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 Truong Nam	**Student ID**	BH00888
Class	SE06301	**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	Nam

Grading grid

P1	P2	P3	P4	P5	P6	P7	M1	M2	M3	M4	M5	D1	D2	D3	D4

☐ Summative Feedback:	☐ Resubmission Feedback:	
Grade:	**Assessor Signature:**	**Date:**
Internal Verifier's Comments:		
Signature & Date:		

Table of Contents

Table of Figures

I. Introduction

This article provides a one-stop solution for understanding key techniques required to write perfect design specifications of data structures. What is commendable is that it accurately describes a range of procedures that can be easily executed on these structures – they give a very deep insight. Moreover, it goes deep into the fundamental concepts of Abstract Data Types (ADTs), and goes through the essential facet of memory management in an implemented program. A particular focus is on the Stack – an elemental data structure managed based on the Last-In, First-Out (LIFO) concept. To culminate this exploration, we present a robust code example that vividly illustrates the fundamental stack operations: particularly: push, pop and isFull adding practical tips on how all of them can be implemented.

II. Contents

Part 1: Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

1. Identify the Data Structures

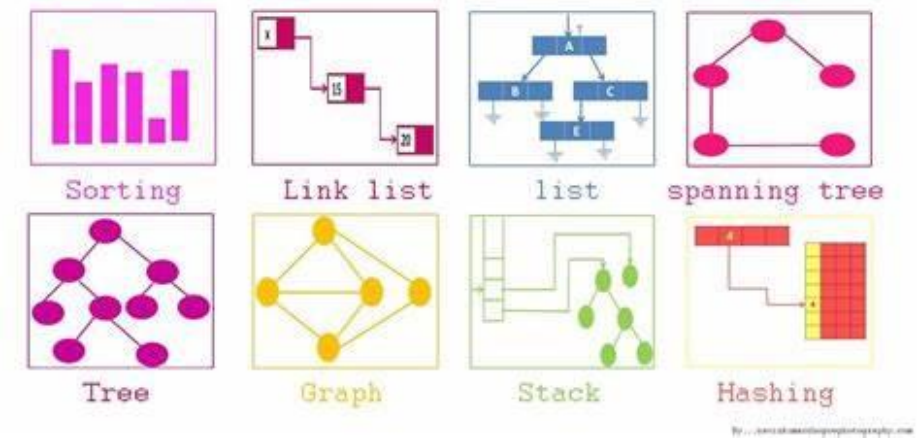


Figure 1:Data Structures

- Arrays/Lists: Ordered collections of elements, often used for sequential access.
- Linked Lists: Elements are nodes with pointers to the next (and sometimes previous) node.
- Stacks: Follows LIFO (Last-In-First-Out) principle.
- Queues: Follows FIFO (First-In-First-Out) principle.
- Trees (Binary, AVL, etc.): Hierarchical structures with nodes and parent-child relationships.

2. Define the Operations

- **Insertion**: Adds an element to a data structure. The insertion position depends on the type of data structure (beginning, end, or specified position).
- **Deletion:** Removes an element from a data structure, usually identified by an index or value.

- **Traversal:** Accesses each element in a data structure to retrieve data or process it.
- **Search**: Searches for an element in a data structure based on a specified value or key.
- **Update**: Modifies the value of an element in a data structure.

Specific Operations for Each Data Structure

Array

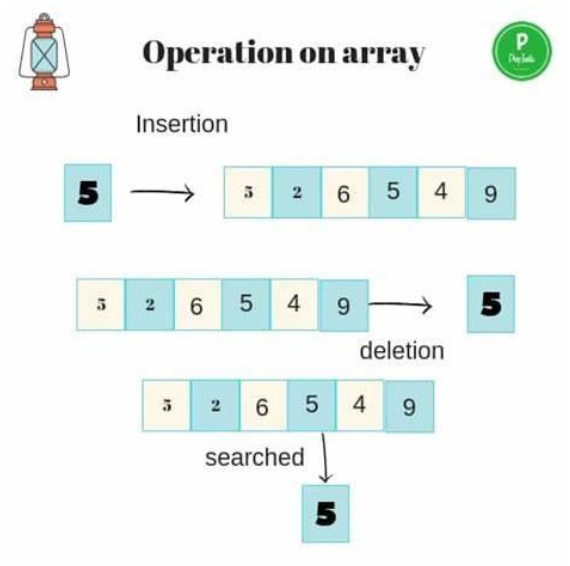


Figure 2:Operation of Array

- **Insertion**: Adds an element at the end or at a specified position. For a fixed-length array, insertion is not possible if it is full.
- **Deletion**: Removes an element at a specified position and shifts subsequent elements to fill the gap.
- **Traversal**: Accesses each element in the array sequentially from start to end.
- **Search**: Linear search (O(n)) or binary search (O(log n)) if the array is sorted.

Linked List

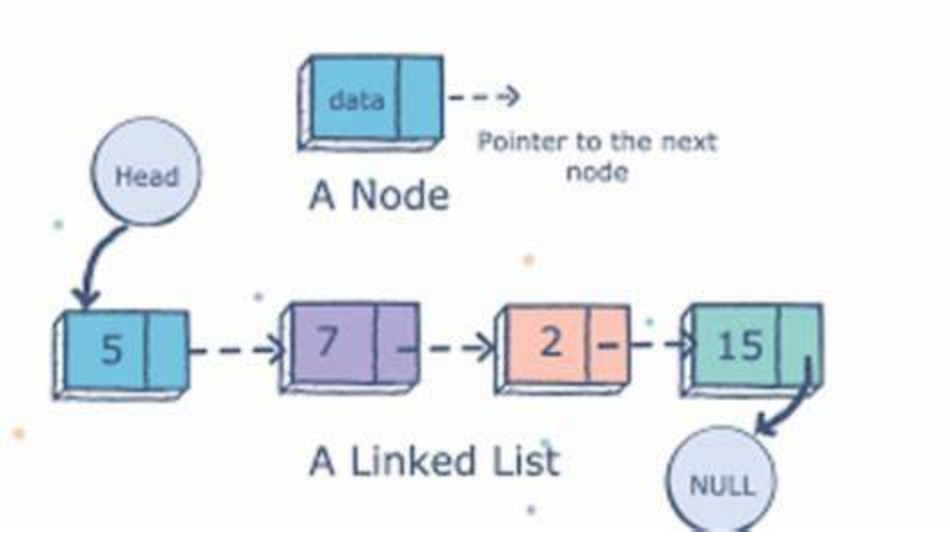


Figure 3:Operations of Linked List

- **Insertion:**
  - Insert at head: Adds an element to the start of the list.
  - Insert at tail: Adds an element to the end of the list.
  - Insert at position: Inserts an element at a specified position by updating links.
- **Deletion:**
  - Delete head: Removes the first element.
  - Delete tail: Removes the last element.
  - Delete at position: Finds and removes the element at a specified position.
- **Traversal:** Accesses each element from the start to the end using pointers.
- **Search:** Finds an element in the list based on a specified value.

Stack

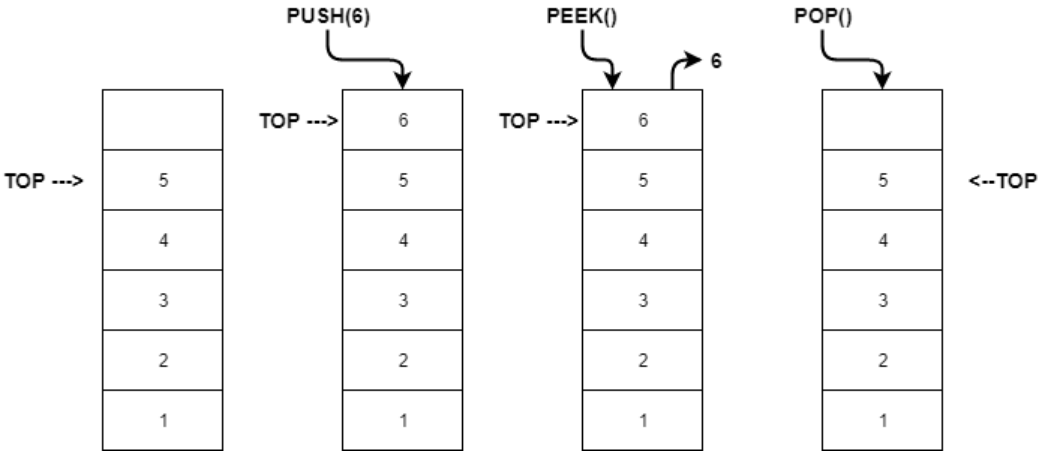


Figure 4:Operations of Stack

- **Push**: Adds an element to the top of the stack.
- **Pop**: Removes and retrieves the element at the top of the stack.
- **Peek**: Retrieves the element at the top without removing it.
- **IsEmpty**: Checks if the stack is empty.

Queue

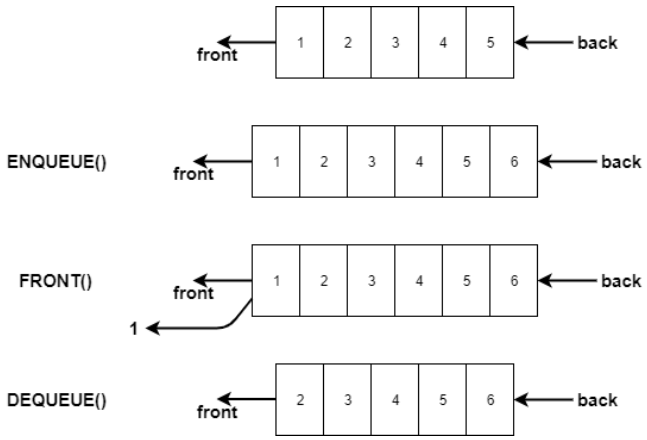
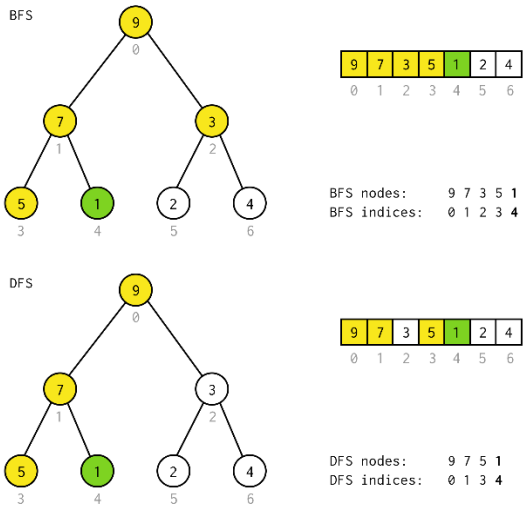


Figure 5:Operations of Queue

- **Enqueue**: Adds an element to the end of the queue.
- **Dequeue**: Removes and retrieves the element at the front of the queue.
- **Peek**: Retrieves the front element without removing it.
- **IsEmpty**: Checks if the queue is empty.

Binary Tree



- **Insertion:** Adds a node to the tree according to binary rules (e.g., binary search trees insert nodes based on their values).
- **Deletion:** Removes a node and restructures the tree to maintain binary properties.
- **Traversal:**
  - Pre-order: Visits the root node first, then the left and right subtrees.
  - In-order: Visits the left subtree, then the root, and then the right subtree.
  - Post-order: Visits the left subtree, then the right subtree, and finally the root.
- **Search:** Finds a node with a specified value.

3. Specify Input Parameters

3.1 Array

- **Insertion**:
  - **Parameters**
    index: The position where the element will be inserted (0-based index)
    value: The value to be inserted.
- **Deletion:**
  - **Parameters**:
    index: The position of the element to be deleted.
- **Traversal**:
  - **Parameters**: None (starts from the first element to the last).
- **Search**:
  - **Parameters**:
    value: The value to search for in the array.

3.2 Linked List

- **Insertion:**
  - **Parameters:**
    value: The value to be inserted.
    position: (Optional) The position where the value should be inserted (e.g., , , or a specific index).headtail
- **Deletion:**
  - **Parameters**:
    value: The value to be deleted, or
    position: The position of the element to be deleted.
- **Traversal:**
  - **Parameters**: None (visits each node sequentially).
- **Search:**
  - **Parameters**:
    value: The value to search for in the list.

3.3 Stack

- **Push:**
  - **Parameters**:
    value: The value to be pushed onto the stack.
- **Pop**:
  - **Parameters:** None (removes the top element).
- **Peek:**
  - **Parameters**: None (returns the top element without removing it).
- **IsEmpty:**
  - **Parameters**: None (checks if the stack is empty).

3.4 Queue

- **Enqueue:**
  - **Parameters**:
    value: The value to be added to the queue.
- **Dequeue:**
  - **Parameters**: None (removes the front element).
- **Peek:**
  - **Parameters**: None (returns the front element without removing it).
- **IsEmpty:**
  - **Parameters**: None (checks if the queue is empty).

3.5 Binary Tree

- **Insertion**:
  - **Parameters**:
    value: The value to be inserted into the tree.
- **Deletion:**
  - **Parameters**:
    value: The value of the node to be deleted.
- **Traversal:**
  - **Parameters**: None (traverses the tree in specified order).
- **Search:**
  - **Parameters**:
    value: The value to search for in the tree.

4. Define Pre- and Post-conditions

- Pre-conditions are the things that must be true before a method is called. The method tells clients “This is what I expect from you".
- Post-conditions are the things that must be true after the method is complete. The method tells clients “This is what I promise to do for you".

Each operation often has requirements for consistency:

- Insertion: Pre-condition – data structure is not at capacity (if fixed size); post-condition – element is present.
- Deletion: Pre-condition – element exists in the structure; post-condition – element is removed.
- Traversal/Search: Pre-condition – structure has at least one element; post-condition – accessed elements or located key/value.

5. Discuss Time and Space Complexity

Data Structure	Operation	Time Complexity	Space Complexity
Array	Access	O(1)	O(n)
	Insertion (end)	O(1)	O(n)
	Insertion (beginning)	O(n)	O(n)
	Deletion (specific)	O(n)	O(n)
Linked List	Access (by index	O(n)	O(n)
	Insertion (head/tail)	O(1)	O(n)
	Insertion (middle)	O(n)	O(n)
	Deletion (head/tail)	O(1)	O(n)
	Deletion (middle)	O(n)	O(n)
Stack	Push	O(1)	O(n)
	Pop	O(1)	O(n)
	Peek	O(1)	O(n)
Queue	Enqueue	O(1)	O(n)
	Dequeue	O(1)	O(n)
	Peek	O(1)	O(n)
Tree	Insertion	O(logn)*	O(n)
	Deletion	O(logn)*	O(n)
	Traversal	O(n)	O(n)

6. Provide Examples and Code Snippets (if applicable)

```
import java.util.Stack;
public class Example {
    public static void main(String[] args){
            //Stack stack1 = new Stack<>();
            Stack<String>stack2 = new Stack<>();
            // stack1.push(1);
            // stack1.push(2);
            // stack1.push(3);
            stack2.push("4");
            stack2.push("5");
```

```
            stack2.push("6");
            // System.out.println(stack1);
            System.out.println(stack2);
            //Pop operation
            System.out.println("Stack after pop operation: "+stack2.pop());
            System.out.println(stack2);
            //peek operation
            System.out.println("The element at the top of the "+"stack is
"+stack2.peek());
            //Empty
            System.out.println("Is stack empty ? "+stack2.empty());

        }
    }
```

Part 2: Determine the operations of a memory stack and how it is used to implement function calls in a computer.

1. **Define a Memory Stack**

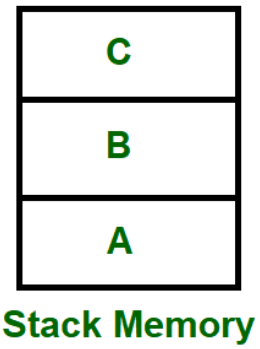


Figure 6:Memory Stack

Stack memory 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.

2. **Identify Operations**

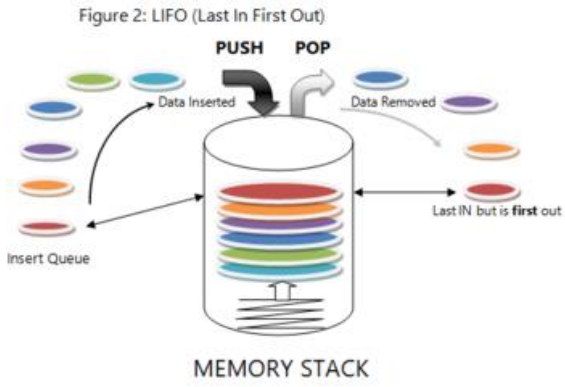


Figure 7:Operations of Memory Stack

A memory stack, also known as a stack, is a data structure that follows the Last-In-First-Out (LIFO) principle. It is used to store and manage data in a particular order. The operations typically associated with a memory stack are:

- Push: The push operation adds an element to the top of the stack. It takes the element to be inserted as an input and places it on top of the existing elements. As a result, the newly added element becomes the new top of the stack.
- Pop: The pop operation removes the top element from the stack. It retrieves the top element and removes it from the stack. After the pop operation, the element below the popped element becomes the new top of the stack.
- Peek or Top: The peek operation retrieves the top element of the stack without removing it. It allows you to examine the element at the top of the stack without modifying the stack itself.
- IsEmpty: The IsEmpty operation checks whether the stack is empty or not. It returns a Boolean value (true or false) indicating whether there are any elements present in the stack.

These operations are fundamental to working with a memory stack and allow you to manipulate the stack's contents by adding, removing, and accessing elements. The stack's LIFO behavior ensures that the last element added is the first one to be removed.

3. Function Call Implementation

Function calls in a computer are handled by means of the stack memory, pointers to instructions, and registers, with most architectures to adhere to one or another calling standard. Here's how it works in detail:

Stack Frames:

- During the time of the calling of the function, space for storing the parameters and local variables of the function is called a stack frame and also the address of the return as well as sometimes

the old value of the register. The stack frame ensures separation when one function is called and when another function ceases the management of memory.

Calling Convention:

- Both architectures make use of a special calling convention by which the behaviors of function calls are predictable. In this convention, it is stated how arguments are passed which is normally through certain registers or stack and where results are stored after the function call or how the stack is cleared after the call is completed.

Pushing Parameters and Return Address:

- Prior to this call the caller pushes parameters to the stack (or passes them in specific registers) and saves the return address (the next instruction to be executed in the calling routine).
- This return address is useful because it can be used for the called function to address the caller once it is done executing.

Instruction Pointer:

- A new value is assigned to the program's instruction pointer (otherwise known as the program counter) to the address of the called function. This shift actually realizes the code belonging to the called function.

Function Execution:

- As we saw in the called function, the stack frame is newly established.
- Local variables are created, and computation is made based on the code of the function.

Return from the Function:

- On finish from the function, the relative address is derived from the stack.
- The function returns by storing the address on the instruction pointer as the next instruction to be executed, and therefore passes back control to whatever called it.

Stack Cleanup:

- The function parameters are removed from stack and the stack frame removing is also done by the calling's convention. This cleanup also ensures the cleanliness of call stack and avoids getting stack overflows or leaks.

4. Discuss the Importance

Function call and local data storage within a stack memory efficiently handles dynamic memory allocations for the system to perform stably. Key points of stack memory include:

- Function call management: Stores the return addresses, local variables, helping function calls know at which area it can return after a function ends.
- Fast memory allocation: The design runs as a LIFO structure where memory allocation and deallocation are fast.
- Automatic variable management: These are automatically allocated at the start of a function and automatically deallocated at the end of that function to minimize the possibility of memory leaks.
- Data isolation: A stack frame that is employed in each of the functions ensures specific datasets are preserved from being overwritten, for other functions.
- Supports recursion: Save each recursive call's state, so functions can call other instances of the same type of function by utilizing the stack.
- Ensures stability: Protects the memory during operations for each function call so that it would not be easily overflowed that adds to its stability.
- Essential for multithreading: Every thread has his own stack, what means that local variables and function calls can be managed independently thus the threads can be safely run concurrently.

Part 3: Illustrate, with an example, a concrete data structure for a First in First out (FIFO) queue.

1. Introduction FIFO

FIFO is an abbreviation for first in, first out. It is a method for handling data structures where the first element is processed first and the newest element is processed last.

A Queue is a data structure that operates on the FIFO (First In, First Out) principle, meaning that the element that comes in first is taken out first. This is like a line of people waiting: the person at the top is served first, then the next person's turn.

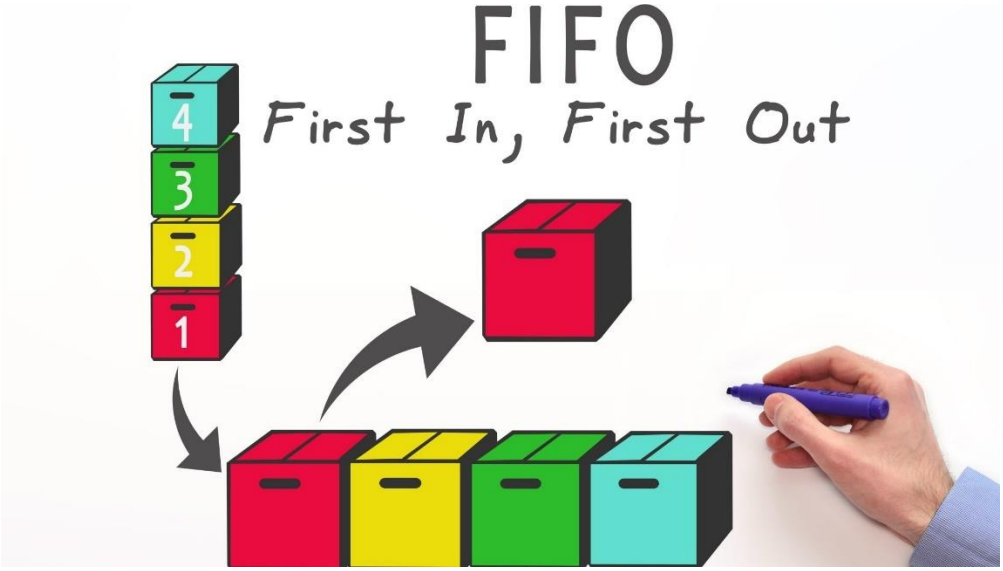


Figure 8:FIFO

2. Define the Structure

- enqueue: Adds an element to the end of the queue.

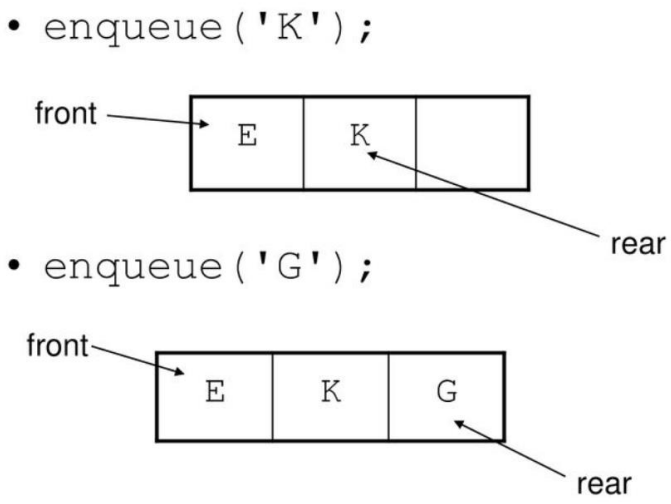


Figure 9:Enqueuce

- dequeue: Removes and returns the element at the head of the queue.

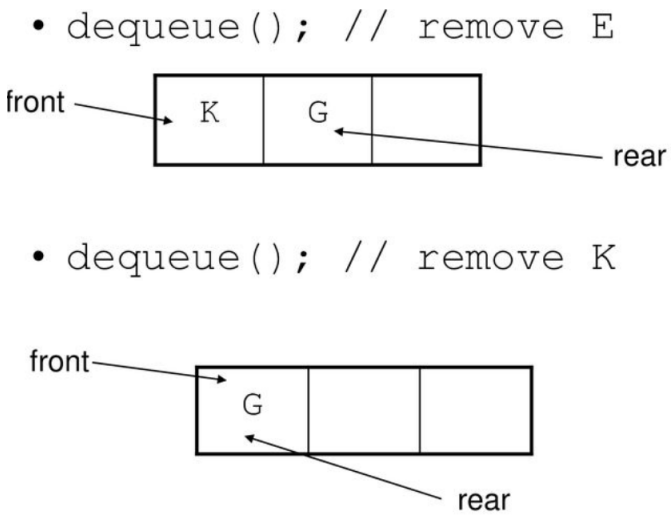


Figure 10:Dequeue

- peek: Returns the element at the head of the queue without removing it.
- isEmpty: Checks whether the queue is empty or not.
- size: Returns the number of elements in the queue.

3. Array-Based Implementation

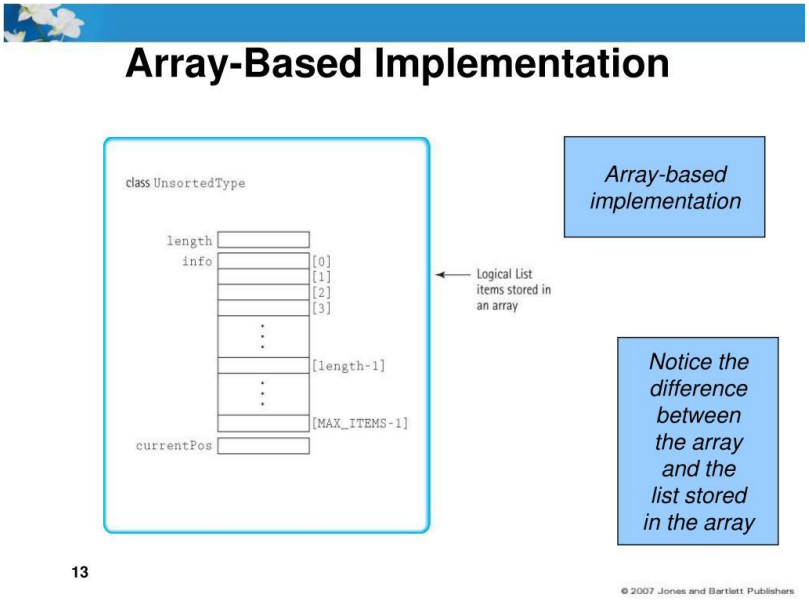


Figure 11:Array-Based Implementation

When using an array for a queue, there is always the need for the first index and last index for the queue. This can be inconvenience especially when the queue is full, a circular buffer can be used so that the queue can reallocate the spare spaces after an element has been deleted.

4. **Linked List-Based Implementation**

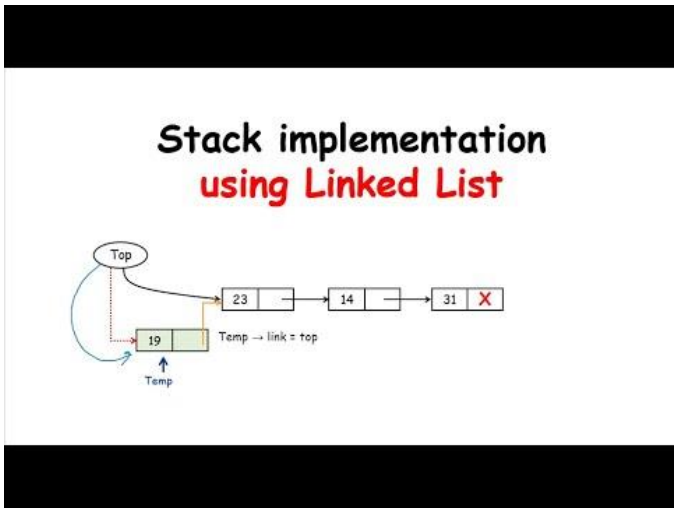


Figure 12: Linked List-Based Implementation

Queue can also be implemented using Linked List. Each element is stored in a node and has two indices: front and rear. This method is more memory efficient than arrays because as we don't have any predefined size.

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

Example:

It's just like a line at a ticket counter of a movie theater where people wait for their turn to be served. Individuals are processed in a first come first served system and each attends to the server one at a time.

- Customer A appears first and stays in the foremost point of the queue.
  - Queue: [A]
- Customer B gets in the line right behind Customer A.
  - Queue: [A, B]
- Customer C then joins the process and stands next to customer B.
  - Queue: [A, B, C]
- The ticket seller moves to the first customer, Customer A following the first come first served basis.
  - Queue after A is served: [B, C]
- Customers D gets to the counter and stands closer to Customer C.
  - Queue: [B, C, D]
- Although the ticket seller interacts with Customer A, the interaction with the next customer, that is, Customer B is also recorded.
  - Queue after B is served: [C, D]

- Last but not the least, Customer C is served next.
  - Queue after C is served: [D]

Final State of the Queue

For every client gets served after the other in the order they were offered the seats, the only one still waiting is Customer D. This follows the FIFO principle: "First In, First Out."

Part 4: Compare the performance of two sorting algorithms.

1. **Introducing the two sorting algorithms you will be comparing**
   a. Quick Sort: This is a partitioning type and recursive sorting Technique, it is also called the quick sorting technique. Quick Sort takes an element (called a “pivot”) and divides the array into two parts: About pivot: All elements smaller than pivot are located to its left; all the elements greater than pivot are located to its right. After that the process is repeated on each of its parts until the array is wholly sorted.

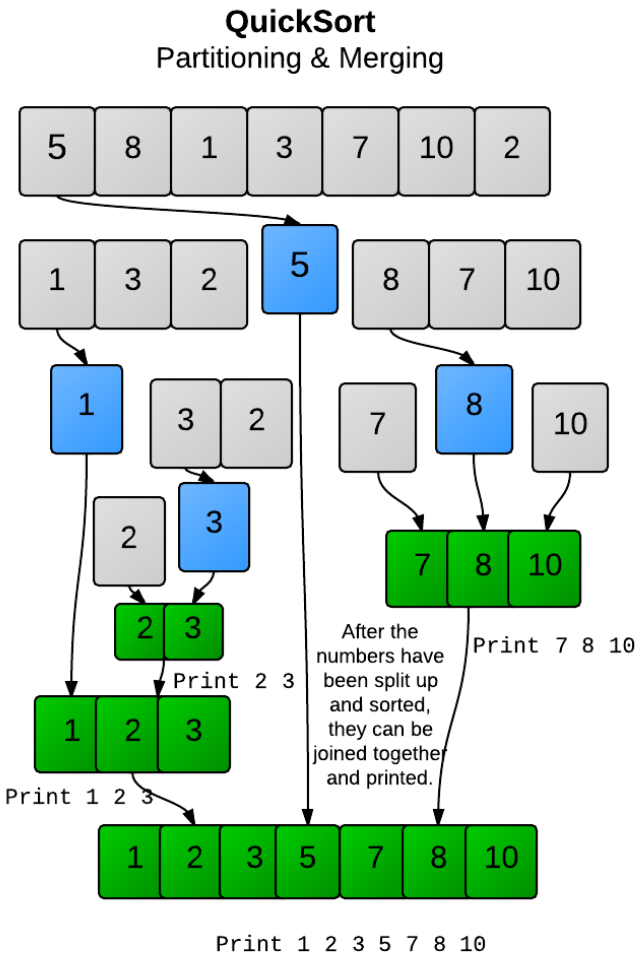


Figure 13:Quick Sort

b. Merge Sort: This is a divide and conquer algorithm. Merge Sort divides the array into two halves, then sorts each half and merges them in order. This algorithm also works recursively.

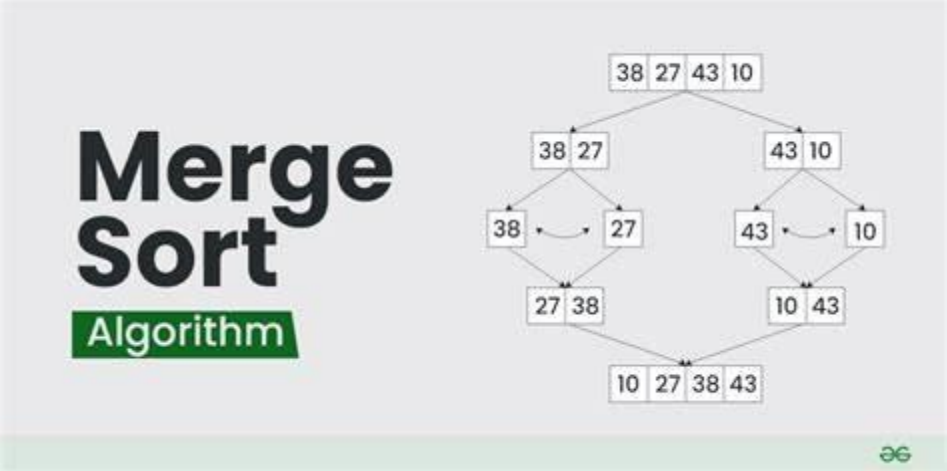


Figure 14:Merge Sort

2. Time and Space Complexity Analysis

Algorithm	Average Time Complexity	Best Time Complexity	Worst Time Complexity	Space Complexity
Quick Sort	O(n log n)	O(n log n)	O(n^2)	O(log n)
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)

Explain:

Quick Sort:

- Average Time: Many of the time on average Quick Sort takes its best time complexity of O (n log n) O(nlogn) because the array is divided into two halves for partitioning.
- Best Time: If the pivot is selected such a way that both the partitions are equal then Quick Sort takes the minimum possible time that is O (n log n).
- Worst Time: If the pivot selects the largest or the smallest element (subpartitioning leads to highly unbalanced subarrays of size s as opposed to p), the runtime can be O (n 2) O(n 2).
- Space: As Quick Sort is in-place (does not use an auxiliary array), it occupies O (log n) O(logn) spaces only because of call stack.

Merge Sort

- Average and Best Time: Merge Sort always has O (n log n) O(nlogn) time complexity because it consistently divides and merges the array evenly.
- Worst Time: Merge Sort still maintains O (n log n) O(nlogn) efficiency regardless of the initial arrangement of the array.
- Space: Merge Sort requires O (n) O(n) extra space to store elements during the merging process.

3. Stability

a. Quick Sort

Quick Sort is an unstable sorting algorithm. This is because it always take an element and sort the elements with respect to or around that particular selected element, which we call as 'pivot.

- **How it works**: Quick sort partitions the array on the basis of pivot element under which all elements smaller than pivot is on left and all elements greater than pivot is on right. This means that while shifting elements in the array other elements of equal value may assume the opposite order to that which they had initially.
- **Result**: hence the relative sequences of equal values are not preserved and hence quick sort can be categorized as unstable .

b. Merge Sort

Merge Sort is considered a stable sorting algorithm. The reason is to be sought in the method employed for dividing and remixing the elements at issue.

- **How it works:** Merge Sort works by dividing the array into two or more sub-arrays then sort each of the sub-arrays. Then the sorted two subarrays will be combined Then the sorted two subarrays Then, two new sorted subarrays Then merged. When merging, it means if two elements from the sub-arrays have equal values they meant that their first appearance will be in that order in the given array.
- **Result**: Hence, the order of equal elements will not be changed and that is why we can classify Merge Sort as a stable sorting algorithm

4. Comparison Table

Criteria	Quick Sort	Merge Sort
Time Complexity	Average: O(n log n) Worst-case: O(n^2)	O(n log n)
Space Complexity	O(log n)	O(n)
Stability	Not stable	Stable
Suitable for Arrays	Small to medium arrays	Large arrays, data needing stability
Execution Time	Faster in most cases	Slightly slower than Quick Sort

5. Performance Comparison

Quick Sort has lesser mean time complexity $O(n \log n)$ and better space complexity than those of insertion sort, and is not stable sort but can perform O (n ^2) in the Worst Case.
Merge Sort is stable, and performs well, on an average, as O (n log n) and therefore is used when a word of the algorithm's stability, however, requires additional memory.
So in case, the speed and memory are critical, then implement Quick Sort with good pivot. When stability is required and sort requires no extra memory then Merge Sort should be used.

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

   Suppose we have an array of 1 million integers, sorted in different states:

   - Random array: Elements are sorted randomly.
   - Ascending sort table: Elements are sorted in ascending order.
   - Ascending sort table: Elements are sorted in descending order.

   For these cases, we will compare the execution time of Quick Sort and Merge Sort to see the difference:

   Quick Sort:

   - Random case: Quick Sort usually works very efficiently with an average time complexity of O (n log n). Quick Sort has the advantage of using less memory space because it is an in-place (in-place) sorting algorithm.
   - Ascending/Descending sorted merged array: Quick Sort can have problems with choosing the wrong axis, especially if it always chooses the axis at the first or last array. In the worst case, the time will be O (n ^2) .

   Merge Sort:

   - All cases: Merge Sort always has a complexity of O (n log n) , regardless of the initial arrangement of the array. Merge Sort is more stable and performant than Quick Sort, especially with random or sorted data. However, the downside of Merge Sort is that it requires an additional space of O (n) O(n) to temporarily store the array when splitting and merging

Part 5: Analyses the operation, using illustrations, of two network shortest path algorithms, providing an example of each.

1. **Introducing the concept of network shortest path algorithms**

Network shortest path algorithms are employed and can identify the shortest path between two nodes on a graph. This is basic to many problems such as routing in telecommunication networks, Global Satellite Positioning System, and data telecommunication. A graph in this case is a collection of nodes, entities that are placed in a network or devices, and edges which depict the connection between the nodes together with the weight that may be a distance between them.

Shortest path algorithms determining the cheapest path between two nodes with reference to distance, time or any other parameter in accordance with the specific usage. Of these algorithms, two most widespread types are called Dijkstra's Algorithm and the Prim-Jarnik Algorithm.

2. **Algorithm 1: Dijkstra's Algorithm**

The proposed algorithm is a special type of the greedy algorithm called Dijkstra's Algorithm, which is devised by Edsger Dijkstra; it locally determines the shortest paths between the nodes in the weighted

graph and the given source node. That is why it is assumed that edge weight must be non-negative; it is perfect for routing and distance minimal problems.

Steps of Dijkstra's Algorithm

- Initialization: Even for the first node, its distance is set to 0 and that of all other nodes to a very large number or, in strict standard, to infinity.
- Visit Nearest Node: The step is choosing the square that was not visited and has the least total distance proposed or calculated to it.
- Update Neighboring Nodes: For each neighbor of the current Node if path from this current Node to this neighbor is shorter than any recorded path to that neighbor, set it equal to that shorter path.
- Repeat: Keep going to the nearest unvisited node found and adjust its distances as applicable until all nodes are visited or the target node is reached.

3. Algorithm 2: Prim-Jarnik Algorithm

The Prim-Jarnik Algorithm widely known as the Prim's Algorithm is one of the minimum spanning tree (MST) algorithms. Although this algorithm is not used to find the shortest path between two nodes, it's important when finding the least cost to connect all the nodes, which is helpful in connection of nodes at least cost which is helpful in network connection. We are including it here for the sake of reference, and for the difference of functioning from the original Dijkstra's.

Like Kruskal's Algorithm, Prim's Algorithm is also based on the Greedy Approach, it goes for edges with lowest weights to form an MST. Similar to Dijkstra's it assumes non negative edge weights.

Steps of Prim's Algorithm

- Initialization: Choose any initial node and place in MST set.
- Select Minimum Edge: Out of the number of nodes associated with the MST first select the minimum weight edge which links the MST set to a new node that has not been incorporated in to the MST.
- Add Node to MST: Include the selected node and edge to MST- set.
- Repeat: They are continued until all of the nodes in the graph are incorporated in the MST

4. Performance Analysis

Algorithm	Purpose	Optimal for	Time Complexity	Applications
Dijkstra	Shortest path from a single source	Directed or undirected graphs with non-negative weights	O((V+E)logV) (with heap)	GPS navigation, internet routing

Prim-Jarnik	Minimum spanning tree	Connected, undirected graphs with non-negative weights	O(ElogV)	Network design, electrical grids

While both of these algorithms are quite similar in their general approach, their ultimate goals are quite different from each other and they form a perfect pair. Dijkstra is meant to look for the best of distance from a source while Prim is meant to make certain the minimal total cost of the network. The knowledge about what algorithm has to be used, how it has to be implemented, and how complex the implementation of particular algorithm would be enables one to select the right algorithm to be used in given network.

Part 6: Specify the abstract data type for a software stack using an imperative definition.

1. **Define the data structure**

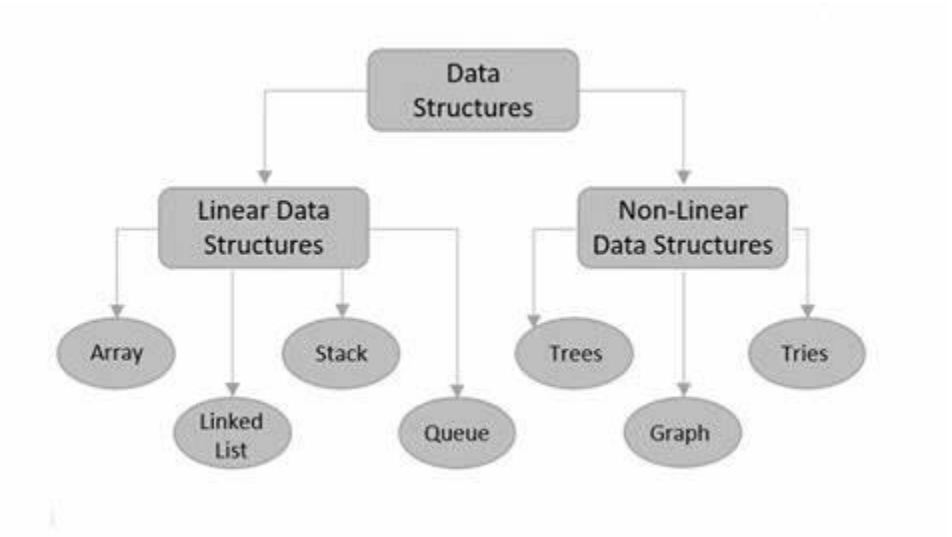


Figure 15: Define the data structure

A stack is a linear data structure that follows the Last In, First Out (LIFO) principle. This means that the most recently added element is the first one to be removed. Stacks are abstract data types with specific operations:

- Push: Adds an element to the top of the stack.
- Pop: Removes and returns the element from the top of the stack.
- Peek (Top): Retrieves the top element without removing it.
- isEmpty: Checks if the stack is empty.

Real-world Applications of Stack

- Function Call Management: Stacks store function calls in programming languages to handle nested or recursive function calls, known as a "call stack."

- Undo Mechanisms: Applications like text editors use stacks to track the most recent actions to enable undoing.
- Expression Evaluation and Syntax Parsing: Compilers use stacks to parse expressions or check for balanced parentheses in expressions.

Stacks can be implemented in two main ways:

- Array-based implementation: Using a fixed-size array.
- Linked-list-based implementation: Flexible handling of sizes- use of linked nodes.

2. Initialize the stack

In order to create a stack, we must import **java.util.stack** package and use the Stack() constructor of this class. The below example creates an empty Stack

```
import java.util.Stack;

public class Main {
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();

    }
}
```

3. Push operation

In order to add an element to the stack, we can use the push() method. This push() operation place the element at the top of the stack.

```
import java.util.Stack;

public class Main {
    public static void main(String[] args) {
        Stack stack1 = new Stack();
        Stack <String> stack2 = new Stack<String>();
        stack1.push( item: "1");
        stack1.push( item: "2");
        stack1.push( item: "3");
        stack2.push( item: "4");
        stack2.push( item: "5");
        stack2.push( item: "6");
        System.out.println(stack1);
        System.out.println(stack2);

    }
}
```

Output:

```
[1, 2, 3]
[4, 5, 6]
```

4. **Pop operation**

   To pop an element from the stack, we can use the pop() method. The element is popped from the top of the stack and is removed from the same.

```java
import java.util.Stack;
public class Main {
    public static void main(String[] args) {
        Stack <String> stack2 = new Stack<String>();
        stack2.push( item: "4");
        stack2.push( item: "5");
        stack2.push( item: "6");
        //Display the stack
        System.out.println(stack2);
        //Pop operation
        System.out.println("Stack after pop operation: " + stack2.pop());
        System.out.println(stack2);
    }
}
```

   Output:

```
[4, 5, 6]
Stack after pop operation: 6
[4, 5]
```

5. **Peek operation**

   To retrieve or fetch the first element of the Stack or the element present at the top of the Stack, we can use peek() method. The element retrieved does not get deleted or removed from the Stack.

```java
import java.util.Stack;

public class Main {
    public static void main(String[] args) {
        Stack <String> stack2 = new Stack<String>();
        stack2.push( item: "4");
        stack2.push( item: "5");
        stack2.push( item: "6");
        //Display the stack
        System.out.println(stack2);
        //Fetching the element at  the head of the stack
        System.out.println("The element at the top of the "+"stack is: "+stack2.peek());

    }
}
```

   Output:

```
[4, 5, 6]
The element at the top of the stack is: 6
```

6. **Check if the stack is empty**

   It returns true if nothing is on the top of the stack. Else, returns false.

```java
import java.util.Stack;
public class Main {
    public static void main(String[] args) {
        Stack <String> stack2 = new Stack<String>();
        stack2.push( item: "4");
        stack2.push( item: "5");
        stack2.push( item: "6");
        //Display the stack
        System.out.println(stack2);
        //Pop operation
        System.out.println("Is stack empty?  \n"+  stack2.empty());
        System.out.println(stack2);
    }
}
```

Output:

```
[4, 5, 6]
Is stack empty?
false
```

7. **Full source code(stack using Array)**

```
public class ArrayStack {
   private int[] stack;
   private int top;
   private int capacity;

   public ArrayStack(int size) {
      stack = new int[size];
      capacity = size;
      top = -1;
   }

   public void push(int value) {
      if (top == capacity - 1) {
         System.out.println("Stack is full");
         return;
      }
```

```
    stack[++top] = value;
}

public int pop() {
    if (isEmpty()) {
        System.out.println("Stack is empty");
        return -1;
    }
    return stack[top--];
}

public int peek() {
    if (!isEmpty()) {
        return stack[top];
    }
    System.out.println("Stack is empty");
    return -1;
}

public boolean isEmpty() {
    return top == -1;
}

public static void main(String[] args) {
    ArrayStack stack = new ArrayStack(5);

    // Push values onto the stack
    stack.push(10);
    stack.push(20);
    stack.push(30);

    // Peek at the top value
    System.out.println("Top value: " + stack.peek());

    // Pop values from the stack
    System.out.println("Popped from stack: " + stack.pop());
    System.out.println("Popped from stack: " + stack.pop());

    // Check if the stack is empty
    if (stack.isEmpty()) {
        System.out.println("Stack is empty");
    } else {
        System.out.println("Stack is not empty");
    }
}
```

```
}
```

8. Full source code (stack using LinkList)

```
public class StackLinkedList {
   // Node class represents each element in the stack
   class Node {
      int data;
      Node next;

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

   // LinkedStack class represents the stack using linked list
   class LinkedStack {
      private Node top;  // The top of the stack

      public LinkedStack() {
         this.top = null;
      }

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

      // Add an element to the stack
      public void push(int data) {
         Node newNode = new Node(data);  // Create a new node
         newNode.next = top;           // Link the new node with the current top
         top = newNode;               // Update top to be the new node
      }

      // Remove and return the top element from the stack
      public int pop() {
         if (isEmpty()) {             // If the stack is empty
            System.out.println("Stack is empty.");
            return -1;
         }
         int value = top.data;         // Store the top value
```

```
        top = top.next;              // Move top down to the next node
        return value;               // Return the popped value
    }

    // View the top element of the stack without removing it
    public int peek() {
        if (isEmpty()) {
            System.out.println("Stack is empty.");
            return -1;
        }
        return top.data;
    }
  }

  // Main method to test stack operations
  public static void main(String[] args) {
    StackLinkedList outer = new StackLinkedList();  // Create an instance of the outer class
    LinkedStack stack = outer.new LinkedStack();    // Create a stack instance

    stack.push(5);
    stack.push(10);
    stack.push(15);

    System.out.println("Top element is: " + stack.peek()); // View the top element
    System.out.println("Popped element: " + stack.pop());  // Remove an element from the stack
    System.out.println("Top element after pop: " + stack.peek()); // View the new top after popping
  }
}
```

9. Comparing

Feature	Array-based Stack	Linked-list-based Stack
Memory	Fixed size, set during initialization	Dynamic size, grows with new elements
Time Complexity	O(1) for push, pop, and peek	O(1) for push, pop, and peek
Implementation	Simple, requires bounds check	Slightly complex, requires node handling
Memory Usage	May waste memory if capacity isn't fully utilized	Efficient for large numbers of elements

Part 7: Examine the advantages of encapsulation and information hiding when using an ADT.

1. Encapsulation

a. What is encapsulation

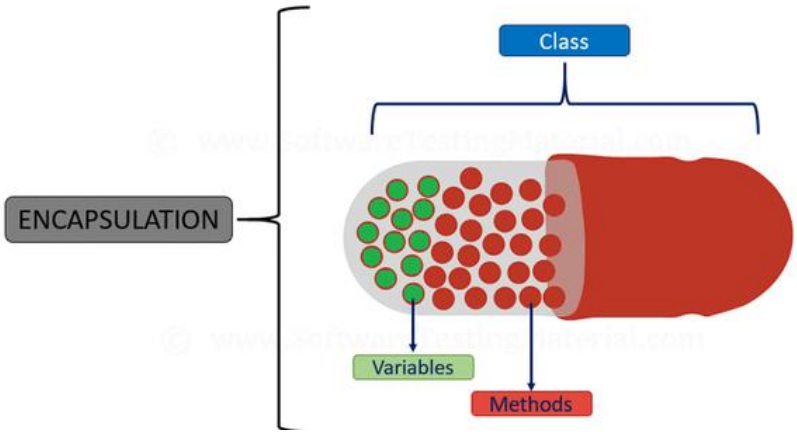


Figure 16:Encapsulation

Encapsulation is the concept of bundling data (variables) and methods that operate on the data into a single unit, called a class. By hiding the internal details and exposing only necessary methods, encapsulation prevents outside interference and misuse of an object's data. This ensures that objects manage their own state and changes to data occur in a controlled manner.

- Example: In a class BankAccount, the balance is a private variable, and methods like deposit() and withdraw() are provided to update the balance. This restricts access to the balance variable from outside the class, ensuring that changes are only made through predefined operations.

b. Example of encapsulations

```
public class Exampleofencapsulations {
    private int[] elements;
    private int top;
    private int maxSize;

    public Exampleofencapsulations(int size) {
        elements = new int[size];
        maxSize = size;
        top = -1;
    }

    public void push(int value) {
        if (top < maxSize - 1) {
            elements[++top] = value;
        } else {
            System.out.println("Stack overflow");
        }
    }

    public int pop() {
```

```
        if (top >= 0) {
            return elements[top--];
        } else {
            System.out.println("Stack underflow");
            return -1;
        }
    }

    public boolean isEmpty() {
        return top == -1;
    }
}
```

c. Data Protection

Encapsulation protects data by preventing direct access to the variables of the object, because access to them is only allowed through the methods of the class. Private access modifiers help to hide important information and make their view possible only through defined ways. This approach to programming cuts off certain frivolous changes which are likely to bring about instabilities in the code base by lowering or removing their securities altogether.

- Example: In the BankAccount class, and if balance is private and can only be accessed through methods such as deposit() and withdraw(), and other functionalities, it means you have the flexibility of putting some extra work (e.g., implementing code to prevent a negative balance) and also, and most importantly, nobody else is going to come and change the balance.

d. Modularity and Maintainability

Encapsulation improves modularity because related data and functions are packaged in different classes thus complex system are easy to comprehend and manage. …modification of one of the classes does not impose its demands on other classes as long as the interface has remained unaltered, this fosters elasticity.

- Example: In large systems encapsulation helps the developer to change the implementation of a class (for example, the way Stack is implemented on the device's disk) while not affecting the code that uses the class. This separation of concerns proves to optimize testing, update and debugging of any code developed.

e. Code Reusability

Encapsulation also makes code reusable by development of boundaries that can be implemented in one part of the program or in another project. That is why encapsulated classes are not an exception: they do not require sharp modifications when reused.

- Example: When the Stack ADT has been implemented, the 'template' can be used for many projects in which a stack data structure is needed. Since it contains all data management internally, there is no need for other parts of a program to understand how push(), pop() or even isEmpty() work.

2. **Information Hiding**

a. Abstraction

Abstraction is closely related to information hiding. It focuses on exposing only the relevant features and functionalities of an object while concealing unnecessary details. This simplification allows users to interact with complex systems without needing to understand their inner 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.

b. Reduced complexity

By hiding unnecessary details, information hiding helps reduce the complexity of software systems. Developers can interact with objects at a high level without needing to understand the intricate details of their implementations. This leads to cleaner and more manageable code.

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 enqueuing and dequeuing operations, which streamlines development and fosters a clearer understanding of program logic.

c. Improved Security

Information hiding contributes to improved security by preventing unauthorized access to sensitive data. By restricting access to an object's internal state, it reduces the likelihood of unintended modifications or misuse.

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.

Part 8: Discuss the view that imperative ADTs are a basis for object orientation offering a justification for the view.

1. **Encapsulation and Information Hiding**

Data encapsulation or data hiding is an important property of ADTs as well as Object Oriented Programming. In imperative ADTs, data hiding is the bundling of data with operations on the data making the operations the only means of accessing the data. This saves data because direct access to

the data is prohibited, which enhances the security of the data. The concept is like private fields and public methods used in many object-oriented languages where an objects data state can only be modified or referenced using specially set methods.

For example:

- ADT corresponding to a queue may include enqueue and dequeue but does not include direct access or modification of the list of objects in the queue. However, any manipulation of the data can only be done within these methods.

Reason: The shielding or encapsulation of data in ADT structures makes classes and objects in OO. As it is the case with ADTs in OO, objects prevent outside interference and offer the possibility of regulated access. This not only makes code secure and more stable to avoid unauthorized access to data by directly interfacing with it.

2. Modularity and Reusability

The use of ADTs also promotes modularity and an important structure to these functions, which is based on the data they handle, promotes modularity. Every ADT is a standalone unit on its own, is implemented to perform a particular task and holds detailed implementation of the same task. This makes the work of these modules easily reusable in other parts of a program or in other programs altogether.

For example:

- An ADT like Stack can be implemented and used in many different tasks such as mathematical expression processing, or in undo/ redo navigation, or in any parsing algorithms. Each time as needed; the ADT can hence just be called without having to reconstruct it from spade since its comprehensive structure is already set.

Reason: Modularity that is in ADTs is the basis of classes and modules in OO. OO carries forward this concept and adds inheritance and polymorphism whereby one class can be inherited and/or reused in more selective manners. This makes it easier to extend the source code in OO more reusable without altering the previous source codes.

3. Procedural Approach

Imperative ADTs are fully implemented in an imperative manner in that the program is consistently implemented as procedures or functions that work on the data. While OO employs objects and methods, both are similar in the sense that they entail well defined operations for the accomplishment

of specific missions. The procedural mechanisms in ADTs give a direct set of instructions on how to operate on data – something that OO builds on by use of methods.

For example:

- Suppose an ADT is List that has other operations as add, remove, and get. These ones are directly called in the program to perform the operations in the correct sequence. This is the same way we have Object in OO invoking methods to affect a behavior.

Reason: The procedural style in ADTs is the building block of methods in OO. Since, ADT prescribes defined operations on data, they provide a sequential format of data processing, which OO further builds upon by linking such ops to the stages of object life cycle, resulting into relatively easier modeling of entities and how they behave.

4. Language Transition

It can be stated that numerous pivotal imperative programming languages comparable to C along with Pascal changed into object-oriented languages comparable to C++ and Object Pascal; within which ADTs are structural. ADTs assumed responsibility of migrating programmers from procedural style to OO style of coding by offering data structures complete with routines which shifted the programmer to OO languages when features like inheritance and polymorphism were incorporated

For example:

- Courses in C++ might begin with writing ADTs for structures that include linked lists, stacks, and so on and then add classes and objects. These ADTs may initially be just data structures and functions, and much later classes that may have inheritance and polymorphic method

Reason: As seen above, ADTs offered a familiar environment which procedural programmers could easily transfer to OO environment. Due to the characteristic of ADTs being designed to be both modular and abstract, when OO came into existence, it only extends from this model, which made it easier for programmer to adapt getting closer to OO.

III. Conclusion

In this presentation I included the basics of Abstract Data Types (ADTs) and explained the approach and operations on this type. Also, I made an elaborate effort of giving an example of how a Linked List can be implemented, thus expanding on the idea of an ADTs. In the future, the section of this book reviewing Memory Layout and Stack Frames allowed me to code for stack frames and also acquire a better understanding of the memory layouts. The information about stack definition and the examples of stack usage in real life were interesting. Although some mistakes and absences of the information are occurred this article, I have benefited from it in one way or the other. During the testing process themselves I met

some difficulties and found a number of ideas that were seemed to me quite ambiguous. Fortunately friends and teachers were supportive and the challenge enabled me to complete them and enhance my understanding of ADTs and Stacks.

IV. References

1. InquisitiveInquisitive 7 and duffymoduffymo 308k4646 gold badges374374 silver badges565565 bronze badges (1957) *What are the differences pre condition ,post condition and invariant in computer terminology*, *Stack Overflow*. Available at: https://stackoverflow.com/questions/11331964/what-are-the-differences-pre-condition-post-condition-and-invariant-in-computer .
2. *Stack memory* (no date) *Stack Memory - an overview | ScienceDirect Topics*. Available at: https://www.sciencedirect.com/topics/engineering/stack-memory.
3. GeeksforGeeks (2022) *FIFO (first-in-first-out) approach in programming*, *GeeksforGeeks*. Available at: https://www.geeksforgeeks.org/fifo-first-in-first-out-approach-in-programming/ .
4. GeeksforGeeks (2024) *Encapsulation in Java*, *GeeksforGeeks*. Available at: https://www.geeksforgeeks.org/encapsulation-in-java/ .
5. Link Github: https://github.com/namnt2004/ASM_DSA.git