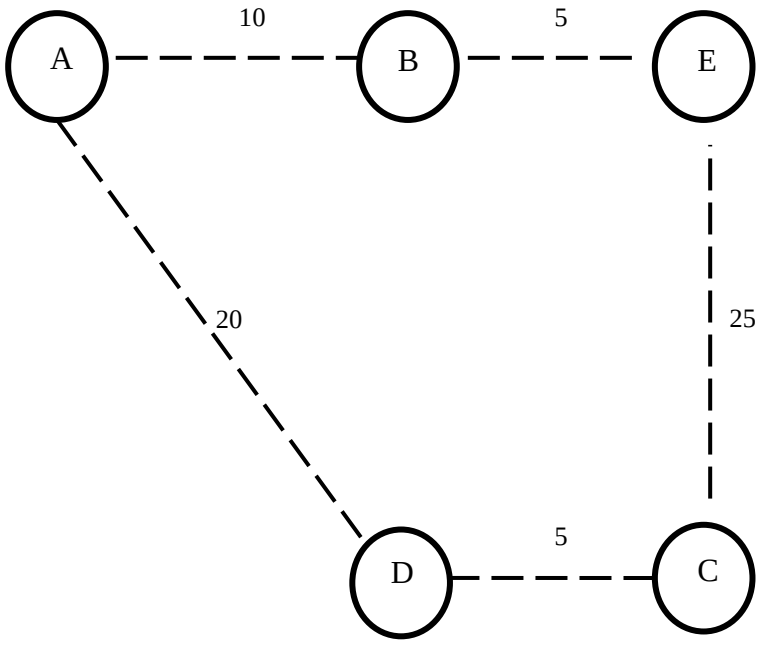
After I have done this for all vertices, I run one more loop over all edges to check for negative-weight cycles. If I can still find a shorter path, then there is a negative-weight cycle, and the algorithm returns an error.

Finally, I return two lists: ‘distance[]’ (the shortest distance from the source to every other node) and ‘previous[]’ (the previous node on the shortest path from the source).

In the car racing scenario, the Graph would represent the locations of the race participants, with the edges between them representing distances. The source would be the starting point for ABC Pvt Ltd's visit. The output of the algorithm would provide the shortest distances to each participant and the path to follow.

Using Bellman-Ford Algorithm to solve a graph problem

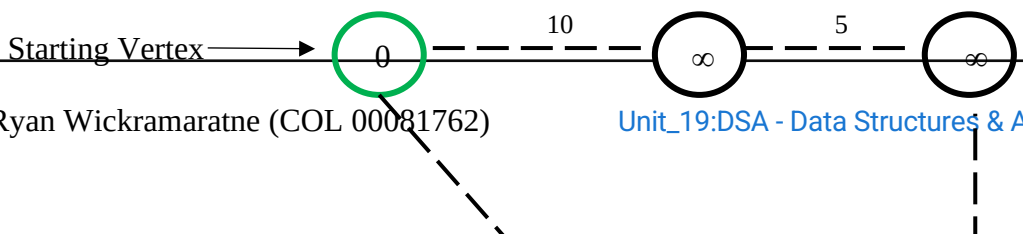
Consider the following same graph used in Dijkstra's algorithm. Let's now run through the Bellman-Ford algorithm. Here, I have 5 nodes, so I will perform relaxation for 5-1=4 times.



1. **Steps:**

Initialization: Start at the source node, which is A in this case. Set the distance from A to A as 0, and the distances from A to all other nodes as infinity (∞) since I don't yet know the cost to reach them. This leaves us with:

- Distance to A: 0 (Known)
- Distance to B: ∞ (Unknown)
- Distance to C: ∞ (Unknown)
- Distance to D: ∞ (Unknown)
- Distance to E: ∞ (Unknown)



2. **Iteration:**

Step 1 (1st iteration): Start with node A. The neighbours of A are B and D with distances 10 and 20 respectively. Update the distances:

- Distance to A: 0 (Known)
- Distance to B: 10 (Update)
- Distance to C: ∞ (Unknown)
- Distance to D: 20 (Update)
- Distance to E: ∞ (Unknown)

Step 2 (2nd iteration): Now I perform the relaxation on all edges. From B (distance 10), I can move to E at an additional distance of 5, hence update E. From D (distance 20), I can move to C at an additional distance of 5, hence update C:

- Distance to A: 0 (Known)
- Distance to B: 10 (Known)
- Distance to C: 25 (Update) (20 units from A to D + 5 units from D to C)
- Distance to D: 20 (Known)
- Distance to E: 15 (Update) (10 units from A to B + 5 units from B to E)

Step 3 (3rd iteration): Again, perform the relaxation on all edges. But at this point, no further updates can be made to the distances:

- Distance to A: 0 (Known)
- Distance to B: 10 (Known)
- Distance to C: 25 (Known)
- Distance to D: 20 (Known)
- Distance to E: 15 (Known)

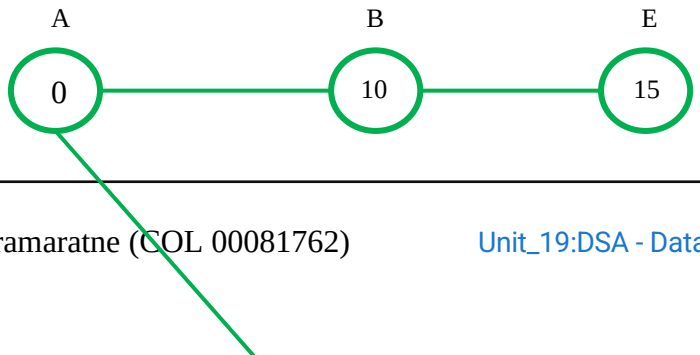
Step 4 (4th iteration): One final time, I perform the relaxation on all edges. No updates are made:

- Distance to A: 0 (Known)
- Distance to B: 10 (Known)
- Distance to C: 25 (Known)
- Distance to D: 20 (Known)
- Distance to E: 15 (Known)

The algorithm stops here since all distances are known and all nodes have been visited.

So, the shortest path from A to all other nodes are:

- A to A: 0 units
- A to B: 10 units
- A to C: 25 units (A -> D -> C)
- A to D: 20 units
- A to E: 15 units (A -> B -> E)



As we can see, Bellman-Ford and Dijkstra's algorithm have yielded the same results in this case. Bellman-Ford, however, can handle negative weights and can also detect negative weight cycles, whereas Dijkstra's cannot.

Limitations and Assumptions

Handling of Negative Weights: Unlike Dijkstra's algorithm, the Bellman-Ford algorithm can handle negative edge weights, making it more versatile. However, it cannot handle graphs with negative cycles. If a negative cycle is reachable from the source, then there is no shortest path, as the total weight can be reduced indefinitely by looping around the negative cycle. The Bellman-Ford algorithm can detect such cycles, though, and report their presence.

Weighted Graphs: Similar to Dijkstra's, Bellman-Ford's algorithm works on weighted graphs. There should be a cost/weight associated with each edge.

Time Complexity: The time complexity of the Bellman-Ford algorithm is O(VE), where V is the number of vertices and E is the number of edges. This makes it slower than Dijkstra's algorithm for graphs without negative edge weights. So, even though it's more powerful because it can handle negative weights, it's generally only used when necessary due to its higher time complexity. It's worth noting that while the time complexity is higher, it's the same for both sparse and dense graphs, which is not the case for Dijkstra's algorithm.

In summary, both algorithms have their strengths and weaknesses. Dijkstra's algorithm is faster but cannot handle negative weights, while Bellman-Ford's algorithm can handle negative weights but is slower and more complex. The choice of which algorithm to use would depend on the specifics of the problem at hand, such as whether the graph has negative weights and the trade-off between speed and complexity.

Sorting Algorithms - Bubble Sort

Introduction and Basic Principle to Bubble Sort

Overview:

Bubble Sort is one of the simplest sorting algorithms and is a good introduction to the concept of sorting for new programmers. The name "Bubble Sort" comes from the way that smaller elements "bubble" to the top of the list.

Basic Principle:

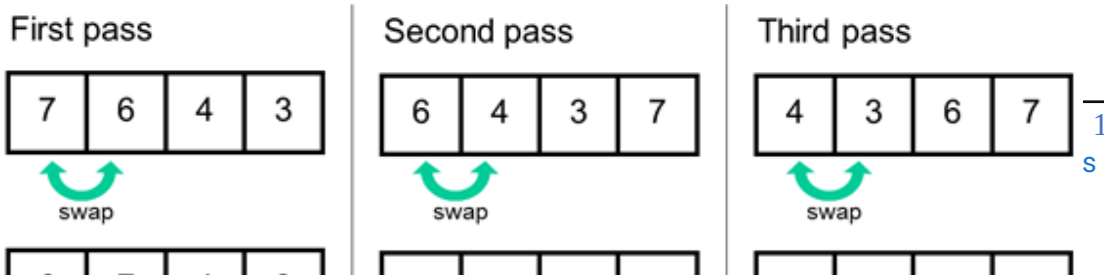
Bubble Sort works by repeatedly swapping the adjacent elements if they are in the wrong order. It continues this process until it makes a pass through the list without having to perform any swaps, at which point the list is sorted.

How It Works:

The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass.

Performance:

Bubble Sort has a worst-case and average time complexity of $O(n^2)$, where n is the number of items being sorted. This makes Bubble Sort highly inefficient on large lists and generally inferior to other sort algorithms like quicksort, mergesort, or heapsort. However, Bubble Sort has the advantage of simplicity and has a best case time complexity of O(n) when the input list is already sorted.



Operation of Bubble Sort

Step 1: Compare the first and second elements of the list. If the first element is larger than the second, swap their positions. If it is not, leave them as they are.

Step 2: Move to the next pair of elements (i.e., the second and third elements). Repeat the process. Continue this operation for every pair of adjacent elements in the list.

Step 3: After the last pair of elements is compared and swapped if necessary, repeat the entire process from the first pair of elements.

Step 4: Continue this process until a pass through the list is completed without any swaps being made. At this point, the list is sorted.

Pseudocode for Bubble Sort

Here's a simple pseudocode representation of Bubble Sort:

```
function bubbleSort(list)
  do
    swapped = false
    for i = 1 to indexOfLastUnsortedElement-1
      if list[i] > list[i+1]
        swap list[i] and list[i+1]
        swapped = true
      end if
    end for
    indexOfLastUnsortedElement = indexOfLastUnsortedElement - 1
  while swapped
end function
```

Explanation of pseudocode:

- In the pseudocode above, list is the ‘list’ of elements to be sorted.
- A ‘do-while’ loop is used to repeat the process of comparing and swapping adjacent elements as long as swaps are being made (i.e., as long as the list is not yet completely sorted). The boolean variable ‘swapped’ is used to check if a swap has been made in each iteration.

- Within the ‘do-while loop’, a ‘for’ loop is used to iterate through each pair of elements in the list, from the first pair to the last unsorted pair.
- If the first element of the pair (‘list[i]’) is larger than the second element (‘list[i+1]’), then these two elements are swapped, and ‘swapped’ is set to ‘true’.
- After each pass through the list, ‘indexOfLastUnsortedElement’ is reduced by one because with each pass through the list, the largest unsorted element is moved to its correct position at the end of the unsorted section of the list.
- The process repeats until a pass through the list is completed with no swaps being made (‘swapped’ remains ‘false’), indicating that the list is now sorted. At this point, the ‘do-while loop’ ends and the ‘bubbleSort’ function finishes.

Implement Bubble Sort in Java to sort the car numbers

Code:

```
public class BubbleSortExample {
    // BubbleSort method
    public static void bubbleSort(int[] array) {
        int n = array.length; // Get the length of the array

        // Outer loop: Go through the array as many times as there are
elements
        for (int i = 0; i < n-1; i++) {

            // Inner loop: Go through the array from the beginning to
the end not yet sorted
            for (int j = 0; j < n-i-1; j++) {

                // If the current element is greater than the next
one...
                if (array[j] > array[j+1]) {

                    // Swap array[j+1] and array[j] using a temporary
variable
                    int temp = array[j];
                    array[j] = array[j+1];
                    array[j+1] = temp;
                }
            }
        }
    }

    // Method to print the elements of an array
    public static void printArray(int[] array) {
        int n = array.length; // Get the length of the array

        // Go through each element of the array
        for (int i = 0; i < n; ++i)

            // Print the current element followed by a space
            System.out.print(array[i] + " ");

        // Print a new line after all elements have been printed
        System.out.println();
    }

    // Main method
    public static void main(String[] args) {
        // Declare an array of car numbers
        int[] carNumbers = {45, 78, 23, 56, 89, 12};

        // Print the original array
        System.out.println("Original array:");
        printArray(carNumbers);
```

```
        // Sort the array using BubbleSort
        bubbleSort(carNumbers);

        // Print the sorted array
        System.out.println("Sorted array:");
        printArray(carNumbers);
    }
}
```

Output:

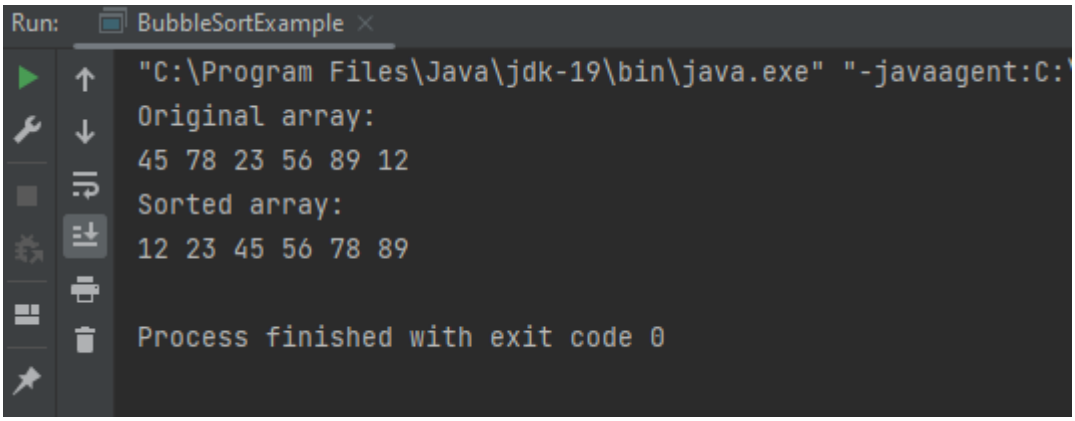


Figure 4. 3 Output of Bubble Sort in Java to sort the car numbers

Performance Analysis for Bubble Sort

1. Time Complexity:

Bubble Sort has a time complexity of:

- Best Case: O(n). The best case occurs when the input array is already sorted. In this case, Bubble Sort would only need to traverse through the array once and make no swaps.
- Average Case: $O(n^2)$. If the elements are in random order (neither sorted nor reversed), Bubble Sort can end up with a quadratic number of comparisons and swaps.
- Worst Case: $O(n^2)$. The worst case happens when the input array is reverse sorted. Bubble Sort will need to swap every pair of elements, requiring a quadratic number of operations.

2. Space Complexity:

Bubble Sort is an in-place sorting algorithm. An in-place algorithm doesn't require extra space or temporary arrays for its operation, so its space complexity is:

- O(1). It uses a constant amount of space to store the input and does not require any extra space for its operation beyond that.

This makes Bubble Sort a good choice when memory space is a crucial factor, despite its inefficiency in terms of time complexity.

3. Other Factors:

- Stability: Bubble Sort is a stable algorithm, as it does not change the relative order of elements with equal keys.
- Adaptivity: Bubble Sort is adaptive, meaning its efficiency improves if the input array is partially sorted.

In conclusion, while Bubble Sort is easy to understand and implement, its O(n^2) average and worst-case time complexities make it inefficient for large data sets. It's more suitable for small or nearly sorted data sets.

Strengths and Limitations of Bubble Sort

Strengths:

- Simplicity: The Bubble Sort algorithm is straightforward to understand and implement. It's a good option when the code's readability is more important than efficiency.
- In-place: Bubble Sort is an in-place sorting algorithm, which means it only requires a constant amount of memory (O(1)) beyond the input list, making it space efficient.
- Detects Sorted List: An optimized version of bubble sort can detect an already sorted list and stop early, giving it a best-case time complexity of O(n) when the list is already or nearly sorted.
- Stable: It is a stable sorting algorithm, meaning that it maintains the relative order of equal sort items.

Limitations:

- Efficiency: Bubble Sort is not suitable for large data sets due to its poor efficiency. It has a worst-case and average time complexity of $O(n^2)$, where n is the number of items being sorted.
- Performance: Bubble Sort makes multiple passes through the list, comparing and swapping adjacent elements, which makes it slower than many other sorting algorithms for larger lists.

Sorting Algorithms - Merge Sort

Introduction and Basic Principle to Merge Sort:

Overview:

Merge Sort is a divide-and-conquer sorting algorithm that was invented by John von Neumann in 1945. It's known for its efficiency and ability to handle large data sets.

Basic Principle:

The principle behind merge sort is that it's easier to sort two sorted lists rather than one unsorted one. Merge sort works by dividing the unsorted list into n sublists, each containing one element (as a list of one element is considered sorted), and then repeatedly merges sublists to produce new sorted sublists until there is only one sublist remaining.

How It Works:

The algorithm starts by dividing the unsorted list into two halves. If the list contains one or zero elements, then it is considered sorted and returned as is. If the list has more than one element, the list is split into two roughly equal halves. Each half is then sorted using Merge Sort. Once the two halves are sorted, they are merged back together to produce a single sorted list.

Performance:

Merge Sort is very efficient with time complexity of O(n log n) in all three cases (worst, average, and best) as merge sort always divides the array into two halves and takes linear time to merge two halves. While it's one of the more efficient sort algorithms, Merge Sort requires equal amount of additional space as the unsorted list, which can be a limitation if the list is large.

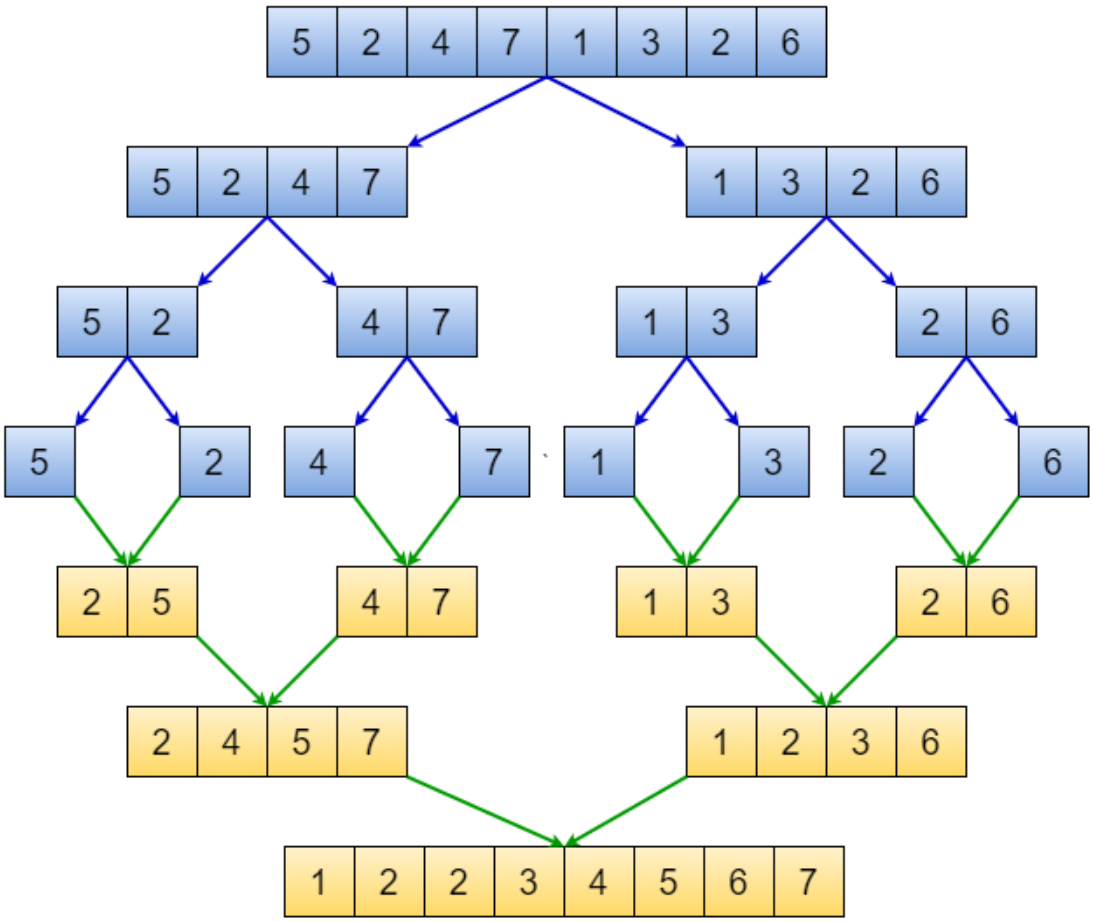


Figure 4. 4 Basic Principle to Merge Sort:

Operation of Merge Sort

Merge Sort is a divide-and-conquer sorting algorithm that works by recursively splitting a list in half. If the list is empty or has one item, it is sorted by definition (base case). If the list has more than one item, I split the list and recursively invoke a merge sort on both halves.

Here are the steps for Merge Sort:

Step 1 - Divide: If the list is of length 0 or 1, then it is considered sorted. Otherwise, divide the unsorted list into two sublists, each containing about half of the elements. This is done recursively, until I am comparing lists of size one against each other.

Step 2 - Conquer: Recursively sort both sublists.

Step 3 - Combine: Merge the two sorted sublists back into one sorted list. Comparing the elements at the start of the sublists allows us to find which is smaller. This smallest element is then moved into the sorted list, and the pointer to the start of the sublist (which now has a new starting element) is incremented. If I have moved all the elements of one sublist, then I can add all of the elements of the other to the sorted list.

Pseudocode for Merge Sort

Here's a simple pseudocode representation of Bubble Sort:

```
function mergeSort(list)
   if length(list) <= 1
      return list
   else
      mid = length(list) / 2
      left = mergeSort(list[1..mid])
      right = mergeSort(list[mid+1..end])
      return merge(left, right)
   end if
end function

function merge(left, right)
   result = []
   while length(left) > 0 and length(right) > 0
      if left[0] <= right[0]
         append left[0] to result
         left = left[1..end]
      else
         append right[0] to result
         right = right[1..end]
      end if
   end while
   if length(left) > 0
      append left to result
   end if
   if length(right) > 0
      append right to result
   end if
   return result
end function
```

Explanation of pseudocode:

- In the pseudocode above, list is the 'list' of elements to be sorted.
- 'mergeSort()' function is the main function that takes a list as an argument and returns a sorted version of the list.
- If the length of the list is 1 or less, the list is already sorted, so I just return the list.

- If the list has more than 1 item, I split the list into two and recursively sort each half by calling ‘mergeSort()’.
- After sorting each half, I merge the sorted lists together using ‘merge()’ function and return the result.
- ‘merge()’ function takes two sorted lists (left and right) and merges them into a single sorted list. It compares the first items of each list and adds the smaller item to the result list. If one list becomes empty before the other, it appends the remaining items of the non-empty list to ‘result’.
- Finally, ‘mergeSort()’ returns a sorted list.

Implement Merge Sort in Java to sort the car numbers

Code:

```
public class MergeSortExample {
    // Merge Sort function
    public static void mergeSort(int[] array, int left, int right) {
        if (right <= left) return;
        int mid = (left+right)/2;
        mergeSort(array, left, mid);
        mergeSort(array, mid+1, right);
        merge(array, left, mid, right);
    }

    // Merge function to merge two halves of a subarray
    static void merge(int[] array, int left, int mid, int right) {
        // Find sizes of two subarrays to be merged
        int n1 = mid - left + 1;
        int n2 = right - mid;

        /* Create temporary arrays */
        int[] leftArray = new int[n1];
        int[] rightArray = new int[n2];

        /*Copy data to temporary arrays*/
        for (int i=0; i<n1; ++i)
            leftArray[i] = array[left + i];
        for (int j=0; j<n2; ++j)
            rightArray[j] = array[mid + 1+ j];

        /* Merge the temp arrays */

        // Initial indexes of first and second subarrays
        int i = 0, j = 0;

        // Initial index of merged subarray array
        int k = left;
        while (i < n1 && j < n2) {
            if (leftArray[i] <= rightArray[j]) {
                array[k] = leftArray[i];
                i++;
            }
            else {
                array[k] = rightArray[j];
                j++;
            }
            k++;
        }

        /* Copy remaining elements of leftArray[] if any */
        while (i < n1) {
            array[k] = leftArray[i];
            i++;
            k++;
```

```
        }

        /* Copy remaining elements of rightArray[] if any */
        while (j < n2) {
            array[k] = rightArray[j];
            j++;
            k++;
        }
    }

    // Method to print the elements of an array
    public static void printArray(int[] array) {
        int n = array.length;
        for (int i = 0; i < n; ++i)
            System.out.print(array[i] + " ");
        System.out.println();
    }

    // Main method
    public static void main(String[] args) {
        int[] carNumbers = {45, 78, 23, 56, 89, 12};
        System.out.println("Original array:");
        printArray(carNumbers);
        mergeSort(carNumbers, 0, carNumbers.length-1);
        System.out.println("Sorted array:");
        printArray(carNumbers);
    }
}
```

Output:

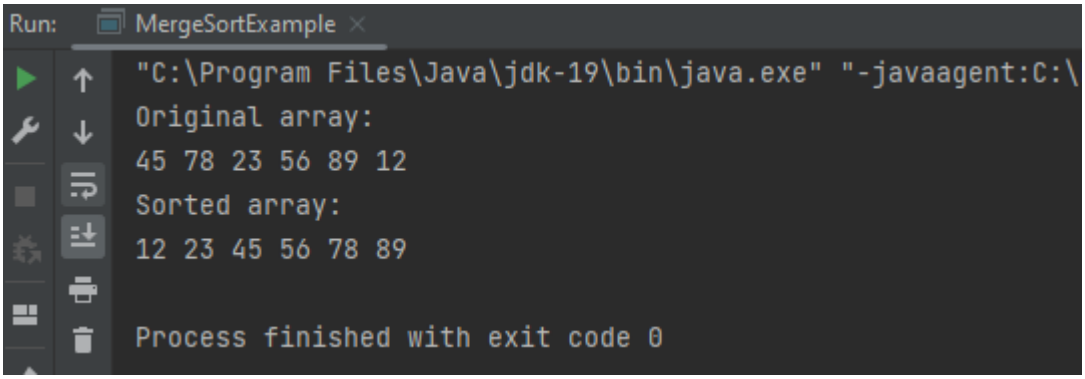


Figure 4. 5 Output of Merge Sort in Java to sort the car numbers

Compare the sorted Merge Sort output with the Bubble Sort output

Sorted Output for Bubble Sort:

Recall the Bubble Sort output from the previous implementation. Given the array of car numbers {45, 78, 23, 56, 89, 12}, the Bubble Sort algorithm returned the sorted array as {12, 23, 45, 56, 78, 89}.

Sorted Output for Merge Sort:

From our recent Merge Sort implementation with the same array of car numbers {45, 78, 23, 56, 89, 12}, the Merge Sort algorithm also returned the sorted array as {12, 23, 45, 56, 78, 89}.

Comparison of Outputs:

On comparing the sorted arrays obtained from both Bubble Sort and Merge Sort algorithms, I find that they are identical. Both algorithms sorted the input array of car numbers {45, 78, 23, 56, 89, 12} into the array {12, 23, 45, 56, 78, 89}.

Therefore, I can confirm that both the Bubble Sort and Merge Sort algorithms were implemented correctly as they provided the same sorted output when used to sort the same input array.

Performance Analysis for Merge Sort

1. Time Complexity:

Merge Sort has a time complexity of:

- Best Case: O(n log n). The best case occurs when the input array is already sorted. Even in this case, Merge Sort will still divide the array into single element arrays, then merge them back together, which still requires a logarithmic number of steps for each of n elements.
- Average Case: O(n log n). This time complexity remains the same regardless of the original order of the input data because the divide and merge operations have to be performed regardless.
- Worst Case: O(n log n). The worst case happens when the input array is reverse sorted or contains equal elements. But even in this case, the time complexity remains O(n log n) because the algorithm will still divide and merge as usual.

2. Space Complexity:

- Merge Sort is not an in-place sorting algorithm. It requires auxiliary space proportional to the array size to hold the temporary arrays used during the merge steps.
- The space complexity is O(n). This is a downside of Merge Sort compared to in-place sorting algorithms like Bubble Sort or Quick Sort.

3. Other Factors:

- Stability: Merge Sort is a stable algorithm, as it does not change the relative order of elements with equal keys.
- Non-Adaptivity: Merge Sort performs the same number of operations (O(n log n)) regardless of the initial order of the input. So, it doesn't take advantage of any existing order in the input array.

In conclusion, Merge Sort is an efficient algorithm with a better worst-case time complexity than many other sorting algorithms like Bubble Sort or even Quick Sort. Its downside is its space complexity; because it's not an in-place sorting algorithm, it requires additional space proportional to the input size. It's more suitable for larger, and particularly for disk-based, data sets where the additional space isn't a constraint.

Strengths and Limitations of Merge Sort

Strengths:

- Efficiency: Merge Sort performs well with large data sets. It has a time complexity of O(n log n) in all cases (best, average, and worst), making it much more efficient than Bubble Sort for large inputs.
- Stable: Merge Sort is a stable sort, which means that equal elements retain their relative order after sorting. This can be important in certain situations where data stability is required.
- Parallelizable: Merge Sort is straightforward to parallelize over multiple CPUs because it breaks the data into smaller chunks that can be sorted simultaneously.

Limitations:

- Space Complexity: Merge Sort is not an in-place sorting algorithm, and it requires additional space proportional to the size of the input, which can be a downside if we're sorting large lists and memory is a concern. Its space complexity is O(n).
- Complexity: Merge Sort is more complex to understand and implement than simple sorts like Bubble Sort.
- Overhead: For small lists, the additional overhead of the recursive function calls can make Merge Sort slower than simple sorting algorithms, even though it has a better time complexity.

Comparison of Algorithms

Comparison of Dijkstra's and Bellman-Ford

1. Time Complexity:

- **Dijkstra's Algorithm:** The time complexity of Dijkstra's algorithm can vary depending on the data structure used for the priority queue. With a simple binary heap, the time complexity is O((V+E) log V), where V is the number of vertices and E is the number of edges. However, with a Fibonacci heap, this can be improved to O(E + V log V).
- **Bellman-Ford Algorithm:** The Bellman-Ford algorithm has a time complexity of O(VE), where V is the number of vertices and E is the number of edges.

So, generally speaking, Dijkstra's algorithm can be faster than Bellman-Ford, especially for sparse graphs (where E is much less than V^2), but the exact speed can depend on the data structures used and the specifics of the graph.

2. Space Complexity:

- Both algorithms have a space complexity of O(V) because they need to keep track of the shortest distance to each vertex.

3. Practical Uses:

- **Dijkstra's Algorithm:** Dijkstra's algorithm is very efficient and is often used for routing and as a subroutine in other graph algorithms. However, it doesn't work with graphs that have negative edge weights.
- **Bellman-Ford Algorithm:** The Bellman-Ford algorithm, while slower, has the advantage of being able to handle graphs with negative edge weights, as long as there are no negative weight cycles. This makes it useful in situations where there may be negative weights, such as for certain problems in financial derivatives pricing.

The Bellman-Ford algorithm is also used in the field of computer networks to route packets of data between multiple network nodes. One of the classic applications is in the Routing Information Protocol (RIP), an older distance-vector routing protocol that employs Bellman-Ford.

In the context of ABC Pvt Ltd's scenario, Dijkstra's Algorithm would be an excellent choice if they are certain that there will be no negative distances between their racing locations. It will find the shortest path between any two points more quickly than Bellman-Ford. However, if they were to include negative distances for any reason (perhaps to represent some kind of advantage), they would need to use Bellman-Ford instead.

Comparison of Bubble Sort and Merge Sort

1. Time Complexity

- **Bubble Sort** has a time complexity of $O(n^2)$ in its worst and average cases, where 'n' is the number of items being sorted. The best case (when the input is already sorted) is O(n), but this is a rare scenario. This quadratic time complexity makes Bubble Sort inefficient on larger lists.
- On the other hand, **Merge Sort** operates in O(n log n) time in all cases (best, average, and worst). This logarithmic time complexity makes Merge Sort much more efficient than Bubble Sort when dealing with larger lists.

2. Space Complexity

- **Bubble Sort** is an in-place sorting algorithm, meaning it only needs a constant amount O(1) of additional space. Therefore, the space complexity of Bubble Sort is very low, which is an advantage when memory is a significant consideration.
- **Merge Sort**, conversely, is not an in-place sorting algorithm. It divides the list into two halves, and it needs additional space to store them. Thus, it has a space complexity of O(n), where 'n' is the number of items to be sorted. This is a disadvantage of Merge Sort, particularly when working with larger lists and memory is a concern.

3. Practical Uses

- **Bubble Sort** is practically used when simplicity is favored, or the dataset is small. Given its ease of implementation and the fact that it's in-place, Bubble Sort is a good choice for small datasets, teaching purposes, or scenarios where the overhead of more complex algorithms isn't necessary.
- **Merge Sort** is useful when dealing with larger datasets, and stability is required. While it requires more space than Bubble Sort, its efficiency makes it suitable for large datasets where time complexity could be a critical factor. It's also beneficial

when we're working with data structures like linked lists, which can be divided and merged efficiently.

In summary, while Bubble Sort might be suitable for small datasets or teaching purposes due to its simplicity and in-place nature, Merge Sort is a far more efficient choice for larger, more complex datasets despite its higher space requirements. The selection between these two algorithms ultimately depends on the specific requirements and constraints of the problem.

Determine which algorithms might be more suitable for ABC Pvt Ltd scenario

Determine between Bubble Sort and Merge Sort Algorithms

Bubble Sort Algorithm and ABC Pvt Ltd:

If ABC Pvt Ltd is looking to sort smaller sets of data, such as arranging the start times of the races or organizing a small number of participants, they might consider using the Bubble Sort algorithm. Its simplicity would be advantageous in these contexts, and it's unlikely that the quadratic time complexity would pose a problem given the small size of the data.

However, if they were to use Bubble Sort for larger datasets, such as sorting a significant number of participants or large sets of timing data, they might find that it becomes too slow. Similarly, since Bubble Sort is not a stable sort, if they needed to sort by multiple criteria (e.g., sort by country, then by time), Bubble Sort might not maintain the order of the initial sort when the second sort is performed.

Merge Sort Algorithm and ABC Pvt Ltd:

When it comes to handling larger datasets, such as sorting all participants of the event, the times they finished, or sorting the paths from longest to shortest, Merge Sort would be an appropriate choice for ABC Pvt Ltd. Despite its higher space requirement, its time complexity of O(n log n) would ensure more efficient performance on larger datasets.

Merge Sort is also a stable sort, so if ABC Pvt Ltd needed to sort by multiple criteria (e.g., sort by country, then by time), Merge Sort would maintain the initial sort order when the second sort is performed. This stability would be advantageous in a scenario where sorting by multiple criteria is necessary.

Conclusion:

In the end, the best algorithm for ABC Pvt Ltd to use depends on their specific needs. If they are sorting small datasets or memory is a critical constraint, Bubble Sort may be sufficient. However, for larger datasets, or when sorting stability is needed, Merge Sort would likely be the superior choice.

Since ABC Pvt Ltd will only have a maximum of 6 participants for the racing event, the **Bubble Sort algorithm** could indeed be a suitable choice. Here are some reasons why:

- Small Dataset: Bubble sort's simplicity shines with small datasets. With a maximum of 6 cars, the relative inefficiency of Bubble Sort compared to other more advanced sorting algorithms becomes negligible. The simplicity and directness of Bubble Sort make it a reasonable choice here.
- Ease of Implementation: Bubble Sort is relatively easier to implement and understand than many other sorting algorithms. It might be more efficient in terms of development and maintenance time for the company if they decide to use Bubble Sort.
- In-Place Sorting: Bubble Sort is an in-place sorting algorithm, meaning it only requires a constant amount O(1) of additional space. While this might not be a deciding factor with a small dataset like six participants, it's still an advantage if the company has memory constraints.
- Stable Sorting: As mentioned earlier, Bubble Sort is a stable sorting algorithm. If the company wanted to sort the cars based on multiple factors (say, based on the model of the car and then by the number of the car), Bubble Sort would maintain the relative order of equal sort elements.

Thus, for ABC Pvt Ltd's use case, Bubble Sort does seem to be a viable and efficient solution given the small dataset size.

However, if the company's dataset size were to significantly increase in the future, they might want to reconsider using a more efficient sorting algorithm like Merge Sort.

Determine between Dijkstra's and Bellman-Ford Algorithms

In ABC Pvt Ltd's case, they are looking to find the shortest possible route between multiple race locations. To accomplish this, they could use either Dijkstra's Algorithm or the Bellman-Ford Algorithm.

Dijkstra's Algorithm and ABC Pvt Ltd:

Given the nature of their problem, Dijkstra's Algorithm is quite suitable. It is faster and more efficient, particularly in situations where I have a large number of locations (or nodes), but the connections (or edges) between them are relatively few (a sparse graph). This seems to align with ABC Pvt Ltd's scenario, where they have multiple race locations but the direct connections (or paths) between them might be relatively few. Moreover, since it's a racing event, it's logical to assume that there would be no negative distances (it's hard to conceive a scenario where moving from one point to another would result in a "negative" distance), hence the limitation of Dijkstra's algorithm won't be an issue here.

Bellman-Ford Algorithm and ABC Pvt Ltd:

On the other hand, the Bellman-Ford Algorithm has the advantage of being able to handle negative weights. However, in the context of ABC Pvt Ltd's racing event, this capability is not needed since the concept of 'negative distance' is practically impossible. Furthermore, the Bellman-Ford algorithm tends to be slower, especially when dealing with large graphs. Given that ABC Pvt Ltd may add more racing locations over time, a more efficient algorithm would be preferable.

Conclusion:

Thus, considering the nature of the racing event, the non-existence of negative distances, and potential scalability in terms of adding more locations in the future, **Dijkstra's Algorithm** would be more suitable for ABC Pvt Ltd's scenario. Here are some reasons why:

- Non-negative weights: Dijkstra's Algorithm assumes that all weights are non-negative. In the context of ABC Pvt Ltd, weights represent distances between race locations, and it is practical to assume that distances cannot be negative. Therefore, this assumption aligns perfectly with the requirements of the scenario.
- Efficiency: Dijkstra's Algorithm is known for its efficiency. It works well especially when the graph is sparse, i.e., when there are fewer edges relative to the number of nodes. In ABC Pvt Ltd's case, if there are many race locations (nodes) but relatively few direct paths between them (edges), Dijkstra's Algorithm would provide a solution more quickly and efficiently.
- Scalability: As ABC Pvt Ltd may add more race locations over time, Dijkstra's Algorithm's efficiency makes it a more scalable solution. It can handle an increasing number of nodes well, providing solutions quickly even as the graph grows larger.
- Optimal solution: Dijkstra's Algorithm is a greedy algorithm, meaning it makes the best choice at each decision point with the goal of finding the global optimum. In the context of planning race routes, this means that Dijkstra's Algorithm will always find the shortest possible route between the chosen starting location and all other locations.

Hence, given the nature of the racing event, the non-existence of negative distances, and potential scalability with regards to the addition of more locations in the future, Dijkstra's Algorithm is a more fitting solution for ABC Pvt Ltd's requirements.

However, it's also important for ABC Pvt Ltd to consider that if their problem changes and they do end up needing to accommodate negative weights for any reason, they would then need to switch to using the Bellman-Ford Algorithm.

Necessary Theory Parts

Types of sorting algorithms

Sorting algorithms are crucial in computer science as they organize data in some particular order, which helps in optimizing complex problem-solving and searching algorithms. Here are several types of sorting algorithms, along with their time complexities:

1. Bubble Sort:

Bubble Sort is a simple comparison-based algorithm that works by repeatedly swapping adjacent elements if they are in the wrong order.

- Best-case time complexity: O(n)
- Average-case time complexity: $O(n^2)$
- Worst-case time complexity: $O(n^2)$

2. Selection Sort:

This algorithm sorts an array by repeatedly finding the minimum element from the unsorted part and putting it at the beginning of the unsorted part.

- Best, average, and worst-case time complexity: $O(n^2)$

3. Insertion Sort:

Insertion sort is a comparison-based algorithm that builds a sorted array one item at a time. It's much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort.

- Best-case time complexity: O(n)
- Average-case time complexity: $O(n^2)$
- Worst-case time complexity: $O(n^2)$

4. Quick Sort:

Quick Sort is a divide-and-conquer algorithm, which works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.

- Best-case time complexity: O(n log n)
- Average-case time complexity: O(n log n)
- Worst-case time complexity: $O(n^2)$

5. Merge Sort:

Merge Sort is also a divide-and-conquer algorithm that involves dividing the unsorted list into n sublists, each containing one element, and then repeatedly merging sublists to produce new sorted sublists until there is only one sublist remaining.

- Best, average, and worst-case time complexity: O(n log n)

6. Heap Sort:

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure. It divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element and moving that to the sorted region.

- Best, average, and worst-case time complexity: O(n log n)

7. Radix Sort:

Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value.

- Best, average, and worst-case time complexity: O(nk), where n is the number of elements and k is the number of digits in the maximum number.

8. Bucket Sort:

Bucket sort, or bin sort, is a distribution sort that works by distributing the elements of an array into a number of buckets, and then each bucket is sorted individually, either using a different sorting algorithm or recursively applying the bucket sort algorithm.

- Best and average-case time complexity: O(n+k)
- Worst-case time complexity: $O(n^2)$

9. Counting Sort:

Counting sort is a linear time sorting algorithm that sorts items based on keys between a specific range. It works by counting the number of objects having distinct key values.

- Best, average, and worst-case time complexity: O(n+k), where n is the number of elements and k is the range of input.

Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
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)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(n)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(1)
Counting Sort	O(n + k)	O(n + k)	O(n + k)	O(k)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n + k)
Bucket Sort	O(n + k)	O(n + k)	O(n^2)	O(n)

Figure 4. 6 Types of sorting algorithms and their time complexities

Types of shortest path algorithms

Shortest path algorithms are used in network routing, map navigation, and many other applications. They find the shortest (or least costly) path between two points, known as the source and the destination. Below are a few types of shortest path algorithms:

1. Dijkstra's Algorithm:

This algorithm finds the shortest path from a given source node to all other nodes in a graph. It is very efficient and widely used, especially in navigation systems and routing protocols in networks. However, it does not work with graphs that have negative edge weights.

2. Bellman-Ford Algorithm:

This algorithm also calculates the shortest path from a single source node to all other nodes in a weighted graph. Unlike Dijkstra's, the Bellman-Ford algorithm can handle graphs with negative edge weights, provided there are no negative cycles. It's less efficient than Dijkstra's, but its ability to handle negative weights can be an advantage.

3. Floyd-Warshall Algorithm:

The Floyd-Warshall algorithm finds the shortest path between all pairs of nodes in a graph. It works with both positive and negative edge weights but cannot handle negative cycles. The algorithm is efficient for dense graphs, but it can be overkill when I only need the shortest path from one source to one destination, or from one source to all destinations.

4. A* Search Algorithm:

This is a popular heuristic-based algorithm used in pathfinding and graph traversal. A* selects the path that minimizes the cost from the start node to the goal node, plus the cost

from the current node to the goal. This approach helps the A* algorithm to find the most cost-effective path.

5. Johnson's Algorithm:

Johnson's algorithm is used to find the shortest paths between every pair of vertices in a given weighted, directed graph and here weights may be negative.

6. Breadth-First Search (BFS):

In an unweighted graph or when all the weights are equal, a BFS will also find the shortest path. This algorithm is not usually classified as a shortest path algorithm because it only works under those specific conditions.

These algorithms have their strengths and weaknesses, and the right choice often depends on the specific problem. For example, for a graph with no negative weights, Dijkstra's algorithm is often the best choice, but if the graph may have negative weights, I might need to use the Bellman-Ford algorithm instead.

Task 5

Understanding Asymptotic Analysis

Explanation of Asymptotic Analysis

1. Introduction

Asymptotic Analysis is a methodology used to evaluate the performance of algorithms, particularly their runtime and space requirements. The term 'asymptotic' refers to 'approaching a value or curve arbitrarily closely' (i.e., as some sort of limit is taken). In the context of algorithms, it's about understanding the behavior of the algorithm as the input size approaches infinity.

2. Why Asymptotic Analysis

It's impractical to measure an algorithm's efficiency by simply implementing it and timing how long it takes to run. This is because the same algorithm can run at different speeds on different machines, or even on the same machine under different conditions. Asymptotic Analysis provides a machine-independent way to measure algorithm efficiency. It abstracts the details of the machine and execution environment and focuses on how the runtime or space requirements grow as the size of the input grows.

3. How Asymptotic Analysis works

- Asymptotic Analysis describes the limiting behavior of an algorithm, i.e., it provides an upper bound, lower bound, or tight bound on the time complexity or space complexity of an algorithm.
- It simplifies the expression for the time complexity or space complexity by dropping lower order terms and ignoring leading constants. For instance, if an

algorithm's time complexity is $3n^2 + 2n + 1$, in Asymptotic Analysis, I focus on the highest order term and ignore the rest, hence it becomes $O(n^2)$.

4. Notations in Asymptotic Analysis

There are three notations primarily used in Asymptotic Analysis:

- Big O Notation (O): It describes an upper bound on the time complexity of an algorithm. It provides an asymptotic upper limit for the growth rate of runtime of an algorithm.
- Omega Notation (Ω): It describes a lower bound on the time complexity of an algorithm. It provides an asymptotic lower limit on the growth rate of runtime of an algorithm.
- Theta Notation (Θ): It describes both the lower bound and the upper bound of the time complexity of an algorithm. It provides an asymptotic tight bound on the growth rate of runtime of an algorithm.

Through Asymptotic Analysis, I can get a fair idea of the best, average, and worst-case scenarios for a given algorithm in terms of its time complexity and space complexity. This understanding is vital in making informed choices about which algorithms to use when developing software.

Asymptotic Analysis and Infinite Input Size

1. Importance of Infinite Input Size

The main idea behind considering an infinite input size is to have a measure of the efficiency of the algorithm when dealing with large inputs. When I design an algorithm, it's crucial to understand how it will perform when subjected to very large inputs, as this is often where efficiency matters most.

2. Asymptotic Notations

To analyze an algorithm's efficiency with an infinite input size, Asymptotic Analysis utilizes three notations: Big O (O), Omega (Ω), and Theta (Θ). These notations represent the upper bound, lower bound, and both the upper and lower bound of an algorithm's running time or space complexity, respectively. They give us a way to describe how the algorithm behaves as the size of its input approaches infinity.

- Big O Notation :The Big O notation provides an upper bound on the time complexity of an algorithm, essentially giving the worst-case scenario. This notation is important because it tells us the maximum time an algorithm might take, which ensures the system won't crash or hang if the data input is unexpectedly large.
- Omega Notation: The Omega notation, on the other hand, provides a lower bound on the time complexity. It represents the best-case scenario. This notation gives the minimum time in which the algorithm will execute.
- Theta Notation: The Theta notation bounds a functions from above and below, so it defines exact asymptotic behavior. This provides a tight bound on the time complexity, which essentially shows the average-case scenario.

3. Ignoring Constants and Lower-Order Terms

As the input size tends towards infinity, constants and smaller order terms have less and less impact on the outcome. As such, in Asymptotic Analysis, they are usually ignored. For instance, if I have a time complexity of $3n^2 + 2n + 1$, I focus on the term with the highest

growth rate (n^2) and ignore the rest, so it becomes $O(n^2)$. This is because, for large n, the lower order terms and constants become negligible.

For another example, in the function $f(n) = 5n^2 + 3n + 2$, the $5n^2$ term will quickly become much larger than the other terms as n increases, such that the $3n + 2$ becomes relatively insignificant. Hence, I can simplify our analysis by focusing on the term with the highest growth rate, which is n^2 in this case, and express the time complexity as $O(n^2)$.

This simplification makes it much easier to compare different algorithms. For example, comparing $O(n^2)$ with $O(n \log n)$ is more straightforward than comparing $5n^2 + 3n + 2$ with $2n \log n + n$.

In summary, Asymptotic Analysis is a vital tool in computer science because it provides a way to evaluate an algorithm's efficiency in a way that's independent of hardware and software variations. It helps us understand how the running time (or space) of an algorithm increases with the input size and allows us to compare the efficiency of different algorithms in a meaningful and consistent way.

Asymptotic Analysis to assess algorithm effectiveness

Asymptotic Notations effectiveness

1. Big O Notation

- Big O notation provides an upper bound on a function, i.e., it gives the maximum possible time taken by an algorithm for any input size. In other words, it describes the worst-case scenario of an algorithm's time or space complexity.
- For example, if an algorithm has a time complexity of $O(n^2)$, it means that in the worst case, the running time will increase quadratically with the size of the input.
- Thus, Big O notation is a critical tool in understanding the performance limitations of an algorithm.

2. Omega Notation

- Omega notation provides a lower bound on a function, i.e., it gives the minimum time taken by an algorithm for any input size. It describes the best-case scenario of an algorithm's time or space complexity.
- For example, if an algorithm has a time complexity of $\Omega(n)$, it means that in the best case, the running time will increase linearly with the size of the input.
- This notation helps us understand the optimal performance capabilities of an algorithm.

3. Theta Notation

- Theta notation provides both an upper and lower bound on a function. It gives a tight bound on the time complexity of an algorithm. In other words, it accurately describes the running time of an algorithm for any given input size, providing a representation of the average-case scenario.
- For example, if an algorithm has a time complexity of $\Theta(n \log n)$, it means that the running time will increase logarithmically with the size of the input on average.

- This notation gives us a balanced perspective of an algorithm's performance. It's not as optimistic as the best case, nor as pessimistic as the worst case.

By understanding and applying these notations, I can evaluate an algorithm's efficiency and performance. This analysis will give us a thorough understanding of how our algorithm behaves in the best, average, and worst-case scenarios, which is crucial when choosing an appropriate algorithm for a particular task.

Derive time complexity using Asymptotic analysis

1. Constant Time Complexity: O(1)

This time complexity represents an operation (or series of operations) that requires the same amount of time to complete regardless of the size of the input data. The 'O' is a Big O notation, which describes an upper bound of an algorithm in worst-case scenarios.

For example, let's consider a scenario where I just need to print the first item in a list of integers. Regardless of how large the list is, I only need one step to print the first item.

```
public class ConstantTimeComplexity {
    public static void main(String[] args) {
        int[] array = {1, 2, 3, 4, 5};
        int index = 2;
        System.out.println(array[index]);  // Always prints array[2]
regardless of array size
    }
}
```

In the above example, no matter how big the array gets, it always takes the same amount of time to fetch an item using an index. So, this operation is O(1).

2. Linear Time Complexity: O(n)

This time complexity represents an operation (or series of operations) that completes in proportion to the size of the input data.

For example, if I want to print each item in a list of integers, the number of steps increases linearly with the size of the list. If I have 10 items, I print 10 times. If I have 1000 items, I print 1000 times.

```
public class LinearTimeComplexity {
    public static void main(String[] args) {
        int[] array = {1, 2, 3, 4, 5};
        for (int num : array) {
            System.out.println(num);
        }
    }
}
```

In the above example, the time it takes to print all the numbers in the array grows linearly with the size of the array. This is why it's considered O(n).

3. Quadratic Time Complexity: $O(n^2)$

This time complexity represents an operation (or series of operations) that completes in proportion to the square of the size of the input data.

For example, if I want to print all pairs of integers in a list, the number of steps increases quadratically with the size of the list.

```
public class QuadraticTimeComplexity {
    void bubbleSort(int arr[]) {
        int n = arr.length;
        for (int i = 0; i < n-1; i++) {
            for (int j = 0; j < n-i-1; j++) {
                if (arr[j] > arr[j+1]) {
                    // swap arr[j+1] and arr[i]
                    int temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
            }
        }
    }
}
```

The time complexity of the above code is $O(n^2)$, where 'n' is the number of items in the array. For each item in the array, I go through the entire array once. Therefore, if the size of the array is 'n', I perform 'n*n' operations.

4. Logarithmic Time Complexity: O(log n)

Algorithms with logarithmic time complexity are often associated with scenarios where the data set size is halved (or generally, reduced) after each operation. Binary search is a common example. In a binary search, you start in the middle of a sorted list and check whether the number you're searching for is in the middle, to the left, or to the right. Depending on the result, you'll then look at the left half or the right half of the list, in essence halving the list.

```
public class BinarySearch {
    int binarySearch(int arr[], int x) {
        int l = 0, r = arr.length - 1;
        while (l <= r) {
            int m = l + (r - l) / 2;

            // Check if x is present at mid
            if (arr[m] == x)
                return m;

            // If x greater, ignore left half
            if (arr[m] < x)
                l = m + 1;

                // If x is smaller, ignore right half
            else
                r = m - 1;
        }
        // if element is not present
        return -1;
    }
}
```

In the above code, I am halving the search space with every comparison. This makes the time complexity O(log n).

5. Linearithmic Time Complexity: O(n log n)

Algorithms with linearithmic time complexity often involve a divide-and-conquer strategy, where the problem is divided into smaller subproblems, each of which is solved independently. Merge sort is a common example.

```
public class MergeSort {

    void merge(int arr[], int left, int mid, int right) {
        // Find sizes of two subarrays to be merged
        int n1 = mid - left + 1;
```

```
        int n2 = right - mid;

        // Create temp arrays
        int L[] = new int[n1];
        int R[] = new int[n2];

        // Copy data to temp arrays
        for (int i = 0; i < n1; ++i)
            L[i] = arr[left + i];
        for (int j = 0; j < n2; ++j)
            R[j] = arr[mid + 1 + j];

        // Merge the temp arrays

        // Initial indexes of first and second subarrays
        int i = 0, j = 0;

        // Initial index of merged subarray array
        int k = left;
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
            } else {
                arr[k] = R[j];
                j++;
            }
            k++;
        }

        // Copy remaining elements of L[] if any
        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
        }

        // Copy remaining elements of R[] if any
        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
        }
    }

    // Main function that sorts arr[l..r] using
    // merge()
    void sort(int arr[], int left, int right) {
        if (left < right) {
            // Find the middle point
            int mid = (left + right) / 2;

            // Sort first and second halves
            sort(arr, left, mid);
            sort(arr, mid + 1, right);

            // Merge the sorted halves
            merge(arr, left, mid, right);
        }
```

```
    }
}
```

6. Exponential Time Complexity: O(2^n)

Algorithms with exponential time complexity often involve a lot of repeated work, because they explore every single possible combination of solutions. Recursive calculation of Fibonacci numbers is a common example.

```
class Fibonacci {
    static int fib(int n) {
        if (n <= 1)
            return n;
        return fib(n - 1) + fib(n - 2);
    }
}
```

Through these examples, I can see how to derive the time complexity using Asymptotic analysis. Each of these complexities represent different classes of algorithms and problems. It's also important to note that not every algorithm neatly fits into one of these categories. However, they provide a useful guideline for estimating how an algorithm will scale with increased input size. Remember that time complexity doesn't necessarily tell you the 'speed' of an algorithm, but rather how it grows with increased input size.

The various methods for measuring an algorithm's efficiency

Main methods for measuring an algorithm's efficiency

1. Time Complexity

Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run, as a function of the size of the input to the program. It is usually expressed using Big O notation, which describes the upper bound of the time complexity in the worst-case scenario.

The time complexity of an algorithm depends on the number of operations an algorithm performs in proportion to the input size. For example, in a linear search algorithm that checks every element in a list, the time complexity is O(n), because the time it takes grows linearly with the size of the input.

Different types of time complexities include:

- Constant time: O(1)
- Logarithmic time: O(log n)
- Linear time: O(n)
- Linear-Logarithmic time: O(n log n)
- Quadratic time: $O(n^2)$
- Cubic time: $O(n^3)$
- Polynomial time: $n^{O(1)}$
- Exponential time: $2^{O(n)}$

2. Space Complexity

Space complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input. It represents the amount of memory space that the algorithm needs beyond the space taken up by the inputs to the algorithm.

Like time complexity, space complexity is often expressed using Big O notation. For example, a sorting algorithm that requires space proportional to the list size has a space complexity of O(n).

Factors that could affect space complexity include variables (which need storage space), data structures, allocation, and deallocation of memory, etc.

3. Trade-Off between Time and Space Complexity

In many cases, there's a trade-off between time and space complexity. An algorithm might be able to decrease the time it takes to process by using more memory (a lookup table or cache, for instance), or decrease the memory it needs by taking more time (for example, by re-computing values rather than storing them).

This trade-off is often represented as a space-time trade-off. Efficient algorithms are those that manage this balance well, finding a good compromise between running time and memory used.

In summary, both time and space complexity provide crucial information about how an algorithm will perform, particularly with large inputs. Understanding these complexities helps programmers to choose the most effective algorithms for their specific needs, taking into account the resources available and the requirements of the task.

Ways to evaluate the real-world performance of an algorithm

1. Actual Running Time

The most direct way to measure an algorithm's efficiency is to implement it and then run it on various test inputs, recording the actual running time. However, this method has several drawbacks. The actual running time can vary depending on many factors, such as the specific machine the algorithm is running on, what other processes are running at the same time, the specific implementation of the algorithm, and so on.

It is also difficult to accurately measure the running time of very fast algorithms, as the time might be below the resolution of the clock on the machine. Still, actual running time can provide a rough estimate of an algorithm's efficiency and can be especially useful when comparing two algorithms to solve the same problem.

2. Use of Resources

Another way to measure the efficiency of an algorithm is by examining its use of resources, which can include things like memory usage, disk I/O, network I/O, and so on. Measuring the use of these resources can give a better picture of an algorithm's efficiency, especially for algorithms that are heavily dependent on such resources. For example, an algorithm that makes extensive use of disk I/O might be less efficient than it appears if only time complexity is considered, because disk operations are often much slower than in-memory operations.

3. Impact of Architecture and Hardware

The architecture of the machine and the specific hardware that the algorithm runs on can have a significant impact on the efficiency of an algorithm. For example, some algorithms are specifically designed to take advantage of parallel processing capabilities of modern CPUs, while others might perform better on machines with large amounts of memory. Similarly, some algorithms might be more efficient on machines with certain types of hardware. For instance, graphics processing units (GPUs) can dramatically speed up certain

types of calculations, and some algorithms are specifically designed to take advantage of this.

4. Impact of Programming Language and Compiler

The choice of programming language and compiler can also impact the performance of an algorithm. Different languages and compilers can translate the same high-level algorithm into machine code differently, leading to differences in efficiency. Furthermore, different programming languages have different built-in functions and data types that can be leveraged to optimize an algorithm's performance.

5. Size and Nature of Input Data

The size and nature of input data is another significant factor in determining the real-world performance of an algorithm. Algorithms may perform differently on small and large inputs. Additionally, the nature of the input data may also impact performance. For example, a sorting algorithm may perform differently on an already sorted list, a reverse-sorted list, or a list with all elements being identical.

In conclusion, while asymptotic analysis provides a useful tool for analyzing the theoretical efficiency of an algorithm, it's important to also consider the real-world performance of the algorithm. Factors such as the actual running time, use of resources, and the impact of architecture and hardware can all have a significant impact on an algorithm's real-world performance.

Examples of several efficiency measurement techniques

Examples of several efficiency measurement techniques – Time complexity

1. Constant Time Complexity: O(1)

Constant time complexity refers to operations that take the same amount of time to complete, regardless of the size of the input data.

A common example of this is accessing an element in an array by its index in Java:

```
public class Main {
    public static void main(String[] args) {
        int[] array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        int index = 5;
        System.out.println(array[index]);
    }
}
```

```
Run:  Main
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA
5

Process finished with exit code 0
```

Figure 5. 1 Output of example code of accessing an element in an array by its index in Java

In the code above, regardless of the size of the array, retrieving an element by its index will always take the same amount of time.

2. Logarithmic Time Complexity: O(log n)

Logarithmic time complexity is often seen in algorithms that reduce the problem size with each step by a factor, typically seen in divide-and-conquer type algorithms.

A classic example of this is the binary search algorithm in Java:

```
public class Main {
    public static void main(String[] args) {
        int[] array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        int target = 6;
        System.out.println(binarySearch(array, target, 0, array.length -
```

```
1));
    }

    public static int binarySearch(int[] array, int target, int start,
int end) {
        if (end >= start) {
            int mid = start + (end - start) / 2;
            if (array[mid] == target)
                return mid;
            if (array[mid] > target)
                return binarySearch(array, target, start, mid - 1);
            return binarySearch(array, target, mid + 1, end);
        }
        return -1;
    }
}
```

```
Run:  Main
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA
5

Process finished with exit code 0
```

Figure 5. 2 Output of example code of binary search algorithm in Java

In binary search, the array is continually divided in half until the target value is found, leading to a logarithmic time complexity.

3. Linear Time Complexity: O(n)

Linear time complexity is observed when the running time of an operation grows linearly with the size of the input data.

A common example of a linear time complexity operation is searching for an element in an unsorted array:

```
public class Main {
    public static void main(String[] args) {
        int[] array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        int target = 6;
        System.out.println(linearSearch(array, target));
    }

    public static int linearSearch(int[] array, int target) {
        for (int i = 0; i < array.length; i++) {
            if (array[i] == target)
                return i;
```

```
        }
        return -1;
    }
}
```

```
Run:  Main
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA
5

Process finished with exit code 0
```

Figure 5. 3 Output of example code of searching for an element in an unsorted array

In the code above, in the worst-case scenario (the target is at the end of the array or not present at all), the function would have to go through each element in the array once, resulting in a linear time complexity.

4. Linear-Logarithmic Time Complexity: O(n log n)

Linear-Logarithmic time complexity is seen in algorithms which perform a logarithmic operation for each item in the data. These algorithms are usually efficient sorting algorithms like mergesort, heapsort, or quicksort.

Here's an example of MergeSort in Java:

```
public class Main {
    void merge(int arr[], int l, int m, int r) {
        int n1 = m - l + 1;
        int n2 = r - m;

        int L[] = new int[n1];
        int R[] = new int[n2];

        for (int i=0; i<n1; ++i)
            L[i] = arr[l + i];
        for (int j=0; j<n2; ++j)
            R[j] = arr[m + 1+ j];

        int i = 0, j = 0;

        int k = l;
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
            } else {
                arr[k] = R[j];
                j++;
            }
```

```
            k++;
        }

        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
        }

        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
        }
    }

    void sort(int arr[], int l, int r) {
        if (l < r) {
            int m = (l+r)/2;

            sort(arr, l, m);
            sort(arr , m+1, r);

            merge(arr, l, m, r);
        }
    }

    static void printArray(int arr[]) {
        int n = arr.length;
        for (int i=0; i<n; ++i)
            System.out.print(arr[i] + " ");
        System.out.println();
    }

    public static void main(String args[]) {
        int arr[] = {12, 11, 13, 5, 6, 7};

        System.out.println("Given Array");
        printArray(arr);

        Main ob = new Main();
        ob.sort(arr, 0, arr.length-1);

        System.out.println("\nSorted array");
        printArray(arr);
    }
}
```

```
Run:  Main
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA
Given Array
12 11 13 5 6 7

Sorted array
5 6 7 11 12 13
```

Figure 5. 4 Output of example code of MergeSort in Java

The MergeSort algorithm works by repeatedly breaking down a list into several sublists until each sublist consists of a single element, and then merging those sublists to produce new sorted sublists until there is only one sublist remaining.

5. Quadratic Time Complexity: $O(n^2)$

Quadratic time complexity is seen when the time to execute an algorithm is proportional to the square of the input size. This is often seen in algorithms that involve nested iterations over the data, such as bubble sort, selection sort, or insertion sort.

Here's an example of BubbleSort in Java:

```java
public class Main {
    void bubbleSort(int arr[]) {
        int n = arr.length;
        for (int i = 0; i < n-1; i++)
            for (int j = 0; j < n-i-1; j++)
                if (arr[j] > arr[j+1]) {
                    int temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
    }

    void printArray(int arr[]) {
        int n = arr.length;
        for (int i=0; i<n; ++i)
            System.out.print(arr[i] + " ");
        System.out.println();
    }

    public static void main(String args[]) {
        Main ob = new Main();
        int arr[] = {64, 34, 25, 12, 22, 11, 90};
        ob.bubbleSort(arr);
        System.out.println("Sorted array");
        ob.printArray(arr);
    }
}
```

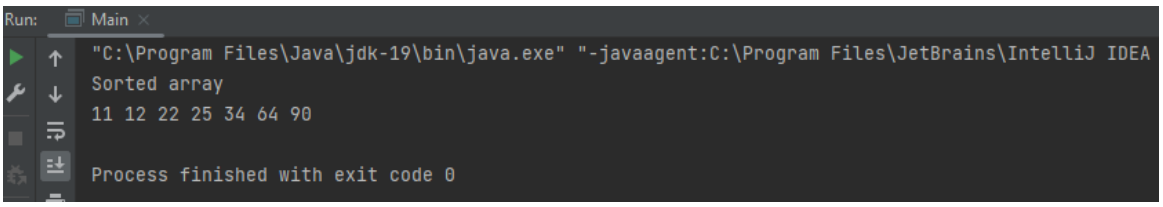


Figure 5. 5 Output of example code of BubbleSort in Java

'BubbleSort' works by repeatedly swapping adjacent elements if they are in the wrong order.

6. Cubic Time Complexity: $O(n^3)$

As mentioned in the previous response, Cubic time complexity appears when an algorithm performs three nested loops over the input data. This time complexity is less common, but can appear in certain numerical computation algorithms or graph algorithms.

Here's an example of a cubic time complexity scenario in Java, where I perform triple nested loops:

```
public class Main {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int count = 0;

        for (int i = 0; i < arr.length; i++) {
            for (int j = 0; j < arr.length; j++) {
                for (int k = 0; k < arr.length; k++) {
                    count += 1;
                }
            }
        }

        System.out.println("Count is: " + count);
    }
}
```

```
Run:  Main
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA
Count is: 125

Process finished with exit code 0
```

Figure 5. 6 Output of example code of triple nested loops

7. Polynomial Time Complexity: n^k (where k is a constant)

Polynomial time complexity covers a variety of time complexities including linear time, quadratic time, cubic time etc, essentially anything where the time complexity can be expressed as n raised to a constant power.

Here's an example of polynomial time complexity scenario with $O(n^4)$ in Java:

```java
public class Main {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int count = 0;

        for (int i = 0; i < arr.length; i++) {
            for (int j = 0; j < arr.length; j++) {
                for (int k = 0; k < arr.length; k++) {
                    for (int l = 0; l < arr.length; l++) {
                        count += 1;
                    }
                }
            }
        }

        System.out.println("Count is: " + count);
    }
}
```

```
Run:  Main
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA
Count is: 625

Process finished with exit code 0
```

Figure 5. 7 Output of example code of polynomial time complexity scenario with O(n4) in Java

8. Exponential Time Complexity: 2^n

Exponential time complexity arises in algorithms where the amount of work doubles with each addition to the input data set. This is seen in problems where the algorithm explores many or all possible configurations of the input data, such as the classic problem of generating all subsets of a set (also known as the power set).

Here's an example of an algorithm with exponential time complexity, a recursive method to generate a power set of a set in Java:

```java
import java.util.*;

class Main {
    static void powerSet(int[] arr, int index, List<Integer> currentSet)
{
        if (index == arr.length) {
            System.out.println(currentSet);
            return;
        }

        // Including the current element
        currentSet.add(arr[index]);
```

```
        powerSet(arr, index + 1, currentSet);

        // Excluding the current element
        currentSet.remove(currentSet.size() - 1);
        powerSet(arr, index + 1, currentSet);
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3};
        powerSet(arr, 0, new ArrayList<>());
    }
}
```

```
Run:  Main
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA
[1, 2, 3]
[1, 2]
[1, 3]
[1]
[2, 3]
[2]
[3]
[]

Process finished with exit code 0
```

Figure 5. 8 Output of example code of recursive method to generate a power set of a set in Java

This code recursively generates all subsets of a set, which is a task that inherently has a time complexity of $O(2^n)$, as there are 2^n possible subsets of a set. Each subset can either include or exclude a given element, hence there are 2^n possible combinations.

Examples of several efficiency measurement techniques – Space complexity

1. Constant Space Complexity: O(1)

An algorithm is said to have a constant space complexity when it needs fixed amount of space for storing data, irrespective of the size of the input. Let's look at an algorithm to find the maximum element in an array. This algorithm only needs space to store the current element and the current maximum, which is independent of the size of the array.

```
public class Main {
    public static void main(String[] args) {
        int[] arr = {2, 11, 5, 1, 8, 7};
        int max = findMax(arr);
        System.out.println("Maximum is: " + max);
    }

    static int findMax(int arr[]) {
        int max = arr[0];
        for (int i = 1; i < arr.length; i++) {
            if (arr[i] > max) {
                max = arr[i];
            }
        }
        return max;
    }
}
```

```
Run:  Main
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA
Maximum is: 11

Process finished with exit code 0
```

*Figure 5. 9 Space complexity - **Constant Space Complexity** - Example code output*

2. Linear Space Complexity: O(n)

An algorithm is said to have a linear space complexity when the space it requires grows linearly with the size of the input. A common example of an algorithm with linear space complexity is one that creates a copy of its input data. For example, let's consider an algorithm that reverses an array by first creating a copy of the array.

In this example, a new array' reversedArr' of the same size as the input array is created, resulting in a space complexity of O(n).

```java
public class Main {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int[] revArr = reverseArray(arr);

        for (int i : revArr) {
            System.out.print(i + " ");
        }
    }

    static int[] reverseArray(int[] arr) {
        int[] reversedArr = new int[arr.length];
        for (int i = 0; i < arr.length; i++) {
            reversedArr[i] = arr[arr.length - 1 - i];
        }
        return reversedArr;
    }
}
```

```
Run:  Main
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA
5 4 3 2 1
Process finished with exit code 0
```

*Figure 5. 10 Space complexity - **Linear Space Complexity** - Example code*

3. Quadratic Space Complexity: $O(n^2)$

An algorithm is said to have a quadratic space complexity when the space it requires grows quadratically with the size of the input. An example can be an algorithm that builds a table with n rows and n columns for dynamic programming purposes, like the Longest Common Subsequence (LCS) problem.

Here is an example implementation of LCS problem which has a quadratic space complexity, as it creates a two-dimensional array, where both dimensions can grow with the size of the input:

```java
public class Main {
    public static void main(String[] args) {
        String str1 = "ABCBDAB";
        String str2 = "BDCABA";

        System.out.println("Length of LCS: " + lcs(str1, str2));
    }

    static int lcs(String str1, String str2) {
        int m = str1.length();
        int n = str2.length();
```

```
        int[][] dp = new int[m+1][n+1];

        for (int i = 0; i <= m; i++) {
            for (int j = 0; j <= n; j++) {
                if (i == 0 || j == 0) {
                    dp[i][j] = 0;
                } else if (str1.charAt(i-1) == str2.charAt(j-1)) {
                    dp[i][j] = dp[i-1][j-1] + 1;
                } else {
                    dp[i][j] = Math.max(dp[i-1][j], dp[i][j-1]);
                }
            }
        }

        return dp[m][n];
    }
}
```

```
Run:  Main
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA
Length of LCS: 4

Process finished with exit code 0
```

Figure 5. 11 Space complexity - Quadratic Space Complexity - Example code

This code finds the longest common subsequence (LCS) of two strings. To do this, it creates a 2D array ‘dp’ of size (m+1)x(n+1) where m and n are the lengths of the two input strings, which gives it a space complexity of O(mn) = O(n^2) in the case where m = n.

It's important to note that space complexity isn't often expressed in terms of Logarithmic time: O(log n), Linear-Logarithmic time: O(n log n), Cubic time: $O(n^3)$, Polynomial time: $n^{O(1)}$, Exponential time: $2^{O(n)}$.

This is because space complexity describes the amount of memory space an algorithm requires to execute. Unlike time complexity, where a longer process can have different stages that operate one after another (allowing the log n and n log n time complexities to come into play), space complexity describes how much memory the algorithm needs at any point in its execution.

Examples of several efficiency measurement techniques – Trade-Off between Time and Space Complexity

Trade-offs between time complexity and space complexity are common in algorithm and software design. Sometimes, I can use more memory (increasing space complexity) to decrease the time complexity of an algorithm. Conversely, I may choose an algorithm with higher time complexity to conserve memory. This balance is a crucial aspect of optimization.

Here are a couple of examples illustrating this trade-off:

1. Memoization (Trade more space for less time)

Memoization is a technique used in dynamic programming where I store the results of expensive function calls and reuse them when necessary. This approach helps us save time by avoiding recomputation of results I already know. But it requires additional space to store the results. Hence, I trade extra space to gain time efficiency.

For example, consider the problem of computing the Fibonacci series. The naive recursive solution has an exponential time complexity because it ends up recomputing subproblems.

Here's a basic implementation of a recursive Fibonacci function in Java:

```java
public class Fibonacci {
    public static int fib(int n) {
        if (n <= 1) {
            return n;
        }
        return fib(n - 1) + fib(n - 2);
    }

    public static void main(String[] args) {
        System.out.println(fib(10));  // Output: 55
    }
}
```

```
Run:  Fibonacci
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA
55

Process finished with exit code 0
```

Figure 5. 12 Output of example code of basic implementation of a recursive Fibonacci function in Java

By using memoization, I can store previously computed values in an array and reuse them, saving time:

```
public class Fibonacci {
    public static int fib(int n) {
        int[] fibArray = new int[n + 1];
        fibArray[0] = 0;
        fibArray[1] = 1;
        for(int i = 2; i <= n; i++){
            fibArray[i] = fibArray[i - 1] + fibArray[i - 2];
        }
        return fibArray[n];
    }

    public static void main(String[] args) {
        System.out.println(fib(10));  // Output: 55
    }
}
```

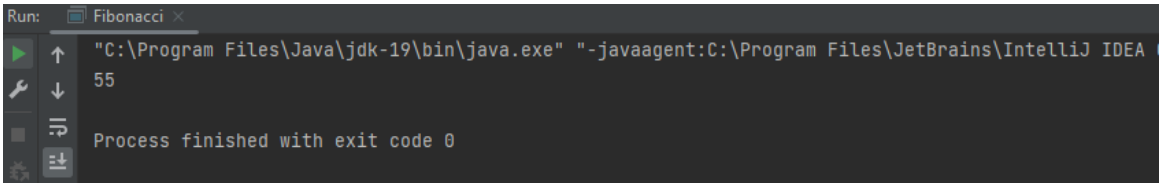


Figure 5. 13 Output of example code of using memoization

In this version, I use O(n) extra space to store our results, but I reduce our time complexity from O(2^n) to O(n).

2. Streaming algorithms (Trade more time for less space)

In contrast to memoization, streaming algorithms are designed to use very little space compared to the size of the input. They process the input piece by piece in a serial sequence, so they can't access the entire input at once. These types of algorithms often have to do more work, thus using more time, to make up for the limitation in space.

For instance, if you want to find the median in a large list of numbers that doesn't fit into memory, you could use an online algorithm (a type of streaming algorithm) that scans the numbers one at a time and calculates a running median. This method would take more time but uses very little space.

Here's an example of a simple streaming algorithm that calculates the running average of a sequence of numbers:

```
import java.util.stream.Stream;

public class RunningAverage {
    private int count = 0;
    private double sum = 0.0;

    public void accept(double value) {
        count++;
        sum += value;
    }

    public double getAverage() {
        return count > 0 ? sum / count : 0;
    }

    public static void main(String[] args) {
        RunningAverage runningAverage = new RunningAverage();
        Stream.of(1.0, 2.0, 3.0, 4.0,
5.0).forEach(runningAverage::accept);
        System.out.println(runningAverage.getAverage());  // Output: 3.0
    }
}
```

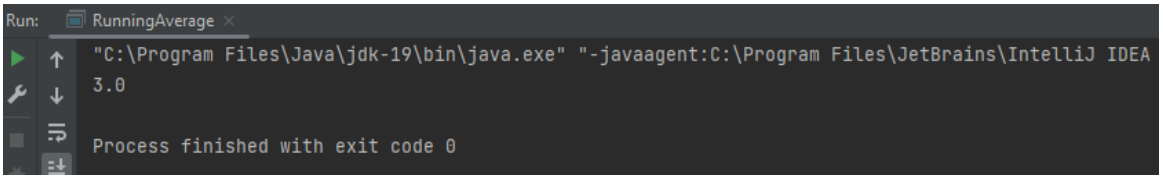


Figure 5. 14 Output of example code of simple streaming algorithm that calculates the running average of a sequence of numbers

The right choice between time and space efficiency depends on the specific constraints of the problem and the computing environment. If memory is scarce but time is plentiful, an algorithm that conserves space at the expense of time may be preferable. If time is the limiting factor, it might be worth using additional memory to speed up the computation.

Understanding Abstract Data Types (ADT) and Their Advantages

What is an Abstract Data Type (ADT)?

An Abstract Data Type (ADT) is a high-level description of data and operations on that data. It provides a blueprint or specification that outlines the required functionality of a data structure, but it doesn't implement the functionality itself. In other words, it defines what operations can be performed but not how these operations are implemented.

For example, an ADT might describe a list as having operations for adding, removing, and retrieving elements, but it wouldn't specify how the list is implemented (e.g., as an array or a linked list).

This separation between the what and the how is a key feature of ADTs and makes them a fundamental tool in programming, as it allows developers to focus on what they want their code to achieve without worrying about the lower-level details of how it works.

Encapsulation and Data Hiding in ADT

One of the major advantages of using an ADT is encapsulation, which is the bundling of data and methods that operate on that data into a single unit or 'object'. Encapsulation allows the internal state of an object to be hidden from the rest of the program, and access to it to be controlled through the object's methods.

Data hiding is a direct consequence of encapsulation and is the practice of making the data inaccessible to outside functions, except through the object's methods. This allows the internal representation of an object to be changed without affecting the rest of the program, as long as the interface stays the same.

The combination of encapsulation and data hiding provides numerous advantages:

- Modularity: The source code for an object can be written and maintained independently of the source code for other objects, making the overall codebase easier to manage.

- Information hiding: By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.
- Code reusability: If an object already exists (perhaps written by another software developer), you can use that object in your program.
- Pluggability and debugging ease: If a particular object turns out to be problematic, you can remove it from your application and plug in a different object as its replacement.

Therefore, an ADT provides a systematic way of organizing and managing software systems, making them easier to design, implement, and modify.

Understanding Trade-Offs and Their Advantages

What is Trade-Offs?

In the realm of computing and software engineering, a 'trade-off' refers to a situation where I must sacrifice a certain aspect or quality to gain or improve another. This could be sacrificing memory to improve execution speed, trading off time complexity for space complexity, or even compromising flexibility for simplicity.

For instance, when I choose to use an iterative approach over a recursive one to save memory space (recursion can lead to significant memory use due to the creation of multiple stack frames), I trade readability and elegance of recursion for the efficiency of iteration. Similarly, in object-oriented programming, I often make a trade-off between encapsulation (which improves modularity and maintainability) and simplicity (as adding layers of abstraction can make the system more complex).

These decisions aren't necessarily about right or wrong but are more about making a suitable choice based on the particular requirements and constraints of your system, such as the availability of resources (time, space, etc.), the specific performance requirements, and even the development team's expertise and comfort with certain paradigms or techniques.

Advantages of Understanding Trade-Offs

Recognizing and understanding trade-offs is a key skill in software development and has several advantages:

- Optimization of Resources: Awareness of trade-offs allows developers to make informed decisions about the optimal use of resources. For instance, in a system with limited memory, developers might choose algorithms with lower space complexity even if they have slightly higher time complexity.
- Tailored Solutions: Understanding trade-offs helps developers create solutions tailored to specific requirements rather than relying on one-size-fits-all approaches.

It allows for a finer degree of control over the performance characteristics of the system.

- Future-Proofing: By understanding the trade-offs made during the design and implementation phases, developers can better anticipate how changes in requirements or resources might affect the system. This helps in creating designs that are more adaptable to future changes.
- Better Debugging and Maintenance: Understanding the trade-offs in a system can help in debugging and maintenance. For instance, if a system is designed to prioritize speed over memory usage, developers can quickly rule out memory leaks when debugging performance issues.
- Collaborative Decision Making: In a team environment, understanding trade-offs facilitates better collaborative decision making. It provides a common language for discussing the pros and cons of different approaches.

In conclusion, trade-offs are an essential part of software development. Understanding them not only helps in making informed design and implementation decisions but also contributes to more efficient use of resources, better maintenance, and adaptability to future changes.

Trade-offs When Using an Abstract Data Type (ADT) for Implementing Programs

Examining the Trade-Offs of Using an Abstract Data Type (ADT) for Implementing Programs

When considering the implementation of programs using Abstract Data Types (ADTs), it is crucial to understand that there will always be trade-offs, which are essentially the balances between different benefits and potential drawbacks. Making the right decision depends on understanding these trade-offs. Let's discuss some of them.

1) Encapsulation vs Complexity:

ADTs provide encapsulation by bundling data with the methods operating on them. This encapsulation is a fundamental principle of Object-Oriented Programming and provides many benefits such as code modularity, improved code readability, and maintainability. However, it also introduces an extra layer of complexity into the software design. Programs implemented using ADTs can be harder to design, and more difficult to understand at first, especially for those not familiar with object-oriented concepts.

2) Data Hiding vs Debugging and Optimization:

Data hiding is another advantage of ADTs. It helps maintain the integrity of the data by preventing unauthorized access and inadvertent modification of the data. However, this feature can sometimes become a hindrance during debugging. Since the data is hidden, it may require additional methods or mechanisms to inspect or modify the data for debugging purposes.

Similarly, because the details of how an ADT is implemented are hidden, it can make certain types of optimization more challenging. The user of an ADT does not know how the ADT is implemented, so they can't make decisions based on that knowledge to make their code more efficient.

3) Interface Stability vs Flexibility:

The operations of an ADT are defined by its interface. Once defined and used, it's challenging to change the interface as it may break existing code. While this stability can be a good thing as it forces careful design and thought, it can also limit flexibility. If a new requirement arises that requires an operation not supported by the ADT's interface, it can be difficult to incorporate that into the ADT without breaking existing code.

4) Memory and Performance Costs:

While ADTs provide excellent abstraction and organization, they can also have memory and performance costs associated with them. As mentioned earlier, the overhead due to method calls, extra memory used for encapsulation, etc., can result in slower performance or higher memory usage.

Each of these trade-offs needs to be carefully considered when deciding to use an ADT for implementing a program. The requirements of the program, such as its performance requirements, its maintenance and development lifecycle, the team's familiarity with ADTs, and many other factors all play into making the right decision.

Trade-offs and Limitations of Abstract Data Types (ADT)

1) Potential Overhead Due to Abstraction:

One of the trade-offs when using an ADT is the potential for additional overhead due to the abstraction it provides. Abstraction simplifies programming by hiding the low-level details of implementation, but this comes at a cost. The use of an ADT can result in extra function calls, memory allocation, and other operations that wouldn't be necessary with a more direct implementation.

For example, an operation that could be performed directly on a raw array might require a method call when performed on an ADT that abstracts the array. This could introduce a small amount of overhead, which might be noticeable when the operation is performed many times, such as in a loop.

2) Potentially Higher Memory Usage:

Another trade-off when using an ADT is potentially higher memory usage. Because an ADT encapsulates data and operations together, it might use more memory than a simpler data structure. This is especially true when the ADT includes additional data fields or metadata to support its operations.

For instance, a linked list ADT typically uses more memory than an equivalent array because each element of the linked list requires additional memory to store the "next" reference. In some contexts, such as embedded systems or other memory-constrained environments, this increased memory usage could be a significant disadvantage.

3) Limitations of Fixed Interface:

While the fixed interface provided by an ADT is beneficial for encapsulation and data hiding, it can sometimes limit flexibility. Once an ADT is defined, its interface—meaning the set of operations it supports—cannot be changed without modifying the ADT itself and potentially all of the code that uses it. This means that if you need to perform an operation

that isn't supported by the ADT's interface, you will need to modify the ADT or find a workaround.

4) Difficulty in Optimization:

Another disadvantage is that the abstraction provided by an ADT might make it more difficult to optimize your code. Because the low-level implementation details are hidden, you may not be able to apply optimizations that would be possible with a more direct implementation.

Despite these trade-offs and limitations, the benefits of using ADTs in terms of code organization, modularity, and reusability often outweigh these potential disadvantages. The decision to use an ADT should be guided by the specific requirements of your project and the trade-offs you are willing to make.

Critical Examination of Trade-Offs When Using an Abstract Data Type (ADT) in Program Implementation

The implementation of programs using Abstract Data Types (ADTs) is a key tenet of structured programming and object-oriented design. ADTs deliver significant benefits, such as code modularity, encapsulation, data hiding, and abstraction. However, these benefits do not come without potential trade-offs. Understanding these trade-offs is crucial in the effective use of ADTs in program design and implementation. Let's critically examine these trade-offs:

1) Flexibility and Generality vs. Specificity and Performance:

ADTs often focus on generality and flexibility, ensuring they can handle a broad range of cases and be used in numerous contexts. This approach can sometimes come at the expense of optimization for specific cases. In contrast, tailor-made data structures and algorithms designed for specific tasks can often achieve better performance.

2) Abstraction Overhead:

The abstraction provided by ADTs simplifies program design and makes code easier to read and maintain. However, it also introduces overhead. The indirection and function calls associated with ADTs can lead to performance hits, especially in time-critical applications. Programmers need to weigh the benefits of abstraction against the potential for performance degradation.

3) Increased Memory Use:

ADTs can sometimes lead to increased memory use due to the extra data and metadata they need to function. For instance, object-oriented ADTs often need to maintain "vtables" or similar structures to track method addresses, which can add memory overhead. For applications with tight memory constraints, this can be a crucial factor.

4) Complexity and Learning Curve:

While ADTs can make code more organized and easier to maintain, they also introduce an additional layer of complexity. This complexity can lead to a steeper learning curve, especially for novice programmers. Misuse of ADTs can even result in code that is harder to understand and maintain.

5) Rigidity of Interface:

ADTs define a set of operations via an interface. While this provides a clean, stable way to interact with the data, it can also make ADTs rigid. If requirements change and new operations need to be added, changing the interface can be challenging and potentially break existing code.

6) Encapsulation vs. Debugging and Inspection:

The data hiding principle of ADTs can sometimes complicate debugging and data inspection. When data is encapsulated within an ADT, additional methods (like getters or setters) may be needed to access the data for debugging or inspection purposes.

In conclusion, while ADTs offer significant advantages, they also present potential trade-offs in areas such as performance, memory use, complexity, learning curve, and rigidity. Developers need to be aware of these trade-offs to make informed decisions when choosing whether to use ADTs in their program implementations. As always, there is no "one size fits all" solution in software engineering, and the use of ADTs should be based on the specific requirements and constraints of the project.

Illustrating Trade-Offs using ADT for Implementing Programs

1) Trade-off Between Time and Space Complexity:

To understand the trade-off between time and space complexity, consider the problem of finding an element in an array. Here are two approaches with different time and space complexities:

- Linear Search (Lower Space, Higher Time)
- Hashing (Higher Space, Lower Time)

1. Linear Search:

Linear search traverses the array linearly to find the target element. This approach does not require any extra space, making its space complexity O(1). However, in the worst case, it could take up to n comparisons (where n is the number of elements), making its time complexity O(n).

```
public class LinearSearchExample {
    public static void main(String[] args) {
        int[] array = {1, 4, 6, 7, 9, 15};
        int target = 6;
        int index = linearSearch(array, target);
        if (index != -1) {
            System.out.println("Element found at index: " + index);
        } else {
            System.out.println("Element not found in the array.");
        }
    }

    public static int linearSearch(int[] arr, int target) {
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == target) {
                return i; // return index of target element
            }
        }
        return -1; // return -1 if target is not found
    }
}
```

```
Run:  LinearSearchExample
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA
Element found at index: 2

Process finished with exit code 0
```

Figure 5. 15 Output of example code of Linear Search in Trade-off Between Time and Space Complexity

2. Hashing:

In contrast, if I use a HashTable to store the elements, I can find the target element in constant time O(1), on average. However, this comes at the cost of additional space to store the HashTable, resulting in space complexity of O(n).

```java
import java.util.HashSet;

public class HashingExample {
    public static void main(String[] args) {
        int[] array = {1, 4, 6, 7, 9, 15};
        int target = 6;
        boolean isPresent = isPresent(array, target);
        if (isPresent) {
            System.out.println("Element is present in the array.");
        } else {
            System.out.println("Element not found in the array.");
        }
    }

    public static boolean isPresent(int[] arr, int target) {
        HashSet<Integer> set = new HashSet<Integer>();
        for (int num : arr) {
            set.add(num);
        }
        return set.contains(target); // return true if target is present
    }
}
```

```
Run:  HashingExample
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA
Element is present in the array.

Process finished with exit code 0
```

Figure 5. 16 Output of example code of Hashing in Trade-off Between Time and Space Complexity

2) Trade-off Between Performance and Code Readability:

Another trade-off I frequently encounter in programming is between performance and code readability. To illustrate this, consider the problem of calculating the Fibonacci series.

1. Recursive Approach:

A recursive approach provides a very elegant and simple-to-understand solution. However, it has an exponential time complexity of O(2^n) due to multiple redundant calculations.

```
public class RecursiveFibonacci {
    public static void main(String[] args) {
        int n = 10; // Calculate the 10th Fibonacci number
        int fibonacciNumber = fibonacci(n);
        System.out.println("The 10th Fibonacci number is: " +
fibonacciNumber);
    }

    public static int fibonacci(int n) {
        if (n <= 1) {
            return n;
        }
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

```
Run:  RecursiveFibonacci
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA
The 10th Fibonacci number is: 55

Process finished with exit code 0
```

Figure 5. 17 Output of example code of Recursive Approach in Trade-off Between Performance and Code Readability

2. Dynamic Programming Approach:

A dynamic programming approach, while being a bit more complex to understand, significantly optimizes the time complexity to O(n) by avoiding redundant calculations.

```
public class DPFibonacci {
    public static void main(String[] args) {
        int n = 10; // Calculate the 10th Fibonacci number
        int fibonacciNumber = fibonacci(n);
        System.out.println("The 10th Fibonacci number is: " +
fibonacciNumber);
    }
```

```
    public static int fibonacci(int n) {
        if (n <= 1) {
            return n;
        }
        int[] fib = new int[n + 1];
        fib[0] = 0;
        fib[1] = 1;
        for (int i = 2; i <= n; i++) {
            fib[i] = fib[i - 1] + fib[i - 2];
        }
        return fib[n];
    }
}
```

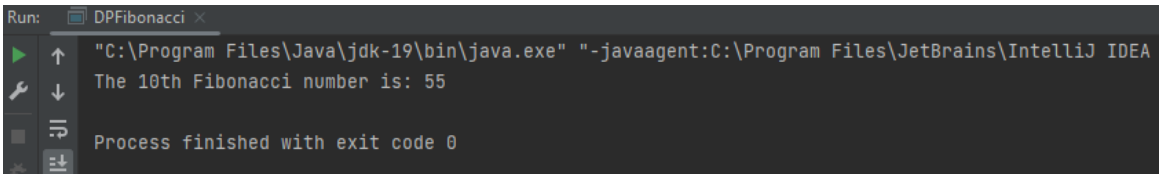


Figure 5. 18 Output of example code of Dynamic Programming Approach in Trade-off Between Performance and Code Readability

3) Trade-Off Between Abstraction and Performance

Let's consider the trade-off between using an Abstract Data Type (ADT) and performance. A good example here is the choice between using a high-level collection class such as an ArrayList in Java and a low-level array.

1. Use of ADT - ArrayList:

ArrayList in Java provides a dynamic array that allows us to add or remove elements, which a primitive array cannot offer without complex manipulations. This convenience comes at a cost, though. An ArrayList uses more memory than a simple array due to the extra features it offers.

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(2);
        list.add(3);
        System.out.println(list.get(1)); // Prints "2"
    }
}
```

```
Run:  Main
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA
2

Process finished with exit code 0
```

Figure 5. 19 Output of example code of Use of ADT - ArrayList in Trade-Off Between Abstraction and Performance

2. Use of Primitive Data Structure - Array:

If I use a primitive array, I use less memory and the access time is slightly faster because there is no overhead from the object methods. However, it comes with a lack of flexibility. Adding or removing elements from an array is a manual and error-prone process.

```
public class Main {
    public static void main(String[] args) {
        int[] array = new int[3];
        array[0] = 1;
        array[1] = 2;
        array[2] = 3;
        System.out.println(array[1]); // Prints "2"
```

```
    }
}
```

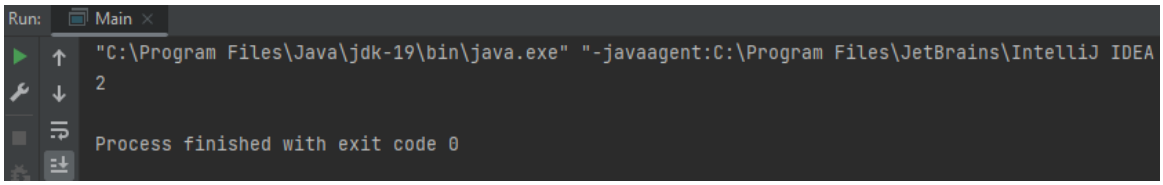


Figure 5. 20 Output of example code of Use of Primitive Data Structure - Array in Trade-Off Between Abstraction and Performance

4) Trade-Off Between Development Time and Runtime Efficiency

Another common trade-off is between the time it takes to develop and test code (development time) and how quickly the code executes (runtime efficiency). For instance, consider a simple task of sorting an array of integers.

1. Built-in Sort Function:

Most programming languages provide built-in functions for common tasks such as sorting. These functions are fast to implement and often optimized for average-case scenarios. However, they may not be the fastest option for a specific dataset or use case.

```
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        int[] array = {3, 2, 1};
        Arrays.sort(array);
        System.out.println(Arrays.toString(array)); // Prints "[1, 2,
3]"
    }
}
```

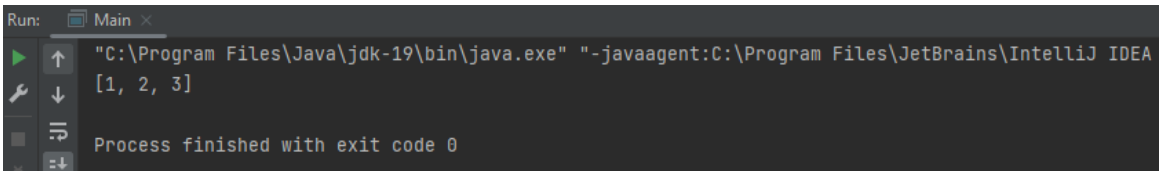


Figure 5. 21 Output of example code of Built-in Sort Function in ***Trade-Off Between Development Time and Runtime Efficiency***

2. Custom Sort Function:

Implementing a custom sorting algorithm can be time-consuming, but it allows for optimizations based on specific knowledge of the dataset or problem constraints. For instance, if I know the input is nearly sorted, I might choose to use insertion sort, which can outperform other more general algorithms in this scenario.

```
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        int[] array = {3, 2, 1};
        insertionSort(array);
```

```
        System.out.println(Arrays.toString(array)); // Prints "[1, 2,
3]"
    }

    public static void insertionSort(int[] array) {
        for (int i = 1; i < array.length; i++) {
            int key = array[i];
            int j = i - 1;
            while (j >= 0 && array[j] > key) {
                array[j + 1] = array[j];
                j = j - 1;
            }
            array[j + 1] = key;
        }
    }
}
```

```
Run:  Main
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA
[1, 2, 3]

Process finished with exit code 0
```

Figure 5. 22 Output of example code of Custom Sort Function in Trade-Off Between Development Time and Runtime Efficiency

These examples clearly illustrate the importance of understanding trade-offs when making decisions about how to implement a program or algorithm. By comprehending these trade-offs, developers can make more informed decisions and build more efficient, effective solutions.

Benefits of using independent data structures for implementing program

Enhancing Code Quality through Independent Data Structures

1) Modularity with Independent Data Structures:

Modularity is the degree to which a system's components may be separated and recombined. In the context of software design, it is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.

Independent data structures can contribute to software modularity in a significant way. By encapsulating data and the operations that can be performed on that data within an independent structure, I can create modules that can be developed, tested, and understood separately from others. This not only simplifies the overall development process but also allows different modules to evolve independently over time.

2) Reusability through Independent Data Structures:

Reusability is a measure of the ability to use components of a code in various contexts. With independent data structures, I can write the code for a particular data structure once and use it in multiple programs. This not only saves time but also helps keep our code DRY (Don't Repeat Yourself).

For example, once I've implemented the ‘LinkedList’ class above, I can reuse it in any program that needs a linked list data structure. I just need to create an instance of the ‘LinkedList’ class.

3) Readability through Independent Data Structures:

Readability refers to how easy it is for a developer to understand a piece of code. Using independent data structures enhances readability by abstracting complex operations behind intuitive methods.

For instance, the ‘append()’ method in the ‘LinkedList’ class provides an intuitive way to add a node to the end of the list. A developer using this class doesn't need to understand the details of how a node is added. This abstraction makes the code easier to read and understand.

Following is a full code example illustrating the concepts of modularity, reusability, and readability using independent data structures. This example is a simple Java application that manages a list of employees using a LinkedList.

```
// Define the Node class, which will hold data for each employee
class EmployeeNode {
    Employee data;
    EmployeeNode next;

    EmployeeNode(Employee d) {
        data = d;
        next = null;
    }
}

// Define the Employee class, which encapsulates employee data
class Employee {
    String name;
    int id;

    Employee(String name, int id) {
        this.name = name;
        this.id = id;
    }

    @Override
    public String toString() {
        return "Employee Name: " + this.name + ", ID: " + this.id;
    }
}

// Define the EmployeeList class, which provides operations on a list of
employees
class EmployeeList {
    EmployeeNode head;

    // Insert a new Employee at the end of the list
    public void addEmployee(Employee new_data) {
        EmployeeNode new_node = new EmployeeNode(new_data);
        if (head == null) {
            head = new EmployeeNode(new_data);
            return;
        }
        new_node.next = null;
        EmployeeNode last = head;
        while (last.next != null) {
```

```
            last = last.next;
        }
        last.next = new_node;
    }

    // Print the list of employees
    public void printList() {
        EmployeeNode node = head;
        while (node != null) {
            System.out.println(node.data);
            node = node.next;
        }
    }
}

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

        Employee emp1 = new Employee("Alice", 101);
        Employee emp2 = new Employee("Bob", 102);
        Employee emp3 = new Employee("Charlie", 103);

        employeeList.addEmployee(emp1);
        employeeList.addEmployee(emp2);
        employeeList.addEmployee(emp3);

        employeeList.printList();
    }
}
```

```
Run:  Main
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA
Employee Name: Alice, ID: 101
Employee Name: Bob, ID: 102
Employee Name: Charlie, ID: 103

Process finished with exit code 0
```

Figure 5. 23 Output of example code of simple Java application that manages a list of employees using a LinkedList

- Modularity is demonstrated by the separation of concerns – 'Employee', 'EmployeeNode', and 'EmployeeList' classes each have their own responsibilities.
- Reusability is demonstrated by the fact that 'Employee' and 'EmployeeList' classes can be reused in other parts of the program, or even in different programs, without any changes.
- Readability is demonstrated by the simple, self-explanatory methods provided by 'EmployeeList', such as 'addEmployee()' and 'printList()'. This makes the code in the 'main' method easy to understand.

Advantages of Independent Data Structures in Separation of Concerns, Maintenance, and Extension

Independent data structures can be instrumental in achieving a higher degree of separation of concerns, easier maintenance, and more straightforward extension of a software system. Here's how:

1) Separation of Concerns with Independent Data Structures

Separation of concerns is a design principle for separating a computer program into distinct sections, each addressing a separate concern. It simplifies development and maintenance by compartmentalizing a complex system into manageable parts.

By using independent data structures, I allow different aspects of a program to reside in separate entities. For instance, an independent stack data structure might contain the operations for pushing, popping, and peeking elements. This keeps the responsibility of managing the underlying data encapsulated within the structure. Any part of the program that utilizes this stack does not need to concern itself with how these operations are implemented, allowing it to focus on its own concerns.

Let's say a music streaming application needs a playlist feature where users can add or remove songs, and also play them in a specific order. I can use a Queue data structure for this, with methods like ‘addSong()’, ‘removeSong()’, and ‘playNext()’.

```
public class Playlist {
    private Queue<Song> songs;

    public Playlist() {
        this.songs = new LinkedList<>();
    }

    public void addSong(Song song) {
        this.songs.add(song);
    }

    public void removeSong(Song song) {
        this.songs.remove(song);
    }

    public Song playNext() {
        return this.songs.poll();
    }
}
```

The Queue here is an independent data structure that encapsulates the operations relevant to handling a list of songs. The rest of the application, such as the UI or networking code, doesn't need to know how the Playlist handles its songs internally.

2) Easier Maintenance with Independent Data Structures

When a data structure is independent, changes to its implementation are less likely to impact the rest of the code. For example, if I need to change how data is stored within the structure, I only need to update the operations within the structure itself. As long as the interface remains the same, no other parts of the program need to be modified. This modularity greatly simplifies maintenance, as bugs can be isolated and fixed within their own module without worrying about unintentional side effects.

Furthermore, it can also improve the testing process. Independent modules can be tested individually before integrating them into the larger system, ensuring that each component works as expected on its own.

Suppose later on, I want to change how songs are played, such that the least recently played song is played next (a kind of history feature). For this, I can change our Playlist to use a different data structure, say a Deque (Double Ended Queue). The operations ‘addSong()’ and ‘removeSong()’ remain the same, but ‘playNext()’ now pops from the front of the Deque.

```
public class Playlist {
    private Deque<Song> songs;

    public Playlist() {
        this.songs = new LinkedList<>();
    }

    public void addSong(Song song) {
        this.songs.addLast(song);
    }

    public void removeSong(Song song) {
        this.songs.remove(song);
    }

    public Song playNext() {
        return this.songs.pollFirst();
    }
}
```

Even though I changed the underlying data structure, the rest of the application doesn't need to change as long as it only uses ‘addSong()’, ‘removeSong()’, and ‘playNext()’.

3) Easier Extension of the Program with Independent Data Structures

Independent data structures can also make it easier to extend a program. If new features require additional operations on a data structure, these can be added to the structure's interface without affecting existing code. For example, suppose I initially implement a list with add and remove operations. Later, I might want to add an operation that finds the smallest element. I can simply add a `findMin` method to the list's interface without needing to change the code that uses the list.

Let's say I want to add a new feature to our Playlist where a user can peek at the next song without actually playing it. I can simply add a new method ‘peekNext()’ to the Playlist.

```
public class Playlist {
    // ...existing code...

    public Song peekNext() {
        return this.songs.peekFirst();
    }
}
```

Again, I've extended our application without having to change any existing code. As long as the code elsewhere in our application uses the Playlist's methods, it's unaffected by the internal changes.

To summarize, the use of independent data structures is a sound strategy to improve the separation of concerns, maintenance, and extension of software. They allow software to be compartmentalized into manageable, testable, and maintainable units, thereby improving the overall software development process. The examples provided shows how using independent data structures can help improve modularity, reusability, readability, and maintainability of code, and how it allows for easier extension of the program.

Conclusion

Throughout this comprehensive assignment, I delved deep into the intricate world of algorithms, data structures, and their interrelationships. I began by exploring the basic operations on data structures such as arrays and dynamic queues. Our journey continued with a thorough examination of different algorithms and their relevancies to real-world scenarios, particularly focusing on ABC Pvt Ltd's unique challenges.

I ventured into understanding the foundational aspects of Asymptotic Analysis, which is pivotal in evaluating the efficiency of algorithms. Through Asymptotic Analysis, I derived insights into how algorithms perform under varying input sizes, helping us make informed decisions in algorithm selection for particular scenarios.

Abstract Data Types (ADTs) were a focal point, showcasing their immense value in encapsulating and hiding data, which ensures robust and modular code structures. Furthermore, the assignment highlighted the significance of understanding trade-offs. Recognizing the potential advantages and limitations of each decision in the world of algorithms and data structures is vital for efficient problem-solving.

Lastly, the assignment underscored the advantages of employing independent data structures. Such structures play a crucial role in improving code quality, enhancing maintenance, ensuring separation of concerns, and facilitating extensibility.

References

Oracle. (n.d.). Arrays. Available at:
https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html

[Accessed 15 May 2023].

GeeksforGeeks. (n.d.). What is an Array? Available at:
https://www.geeksforgeeks.org/what-is-array/

[Accessed 17 May 2023].

GeeksforGeeks. (n.d.). Stack Data Structure. Available at:
https://www.geeksforgeeks.org/stack-data-structure/

[Accessed 20 May 2023].

Programiz. (n.d.). Stack Data Structure. Available at:
https://www.programiz.com/dsa/stack

[Accessed 22 May 2023].

GeeksforGeeks. (n.d.). Queue Data Structure. Available at:
https://www.geeksforgeeks.org/queue-data-structure/

[Accessed 25 May 2023].

Programiz. (n.d.). Queue Data Structure. Available at:
https://www.programiz.com/dsa/queue

[Accessed 28 May 2023].

GeeksforGeeks. (n.d.). Static vs Dynamic Data Structure. Available at: https://www.geeksforgeeks.org/static-data-structure-vs-dynamic-data-structure/

[Accessed 30 May 2023].

Scaler. (n.d.). Static and Dynamic Data Structure. Available at: https://www.scaler.com/topics/static-and-dynamic-data-structure/

[Accessed 2 June 2023].

Swift.org. (n.d.). The Swift Programming Language: Error Handling. Available at: https://docs.swift.org/swift-book/documentation/the-swift-programming-language/errorhandling/

[Accessed 6 June 2023].

Guru99. (n.d.). Test Case. Available at: https://www.guru99.com/test-case.html

[Accessed 10 June 2023].

Guru99. (n.d.). What everybody ought to know about Test Planning. Available at: https://www.guru99.com/what-everybody-ought-to-know-about-test-planing.html

[Accessed 15 June 2023].

Simplilearn. (n.d.). Time and Space Complexity. Available at: https://www.simplilearn.com/tutorials/data-structure-tutorial/time-and-space-complexity

[Accessed 18 June 2023].

GeeksforGeeks. (n.d.). Understanding Time Complexity with Simple Examples. Available at: https://www.geeksforgeeks.org/understanding-time-complexity-simple-examples/

[Accessed 22 June 2023].

GeeksforGeeks. (n.d.). G-Fact 86. Available at: https://www.geeksforgeeks.org/g-fact-86/

[Accessed 25 June 2023].

Stackify. (n.d.). OOP Concept for Beginners: What is Encapsulation? Available at: https://stackify.com/oop-concept-for-beginners-what-is-encapsulation/

[Accessed 28 June 2023].

Medium. (n.d.). Abstract Data Types and Objects. Available at: https://medium.com/@jnkrtech/abstract-data-types-and-objects-17828bd4abdc

[Accessed 30 June 2023].

GeeksforGeeks. (n.d.). Bubble Sort. Available at: https://www.geeksforgeeks.org/bubble-sort/

[Accessed 5 July 2023].

Simplilearn. (n.d.). Merge Sort Algorithm. Available at: https://www.simplilearn.com/tutorials/data-structure-tutorial/merge-sort-algorithm

[Accessed 8 July 2023].

GeeksforGeeks. (n.d.). Bellman-Ford Algorithm. Available at: https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/

[Accessed 12 July 2023].

GeeksforGeeks. (n.d.). Dijkstra's Shortest Path Algorithm. Available at: https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

[Accessed 15 July 2023].

GeeksforGeeks. (n.d.). Asymptotic Analysis. Available at: https://www.geeksforgeeks.org/asymptotic-notation-and-analysis-based-on-input-size-of-algorithms/

[Accessed 20 July 2023].

Data Flair. (n.d.). Asymptotic Analysis of Algorithms in Data Structures. Available at: https://data-flair.training/blogs/asymptotic-analysis-of-algorithms-in-data-structures/

[Accessed 24 July 2023].

Khan Academy. (n.d.). Measuring an Algorithm's Efficiency. Available at: https://www.khanacademy.org/computing/ap-computer-science-principles/algorithms-101/evaluating-algorithms/a/measuring-an-algorithms-efficiency

[Accessed 28 July 2023].

Quescol. (n.d.). Abstract Data Type. Available at: https://quescol.com/data-structure/abstract-data-type

[Accessed 30 July 2023].

Medium. (n.d.). Advantages of Abstract Data Types (ADTs). Available at: https://tsfigueira.medium.com/advantages-of-abstract-data-types-adts-22e502d0d41c

[Accessed 2 August 2023].

Citizen Choice. (n.d.). Time-Space Trade-off & Abstract Data Types (ADT). Available at: https://citizenchoice.in/course/Data-Structures-and-Algorithms/Chapter%201/Time-Space-Trade-off-Abstract-Data-Types-ADT-

[Accessed 8 August 2023].