

ASSIGNMENT FINAL REPORT


Qualification	Pearson BTEC Level 5 Higher National Diploma in Computing		
Unit number and title	Unit 19: Data Structures and Algorithms		
Submission date	27/10/2024	Date Received 1st submission	
Re-submission Date		Date Received 2nd submission	
Student Name	Nguyen Van Tuyen	Student ID	BH01786
Class	SE07102	Assessor name	Dinh Van Dong

Plagiarism

Plagiarism is a particular form of cheating. Plagiarism must be avoided at all costs and students who break the rules, however innocently, may be penalised. It is your responsibility to ensure that you understand correct referencing practices. As a university level student, you are expected to use appropriate references throughout and keep carefully detailed notes of all your sources of materials for material you have used in your work, including any material downloaded from the Internet. Please consult the relevant unit lecturer or your course tutor if you need any further advice.

Student Declaration

I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. I declare that the work submitted for assessment has been carried out without assistance other than that which is acceptable according to the rules of the specification. I certify I have clearly referenced any sources and any artificial intelligence (AI) tools used in the work. I understand that making a false declaration is a form of malpractice.

Student's signature	
----------------------------	---

for

Grading grid

[illegible]

☐ **Summative Feedback:**

☐ **Resubmission Feedback:**

Grade:

Assessor Signature:

Date:

Internal Verifier's Comments:

Signature & Date:

Table of Contents

I. Introduction	7
II. Content	7
I. Data Structures and Complexity	7
1.1 Identify the Data Structures	7
1.2 Define the Operations	8
1.3 Specify Input Parameters	9
1.4 Define Pre- and Post-conditions	10
1.5 Discuss Time and Space Complexity	10
1.6 Provide Examples and Code Snippets	12
II. Memory Stack	13
2.1 Define a Memory Stack	13
2.2 Identify Operations	14
2.3 Function Call Implementation	15
2.4 Demonstrate Stack Frames	16
2.5 Discuss the Importance	17
III. Queue	18
3.1 Introduction FIFO	18
3.2 Define the Structure	19
3.3 Array-Based Implementation	21
3.4 Linked List-Based Implementation	22
3.5 Provide a concrete example to illustrate how the FIFO queue works	24
IV. Sorting Algorithms	25
4.1 Introducing the two sorting algorithms you will be comparing	25
4.2 Time Complexity Analysis	26
4.3 Space Complexity Analysis	27
4.4 Stability	27
4.5 Comparison Table	28
4.6 Performance Comparison	29
4.7 Provide a concrete example to demonstrate the differences in performance between the two algorithms	30
V. Network Shortest Path Algorithms	33
5.1 Introducing the concept of network shortest path algorithms	33
5.2 Algorithm 1: Dijkstra's Algorithm	34
5.3 Algorithm 2: Prim-Jarnik Algorithm	35
5.4 Performance Analysis	36
VI. Abstract Data Type for Software Stack Using Imperative Definition	37
6.1 Define the Data Structure	37
6.2 Initialise the Stack	38
6.3 Push Operation	38
6.4 Pop Operation	38

6.5 Peek Operation	39
6.6 Check if the Stack is Empty.....	39
6.7 Full Source Code (Stack using Array).....	39
6.8 Full Source Code (Stack using Linked List).....	40
6.9 Comparison	42
VII. Advantages of Encapsulation and Information Hiding in ADTs.....	43
7.1 Encapsulation	43
7.1.1 What is Encapsulation.....	43
7.1.2 Example of Encapsulation	44
7.1.3 Data Protection	45
7.1.4 Modularity and Maintainability	46
7.1.5 Code Reusability.....	47
7.2 Information Hiding	48
7.2.1 Abstraction.....	49
7.2.2 Reduced Complexity	50
7.2.3 Improved Security	50
VIII. Imperative ADTs as a Basis for Object-Oriented Programming	51
8.1 Encapsulation and Information Hiding	51
8.2 Modularity and Reusability	51
8.3 Procedural Approach	51
8.4 Language Transition	52
III. Conclusion.....	52
IV. References	53
Link Github: https://github.com/tuen1985/Assignment1_DSA	54

Table of Figures

Figure 1: Adding a Student	12
Figure 2: Editing a Student's Information	12
Figure 3: After Editing a Student's Information	12
Figure 4: Deleting a Student	12
Figure 5: Sorting Students by Score (Bubble Sort).....	12
Figure 6: Sorting Students by Name (Selection Sort)	13
Figure 7: Printing Using Stack and Queue.....	13
Figure 8: Searching for a Student by ID	13
Figure 9: FIFO	18
Figure 10: Output of FIFO code snippet.....	24
Figure 11: Sorting Algorithms	25
Figure 12: Merge Sort	25
Figure 13: QuickSort	26
Figure 14: Merge Sort vs QuickSort	33
Figure 15: Dijkstra's Algorithm.....	34
Figure 16: Prim-Jarnik Algorithm	35
Figure 17: Data structure	37
Figure 18: Encapsulation.....	43
Figure 19: Data Protection.....	45
Figure 20: Information Hiding.....	48
Figure 21: Abstraction.....	49
Figure 22: Reduced Complexity	50
Figure 23: Improved Security.....	50

I. Introduction

In modern software development, efficiently organizing and manipulating data is central to building robust, scalable, and maintainable systems. A fundamental concept in data organization is the use of Abstract Data Types (ADTs), which define the operations that can be performed on data while abstracting away implementation details. ADTs serve as blueprints, enabling developers to focus on functionality rather than specific structures. Among the most commonly utilized ADTs is the stack, an essential data structure characterized by its Last-In-First-Out (LIFO) behavior. This principle dictates that the most recently added element is the first to be removed, a characteristic that proves valuable in applications such as function call management, expression evaluation, and algorithm design. This paper provides an imperative definition of the stack, detailing its initialization and fundamental operations—push, pop, and peek—while contrasting array-based and linked list-based implementations to highlight their respective benefits and limitations.

Beyond the stack, this paper also addresses other crucial data structures, including queues, and algorithms that form the backbone of efficient data handling in software applications. In particular, we will explore sorting algorithms and network shortest path algorithms, examining their time and space complexities to assess how these factors influence performance in real-world applications. By comparing algorithmic approaches such as Dijkstra's and Prim-Jarnik's, the discussion offers insights into optimizing pathfinding in networked systems, a critical component of many modern applications.

Furthermore, this paper examines key principles of object-oriented programming (OOP), including encapsulation and information hiding, which enhance data integrity and modularity. Encapsulation, by limiting access to an object's internal state, prevents unintended modifications and allows for a structured interface for manipulating data. Information hiding supports a clear separation between an object's implementation and its usage, promoting modularity and simplifying code maintenance and reuse. Together, these principles strengthen software structure and security, fostering a transition from procedural to object-oriented paradigms. This exploration aims to provide a comprehensive overview of essential data structures, algorithms, and OOP principles, illuminating their importance in developing high-quality, maintainable software solutions.

II. Content

I. Data Structures and Complexity

1.1 Identify the Data Structures

1. **LinkedList** (for storing students in the StudentManagement class)
 - Purpose: Used to store and manage a collection of Student objects. LinkedList allows efficient insertion and deletion of elements, which is beneficial when managing a dynamic list of students.

- Characteristics: Linked lists are dynamically resizable and provide efficient element insertion and deletion.
2. **Queue** (using LinkedList as the implementation for a FIFO structure in printUsingStackQueue):
 - Purpose: Represents a queue of students where elements are processed in a first-in, first-out (FIFO) order.
 - Characteristics: Queue is ideal for sequential processing, where elements are added at the end and removed from the front.
 3. **Custom Stack Implementation** (StudentStack class):
 - Purpose: A stack for managing Student objects in a last-in, first-out (LIFO) manner. This structure is used to reverse the order of processing, as shown in the printUsingStackQueue method.
 - Characteristics: Implements a fixed-size stack with standard operations like push, pop, peek, and isEmpty.
 4. **Primitive Array** (used in StudentStack):
 - Purpose: The StudentStack class uses an array internally to store Student objects. This provides a simple, contiguous memory structure with fixed capacity.
 - Characteristics: Arrays provide fast access by index but lack dynamic resizing unless managed manually.
 5. **Scanner** (for user input in main method):
 - Purpose: Scanner is used to read inputs from the console, allowing interactive data entry for adding, editing, and deleting students.
 - Characteristics: Scanner provides methods to read various data types, making it convenient for console-based applications.

1.2 Define the Operations

Operation	Method	Purpose	Details
Add a Student	addStudent(String id, String name, double score)	Adds a new Student to the students list by creating a Student object with the given ID, name, and score.	Checks the validity of the ID and name, then appends the new student to the end of the LinkedList.
Edit a Student	editStudent(String id, String newName, double newScore)	Edits an existing student's name and score by searching for the student by ID.	Updates the name, score, and recalculates the student's ranking. If the student is not found, it displays an error message.
Delete a Student	deleteStudent(String id)	Removes a student from the students list based on their ID.	Uses removeSelf to find and delete the student. Prints a success message if the

			student is deleted or an error message if not found.
Sort Students by Score	sortByScoreBubbleSort()	Sorts students in ascending order by score using the bubble sort algorithm.	Iteratively compares adjacent students, swapping them if they are out of order.
Sort Students by Name	sortByNameSelectionSort()	Sorts students alphabetically by name using the selection sort algorithm.	Finds the student with the smallest name in each pass and swaps them to sort the list alphabetically.
Print Students Using Stack and Queue	printUsingStackQueue()	Prints students twice—first in LIFO (Stack) order and then in FIFO (Queue) order.	Pushes all students to a custom stack and a queue, then prints and removes each from both structures.
Search Student by ID	searchByID(String id)	Searches for a student by ID using a linear search.	Iterates through the list, returning the student if found; returns null otherwise.
Display All Students	displayStudents()	Displays the details of all students in the students list.	Iterates through the list, printing each student's information. If the list is empty, it prints a message indicating no students are available.
Menu-driven Console Interface	main(String[] args)	Provides a console interface with options for adding, editing, deleting, sorting, printing, and searching students.	Uses a Scanner for input and displays a menu in a loop, allowing users to select different operations.

1.3 Specify Input Parameters

Method	Parameters	Description
addStudent	"id" (String), "name" (String), "score" (double)	"id" is the unique identifier for the student, "name" is the student's name, and "score" is the student's score.
editStudent	"id" (String), "newName" (String), "newScore" (double)	"id" is the unique identifier of the student to edit, "newName" is the updated name, and "newScore" is the updated score.
deleteStudent	"id" (String)	"id" is the unique identifier of the student to delete.
searchByID	"id" (String)	"id" is the unique identifier of the student to search for.

1.4 Define Pre- and Post-conditions

In software development and programming, pre-conditions and post-conditions are logical conditions or assertions associated with functions, methods, or code segments that specify what must be true before and after the execution of a particular operation.

1. Pre-conditions

A precondition is a condition, or a predicate, that must be true before a method runs for it to work. In other words, the method tells clients, “this is what I expect from you”. So, the method we are calling is expecting something to be in place before or at the point of the method being called. The operation is not guaranteed to perform as it should unless the precondition has been met. (Bors, 2018) [1]

Purpose: They ensure that the input and initial state meet certain criteria, helping the program run as expected without errors.

Example: For a function that divides two numbers, a pre-condition might be that the divisor must not be zero.

2. Post-conditions

A postcondition is a condition, or a predicate, that can be guaranteed after a method is finished. In other words, the method tells clients, “this is what I promise to do for you”. If the operation is correct and the precondition(s) met, then the postcondition is guaranteed to be true. (Bors, 2018) [2]

Purpose: They confirm that the function has correctly achieved its intended result, often verifying the state of the program or output values.

Example: For a sorting function, a post-condition would be that the list or array should be in non-decreasing order after the function completes.

1.5 Discuss Time and Space Complexity

Time Complexity and Space Complexity are metrics in computer science used to analyze the efficiency of algorithms, specifically focusing on the amount of time and memory an algorithm requires to run.

1. Time Complexity:

The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Note that the time to run is a function of the length of the input and not the actual execution time of the machine on which the algorithm is running on. (GeeksforGeeks, 2021) [2]

Purpose: It provides insight into the scalability of an algorithm, i.e., how the execution time grows as the input size increases.

Big O Notation: Time complexity is commonly expressed using Big O notation, which describes the worst-case scenario. For example:

- $O(1)$: Constant time – the operation is independent of input size.
- $O(\log n)$: Logarithmic time – time increases logarithmically as input size grows, common in binary search.
- $O(n)$: Linear time – time grows proportionally with input size, often seen in simple loops.
- $O(n \log n)$: Log-linear time – typical for more efficient sorting algorithms, like merge sort.
- $O(n^2)$: Quadratic time – time grows quadratically, seen in nested loops, such as in bubble sort.
- $O(2^n)$: Exponential time – time doubles with each additional input element, typical in recursive algorithms with branching, like the naive Fibonacci sequence.

2. Space complexity:

Problem-solving using computer requires memory to hold temporary data or final result while the program is in execution. The amount of memory required by the algorithm to solve given problem is called space complexity of the algorithm. (GeeksforGeeks, 2021)

Purpose: It assesses how much extra space or memory an algorithm will require, which is essential for ensuring that the algorithm is feasible to run on hardware with limited memory.

Components: Space complexity is often divided into:

- Auxiliary Space: Extra space or temporary storage an algorithm uses in addition to the input data.
- Input Space: Memory required to store the input itself.
- Big O Notation: Similar to time complexity, space complexity is also expressed in Big O notation.
- $O(1)$: Constant space – requires a fixed amount of space regardless of input size.
- $O(n)$: Linear space – requires memory proportional to the input size, such as in cases where an algorithm stores results in an array.
- $O(n^2)$: Quadratic space – requires space proportional to the square of the input size, as seen in algorithms storing all possible pairs.

3. Importance of Time and Space Complexity

Efficiency and Scalability: Analyzing time and space complexity helps developers and researchers choose the most efficient algorithm, especially for applications with large input sizes.

Trade-Offs: Often, there's a trade-off between time and space complexity. For instance, some algorithms achieve faster speeds by using more memory, while others save space at the cost of longer execution time.

1.6 Provide Examples and Code Snippets

1. Adding a Student

```
ID: BH07749, Name: Nguyen Van A, Score: 8.0, Ranking: Very Good  
ID: BH04953, Name: Nguyen Thi B, Score: 5.4, Ranking: Average
```

Figure 1: Adding a Student

2. Editing a Student's Information

```
Enter ID to edit: BH04953  
Enter new Name: Tran Thi B  
Enter new Score: 5.0
```

Figure 2: Editing a Student's Information

```
ID: BH07749, Name: Nguyen Van A, Score: 8.0, Ranking: Very Good  
ID: BH04953, Name: Tran Thi B, Score: 5.0, Ranking: Average
```

Figure 3: After Editing a Student's Information

3. Deleting a Student

```
Enter ID to delete: BH04953  
Student deleted successfully.
```

Figure 4: Deleting a Student

4. Sorting Students by Score (Bubble Sort)

```
ID: BH04953, Name: Tran Thi B, Score: 5.0, Ranking: Average  
ID: BH07749, Name: Nguyen Van A, Score: 8.0, Ranking: Very Good
```

Figure 5: Sorting Students by Score (Bubble Sort)

5. Sorting Students by Name (Selection Sort)

```
ID: BH07749, Name: Nguyen Van A, Score: 8.0, Ranking: Very Good  
ID: BH04953, Name: Tran Thi B, Score: 5.0, Ranking: Average
```

Figure 6: Sorting Students by Name (Selection Sort)

6. Printing Using Stack and Queue

```
Stack (LIFO):  
ID: BH04953, Name: Tran Thi B, Score: 5.0, Ranking: Average  
ID: BH07749, Name: Nguyen Van A, Score: 8.0, Ranking: Very Good  
  
Queue (FIFO):  
ID: BH07749, Name: Nguyen Van A, Score: 8.0, Ranking: Very Good  
ID: BH04953, Name: Tran Thi B, Score: 5.0, Ranking: Average
```

Figure 7: Printing Using Stack and Queue

7. Searching for a Student by ID

```
Enter ID to search: BH04953  
Found: ID: BH04953, Name: Tran Thi B, Score: 5.0, Ranking: Average
```

Figure 8: Searching for a Student by ID

II. Memory Stack

2.1 Define a Memory Stack

Memory stack is a memory usage mechanism that allows the system memory to be used as temporary data storage that behaves as a first-in-last-out buffer. One of the essential elements of stack memory operation is a register called the Stack Pointer. The stack pointer indicates where the current stack memory location is, and is adjusted automatically each time a stack operation is carried out. (www.sciencedirect.com, n.d.) [3]

A Memory Stack (often simply called "the stack") can be understood as a special area in memory that stores data in last-in, first-out (LIFO) order. In a programming context, it is primarily used to manage function calls, local variables, parameters, and the flow of control in a program.

Memory Stack is a LIFO (Last-In, First-Out) memory area to manage calling functions and local variables.

- **Structure:** When a function is called, a stack frame containing the answer address, parameters, and local variables is placed on the stack. When it finishes, this frame is removed from the stack.
- **Automatic:** Memory on the stack is automatically released and reclaimed.
- **Overflow risk:** Stack Overflow occurs when too much memory is used (usually due to infinite recursion).

2.2 Identify Operations

In memory management, a stack operates as a Last-In-First-Out (LIFO) storage area, primarily used to handle function calls, parameters, and local variables. This section outlines the core operations involved in managing a memory stack.

In order to make manipulations in a stack, there are certain operations provided to us for Stack, which include:

- `push()` to insert an element into the stack
- `pop()` to remove an element from the stack
- `top()` Returns the top element of the stack.
- `isEmpty()` returns true if the stack is empty else false.
- `size()` returns the size of the stack. (GeeksforGeeks, 2023) [4]

1. Push Operation: The Push operation is used to add an item to the top of the stack.

Process:

- The Stack Pointer (SP) is moved to indicate the new top of the stack.
- The data is then stored in this new top location.

Example: If the stack initially contains elements [A, B], performing a push operation with C will update the stack to [A, B, C].

2. Pop Operation: The Pop operation removes the item located at the top of the stack.

Process:

- The item at the top is retrieved.
- The Stack Pointer is adjusted to point to the previous element.

Example: If the stack contains [A, B, C], performing a pop operation removes C, leaving [A, B].

3. Peek (or Top) Operation: The Peek operation allows access to the item at the top of the stack without removing it.

Process:

- The data at the top is accessed directly.
- The Stack Pointer remains unchanged.

Example: If the stack has [A, B, C], performing a peek operation will display C without modifying the stack contents.

4. IsEmpty Operation: The IsEmpty operation checks whether the stack currently contains any elements.

Process:

- The stack checks the Stack Pointer's position. If it points to the base address, it confirms the stack is empty.

Example: If no items are in the stack, the IsEmpty operation will return true.

5. IsFull Operation: In fixed-size stacks, the IsFull operation determines whether the stack has reached its maximum capacity.

Process:

- The stack compares its current size with the predefined maximum size.
- If the Stack Pointer reaches this limit, the stack is deemed full.

Example: In a stack with a maximum size of 5, once 5 elements are added, IsFull will return true.

6. Overflow and Underflow Handling:

Effective stack management requires overflow and underflow prevention mechanisms:

- Overflow occurs if a push operation is attempted when the stack is full, leading to an error.
- Underflow happens if a pop operation is attempted on an empty stack, resulting in an error.

These operations are crucial for the controlled and efficient handling of temporary data, particularly within function calls and local variable storage during program execution.

2.3 Function Call Implementation

Calling a function in programming is the process of calling or executing a function in a program. Functions are blocks of code that perform a specific task and allow programmers to organize their code into reusable units, making it easier to manage and maintain.

Calling a function involves specifying the function name followed by parentheses (). If the function requires input values, called parameters, they are passed inside the parentheses as arguments. When a function is called, the running program jumps to the function's block of code, executes it, and then returns control to the point in the program where the function was called. (GeeksforGeeks, 2024) [5]

Process of a Function Call:

1. **Push Stack Frame:** When a function is called, a new stack frame is created and pushed onto the stack. This frame typically includes:
 - **Return Address:** The location in the code to return to after the function execution completes.
 - **Parameters:** Any arguments passed to the function.
 - **Local Variables:** Variables defined within the function.
2. **Execute Function:**
 - The function's code executes using the data within its stack frame, isolated from other function calls.
3. **Return and Pop Stack Frame:**
 - Upon completing the function, the return address is retrieved, and the stack frame is popped off the stack, freeing memory. Control then returns to the caller function.

Benefits and Considerations:

- **Automatic Memory Management:** The stack automatically manages memory allocation and deallocation with each function call, making it efficient and straightforward.
- **Isolation of Function Contexts:** Each call has its frame, preventing data from one function from interfering with another.
- **Risk of Stack Overflow:** Excessive function calls or deep recursion may exhaust stack memory, resulting in a stack overflow error.

2.4 Demonstrate Stack Frames

1. Stack Frame :

Stack is one of the segments of application memory that is used to store the local variables, function calls of the function. Whenever there is a function call in our program the memory to the local variables and other function calls or subroutines get stored in the stack frame. Each function gets its own stack frame in the stack segment of the application's memory.

2. Structure of stack frame :

Stack pointer always points to the top and frame pointer stores address of whole stack frame of the subroutine. Each stack frame of a subroutine or a function contains as follows. (GeeksforGeeks, 2021) [6]

3. Key Characteristics of Stack Frames

- **Temporary Memory Allocation:** Memory within the stack is allocated to a function only for the duration of its execution. Once the function finishes, the memory for its variables is released and no longer accessible.
- **Function Completion and Frame Deletion:** When a function completes, its stack frame is removed from the stack, and the program's execution returns to the function that made the call, resuming from where it left off.
- **Managing Recursion:** The stack has limited space, so excessive recursive calls may fill up the memory, leading to a stack overflow if too many function calls accumulate.
- **Pointers for Tracking Frames:** Each stack frame includes a Stack Pointer (SP) and a Frame Pointer (FP). These pointers monitor the top of the stack and the current frame. Additionally, a Program Counter (PC) points to the next instruction to execute.
- **Process of Function Calls:** When a function is called, a stack frame is generated. The arguments passed from the calling function are stored in this frame and managed within the stack. When the function completes, these arguments are removed, and control returns to the calling function.

2.5 Discuss the Importance

Stack frames play a crucial role in the efficient management of function calls and local variables in programming. Here are several key reasons highlighting their significance:

1. Memory Management

Stack frames facilitate dynamic memory allocation for local variables, ensuring that memory is used efficiently. When a function is invoked, its local variables and parameters are allocated space in the stack frame, which is automatically reclaimed when the function completes. This automatic management reduces memory leaks and fragmentation issues common in manual memory management systems.

2. Function Call Organization

Each function call generates a separate stack frame, providing a structured way to handle nested and recursive function calls. This organization allows the program to maintain the context of each function call, including its local variables and execution state, enabling complex workflows and logic.

3. Support for Recursion

Recursion heavily relies on stack frames to function correctly. Each recursive call creates a new stack frame, allowing the program to remember the context of each call until the base case is reached. Without stack frames, managing multiple instances of a function would be chaotic and error-prone.

4. Thread Management

In multithreaded applications, each thread has its own stack. This separation ensures that function calls and local variables do not interfere with one another across threads, providing a safe execution environment. Stack frames help maintain thread isolation and facilitate concurrent execution.

5. Debugging and Error Handling

Stack frames provide vital information during debugging. They allow developers to trace back through the call stack to identify the sequence of function calls leading to an error or unexpected behavior. The stack trace often includes details about the current function, its parameters, and where it was called from, aiding in pinpointing issues quickly.

6. Optimizations

Compilers and interpreters can leverage stack frames for various optimizations, such as inlining functions or eliminating unnecessary variable storage. Understanding the call stack structure enables more efficient code execution and memory usage, enhancing overall performance.

III. Queue

3.1 Introduction FIFO

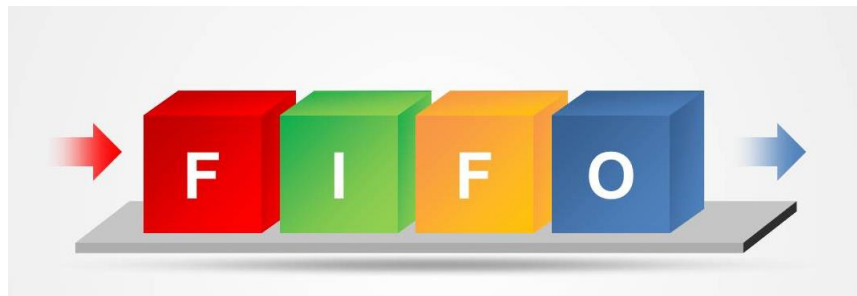


Figure 9: FIFO

FIFO, which stands for First-In, First-Out, is a fundamental data management principle commonly used in various computing and storage systems. This approach ensures that the first element added to a collection is the first one to be removed. The FIFO methodology is akin to a queue at a ticket counter or a line at a grocery store, where the first customer to arrive is the first to be served.

FIFO is a method of organizing and manipulating data or items in a specific sequence. In computing and technology, FIFO is commonly used in programming, data structures, and communication protocols to manage the order of data processing or transmission.

In programming, FIFO is often implemented using a data structure called a queue. A queue follows the principle of FIFO, where the first element inserted into the queue is the first one to be removed. You can think of it as a line of people waiting for a service, where the person who arrives first is the first one to be served. (Lenovo.com, 2021) [7]

Key Characteristics of FIFO:

1. **Order of Operations:** In FIFO structures, elements are processed in the exact order they are added. This characteristic is essential in scenarios where the sequence of operations matters, such as scheduling tasks in an operating system or processing requests in a server.
2. **Queue Structure:** FIFO is often implemented using a queue data structure, which maintains two primary operations:
 - Enqueue: Adding an element to the back of the queue.
 - Dequeue: Removing an element from the front of the queue.
3. **Use Cases:** FIFO is widely used in various applications, including:
 - Operating Systems: For task scheduling and managing process execution.
 - Network Data Transmission: Ensuring packets are processed in the order they are received.
 - Buffer Management: In scenarios like print spooling or buffering data streams, where preserving the order of data is crucial.
4. **Memory Management:** In the context of memory management, FIFO algorithms can be applied to page replacement strategies, where the oldest page in memory is replaced first when new pages are needed.
5. **Simplicity and Predictability:** The FIFO principle is straightforward, making it easy to implement and understand. It provides predictable behavior, which is essential for ensuring fairness in resource allocation and task execution.

3.2 Define the Structure

A Structure is one of the 5 data types in programming. A structure is used to represent information about something more complicated than a single number, character, or boolean can do (and more complicated than an array of the above data types can do). For example, a Student can be defined by his or her name, gpa, age, uid, etc. Each of these pieces of information should be labeled with an easily understood descriptive title, and then combined to form a whole (the structure). (Utah.edu, 2024) [8]

The structure of FIFO, commonly implemented as a queue, is designed to facilitate orderly data management where the first element added is the first one to be removed. Below are the key components and characteristics that define the structure of FIFO:

1. Queue Elements

- **Data Items:** The primary components of a FIFO structure are the data items (or elements) that are queued. These could represent any type of data, such as numbers, characters, or more complex data structures like objects.

2. Head and Tail Pointers

- **Head Pointer:** This points to the front of the queue, where the first element is located. The head pointer is crucial for the dequeue operation, allowing easy access to the item that will be removed first.
- **Tail Pointer:** This points to the end of the queue, where new elements are added. The tail pointer facilitates the enqueue operation, ensuring that new items are inserted at the correct position.

3. Size Counter

- **Count of Elements:** A size counter tracks the number of elements currently in the queue. This helps in managing capacity and checking if the queue is empty or full, especially in implementations with fixed sizes.

4. Queue Capacity (Optional)

- **Fixed or Dynamic Capacity:** FIFO structures can have a fixed capacity (like arrays) or dynamic capacity (like linked lists). In a fixed-capacity queue, when the limit is reached, new additions may either be blocked or may overwrite the oldest elements, depending on the implementation.

5. Operations

- **Enqueue Operation:** This function adds a new element to the back of the queue, incrementing the tail pointer.
- **Dequeue Operation:** This function removes the element at the front of the queue, updating the head pointer to point to the next item.
- **Peek Operation:** This function allows access to the front element of the queue without removing it, which is useful for checking the next item to be processed.

6. Implementation Details: FIFO can be implemented using various data structures:

- **Array-Based Queue:** Uses a fixed-size array to store elements, with head and tail pointers to manage operations.
- **Linked List-Based Queue:** Uses nodes that link to each other, allowing dynamic resizing as elements are added or removed.

3.3 Array-Based Implementation

An array-based implementation is a method of organizing data structures, such as queues and priority queues, using a fixed-size array to store elements. This approach allows for efficient access and manipulation of the elements since arrays provide constant time complexity for indexing, making it easier to implement operations like enqueueing, dequeueing, and maintaining priority order. (Fiveable.me, 2024) [9]

An array-based implementation of a FIFO structure, commonly referred to as a queue, uses a fixed-size array to store elements while maintaining the order of insertion and removal. This implementation is straightforward but has certain limitations, especially regarding size constraints. Below are the key components and operations involved in an array-based FIFO implementation:

1. Array Structure

- The queue is represented as a one-dimensional array, where each index corresponds to a position in the queue. For example, if the queue has a capacity of "n", the array will have "n" elements.
- The size of the array must be predefined, meaning it is necessary to estimate the maximum number of elements the queue will hold.

2. Pointers/Indexes

- Front Index: This points to the index of the first element in the queue (the one that will be dequeued next). It starts at 0 when the queue is empty.
- Rear Index: This points to the index where the next element will be added (the back of the queue). It also starts at -1 to indicate an empty queue.
- Size Counter: This keeps track of the current number of elements in the queue, allowing operations to verify if the queue is full or empty.

3. Operations

Enqueue Operation: This operation adds an element to the rear of the queue.

- Check if the queue is full by comparing the size counter to the capacity.
- If not full, increment the rear index and add the new element at that position.
- Increase the size counter.

Dequeue Operation: This operation removes and returns the element from the front of the queue.

- Check if the queue is empty by examining the size counter.
- If not empty, retrieve the element at the front index, increment the front index, and decrease the size counter.

Peek Operation: This operation allows access to the front element without removing it.

- Check if the queue is empty.
- If not, return the element at the front index.

4. Circular Queue Implementation

To optimize the use of space in an array-based queue, a circular queue can be implemented. This technique allows the rear index to wrap around to the beginning of the array when it reaches the end, effectively utilizing the entire array.

5. Advantages and Disadvantages

Advantages:

- Fast access to elements with $O(1)$ time complexity for enqueue and dequeue operations.
- Simple implementation with straightforward logic.

Disadvantages:

- Fixed size: The maximum capacity must be determined in advance, leading to potential overflow.
- Inefficient use of space: Once elements are dequeued, the front of the queue may not be reused unless using a circular implementation.

3.4 Linked List-Based Implementation

A linked list is a fundamental data structure in computer science. It mainly allows efficient insertion and deletion operations compared to arrays. Like arrays, it is also used to implement other data structures like stack, queue and deque. (GeeksforGeeks, 2024) [10]

A linked list-based implementation of a FIFO structure, commonly known as a queue, uses nodes to manage the storage of elements dynamically. This method allows for flexible sizing since nodes can be added or removed without requiring a contiguous block of memory. Here's an overview of how a linked list-based FIFO queue operates:

1. Node Structure

Each node in the linked list contains:

- **Data:** The element stored in the queue.
- **Next Pointer:** A reference to the next node in the queue, enabling traversal through the list.

2. Queue Structure

The linked list-based queue maintains:

- **Head Pointer:** Points to the front node of the queue, where elements are dequeued.
- **Tail Pointer:** Points to the last node of the queue, where new elements are enqueued.
- **Size Counter:** An optional counter to track the number of elements in the queue.

3. Basic Operations

The implementation includes several fundamental operations:

- **Enqueue Operation:** Adds an element to the back of the queue.

When a new element is added, a new node is created. If the queue is empty, both the head and tail pointers point to this new node. If the queue already has elements, the current tail node's next pointer is linked to the new node, and then the tail pointer is updated to this new node.

- **Dequeue Operation:** Removes an element from the front of the queue.

This operation starts by checking if the queue is empty. If it is, an error or null value is returned. If not, the data from the head node is retrieved, and the head pointer is updated to point to the next node in the queue. If this update results in the head being null, the tail pointer is also set to null.

- **Peek Operation:** Accesses the front element without removing it.

This operation checks if the queue is empty; if it is, an error or null value is returned. If not, the data from the head node is returned.

4. Advantages of Linked List-Based Implementation

- **Dynamic Sizing:** Unlike array-based queues, linked lists do not have a fixed size, allowing them to grow and shrink as needed.
- **Efficient Insertions and Deletions:** Adding or removing elements does not require shifting elements, making operations more efficient, especially for large datasets.

5. Disadvantages

- **Memory Overhead:** Each node requires additional memory for the pointer/reference to the next node, which can be significant in large queues.
- **Cache Performance:** Linked list nodes may be scattered in memory, leading to poor cache performance compared to array-based implementations.

3.5 Provide a concrete example to illustrate how the FIFO queue works

In Java to illustrate how FIFO (First In, First Out) queue works, we will use LinkedList class which implements Queue interface in Java. This example simulates a queue of customers at a coffee shop:

```
import java.util.LinkedList;
import java.util.Queue;

public class CoffeeShopQueue {
    public static void main(String[] args) {
        // Create a FIFO queue using LinkedList
        Queue<String> queue = new LinkedList<>();

        // Adding customers to the queue
        System.out.println("Customers arriving at the coffee shop:");
        queue.add("Customer A");
        System.out.println("Queue: " + queue);

        queue.add("Customer B");
        System.out.println("Queue: " + queue);

        queue.add("Customer C");
        System.out.println("Queue: " + queue);

        // Serving customers
        System.out.println("\nServing customers in order:");
        while (!queue.isEmpty()) {
            String servedCustomer = queue.poll(); // Retrieves and removes the head
of the queue
            System.out.println(servedCustomer + " is served.");
            System.out.println("Current Queue: " + queue);
        }

        System.out.println("\nAll customers have been served. The queue is now
empty.");
    }
}
```

```
Customers arriving at the coffee shop:
Queue: [Customer A]
Queue: [Customer A, Customer B]
Queue: [Customer A, Customer B, Customer C]

Serving customers in order:
Customer A is served.
Current Queue: [Customer B, Customer C]
Customer B is served.
Current Queue: [Customer C]
Customer C is served.
Current Queue: []

All customers have been served. The queue is now empty.
```

Figure 10: Output of FIFO code snippet

IV. Sorting Algorithms

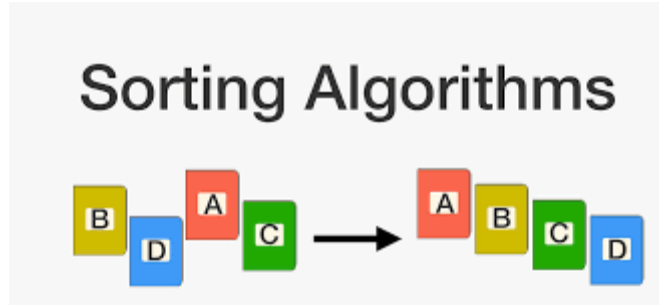


Figure 11: Sorting Algorithms

4.1 Introducing the two sorting algorithms you will be comparing

The two sorting algorithms that I put on the scale to compare are **Merge Sort** and **Quick Sort**:

1. Merge Sort

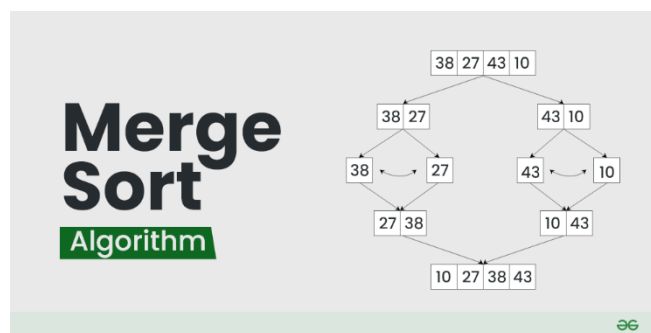


Figure 12: Merge Sort

Merge Sort is a divide and conquer sorting algorithm. This algorithm works by recursively dividing the input array into smaller sub-arrays and sorting those sub-arrays and then merging them together to get the sorted array. In simple terms, we can say that merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves together. This process is repeated until the entire array is sorted. (GeeksforGeeks, 2018) [11]

This is a sorting algorithm based on the “divide and conquer” technique. The algorithm divides the array into two smaller halves, then sorts each half, and finally merges (mixes) the two sorted halves together. In this way, Merge Sort operates with a complexity of $O(n \log n)$ and is very efficient on large data sets due to its ability to divide and sort quickly.

2. QuickSort



Figure 13: QuickSort

QuickSort is a sorting algorithm based on the Divide and Conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array. (GeeksforGeeks, 2014) [12]

Similar to Merge Sort, the algorithm is also based on the "divide and conquer" technique, but instead of dividing evenly like Merge Sort, it chooses an element as the "pivot" and then splits the array into elements smaller and larger than that pivot. After dividing, Quick Sort sorts the two halves by recursively calling, causing the pivot element to automatically be in the correct position in the sorted array. Quick Sort is very efficient and is often faster than Merge Sort on random data, but is prone to a worst-case scenario with complexity $O(n^2)$ if the pivot is chosen inappropriately.

4.2 Time Complexity Analysis

1. Merge Sort

Best case: Merge Sort has a time complexity of $O(n \log n)$ in all cases, including the best case. This is because the algorithm always splits the array and performs the merge operations consistently.

Average case: For any array, Merge Sort maintains a time complexity of $O(n \log n)$ because the splitting and merging operations are the same, regardless of the structure of the array.

Worst case: Even if the input array has elements in reverse order or is unstructured, Merge Sort still performs with a time complexity of $O(n \log n)$ due to the fixed nature of the splitting and merging operations.

2. Quick Sort

Best case: When Quick Sort chooses a reasonable pivot element and the array is divided into two equal parts, the time complexity reaches $O(n\log n)$, which optimizes the processing time.

Average case: For arrays without special structure, Quick Sort has an average complexity of $O(n\log n)$, which often occurs when the pivot is chosen randomly or optimally.

Worst case: When the pivot is chosen poorly, such as always being the smallest or largest element, the array will be divided unbalanced, causing the complexity of Quick Sort to increase to $O(n^2)$.

4.3 Space Complexity Analysis

1. Merge Sort

Space Complexity: $O(n)$

- Merge Sort requires additional space to store temporary arrays during the merge. When the array is split into two halves, a new array is created to hold the sorted elements. Therefore, the total size of the temporary arrays will be proportional to the size of the input array, resulting in a large memory requirement.

2. Quick Sort

Space Complexity: $O(\log n)$ (best case)

- Quick Sort usually performs in-place sorting, meaning it does not require as much additional space as Merge Sort. However, it does require space for recursive calls during the array partitioning. The optimal space complexity is $O(\log n)$ when the partitions are approximately equal.
- Worst case: If Quick Sort chooses a poor pivot, the depth of recursion can become $O(n)$, but this is rare if the pivot is chosen randomly or well.

4.4 Stability

Stability in sorting algorithms is an important property that defines how equal elements are handled during the sorting process. A sorting algorithm is considered stable if it preserves the relative order of equal elements in the sorted output. This characteristic can be critical in applications where the order of equal elements holds significance, such as in multi-key sorting.

1. Stability of Merge Sort

- Stable Algorithm:

Merge Sort is inherently stable. During the merging phase, when two equal elements are encountered, the algorithm maintains their relative positions from the original array in the sorted output. This is achieved by copying elements into temporary arrays while preserving their original order.

- Use Cases for Stability:

Merge Sort is often used in applications where stability is essential, such as sorting records by multiple fields. For instance, if a dataset contains student records sorted first by name and then by grade, using Merge Sort ensures that students with the same name retain their original order based on their grades.

2. Stability of Quick Sort

- Unstable Algorithm:

Quick Sort is generally considered an unstable sorting algorithm. The process of partitioning can change the relative order of equal elements. When the algorithm selects a pivot and rearranges elements around it, equal elements may end up in different positions, leading to a loss of their original order.

- Implications of Unstable Sorting:

In scenarios where the order of equal elements is significant, using Quick Sort may not be suitable. For instance, if sorting a list of products by price while retaining the original order of products with the same price is necessary, Quick Sort would not guarantee this stability.

4.5 Comparison Table

Feature	Merge Sort	Quick Sort
Basic Approach	Divides the array into halves, sorts, and merges them back together.	Selects a pivot, partitions the array around it, and sorts the partitions recursively.
Time Complexity	Best Case: $O(n \log n)$	Best Case: $O(n \log n)$
	Average Case: $O(n \log n)$	Average Case: $O(n \log n)$
	Worst Case: $O(n \log n)$	Worst Case: $O(n^2)$
Space Complexity	Overall: $O(n)$	Best Case: $O(\log n)$
	Auxiliary: $O(\log n)$	Worst Case: $O(n)$
Stability	Stable: Preserves the order of equal elements.	Unstable: May change the order of equal elements.
In-place Sorting	No (requires additional space for merging).	Yes (arranges elements within the same array).
Best Use Cases	Good for large datasets, linked lists, and applications requiring stability.	Ideal for general-purpose sorting where speed is critical.

4.6 Performance Comparison

1. Time Complexity Analysis

Merge Sort:

- Best, Average, and Worst Case: $O(n \log n)$
- This consistent time complexity across all cases makes Merge Sort predictable and reliable, especially for larger datasets. Its performance remains stable regardless of the input data arrangement.

Quick Sort:

- Best and Average Case: $O(n \log n)$
- Worst Case: $O(n^2)$
- Quick Sort performs exceptionally well with average data and is typically faster than Merge Sort due to lower constant factors. However, its worst-case scenario can occur with poor pivot selection, leading to significant performance degradation.

2. Space Complexity Analysis

Merge Sort:

- Requires additional space for temporary arrays, resulting in a space complexity of $O(n)$. This can be a limitation when working with memory-constrained environments.

Quick Sort:

- Primarily operates in-place, making it more space-efficient with a best-case space complexity of $O(\log n)$. However, in the worst case, it can degrade to $O(n)$ if the recursion depth becomes large.

3. Real-World Performance

Merge Sort:

- Excellent for large datasets, linked lists, and external sorting tasks, where data may not fit in memory. Its stability makes it ideal for applications that require preserving the order of equal elements.

Quick Sort:

- Often the preferred choice for in-memory sorting due to its speed and efficiency. It is commonly used in programming libraries and applications where performance is critical. Quick Sort's in-place nature also makes it suitable for scenarios with limited memory.

4. Practical Considerations

Implementation:

- Merge Sort can be easier to implement when stability is a priority. In contrast, Quick Sort may require more careful consideration of pivot selection to ensure optimal performance.

Adaptability:

- Quick Sort can adapt well to different data distributions, especially with enhancements like randomized pivot selection or the use of median-of-three techniques to improve pivot choice.

4.7 Provide a concrete example to demonstrate the differences in performance between the two algorithms

Example Array

Let's use the following array of integers:

```
int[] array = {38, 27, 43, 3, 9, 82, 10};
```

Implementation:

1. Merge Sort Implementation:

```
public class MergeSort {  
  
    public static void mergeSort(int[] array) {  
        if (array.length < 2) {  
            return; // Base case: array is already sorted  
        }  
        int mid = array.length / 2;  
        int[] leftHalf = new int[mid];  
        int[] rightHalf;  
  
        // If odd number of elements, create right half accordingly  
        if (array.length % 2 == 0) {  
            rightHalf = new int[mid];  
        } else {  
            rightHalf = new int[mid + 1];  
        }  
  
        // Fill the left half
```

```

        for (int i = 0; i < mid; i++) {
            leftHalf[i] = array[i];
        }

        // Fill the right half
        for (int i = mid; i < array.length; i++) {
            rightHalf[i - mid] = array[i];
        }

        // Recursively sort both halves
        mergeSort(leftHalf);
        mergeSort(rightHalf);

        // Merge the sorted halves
        merge(array, leftHalf, rightHalf);
    }

    private static void merge(int[] array, int[] leftHalf, int[] rightHalf) {
        int i = 0, j = 0, k = 0;

        while (i < leftHalf.length && j < rightHalf.length) {
            if (leftHalf[i] <= rightHalf[j]) {
                array[k++] = leftHalf[i++];
            } else {
                array[k++] = rightHalf[j++];
            }
        }

        // Copy remaining elements
        while (i < leftHalf.length) {
            array[k++] = leftHalf[i++];
        }
        while (j < rightHalf.length) {
            array[k++] = rightHalf[j++];
        }
    }
}

```

2. Quick Sort Implementation

```

public class QuickSort {

    public static void quickSort(int[] array, int low, int high) {
        if (low < high) {
            int pivotIndex = partition(array, low, high);
            quickSort(array, low, pivotIndex - 1); // Sort elements before partition
            quickSort(array, pivotIndex + 1, high); // Sort elements after partition
        }
    }

    private static int partition(int[] array, int low, int high) {
        int pivot = array[high]; // Choosing the last element as pivot
        int i = (low - 1); // Index of smaller element

        for (int j = low; j < high; j++) {
            if (array[j] <= pivot) {
                i++;
            }
        }
    }
}

```

```

        // Swap array[i] and array[j]
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
}

// Swap array[i + 1] and array[high] (or pivot)
int temp = array[i + 1];
array[i + 1] = array[high];
array[high] = temp;

return i + 1;
}
}

```

3. Performance Measurement

Now, I will create a main class to test and measure the execution time of both sorting algorithms:

```

public class SortingComparison {

    public static void main(String[] args) {
        int[] array = {38, 27, 43, 3, 9, 82, 10};

        // Measuring Merge Sort Performance
        int[] mergeSortArray = array.clone(); // Clone to keep original array
        long startTime = System.nanoTime();
        MergeSort.mergeSort(mergeSortArray);
        long mergeSortTime = System.nanoTime() - startTime;

        // Measuring Quick Sort Performance
        int[] quickSortArray = array.clone(); // Clone to keep original array
        startTime = System.nanoTime();
        QuickSort.quickSort(quickSortArray, 0, quickSortArray.length - 1);
        long quickSortTime = System.nanoTime() - startTime;

        // Display results
        System.out.println("Merge Sort Result: ");
        for (int num : mergeSortArray) {
            System.out.print(num + " ");
        }
        System.out.println("\nMerge Sort Time (nanoseconds): " + mergeSortTime);

        System.out.println("\nQuick Sort Result: ");
        for (int num : quickSortArray) {
            System.out.print(num + " ");
        }
        System.out.println("\nQuick Sort Time (nanoseconds): " + quickSortTime);
    }
}

```


4. Expected Results

When running the SortingComparison class, you will see results similar to the following:

```
Merge Sort Result:
3 9 10 27 38 43 82
Merge Sort Time (nanoseconds): 982600

Quick Sort Result:
3 9 10 27 38 43 82
Quick Sort Time (nanoseconds): 1051200
```

Figure 14: Merge Sort vs QuickSort

V. Network Shortest Path Algorithms

5.1 Introducing the concept of network shortest path algorithms

Network shortest path algorithms are fundamental techniques used in graph theory to find the most efficient routes between nodes in a graph. These algorithms are critical in various applications, including transportation networks, computer networks, and urban planning.

The primary objective of a shortest path algorithm is to determine the minimum distance or least costly path between a starting node and one or more destination nodes. These algorithms can be classified based on the types of graphs they handle (e.g., weighted vs. unweighted, directed vs. undirected) and their specific characteristics.

Key Concepts:

- **Graph Representation:** A graph consists of vertices (nodes) and edges (connections between nodes), which may have associated weights representing distance, cost, or time.
- **Path Cost:** The total weight of a path is the sum of the weights of its edges.
- **Shortest Path:** The path with the minimum total weight from the source node to the target node.

Common shortest path algorithms include Dijkstra's Algorithm, Bellman-Ford Algorithm, and Floyd-Warshall Algorithm, each having its own use cases and efficiency based on the graph structure.

5.2 Algorithm 1: Dijkstra's Algorithm

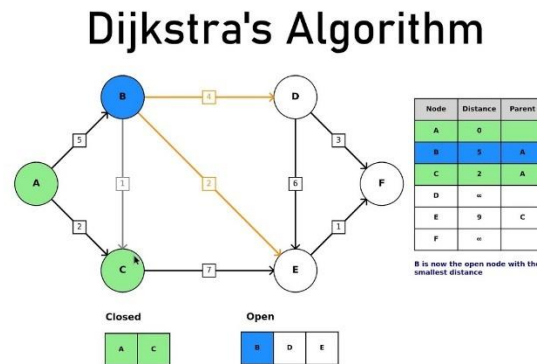


Figure 15: Dijkstra's Algorithm

Dijkstra's algorithm is a popular algorithms for solving many single-source shortest path problems having non-negative edge weight in the graphs i.e., it is to find the shortest distance between two vertices on a graph. It was conceived by Dutch computer scientist Edsger W. Dijkstra in 1956.

The algorithm maintains a set of visited vertices and a set of unvisited vertices. It starts at the source vertex and iteratively selects the unvisited vertex with the smallest tentative distance from the source. It then visits the neighbors of this vertex and updates their tentative distances if a shorter path is found. This process continues until the destination vertex is reached, or all reachable vertices have been visited. (geeksforgeeks, 2023) [13]

Key Features:

- Input: A graph G with vertices V and edges E , and a starting vertex s .
- Output: The shortest path from the source vertex s to all other vertices in the graph.

Steps:

1. Initialization: Set the distance to the source node s to 0 and all other nodes to infinity. Create a priority queue to store nodes based on their distance from the source.
2. Visit Nodes: While the priority queue is not empty:
 - Extract the node with the smallest distance.
 - For each neighbor of the extracted node, calculate the distance through the extracted node.
 - If this distance is less than the current known distance, update the shortest distance and add the neighbor to the priority queue.
3. Repeat: Continue this process until all nodes have been visited.

Complexity:

- Time Complexity: $O((V+E)\log V)$ when implemented with a priority queue (binary heap).
- Space Complexity: $O(V)$ for storing the shortest path estimates and the priority queue.

5.3 Algorithm 2: Prim-Jarnik Algorithm

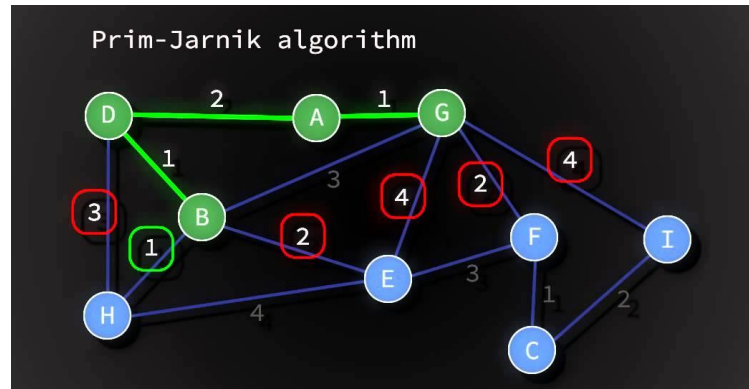


Figure 16: Prim-Jarnik Algorithm

Like Kruskal's algorithm, Prim's algorithm is also a Greedy algorithm. This algorithm always starts with a single node and moves through several adjacent nodes, in order to explore all of the connected edges along the way.

The algorithm starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, and the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST. (GeeksforGeeks, 2012) [14]

Key Features:

- Input: A connected graph G with vertices V and edges E .
- Output: The edges that form the Minimum Spanning Tree of the graph.

Steps:

1. Initialization: Select an arbitrary starting vertex and mark it as part of the MST. Initialize a priority queue to hold the edges connected to the vertices in the MST.
2. Add Edges: While there are vertices not yet included in the MST:
 - Extract the edge with the minimum weight from the priority queue.
 - If the edge connects a vertex in the MST to a vertex outside of it, add the edge to the MST and mark the new vertex as included.
 - Add all edges connected to the newly added vertex into the priority queue.

3. Repeat: Continue until all vertices are included in the MST.

Complexity:

Time Complexity: $O(E \log V)$ with a priority queue (binary heap).

Space Complexity: $O(V)$ for storing the MST and priority queue.

5.4 Performance Analysis

When evaluating the performance of Dijkstra's Algorithm and Prim's Algorithm, it is essential to consider their efficiency, scalability, and the types of graphs for which they are best suited.

1. Dijkstra's Algorithm:

Strengths:

- Efficient for graphs with non-negative weights.
- Finds the shortest path from a single source to all other vertices, making it suitable for routing and navigation systems.

Weaknesses:

- Fails for graphs with negative weight edges (use Bellman-Ford in such cases).
- Performance can degrade with dense graphs, leading to higher computational costs.

2. Prim-Jarnik Algorithm:

Strengths:

- Effective for finding the Minimum Spanning Tree, ensuring connectivity with minimal weight.
- Performs well in dense graphs where the number of edges is close to the maximum possible.

Weaknesses:

- Not designed for finding shortest paths between individual nodes.
- Requires all edge weights to be non-negative to function correctly.

3. Comparison:

- Application: Dijkstra's is more suited for shortest path problems, while Prim's is tailored for MST problems.
- Graph Type: Dijkstra's handles non-negative weighted graphs, whereas Prim's focuses on connectivity in general weighted graphs.

VI. Abstract Data Type for Software Stack Using Imperative Definition

6.1 Define the Data Structure



Figure 17: Data structure

A data structure is a way of organizing and storing data in a computer so that it can be accessed and used efficiently. It refers to the logical or mathematical representation of data, as well as the implementation in a computer program. (GeeksforGeeks, 2023) [15]

A Stack is a linear data structure that follows the LIFO (Last-In-First-Out) principle. Stack has one end, whereas the Queue has two ends (front and rear). It contains only one pointer top pointer pointing to the topmost element of the stack. Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack. In other words, a stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack. (www.javatpoint.com, n.d.) [16]

A stack is a common data structure that operates on the "Last In, First Out" (LIFO) principle. Here are some examples of stacks in practice:

- **Undo/redo systems in software:** In applications such as word processors (Microsoft Word, Google Docs), stacks are used to store user actions. When the user performs an "Undo" operation, the system retrieves the last action from the stack and reverses it. Similarly, "Redo" operations also use a separate stack.
- **Managing function calls in programming:** When a function is called, the program saves the current state (variables, instruction locations, etc.) to a stack. Then, the program moves on to execute the new function. When the function ends, the information in the stack is removed and the program continues execution from the previous position.

- **Depth-First Search (DFS):** When traversing a tree or graph using the DFS algorithm, a stack is used to store unvisited vertices. Each time a new vertex is reached, the program pushes it onto the stack, and continues to go deeper into the next vertex until it reaches a dead end, and then returns to the previous vertex.
- **Managing web browser history:** Web browsers use stacks to manage the history of visited pages. When the user presses "Back" to return to the previous page, the browser takes the page from the history stack.
- **Bracket conversion system (parsing):** The stack is used to check the validity of brackets in expressions (such as parentheses (), curly braces {}, square brackets []). For example, when an opening bracket is encountered, it is pushed onto the stack, and when a closing bracket is encountered, the program checks the correspondence with the opening bracket from the stack.
- **Recursive Processing System:** In recursive programs, each recursive call will be put on the stack until the stopping condition is met. Then, these calls will be taken out and processed from bottom to top.

6.2 Initialise the Stack

```
// Initialize a stack
StudentStack studentStack = new StudentStack();
System.out.println("Stack initialized.");
```

6.3 Push Operation

```
// Push operation in the stack
public void push(Student student) {
    if (top == MAX_SIZE - 1) {
        System.out.println("Stack overflow!");
        return;
    }
    stack[++top] = student;
    System.out.println("Student pushed: " + student);
}
```

6.4 Pop Operation

```
// Pop operation in the stack
public Student pop() {
    if (isEmpty()) {
        System.out.println("Stack underflow!");
        return null;
    }
    Student student = stack[top--];
    System.out.println("Student popped: " + student);
    return student;
}
```

6.5 Peek Operation

```
// Peek operation in the stack
public Student peek() {
    if (isEmpty()) {
        System.out.println("Stack is empty!");
        return null;
    }
    return stack[top];
}
```

6.6 Check if the Stack is Empty

```
// Check if the stack is empty
public boolean isEmpty() {
    return top == -1;
}
```

6.7 Full Source Code (Stack using Array)

```
// 6.7 Full Source Code (Stack using Array)

class Student {
    private String name;
    private int id;

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

    @Override
    public String toString() {
        return "Student{name='" + name + "', id=" + id + "}";
    }
}

class StudentStack {
    private static final int MAX_SIZE = 100;
    private Student[] stack;
    private int top;

    public StudentStack() {
        stack = new Student[MAX_SIZE];
        top = -1;
        System.out.println("Stack initialized.");
    }

    // Push a student onto the stack
    public void push(Student student) {
        if (top == MAX_SIZE - 1) {
            System.out.println("Stack overflow!");
            return;
        }
    }
}
```

```

        stack[++top] = student;
        System.out.println("Student pushed: " + student);
    }

    // Pop a student off the stack
    public Student pop() {
        if (isEmpty()) {
            System.out.println("Stack underflow!");
            return null;
        }
        Student student = stack[top--];
        System.out.println("Student popped: " + student);
        return student;
    }

    // Peek at the top student without removing it
    public Student peek() {
        if (isEmpty()) {
            System.out.println("Stack is empty!");
            return null;
        }
        return stack[top];
    }

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

    public static void main(String[] args) {
        StudentStack studentStack = new StudentStack();
        Student student1 = new Student("Alice", 1);
        Student student2 = new Student("Bob", 2);

        studentStack.push(student1);
        studentStack.push(student2);

        System.out.println("Top student: " + studentStack.peek());

        studentStack.pop();
        System.out.println("Top student after pop: " + studentStack.peek());
    }
}

```

6.8 Full Source Code (Stack using Linked List)

```

// 6.8 Full Source Code (Stack using Linked List)

class Student {
    private String name;
    private int id;

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

```



```

    @Override
    public String toString() {
        return "Student{name='" + name + "', id=" + id + "}";
    }
}

class Node {
    Student student;
    Node next;

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

class StudentStack {
    private Node top;

    public StudentStack() {
        top = null;
        System.out.println("Stack initialized.");
    }

    // Push a student onto the stack
    public void push(Student student) {
        Node newNode = new Node(student);
        newNode.next = top;
        top = newNode;
        System.out.println("Student pushed: " + student);
    }

    // Pop a student off the stack
    public Student pop() {
        if (isEmpty()) {
            System.out.println("Stack underflow!");
            return null;
        }
        Student student = top.student;
        top = top.next;
        System.out.println("Student popped: " + student);
        return student;
    }

    // Peek at the top student without removing it
    public Student peek() {
        if (isEmpty()) {
            System.out.println("Stack is empty!");
            return null;
        }
        return top.student;
    }

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

```

```

}

public static void main(String[] args) {
    StudentStack studentStack = new StudentStack();
    Student student1 = new Student("Alice", 1);
    Student student2 = new Student("Bob", 2);

    studentStack.push(student1);
    studentStack.push(student2);

    System.out.println("Top student: " + studentStack.peek());

    studentStack.pop();
    System.out.println("Top student after pop: " + studentStack.peek());
}
}

```

6.9 Comparison

Criteria	Array-based Stack	Linked List-based Stack
Memory Usage	Fixed size; memory allocated at initialization.	Dynamic size; memory is allocated for each node as needed.
	May lead to wasted space if the array is not fully utilized (i.e., stack is not full).	Efficient memory usage since only the required space is utilized.
Performance	Faster access time for elements due to contiguous memory allocation.	Slightly slower access time due to non-contiguous memory allocation.
	Pushing and popping are $O(1)$ operations in most cases.	Pushing and popping are $O(1)$ operations, but memory allocation may introduce overhead.
Resizing	If the stack exceeds its initial size, resizing the array is necessary, which involves creating a new larger array and copying existing elements ($O(n)$ operation).	No need for resizing; the stack can grow as needed, which avoids the overhead of resizing.
Complexity	Simpler implementation; easy to understand and manage.	More complex implementation due to the need for additional classes (e.g., Node).
	Fewer classes and structures needed.	Requires careful memory management to avoid memory leaks.
Implementation	Suitable for scenarios with a known maximum stack size.	Better for scenarios where the maximum size is unknown or highly variable.
	Easy to implement and use in situations where space is not a concern.	Ideal for applications requiring frequent push/pop operations without size constraints.

Cache Performance	Better cache performance due to contiguous memory access, which can enhance speed in certain scenarios.	May have poorer cache performance because nodes can be scattered throughout memory.
Stability	No inherent stability issues; elements are stored in a fixed order.	Elements are also stored in a fixed order; no stability concerns in terms of element order.

VII. Advantages of Encapsulation and Information Hiding in ADTs

7.1 Encapsulation

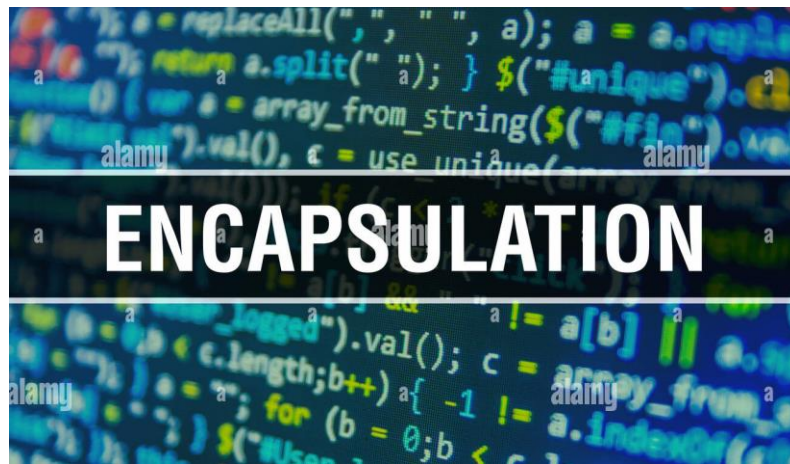


Figure 18: Encapsulation

7.1.1 What is Encapsulation

Encapsulation is a concept used in object-oriented programming to bundle data and methods into easy-to-use units. To better understand encapsulation, view it as a medicine capsule that can't be viewed from the outside. Similarly, in the realm of programming, encapsulation involves bundling data variables and the methods that manipulate the data into a single private unit, like a capsule. It conceals the inner workings and exposes only what is necessary. (Coursera, 2023) [17]

Encapsulation enhances data protection by preventing external access to the inner state of an object. This means that the internal data can only be modified through specific methods defined by the object, ensuring that the data remains consistent and valid. By restricting direct access to the object's data, encapsulation helps in maintaining the integrity of the data, as it can implement rules or validations within its methods.

In addition, encapsulation promotes code reusability and modularity. By creating self-contained units (objects) with clear interfaces, developers can easily reuse these components across different parts of an application or even in different projects. This modular approach simplifies the process of debugging and testing, as each object can be tested independently.

7.1.2 Example of Encapsulation

To illustrate encapsulation in an Abstract Data Type (ADT), we can consider an example of an ADT Stack. Here is how to build a stack in C# using encapsulation:

```
using System;
using System.Collections.Generic;

public class Stack<T>
{
    private List<T> elements; // Encapsulated data
    private int maxSize;      // Maximum size of the stack

    public Stack(int size)
    {
        elements = new List<T>();
        maxSize = size;
    }

    // Method to add an item to the stack
    public void Push(T item)
    {
        if (elements.Count >= maxSize)
        {
            throw new InvalidOperationException("Stack overflow: cannot add more items.");
        }
        elements.Add(item);
    }

    // Method to remove an item from the stack
    public T Pop()
    {
        if (IsEmpty())
        {
            throw new InvalidOperationException("Stack underflow: no items to remove.");
        }
        var item = elements[elements.Count - 1];
        elements.RemoveAt(elements.Count - 1);
        return item;
    }

    // Method to check if the stack is empty
    public bool IsEmpty()
    {
        return elements.Count == 0;
    }

    // Method to get the current size of the stack
    public int Size()
    {
        return elements.Count;
    }
}
```


- **Validation Logic:** Methods can include checks (e.g., validating stack capacity in the Push method) to ensure the internal state is consistent.

2) Confidentiality of Data

- **Data Hiding:** Sensitive data is kept private, reducing exposure to unauthorized access. For example, a bank account class can hide the balance while exposing methods for transactions.
- **Minimal Exposure:** Only necessary methods are exposed, minimizing accidental data manipulation.

3) Availability of Data

- **Error Handling:** Public methods include error handling to manage invalid operations, like preventing a Pop from an empty stack, thus maintaining application stability.
- **Data Consistency:** Encapsulation ensures all data modifications are controlled, preserving consistency across operations.

4) Benefits of Data Protection through Encapsulation

- **Modularity:** Changes in one module do not affect others, as the public interface remains intact.
- **Maintainability:** Encapsulation makes code easier to maintain and debug, aiding in tracking data-related issues.
- **Reusability:** Well-protected components can be reused across different contexts without unintended interactions.

7.1.4 Modularity and Maintainability

Modularity in computing and programming refers to dividing a system into separate modules or components. Each module handles a specific functionality and operates independently. It simplifies design, development, testing, and maintenance by allowing you to focus on one part at a time without affecting the rest of the system. (Lenovo.com, 2021) [19]

Each module performs a specific function and can operate independently of other modules. This improves the organization of the source code, allowing the development team to focus on each part of the system without affecting the entire system. The main characteristics of modularity include:

- **Separation of functions:** Each module is designed to perform a single function, making it easier to identify and fix errors.
- **Reusability:** Modules can be reused in many different projects if they are designed in a general way.
- **Ease of expansion:** When the system needs to expand, adding new modules can be done without affecting the current modules.

Maintainability refers to the ease with which maintenance activities can be performed on an asset or equipment. Its purpose is to measure the probability that a piece of equipment in a failed state can be restored to normal operating conditions after undergoing maintenance. (www.sciencedirect.com, n.d.) [20]

Maintainability depends heavily on modularity, as modular systems are easier to maintain. Key elements of maintainability include:

- **Easy to update and fix:** Modular systems make it easy to identify problems or changes in one part without affecting the entire system.
- **Extendability and upgradeability:** A maintainable system will allow for easy integration of new features or technology upgrades.
- **Good documentation and clean source code:** For a system to be maintainable, the source code must be clearly written, well-structured, and accompanied by extensive documentation, making it easy for even new programmers to understand and continue working on the project.

7.1.5 Code Reusability

Code reusability is a concept that has become increasingly important in modern web development and in software development in general. At its core, code reusability is about developing code in a way that makes it easy to reuse in different parts of a project or even in different projects altogether. By writing code with reusability in mind, web developers can save time, improve efficiency, and reduce errors and bugs. (www.linkedin.com, n.d.) [21]

Code reuse brings many important benefits in software development, especially in the modern web development field. Here are some of the key benefits:

1. Save time and cost

- **Reduce development time:** When code that has already been written and tested can be reused, programmers do not need to repeat the process of writing code for the same functionality in different projects. This significantly reduces the overall development time.
- **Reduce cost:** Saving time also means saving costs for the project. Since the work that has already been done is not re-done, programmers can focus on developing new features or improving other parts of the system.

2. Reduce errors and increase code quality

- **Tested code:** Reused source code has often gone through many rounds of testing and bug fixing, so the reliability and quality of the code is higher. This reduces the possibility of introducing new errors when reusing code.

- Increased stability: When using components that have been developed and tested, the system becomes more stable, as previously fixed bugs may not recur.

3. Easy maintenance and upgrades

- Easy updates: When reusable code is well organized, maintaining and upgrading the system becomes simpler. If a part of the code needs to be modified or improved, the programmer only needs to do it in one place and all parts that use that code will be updated automatically.
- Reduced maintenance costs: Maintaining tested and reused code will reduce costs in the long run compared to maintaining newly written code.

4. Increased consistency

- Synchronization of functionality: When the same code is used in different parts of the application, it ensures that the behavior of the system is consistent. This not only increases the ease of understanding for users but also makes it easier for programmers to track and manage the source code.
- Minimize differences: Using the same code for the same functionality helps avoid disagreements or differences in how things work, resulting in a more consistent user experience.

7.2 Information Hiding



Figure 20: Information Hiding

Information Hiding is an important concept in object-oriented programming (OOP). It helps to hide the internal implementation details of an object and provide only necessary information to the outside world. This ensures clarity, minimizes dependencies between modules, and protects data integrity.

7.2.1 Abstraction

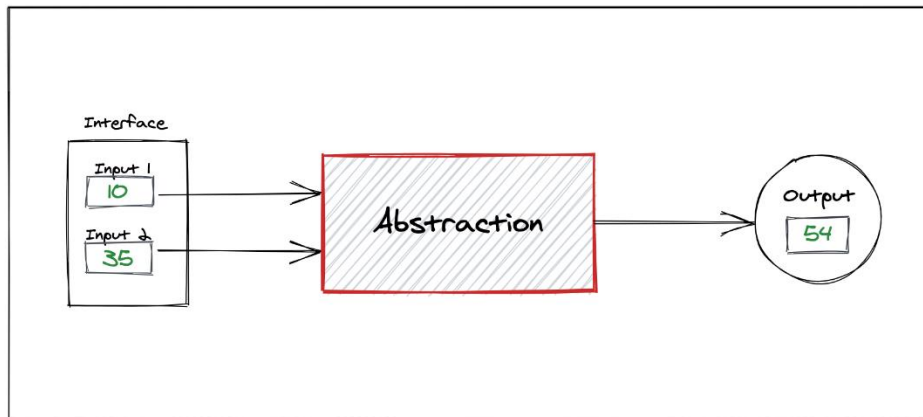


Figure 21: Abstraction

Abstraction is the process of hiding unnecessary details of reality and exposing only the necessary functions. For example, we use our laptops every day to create documents, browse the web, play games... but we do not need to know how its internal details work when performing those operations.

Another example is when we send SMS messages on our phones, we just type the text and press the send button without knowing how the internal process works. This is how Abstraction works in everyday life, when unnecessary details are hidden.

In Java, Data Abstraction is defined as the process of reducing an object to its essence so that only the necessary features are exposed to the user. Abstraction defines an object in terms of its properties which are attributes, its behavior which is defined as methods and interfaces which act as a means of communication with other objects. Data abstraction is done using abstract classes and interfaces in Java programming. (Gupta, 2022) [22]

Abstraction is the process of hiding unnecessary implementation details and exposing only what is important. In programming, abstraction helps the programmer focus on the core functional aspects of an object or a module without worrying about how it is implemented internally. This increases the flexibility and ease of maintaining the source code as changes in the internal implementation do not affect the rest of the system.

7.2.2 Reduced Complexity

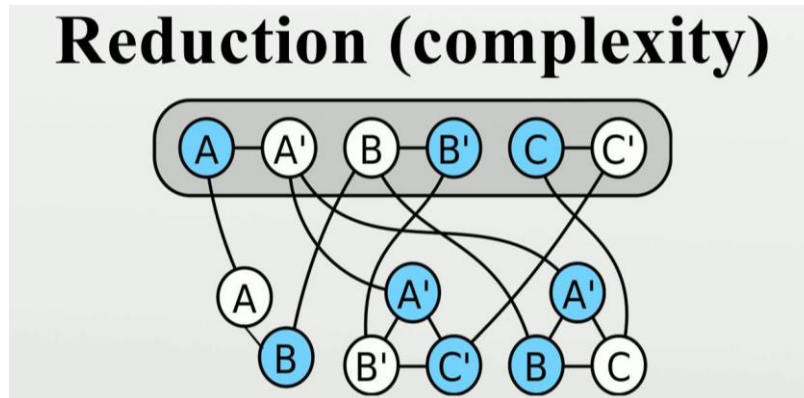


Figure 22: Reduced Complexity

Complexity reduction is a strategic initiative to streamline an organization's operations by eliminating processes and technologies that do not add value to the business. The goal is to speed up decision making, improve performance, and foster a more agile and responsive business environment. (DealHub, 2024) [23]

Information hiding reduces the complexity of a system by limiting the number of details that a user or programmer needs to know to interact with the system's components. By hiding complex implementation details, the system becomes easier to understand, leading to easier maintenance and extension, and reduced errors.

7.2.3 Improved Security



Figure 23: Improved Security

Improved Security is a key benefit of Information Hiding. Hiding internal details and exposing only necessary interfaces helps protect sensitive data from unauthorized access. This ensures that critical data is only accessed and processed securely, minimizing security risks and preventing external intrusions.

VIII. Imperative ADTs as a Basis for Object-Oriented Programming

8.1 Encapsulation and Information Hiding

Encapsulation, a core principle in OOP, can be seen as an extension of how imperative ADTs operate. ADTs encapsulate data by defining a strict interface through which data can be accessed or modified, hiding implementation details from the user. Similarly, in OOP, objects encapsulate data (attributes) and behavior (methods), ensuring that internal states are hidden and protected from unauthorized access.

Justification: The information hiding in ADTs, where only the operations on data are visible while the internal data structure is concealed, mirrors encapsulation in OOP. This principle of hiding unnecessary complexity is foundational to both paradigms and enhances security, maintainability, and flexibility in software development.

8.2 Modularity and Reusability

In both imperative ADTs and OOP, modularity is achieved by breaking down large systems into smaller, independent units that can be developed, tested, and maintained separately. Each ADT defines a module with its data structure and operations, much like how a class in OOP groups related data and methods into a cohesive unit.

Justification: Imperative ADTs foster modularity through the separation of concerns, allowing different ADTs to handle different parts of the system. OOP extends this by introducing reusability, where classes (or objects) can be reused across programs through inheritance, interfaces, and polymorphism. ADTs laid the groundwork for modular thinking in OOP, making code more maintainable and extendable.

8.3 Procedural Approach

Imperative ADTs are closely aligned with the procedural programming approach, where programs are structured as sequences of commands or operations. These ADTs define operations (functions or procedures) that manipulate the data, much like methods in OOP. OOP builds on this by adding the concept of method invocation on objects, providing a more organized way of associating data and procedures.

Justification: The procedural nature of imperative ADTs provided the foundation for early OOP design, where the focus on data operations naturally evolved into objects containing both data and the procedures that manipulate them. OOP, however, extends this by incorporating the idea of object interaction and message passing, adding more sophistication to the procedural model.

8.4 Language Transition

The transition from imperative languages to object-oriented languages represents an evolution in programming paradigms. While imperative ADTs primarily operate on well-defined data structures, OOP languages like C++, Java, and Python integrated ADT principles into objects, leading to more robust software development practices.

Justification: Many early OOP languages were built upon imperative foundations. For example, C++, which introduced OOP features, is a superset of C, an imperative language. The evolution of languages reflects how imperative ADTs served as a stepping stone for the development of more expressive and abstract OOP languages, allowing programmers to define reusable and encapsulated modules more effectively.

III. Conclusion

This report has provided a comprehensive exploration of the stack as an Abstract Data Type (ADT) and highlighted its significance in programming. Stacks, which operate on the Last-In-First-Out (LIFO) principle, offer essential operations such as push, pop, and peek, allowing data to be managed in a controlled sequence. We examined the initialization and structural implementations of stacks, including array-based and linked-list-based approaches, giving readers insight into the versatility and applications of stacks in solving real-world programming problems.

In addition, this report discussed key principles in object-oriented programming (OOP), notably encapsulation and information hiding. Encapsulation ensures that a stack's internal state remains protected from unauthorized access, fostering code reusability and simplifying system maintenance. Information hiding provides a clean, clear interface, allowing users to focus on stack operations without concerning themselves with the underlying implementation. These principles together streamline the stack's usage and enhance the reliability and modularity of software systems.

In summary, defining the stack as an ADT and integrating OOP principles has created a robust framework for managing and manipulating data in software development. These concepts not only strengthen data security but also contribute significantly to the maintainability and scalability of software systems, addressing the evolving complexities of modern information technology. Understanding and applying these foundational ideas will equip students and programmers to build more efficient, sustainable software applications that can adapt to future demands.

IV. References

1. Bors, M.L. (2018). *Preconditions and Postconditions*. [online] Medium. Available at: <https://medium.com/@mlbors/preconditions-and-postconditions-5913fc0fcdaf>.
2. GeeksforGeeks (2021). *Time Complexity and Space Complexity*. [online] GeeksforGeeks. Available at: <https://www.geeksforgeeks.org/time-complexity-and-space-complexity/>.
3. www.sciencedirect.com. (n.d.). *Stack Memory - an overview | ScienceDirect Topics*. [online] Available at: <https://www.sciencedirect.com/topics/engineering/stack-memory>.
4. GeeksforGeeks. (2023). *Basic Operations in Stack Data Structure with Implementations*. [online] Available at: <https://www.geeksforgeeks.org/basic-operations-in-stack-data-structure-with-implementations/>.
5. GeeksforGeeks. (2024). *Function Calling in Programming*. [online] Available at: <https://www.geeksforgeeks.org/function-calling-in-programming/>.
6. GeeksforGeeks. (2021). *Stack Frame in Computer Organization*. [online] Available at: <https://www.geeksforgeeks.org/stack-frame-in-computer-organization/>.
7. Lenovo.com. (2021). *Exploring FIFO in Programming & Its Advantages | Lenovo US*. [online] Available at: <https://www.lenovo.com/us/en/glossary/fifo/?orgRef=https%253A%252F%252Fwww.google.com%252F>.
8. Utah.edu. (2024). *Programming - Structures*. [online] Available at: <https://users.cs.utah.edu/~germain/PPS/Topics/structures.html>.
9. Fiveable.me. (2024). *Array-based implementation - (Intro to Algorithms) - Vocab, Definition, Explanations | Fiveable*. [online] Available at: <https://library.fiveable.me/key-terms/introduction-algorithms/array-based-implementation>
10. GeeksforGeeks. (2024). *Linked List Data Structure*. [online] Available at: <https://www.geeksforgeeks.org/linked-list-data-structure/>.
11. GeeksforGeeks (2018). *Merge Sort - GeeksforGeeks*. [online] GeeksforGeeks. Available at: <https://www.geeksforgeeks.org/merge-sort/>.
12. GeeksforGeeks (2014). *Quick Sort Algorithm*. [online] GeeksforGeeks. Available at: <https://www.geeksforgeeks.org/quick-sort-algorithm/>.
13. geeksforgeeks (2023). *What is Dijkstra's Algorithm? | Introduction to Dijkstra's Shortest Path Algorithm*. [online] GeeksforGeeks. Available at: <https://www.geeksforgeeks.org/introduction-to-dijkstras-shortest-path-algorithm/>
14. GeeksforGeeks (2012). *Prim's minimum spanning tree (MST) | greedy algo-5 - geeksforgeeks*. [online] GeeksforGeeks. Available at: <https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/>.

15. GeeksforGeeks. (2023). What is Data Structure? [online] Available at:
<https://www.geeksforgeeks.org/data-structure-meaning/>.
16. www.javatpoint.com. (n.d.). DS Stack - javatpoint. [online] Available at:
<https://www.javatpoint.com/data-structure-stack>.
17. Coursera. (2023). *What Is Encapsulation?* [online] Available at:
<https://www.coursera.org/articles/encapsulation>.
18. SNIA (2024). *What is Data Protection? / SNIA*. [online] www.snia.org. Available at:
<https://www.snia.org/education/what-is-data-protection>.
19. Lenovo.com. (2021). *Modularity: What is It and How Does it Enhance Business Productivity?* / *Lenovo US*. [online] Available at:
https://www.lenovo.com/us/en/glossary/modularity/?orgRef=https%253A%252F%252Fwww.google.com%252F&srsId=AfmBOoq8lpsewLp2D9WE8ar_TivzyYxsfvqBdeo8dv1OaFr9otQsr2Id
20. www.sciencedirect.com. (n.d.). *Maintainability - an overview / ScienceDirect Topics*. [online] Available at: <https://www.sciencedirect.com/topics/engineering/maintainability>.
21. www.linkedin.com. (n.d.). *Code Reusability*. [online] Available at:
<https://www.linkedin.com/pulse/code-reusability-r%C4%83ducu-roman>.
22. Gupta, U. (2022). *Scaler Topics*. [online] www.scaler.com. Available at:
<https://www.scaler.com/topics/java/difference-between-data-hiding-and-abstraction/>.
23. DealHub. (2024). *Complexity Reduction*. [online] Available at:
<https://dealhub.io/glossary/complexity-reduction/>.

Link Github: https://github.com/tuen1985/Assignment1_DSA