

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 Quang Linh	Student ID	BH01804
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.

 **Summative Feedback:**

 **Resubmission Feedback:**

Grade:

Assessor Signature:

Date:

Internal Verifier's Comments:

Signature & Date:

Table of Contents

I. Data Structures and Complexity	7
1. Identify the Data Structures	7
2. Define the Operations.....	7
3. Specify Input Parameters.....	7
4. Define Pre- and Post-conditions	8
5. Discuss Time and Space Complexity	8
II. Memory Stack	8
1. Define a Memory Stack.....	8
2. Identify Operations	9
3. Function Call Implementation	10
4. Demonstrate Stack Frames	10
5. Discuss the Importance	10
III. Queue	11
1. Introduction FIFO	11
2. Define the Structure.....	11
3. Array-Based Implementation	11
4. Linked List-Based Implementation	12
5. Provide a concrete example to illustrate how the FIFO queue works	12
IV. Sorting algorithms	14
1. Introducing the two sorting algorithms you will be comparing	14
2. Time Complexity Analysis	14
3. Space Complexity Analysis.....	15
4. Stability	15
5. Comparison Table	15
6. Performance Comparison	16
7. Provide a concrete example to demonstrate the differences in performance between the two algorithms	16
V. Network shortest path algorithms.....	16
1. Introducing the concept of network shortest path algorithms	16
2. Algorithm 1: Dijkstra's Algorithm	17
3. Algorithm 2: Prim-Jarnik Algorithm.....	18

4. Performance Analysis	18
VI. Specify the abstract data type for a software stack using an imperative definition.	20
1. Define the Data Structure: Stack (Imperative Definition)	20
2. Initialise the stack	21
3. Push Operation.....	21
4. Pop Operation	21
6. Check if the Stack is Empty	22
5. Peek Operation.....	23
7. Full Source Code (Stack Using Array).....	23
8. Full Source Code (Stack Using Linked List)	25
9. Comparison: Array vs Linked List Implementation	26
VII. Examine the advantages of encapsulation and information hiding when using an ADT.	26
1. Encapsulation	26
1.1. What is Encapsulation?	26
1.2. Example of encapsulations	28
1.3. Data Protection in Encapsulation	28
1.4. Modularity and Maintainability	30
1.5. Code Reusability in Encapsulation.....	31
2. Information Hiding	32
2.1. Abstraction	32
2.2. Reduced Complexity	33
2.3. Improved Security	33
VIII. Imperative ADTs as a Basis for Object Orientation	33
1. Encapsulation and Information Hiding.....	33
2. Modularity and Reusability	34
3. Procedural Approach	34
4. Language Transition	35
IX. CONCLUSION	35
X. REFERENCES	36

Table of Figures

Figure 1: Example: FIFO Queue.....	13
Figure 2: Initialise the stack	21
Figure 3: Push Operation	21
Figure 4: Pop Operation	22
Figure 5: Check if the Stack is Empty	22
Figure 6: Peek Operation	23
Figure 7: Full Source Code.....	24
Figure 8: Full Source Code.....	25
Figure 9: Example of encapsulations	28

I. Data Structures and Complexity

1. Identify the Data Structures

Objective: Classify and explain basic data structures.

Types of data structures:

- **Array:** Static arrays, dynamic arrays and index types.
- **Linked List:** Single, double, circular linked lists. Advantages over arrays (flexible insertion and deletion).
- **Stack:** Stacks are used in tree traversal, recursive processing.
- **Queue:** Regular queue and priority queue.
- **Tree:** Binary tree, binary search tree (BST), AVL tree.
- **Hash Table:** Conflict handling techniques, such as linear probing, double hashing.

2. Define the Operations

- **Insert:** How to insert in dynamic arrays and linked lists.
- **Delete:** Difference between deleting at the beginning/end and deleting at any position in linked lists and arrays.
- **Search:** Linear Search ($O(n)$) and Binary Search ($O(\log n)$).
- **Sort:** Popular sorting algorithms, from simple (Bubble Sort) to complex (Merge Sort, Quick Sort).
- **Traverse:** Traverse arrays and trees in pre-order, in-order, and post-order.

3. Specify Input Parameters

For each operation on a data structure, it is necessary to specify the input parameters to ensure that the operations are performed correctly. The parameters include the format, data type, and specific location for the operations.

a. Insert into Array:

Required parameters:

- **Value:** The value to be inserted into the array. This can be an integer, a string, or any data type depending on the array.
- **Reference:** The index in the array where the new value will be inserted.

b. Delete from Linked List:

Required parameters:

- **Index:** Index of the element to be deleted (used for singly or doubly linked lists).
- **Reference:** Direct reference to the element to be deleted (can be a pointer or a specific object).

c. Push into Stack:

Request Parameters:

- Value: The value to be placed on the stack. The data type can be any (integer, string, object).

d. Enqueue into Queue:

Request Parameters:

- Value: The value to be placed on the queue. This can be a number, string, or object depending on the specific application.

4. Define Pre- and Post-conditions

- **Pre-condition:** Are conditions that must be satisfied before performing an operation.
- **Post-condition:** Are conditions that must be satisfied after performing an operation.

5. Discuss Time and Space Complexity

Big O notation helps evaluate the performance of algorithms, showing how they perform as the input size increases. It describes how quickly the time or memory required increases as the input data increases, helping us choose the right data structures and algorithms.

- $O(1)$: Immediate operations such as accessing elements in an array.
- $O(n)$: Traversing a linked list or array.
- $O(\log n)$: Traversing a binary search tree.
- $O(n \log n)$: Merge Sort, Quick Sort.
- $O(n^2)$: Simple sorting algorithms such as Bubble Sort.

II. Memory Stack

1. Define a Memory Stack

Memory Stack is an important data structure in programming, especially in systems programming. Stacks are used to manage memory for local variables, store program state, and make function calls during execution.

Functions and Roles of Stack Memory

a. Memory Management:

- The stack is used to store local variables of a function, including parameters and return values.

- When a function is called, a new stack frame is created, which contains information about the local variables of that function.

b. Function Call Implementation:

- The stack holds information about the return address to know where to continue after the function completes.
- When a function is called, the address of that function is pushed onto the stack. When the function completes, this address is taken from the stack and the program control returns to the next point.

c. Recursion Management:

- The stack allows recursive functions to be executed without running out of memory, because each recursive function call creates a new stack frame.
- Each stack frame contains the state of the function calls, allowing the program to return to the correct state when the function completes.

2. Identify Operations

The memory stack supports several basic operations that help manage stack frames during program execution. Here are the three main operations: Push, Pop, and Peek.

a. Push: Add New Stack Frame

Description:

- When a function is called, a new stack frame is created and added to the top of the stack.
- This frame contains the function's parameters and return address, allowing the program to know where to continue when the function finishes.

b. Pop: Remove Stack Frame

Description:

- When a function completes, the corresponding stack frame is removed (pop) from the top of the stack.
- This frees up memory and returns control to the calling function.

c. Peek: View Information Without Performing an Operation

Description:

- The Peek operation allows you to view the return address or parameters of a function without removing the stack frame.
- This is useful when you need to check information without changing the state of the stack.

3. Function Call Implementation

When a function is called in a program, the memory stack plays an essential role in managing the state of the program. Here is the detailed process of how the stack performs a function call.

- When Calling Function: Push Operation
- Executing Function
- When Function Ends: Pop Operation

4. Demonstrate Stack Frames

The Stack Frame is an important structure in the memory stack that stores all the information needed to execute a function call. Each time a function is called, a new stack frame is created to hold the function's local variables, parameters, and return address.

Structure of a Stack Frame. A stack frame typically contains the following main components:

a. Base Pointer (BP):

Function:

- BP is used to store the address of the bottom of the current stack frame. It holds information about the location of parameters and local variables in the stack frame.

b. Stack Pointer (SP):

Function:

- SP points to the top of the stack and moves up and down as stack frames are created or released. When a function is called, SP is incremented to store the new stack frame. When the function ends, SP is decremented as the stack frame is removed.

5. Discuss the Importance

The memory stack plays a crucial role in handling recursive functions. When a function calls itself, the memory stack helps keep track of function calls and maintains the state of each call, thus allowing for correct and efficient execution.

III. Queue

1. Introduction FIFO

a. Definition of FIFO Queue

A FIFO (First In, First Out) queue is a data structure that allows elements to be added and removed on a “first in, first out” basis. This means that the first element added to the queue will be the first element removed. A FIFO queue works similar to a queue, where the person at the head of the queue will be served first.

b. FIFO Queue Structure:

- Enqueue: The new element is added to the tail of the queue.
- Dequeue: The element is removed from the head of the queue.

c. Applications of FIFO Queue

- Printing System
- CPU Scheduling
- Data Transmission Network
- Resource Management

2. Define the Structure

A Queue is a linear data structure that allows elements to be managed according to the FIFO (First In, First Out) principle. A queue has two main operations:

- **Enqueue:** The Enqueue operation is used to add a new element to the tail of the queue.
- **Dequeue:** The Dequeue operation is used to remove an element from the head of the queue.

Queue Data Structures: Queues can be implemented in a variety of ways, including:

a. Array-Based Queues:

- The queue is stored in an array. When Enqueue and Dequeue are performed, the index of the head and tail of the queue will change accordingly. The size of the array must be carefully managed to avoid overflow or underflow.

b. Linked List-Based Queues:

- The queue is implemented using a linked list, allowing easy addition and removal of elements without worrying about the size. Each element in the queue will be a node in the list.

3. Array-Based Implementation

a. Array Structure: Queue can be implemented using an array with fixed length.

b. How it works:

- Enqueue: Add element to "rear" index.
- Dequeue: Remove element at "front" and increment "front" index.
- When the queue is full, use circular queue or re-initialize.

4. Linked List-Based Implementation

a. Dynamic Linked Structure: Linked queue has no size limit and makes good use of memory.

b. How it works:

- Enqueue: Add element to the end of the linked list.
- Dequeue: Remove element at the beginning of the list, freeing the memory of the head node.

5. Provide a concrete example to illustrate how the FIFO queue works

a. Operating Principle:

- When a user sends a print command, it will be added to the print queue (Enqueue).
- The printing system will take the first command in the queue to execute (Dequeue).

b. Process:

- Step 1: User A sends a command to print document A.
- Step 2: User B sends a command to print document B.
- Step 3: User C sends a command to print document C.
- The print queue will now have the order: [A, B, C].

c. Processing Print Commands:

- Step 4: The system takes the first print command (document A) out of the queue to print.
- Step 5: When document A finishes printing, it will be removed from the queue.
- Step 6: Next, the print command for document B will be processed.
- The print queue will now have: [B, C].

d. Result:

- Printed document A will be completed first, then document B and finally document C, keeping the FIFO principle.

```
class PrintQueue:
    def __init__(self):
        self.queue = []

    def enqueue(self, document):
        self.queue.append(document)
        print(f"Document {document} added to the print queue.")

    def dequeue(self):
        if self.queue:
            return self.queue.pop(0)
        else:
            return None

    def print_documents(self):
        while self.queue:
            document = self.dequeue()
            print(f"Printing document: {document}")
            print("All documents have been printed.")

# Khởi tạo hàng đợi in
print_queue = PrintQueue()

# Thêm lệnh in
print_queue.enqueue("Document A")
print_queue.enqueue("Document B")
print_queue.enqueue("Document C")

# Bắt đầu in tài liệu
print_queue.print_documents()
```

Figure 1: Example: FIFO Queue

IV. Sorting algorithms

1. Introducing the two sorting algorithms you will be comparing

1.1. Merge Sort

a. Definition:

- Merge Sort is a divide and conquer sorting algorithm. It divides the list to be sorted into two halves, sorts each halves, and then merges them into a sorted list.

b. Process:

- Divide: The list is divided into two halves until each half has only one element.
- Sort and Merge: The halves are merged during the sorting process.

c. Application:

- Suitable for sorting large data, such as sorting files or databases.
- Used in applications that require stability, where the order between equal elements needs to be maintained.

1.2. Quick Sort

a. Definition:

- Quick Sort is also a divide and conquer sorting algorithm, but it works by selecting an element as a "pivot" and dividing the list into elements smaller and larger than the pivot. The algorithm is then applied recursively to the elements to the left and right of the pivot.

b. Process:

- Pivot Selection: Select an element as a pivot (usually the first, middle, or last element).
- Divide: Reorder the list so that all elements smaller than the pivot are on the left, and all elements larger than the pivot are on the right.
- Recursive: Apply Quick Sort to the left and right parts.

c. Application:

- Suitable for in-memory sorting.
- Widely used in applications that require high performance and low memory.

2. Time Complexity Analysis

a. Merge Sort

- Merge Sort has $O(n \log n)$ time complexity in all cases (best, average, and worst cases).
- This means that even if the list is sorted or random, Merge Sort always has stable performance, dividing the data into small arrays and merging them.

b. Quick Sort

- Quick Sort has $O(n \log n)$ time complexity in the average and best cases, when the pivot element is chosen wisely and the data is evenly distributed.
- However, in the worst case, when the pivot is chosen poorly (for example, choosing the first or last element of a sorted array), Quick Sort can have $O(n^2)$ complexity, because the algorithm has to iterate through many inefficient splits.

3. Space Complexity Analysis

Merge Sort:

- Requires additional memory to store the sub-array for the merge, so the space complexity is $O(n)$.

Quick Sort:

- Uses a stack to store the recursion state, so the space requirement depends on the depth of the recursion. The best case is $O(\log n)$, but the worst case can be up to $O(n)$.

4. Stability

- Merge Sort: Stable because it always maintains the original order of equal elements during the merge. This is useful when it is important to maintain the original order in data.
- Quick Sort: Unstable because the process of selecting pivots and swapping elements can change the order of equal elements. This may not be important in some applications, but is a factor to consider in special situations.

5. Comparison Table

Criteria	Merge Sort	Quick Sort
Time Complexity	$O(n \log n)$ (best, average, worst)	$O(n \log n)$ (average), $O(n^2)$ (worst)

Space complexity	$O(n)$ (due to need for sub-array)	$O(\log n)$ (average), $O(n)$ (worst)
Stability	Stable	Unstable
Practical Applications	Arrange large files, data needs to be stable	In-memory sorting, high performance

6. Performance Comparison

- Merge Sort is suitable for large data and needs stability, but consumes secondary memory.
- Quick Sort is fast in most cases and does not require much secondary memory, but can be slow and resource-consuming in case of uneven partitioning.

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

For example, suppose we need to sort a list of 1000 elements that are sorted in reverse order.

- **Merge Sort** will process this list in $O(n \log n)$ time, and will sort it evenly and consistently.
- **Quick Sort** can achieve $O(n^2)$ time if it always chooses the first or last pivot, which results in a significant performance degradation compared to Merge Sort in this case.

V. Network shortest path algorithms

1. Introducing the concept of network shortest path algorithms

a. Concept:

Shortest path: Is the problem of finding the shortest path between two vertices in a weighted graph.

Shortest path algorithms help optimize the search for optimal paths through complex networks such as:

- Transportation: Find the fastest path between two points in the transportation system.
- Computer networks: Find the most efficient route to transmit data through computer networks.

b. Applications:

- Path planning: Used in mapping applications, GPS to find the fastest path.

- Network optimization: Used to route data through computer networks to reduce latency and maximize bandwidth.
- Graph search: Analyze data, such as social networks or connecting cities.

c. Popular algorithms:

- Dijkstra's Algorithm.
- Prim-Jarnik Algorithm.

2. Algorithm 1: Dijkstra's Algorithm

a. Objective: Find the shortest path from a source vertex to all other vertices in a non-negative weighted graph.

b. How it works:

- The algorithm starts at the source vertex, marking its distance as 0.
- Continue to traverse the weighted adjacent vertices, updating the shortest distance to each unvisited vertex.
- Choose the unvisited vertex with the smallest distance, repeat the process until all vertices are processed.

c. Detailed Steps:

- Set the distance from the source vertex to itself to 0, other vertices to infinity (∞).
- Choose the vertex with the smallest distance, update the distances to the adjacent vertices.
- Repeat step 2 until all vertices are visited.

d. Complexity:

- Time: $O(V^2)$ for the basic implementation, or $O(E + V \log V)$ for the Priority Queue implementation.
- Memory: $O(V)$ to store information about vertices and shortest distances.

e. Applications:

- GPS navigation systems: Find the fastest path between locations.
- Computer networks: Find the optimal route to transmit data.
- Path planning: Applications in games, robot navigation.

3. Algorithm 2: Prim-Jarnik Algorithm

a. Objective: Prim-Jarnik Algorithm (also known as Prim's Algorithm) finds the Minimum Spanning Tree (MST) in a weighted graph. The minimum spanning tree is the set of edges in the graph that connect all vertices with the smallest total weight without forming a cycle.

b. How it works:

- Start from any vertex (usually the source vertex).
- Choose the edge with the smallest weight connecting the current vertex to an unvisited vertex.
- Add this edge to the minimum spanning tree and continue traversing the adjacent edges from the visited vertex set.
- Repeat the process until all vertices in the graph are added to the spanning tree.

c. Detailed Steps:

- Initialization: Select a source vertex, mark it as visited.
- Traverse: Each time select the edge with the smallest weight connecting a visited vertex and an unvisited vertex.
- Repeat: Continue traversing until all vertices are added to the spanning tree.

d. Complexity:

- Time: $O(V^2)$ with basic implementation, or $O(E \log V)$ if using priority queue.
- Memory: $O(V)$ for storing information about vertices and minimum spanning trees.

e. Applications:

- Communication networks: Optimizing the connection of communication networks with the lowest cost.
- Circuit design: Creating connections between nodes in a circuit with the shortest total cost.
- Transportation: Building optimal road and railway systems, minimizing construction costs.

4. Performance Analysis

Criteria	Dijkstra's algorithm	Prim-Jarnik algorithm

Target	Find the shortest path from a source vertex to all other vertices in the graph.	Find the minimum spanning tree that connects all vertices in the graph.
Graph Type	The graph has non-negative weights.	The graph has arbitrary weights (non-negative or negative).
Time Complexity	$O(V^2)$ or $O(E + V \log V)$ with priority queue.	$O(V^2)$ or $O(E \log V)$ with priority queue.
Application	GPS systems, computer networks, route planning.	Communication network, circuit design, connection optimization.
Features	Find the shortest path from one vertex to multiple vertices.	Build the smallest tree span at the lowest total cost.

Performance Analysis:

a. Dijkstra:

- **Advantages:** Finds the shortest path from a vertex to all other vertices in a graph, suitable for routing and navigation systems.
- **Disadvantages:** Only works well for graphs with non-negative weights. Not optimal for finding connections to the entire graph.

b. Prim-Jarnik:

- **Advantages:** Finds the minimum spanning tree, optimal when connecting all vertices in a graph with the smallest cost. Often used in network design, where the connection cost is minimized.
- **Disadvantages:** Does not solve the shortest path problem for a specific vertex.

VI. Specify the abstract data type for a software stack using an imperative definition.

1. Define the Data Structure: Stack (Imperative Definition)

A **stack** is an abstract data type (ADT) that follows the **Last In, First Out (LIFO)** principle. In a stack, the most recently added element is the first one to be removed.

- Key Operations of a Stack:

- **Push:** Add an element to the top of the stack.
- **Pop:** Remove the topmost element from the stack.
- **Peek (Top):** Return the topmost element without removing it.
- **IsEmpty:** Check whether the stack is empty.
- **Size:** Return the number of elements in the stack.

Real-World Examples of Stack Applications:

- **Undo/Redo Functionality:** In applications like text editors, a stack is used to keep track of actions. Each action is pushed onto the stack, and the undo operation pops the most recent action.
- **Expression Evaluation:** In compilers or calculators, stacks are used to evaluate expressions, especially those written in postfix or prefix notation.
- **Backtracking Algorithms:** Stack is utilized in algorithms like depth-first search (DFS) to keep track of visited nodes and to backtrack when necessary.
- **Function Call Management:** Stacks are used by programming languages for managing function calls through the call stack. When a function is called, its state (local variables, execution point) is pushed onto the stack. When the function finishes, the state is popped.

- **Browser History Navigation:** Web browsers use a stack to manage navigation history. Every page visited is pushed onto the stack, and clicking "Back" pops the most recent page from the stack to go back to the previous one.

2. Initialise the stack

In Java, you can initialize a stack using an array. Since arrays in Java have a fixed size, you'll need to define the maximum size of the stack.

```
class Stack { no usages
    private int maxSize; 2 usages
    private int top; 1 usage
    private int[] stackArray; 1 usage

    public Stack(int size) { no usages
        maxSize = size;
        stackArray = new int[maxSize];
        top = -1; // Indicates an empty stack
    }
}
```

Figure 2: Initialise the stack

3. Push Operation

The push operation adds an element to the top of the stack. You need to check if the stack is full before pushing.

```
// Push operation
public void push(int value) {
    if (top == maxSize - 1) {
        System.out.println("Stack is full. Cannot push " + value);
    } else {
        stackArray[++top] = value; // Increment top and insert value
        System.out.println("Pushed " + value + " onto the stack.");
    }
}
```

Figure 3: Push Operation

4. Pop Operation

The pop operation removes and returns the top element from the stack. Ensure the stack is not empty before popping.

```
// Pop operation
public int pop() {
    if (top == -1) {
        System.out.println("Stack is empty. Cannot pop.");
        return -1; // Return a sentinel value or throw an exception
    } else {
        int value = stackArray[top--]; // Return top value and decrement top
        System.out.println("Popped " + value + " from the stack.");
        return value;
    }
}
```

Figure 4: Pop Operation

6. Check if the Stack is Empty

To check if the stack is empty, verify whether top is -1.

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

Figure 5: Check if the Stack is Empty

5. Peek Operation

The peek operation allows you to view the top element without removing it from the stack.

```
// Peek operation
public int peek() {
    if (top == -1) {
        System.out.println("Stack is empty. Nothing to peek.");
        return -1; // Return a sentinel value or throw an exception
    } else {
        return stackArray[top]; // Return the top element
    }
}
```

Figure 6: Peek Operation

7. Full Source Code (Stack Using Array)

```
class Stack {
    private int maxSize;
    private int top;
    private int[] stackArray;

    public Stack(int size) {
        maxSize = size;
        stackArray = new int[maxSize];
        top = -1; // Indicates an empty stack
    }

    // Push operation
    public void push(int value) {
        if (top == maxSize - 1) {
            System.out.println("Stack is full. Cannot push " + value);
        } else {
            stackArray[++top] = value; // Increment top and insert value
            System.out.println("Pushed " + value + " onto the stack.");
        }
    }
}
```

```

public int pop() {
    if (top == -1) {
        System.out.println("Stack is empty. Cannot pop.");
        return -1; // Return a sentinel value or throw an exception
    } else {
        int value = stackArray[top--]; // Return top value and decrement top
        System.out.println("Popped " + value + " from the stack.");
        return value;
    }
}

// Peek operation
public int peek() {
    if (top == -1) {
        System.out.println("Stack is empty. Nothing to peek.");
        return -1; // Return a sentinel value or throw an exception
    } else {
        return stackArray[top]; // Return the top element
    }
}

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

```

Figure 7: Full Source Code

8. Full Source Code (Stack Using Linked List)

```
class Node { 4 usages
    int data; 3 usages
    Node next; 3 usages

    public Node(int data) { 1 usage
        this.data = data;
        this.next = null;
    }
}

class StackLinkedList { no usages
    private Node top; 10 usages

    public StackLinkedList() { no usages
        top = null;
    }

    // Push operation
    public void push(int value) { no usages
        Node newNode = new Node(value);
        newNode.next = top;
        top = newNode;
        System.out.println("Pushed " + value + " onto the stack.");
    }

    public int pop() { no usages
        if (top == null) {
            System.out.println("Stack is empty. Cannot pop.");
            return -1; // Return a sentinel value or throw an exception
        } else {
            int value = top.data;
            top = top.next; // Move top to the next node
            System.out.println("Popped " + value + " from the stack.");
            return value;
        }
    }

    // Peek operation
    public int peek() { no usages
        if (top == null) {
            System.out.println("Stack is empty. Nothing to peek.");
            return -1; // Return a sentinel value or throw an exception
        } else {
            return top.data; // Return the top element
        }
    }

    // Check if the stack is empty
    public boolean isEmpty() { no usages
        return top == null;
    }
}
```

Figure 8: Full Source Code

9. Comparison: Array vs Linked List Implementation

a. Memory Management:

- **Array:** Requires a predefined size, which can lead to wasted space or stack overflow if the size limit is exceeded.
- **Linked List:** Grows dynamically as needed, so it uses memory efficiently without any predefined size.

b. Speed:

- **Array:** Direct access to elements via index makes it faster in cases where random access is needed. Push and pop operations are $O(1)$.
- **Linked List:** Each element is accessed sequentially, making it slower for certain operations. Push and pop operations are $O(1)$, but space overhead exists due to storing pointers.

c. Complexity:

- **Array:** Easier to implement but less flexible regarding dynamic resizing.
- **Linked List:** More complex to implement but offers better flexibility for growing stack sizes dynamically. Each node requires additional memory for the pointer to the next node.

d. Use Cases:

- **Array:** Best for situations where the maximum size of the stack is known in advance.
- **Linked List:** Preferred when the number of elements is unknown or may vary significantly over time.

VII. Examine the advantages of encapsulation and information hiding when using an ADT.

1. Encapsulation

1.1. What is Encapsulation?

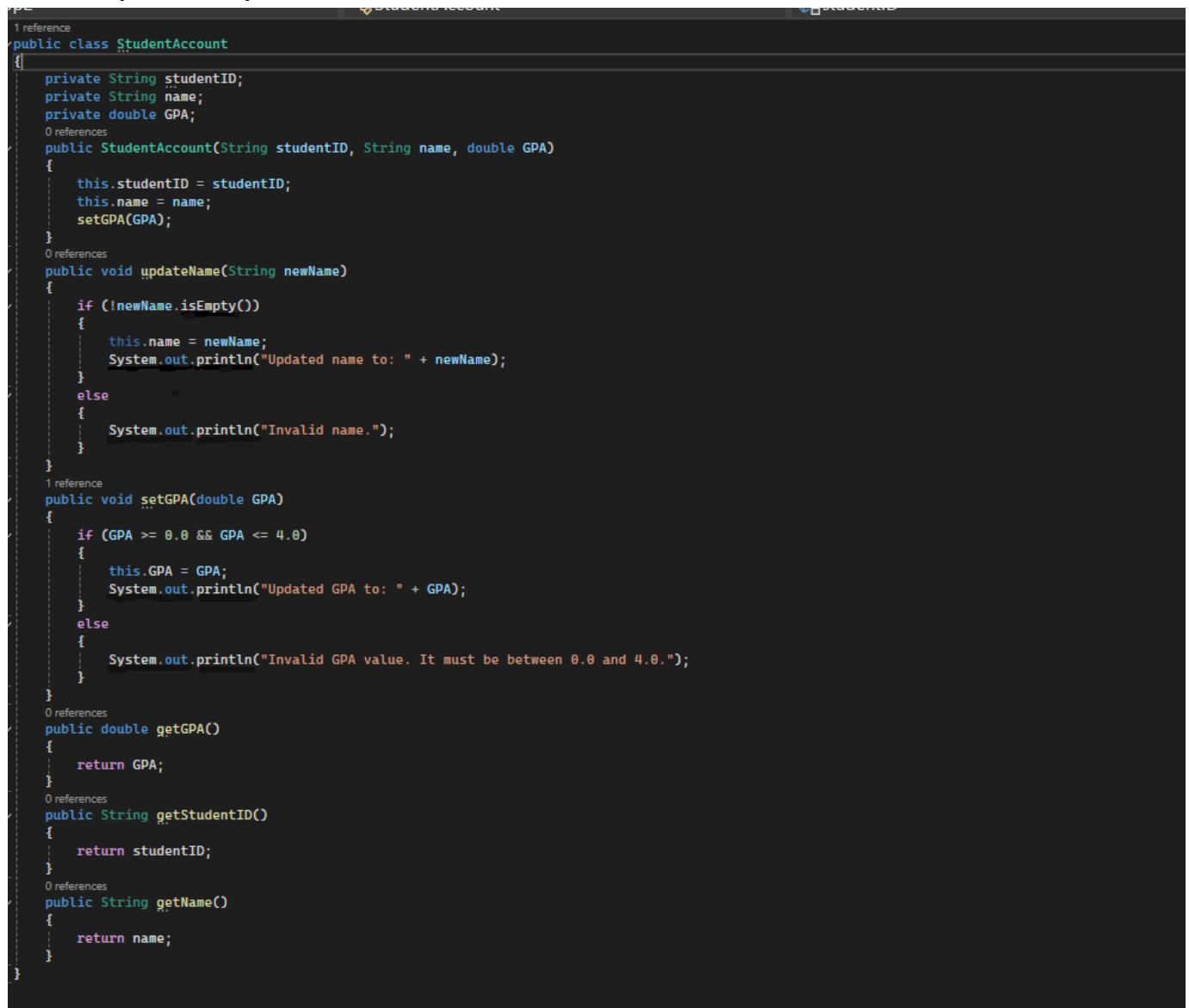
In object-oriented programming (OOP), **encapsulation** is the concept of bundling data (variables) and methods (functions) that operate on the data into a single unit, or **class**. Encapsulation allows an object to control its own state by restricting direct access to its internal data and only allowing modification through well-defined methods.

This practice helps ensure that the internal representation of an object is hidden from the outside world, promoting the idea of **information hiding**. By hiding the internal state, encapsulation helps to protect the integrity of an object and ensures that its data cannot be directly modified or accessed in unintended ways.

Advantages of Encapsulation and Information Hiding in ADTs:

- **Modularity:** By hiding implementation details, encapsulation allows developers to change the internals of an ADT (Abstract Data Type) without affecting external code that uses the ADT. This leads to better modularity and easier maintenance.
- **Data Protection:** Encapsulation ensures that the internal data of an object is protected from outside interference and misuse. Direct manipulation of data is not allowed, which helps maintain consistency and correctness.
- **Easier Debugging:** By isolating the data and its manipulation within methods, bugs and issues are easier to detect and resolve since changes are limited to specific parts of the code.
- **Flexibility:** The internal implementation of an ADT can be modified without requiring changes to the code that interacts with it. As long as the public interface (the methods exposed) remains the same, users of the ADT will not be affected.
- **Reusability:** Well-encapsulated code is more reusable since it abstracts the complexity of how operations are performed internally, allowing developers to use and adapt the ADT in different situations.

1.2. Example of encapsulations



```
1 reference
public class StudentAccount
{
    private String studentID;
    private String name;
    private double GPA;
    0 references
    public StudentAccount(String studentID, String name, double GPA)
    {
        this.studentID = studentID;
        this.name = name;
        setGPA(GPA);
    }
    0 references
    public void updateName(String newName)
    {
        if (!newName.isEmpty())
        {
            this.name = newName;
            System.out.println("Updated name to: " + newName);
        }
        else
        {
            System.out.println("Invalid name.");
        }
    }
    1 reference
    public void setGPA(double GPA)
    {
        if (GPA >= 0.0 && GPA <= 4.0)
        {
            this.GPA = GPA;
            System.out.println("Updated GPA to: " + GPA);
        }
        else
        {
            System.out.println("Invalid GPA value. It must be between 0.0 and 4.0.");
        }
    }
    0 references
    public double getGPA()
    {
        return GPA;
    }
    0 references
    public String getStudentID()
    {
        return studentID;
    }
    0 references
    public String getName()
    {
        return name;
    }
}
```

Figure 9: Example of encapsulations

1.3. Data Protection in Encapsulation

Data protection is one of the core benefits of encapsulation in object-oriented programming (OOP). By restricting direct access to the internal data of an object, encapsulation ensures that the data is safeguarded from unauthorized modification or misuse. Here's a deeper analysis of how encapsulation contributes to data protection:

Key Aspects of Data Protection in Encapsulation

a. Private Access Modifiers:

- Encapsulation typically involves using the private keyword to prevent external access to an object's internal variables (fields or attributes). This means that data cannot be changed directly by code outside of the class. Only the methods within the class can modify or retrieve the data.
- Example: In the StudentAccount class from the previous example, the fields studentID, name, and GPA are marked as private, ensuring that external objects cannot modify them directly.

b. Controlled Access through Methods:

- Encapsulation provides **getter** and **setter** methods to control how the data is accessed or modified. For instance, in the StudentAccount class, the setGPA(double GPA) method validates that the GPA falls within the acceptable range (0.0 to 4.0) before modifying the internal state. This prevents invalid data from corrupting the object.

c. Preventing Inconsistent States:

- Without encapsulation, if external code had direct access to an object's fields, it could set invalid or conflicting values that would cause the object to be in an inconsistent or erroneous state. Encapsulation ensures that all modifications to data are done in a controlled and consistent way.
- Example: In the BankAccount class, if direct access to the balance field was allowed, external code could inadvertently set it to a negative value. Encapsulation prevents this by only allowing the withdraw and deposit methods to modify the balance under predefined conditions.

d. Enhanced Security:

- By limiting access to sensitive data, encapsulation enhances the security of the application. For example, in a banking application, information such as account numbers, passwords, and balances should be hidden from external systems and only accessible through secure, controlled operations.
- Example: In the BankAccount class, the accountNumber is private and can only be accessed through a getter method, ensuring that unauthorized access or modifications to the account number are prevented.

e. Maintenance and Flexibility:

- Encapsulation makes it easier to maintain and update code without affecting external parts of the program that rely on the class. If the internal implementation needs to change (e.g., modifying how a student's GPA is calculated), these changes can be done within the class without impacting other parts of the system.
- This ensures that sensitive data can be protected from unintentional consequences during updates or bug fixes.

1.4. Modularity and Maintainability

Modularity and **maintainability** are two significant benefits of encapsulation in object-oriented programming (OOP). Encapsulation promotes the separation of concerns and simplifies managing complex software by breaking it into smaller, self-contained modules. Here's a detailed analysis of how encapsulation enhances modularity and maintainability:

a. Modularity

Modularity refers to the practice of dividing a program into separate, independent units or modules that encapsulate specific functionalities. Each module has well-defined interfaces, and its internal implementation is hidden from the rest of the system. Encapsulation supports modularity by ensuring that data and methods within a class are self-contained and only interact with other parts of the program through controlled interfaces.

How Encapsulation Enhances Modularity:

- **Self-contained units:** Each class in an encapsulated system acts as a module. The class contains both the data and methods needed to operate on that data, which makes it independent and self-contained.
- **Separation of concerns:** By keeping the internal details of a class hidden, encapsulation allows developers to focus on specific functionalities without worrying about other parts of the program. Each class/module is responsible for its own data and behavior.
- **Interchangeability and reusability:** Modular code is more reusable. Since classes are self-contained, they can be easily reused in different parts of the application or even across projects. If a new type of bank account or student record needs to be created, existing classes can be extended or reused without modifying their internal implementation.

b. Maintainability

Maintainability refers to the ease with which a program can be updated, modified, or extended over time without introducing bugs or errors. Encapsulation improves maintainability by limiting the scope of changes and ensuring that modifications to one part of the system do not affect other parts.

How Encapsulation Enhances Maintainability:

- **Controlled access:** Since encapsulation restricts direct access to an object's internal state, any changes to the internal implementation can be done without affecting external code that depends on the object. Only the public methods need to remain consistent, allowing for easy modification of the internal workings of a class.
- **Easier debugging and fixing:** Encapsulation isolates changes to specific classes or modules, which makes it easier to track down bugs and fix them. Since the internal implementation is hidden, you can ensure that changes are localized and don't affect the overall system.
- **Minimized ripple effects:** In a well-encapsulated system, changes made to one class will not impact other classes, as long as the public interfaces remain the same. This minimizes the risk of breaking other parts of the program when making updates.
- **Simplified testing:** Encapsulation makes it easier to test individual classes (modules) in isolation. Since each class has well-defined responsibilities and interfaces, you can write unit tests to ensure that the class functions as expected without needing to test the entire system.

1.5. Code Reusability in Encapsulation

Code reusability is a significant advantage of encapsulation in object-oriented programming (OOP). Encapsulation allows developers to write self-contained, modular classes that can be reused across different parts of a program or even in different projects. By packaging both data and behavior into classes, encapsulation makes it easy to extend, adapt, and reuse code without needing to rewrite it from scratch.

Key Aspects of Code Reusability in Encapsulation:

a. Self-contained Classes:

- In encapsulation, a class contains both the data (attributes) and methods (functions) that operate on that data. This self-contained nature of a class makes it an ideal reusable component because the class can be used in different contexts without modification.

b. Inheritance and Extension:

- Encapsulation allows a class to be extended or inherited in other classes, making it easier to reuse the core functionality while adding or modifying specific behaviors. Inheritance allows you to build new classes based on existing ones, promoting reusability without duplication of code.

c. Generalization and Specialization:

- Encapsulation supports **generalization** by allowing developers to create generalized classes that can be reused in various contexts, while also enabling **specialization** where specific behaviors can be introduced in derived classes. The original class remains reusable and intact, while derived classes add specialized behavior as needed.

d. Decoupling and Flexibility:

- Encapsulation decouples the internal implementation of a class from the code that uses it, making classes more flexible and adaptable. As long as the public methods (interface) of a class remain the same, the class can be reused without modification, even if its internal implementation changes.
- This decoupling allows a class to be reused in different projects or systems, as external code doesn't depend on how the class performs its tasks internally, but only on its public interface.

e. Maintainability and Scalability:

- Code that is encapsulated and reusable tends to be more maintainable. If a class needs to be updated or optimized, the changes are localized within that class. This enhances maintainability and scalability because the class can be improved over time without rewriting all the code that uses it.

2. Information Hiding

2.1. Abstraction

- Abstraction is the process of simplifying complex systems by focusing on the essential features while hiding the underlying implementation details. In the context of ADTs, abstraction allows users to interact with data types through a well-defined interface without needing to know how these operations are implemented. This means that users can utilise data structures like lists or stacks based on their functionalities rather than their internal workings.
- For example, when using a stack ADT, a programmer can push or pop elements without understanding whether the stack is implemented using an array or a linked list. This separation of

concerns not only simplifies usage but also allows for flexibility in changing implementations without affecting the user's code.

2.2. Reduced Complexity

- By hiding implementation details, ADTs significantly reduce complexity for developers. Users of an ADT are presented with a simplified interface that exposes only necessary operations, making it easier to understand and use the data structure. This reduction in complexity leads to fewer errors and more straightforward debugging processes.
- For instance, when working with a queue ADT, developers do not have to grapple with the intricacies of how elements are stored or managed internally. They can focus on enqueueing and dequeuing operations, which streamlines development and fosters a clearer understanding of program logic.

2.3. Improved Security

- Information hiding enhances security by restricting access to sensitive data within an ADT. By exposing only necessary methods and keeping data members private, ADTs prevent unauthorised access and modifications. This encapsulation protects the integrity of the data and ensures that it can only be manipulated through controlled interfaces.
- For example, consider a bank account ADT that allows deposits and withdrawals but hides the balance from direct access. This means that external code cannot alter the balance directly; it can only do so through designated methods that enforce rules (e.g., preventing overdrafts). This mechanism not only secures the data but also enforces business logic consistently across the application.

VIII. Imperative ADTs as a Basis for Object Orientation

Imperative Abstract Data Types (ADTs) are foundational to understanding how object-oriented programming (OOP) evolved. ADTs describe the structure and behavior of data types by specifying the operations that can be performed on them, without exposing their internal workings. This is strongly aligned with core principles in OOP, such as encapsulation, information hiding, modularity, reusability, and the procedural approach. Below, we explore the connection between imperative ADTs and OOP principles and how they justify the view that ADTs laid the groundwork.

1. Encapsulation and Information Hiding

Encapsulation and **information hiding** are key concepts both in ADTs and OOP. ADTs in the imperative paradigm define a set of operations that interact with data without exposing how the data is internally

represented. Similarly, in OOP, encapsulation is the process of bundling data (attributes) and methods (operations) into a single unit (class) and restricting direct access to the internal data.

- **Imperative ADTs:** When working with an ADT, users interact with data through a defined set of operations (functions or procedures) without needing to understand or access the data's internal representation. For example, in a stack ADT, users call `push()` or `pop()` without needing to know how the stack is implemented internally (whether it uses arrays or linked lists).
- **OOP:** Encapsulation in OOP formalizes this by explicitly restricting access to an object's internal state through the use of access modifiers (e.g., `private`, `public`, `protected`). This hides the complexity and prevents unauthorized or incorrect modifications to an object's internal data. Information hiding increases reliability by ensuring that data can only be manipulated in controlled ways.

2. Modularity and Reusability

Modularity and **reusability** are central goals in both imperative ADTs and OOP. They refer to breaking down a system into independent, interchangeable modules that are easier to understand, maintain, and reuse.

- **Imperative ADTs:** ADTs promote modularity by providing well-defined operations that operate on specific data structures. An ADT is a self-contained unit with a clear interface, which makes it reusable in different parts of a program. For instance, a stack ADT can be reused in various applications like depth-first search algorithms, expression evaluation, or backtracking algorithms.
- **OOP:** In OOP, classes serve as modular units that encapsulate both data and behavior. These classes can be reused across different contexts. In addition, the concepts of **inheritance** and **polymorphism** allow for even greater flexibility and reusability. A base class defines general behavior, while derived classes can extend or override specific methods to suit specialized tasks.

3. Procedural Approach

The **procedural approach** in imperative ADTs involves defining a sequence of instructions (procedures or functions) that operate on data. This is a precursor to the method-oriented approach in OOP.

- **Imperative ADTs:** In the procedural paradigm, the focus is on the actions or procedures that can be performed on data. Each ADT is accompanied by a set of operations (like `insert()`, `delete()`, `search()` in a list ADT). These procedures define how the ADT behaves but are separate from the data itself.

- **OOP:** In object orientation, the procedural approach is integrated within the objects themselves. Instead of separate procedures, methods belong to the object and operate on its encapsulated data. This shift enables **behavioral encapsulation**, where both data and the procedures that manipulate the data are bundled together.

4. Language Transition

The transition from languages focused on **imperative ADTs** (like C) to **object-oriented languages** (like Java, C++, and Python) reflects a shift in how data and behavior are organized.

- **Imperative Languages:** In languages like C, ADTs are typically implemented using structs and associated procedures. While this is effective, it separates data from the procedures that manipulate it. Programmers must manage access to the data manually, often leading to issues such as data inconsistency and error-prone code.
- **OOP Languages:** Object-oriented languages like Java, C++, and Python introduced formal class constructs to encapsulate data and methods together. This improved data safety through **access control mechanisms** and allowed inheritance and polymorphism to further extend reusability. Object orientation also formalized the relationship between data and behavior, making programs easier to design, understand, and maintain.

IX. Conclusion

In this exploration of how imperative ADTs serve as a foundation for object orientation, we have examined key principles such as encapsulation, information hiding, modularity, reusability, the procedural approach, and the transition from imperative to object-oriented languages. Imperative ADTs promote core software design principles by encapsulating data and operations, protecting internal states, and creating modular units of functionality. These concepts directly influenced object-oriented programming, where encapsulation was formalized through classes and objects, ensuring better data protection, ease of maintainability, and higher reusability.

The procedural approach in ADTs naturally evolved into object-oriented methods that are tied directly to the data within a class, improving cohesion and reducing complexity. Finally, the transition from imperative languages to OOP languages like Java and C++ reflects a refinement of programming models where both data and behavior are integrated, leading to more robust, flexible, and scalable systems.

In conclusion, imperative ADTs laid the groundwork for many of the principles that define object-oriented programming today. They introduced the structure and abstraction needed for designing

complex systems, which OOP built upon to offer a more refined, maintainable, and reusable approach to software development.

X. References

1. Microsoft. "Visual Studio Code." *Visualstudio.com*, Microsoft, 2024, code.visualstudio.com/.
2. JetBrains. "IntelliJ IDEA." *JetBrains*, JetBrains, 2019, www.jetbrains.com/idea/.

GITHUB:

https://github.com/Linhnguyen005/ASM_PART1_NGUYENQUANGLINH_BH01804