A black background with a black square

Description automatically generated with medium confidence

**Investigate and Implement Complex Data Structures and Algorithms to Solve Real World Problems**

Raneem Yahya Sa’deh

Student ID: 22210016

Instructor Name: Dr. Balqees Aldabaibeh/ Eng. Asmaa

­

Course Name: Data Structures & Algorithms

HTU Course No: 40201201

BTEC Unit No: T/618/7430

Academic Year Spring 2023/2024

A grey logo on a black background

Description automatically generated with low confidence

Contents

[Part 1 4](#_Toc168775384)

[Task 1 – Logical Level 4](#_Toc168775385)

[**2.** **What is a Stack?** 8](#_Toc168775386)

[Task 2 - Application (user) level 10](#_Toc168775387)

[**1.** **Advantages of Encapsulation and Information Hiding When Using an ADT** 10](#_Toc168775388)

[Task 3 - Implementation (concrete) level 12](#_Toc168775389)

[1. 12](#_Toc168775390)

[2. 17](#_Toc168775391)

[3. Task and Category ADTs 18](#_Toc168775392)

[Part 2 24](#_Toc168775393)

[2. **Memory Stack Frames in Recursive Merge Sort** 27](#_Toc168775394)

[3. Big-O Notation and Asymptotic Analysis 31](#_Toc168775395)

[4. 4. Alternative Methods for Measuring Efficiency 31](#_Toc168775396)

[Part 3 33](#_Toc168775397)

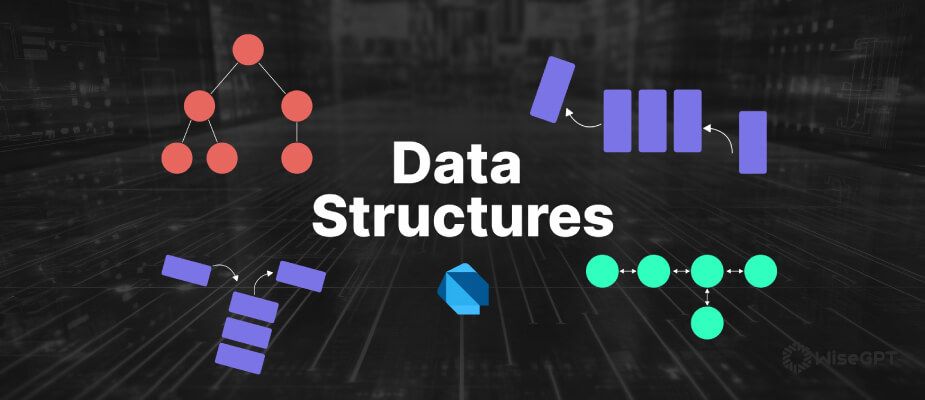
[Complexity Evaluation 34](#_Toc168775398)

[Complexity Evaluation 39](#_Toc168775399)

[1. 40](#_Toc168775400)

[2. 40](#_Toc168775401)

[3. 40](#_Toc168775402)

*****“Within the broad field of computing, where efficiency and organization are very important, data structures become the foundation of digital architecture. It is complex frameworks that are carefully created to maximize data management and expedite computing processes, not just warehouses of information.”*

**What is Data Structure?**

It is the storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently. Depending on your requirement's and project, it is important to choose the right data structure for your project. For example, if you want to store data regularly in memory, then you can go for the array data structure.

**Importance of Data Structures:**

|  |  |
| --- | --- |
| Good Data Management | Organizing data in a structured way with data structures enables quick storing and retrieving, and it make the speed of the process and the performance of the system better. |
| Data Organization | By providing a logical connection to data, the data structure will promote clarity and complexity by making it easier to understand and navigate. |
| Data Abstraction | It is going to protect programmers from dealing with the complex details of how data is stored, allowing them to concentrate on the fundamental concepts of manipulating data. |
| Algorithm Optimization | When you choose the data structure perfectly for the problem that you want to solve, it will effect the effectiveness and efficiency of the algorithm in the data. |

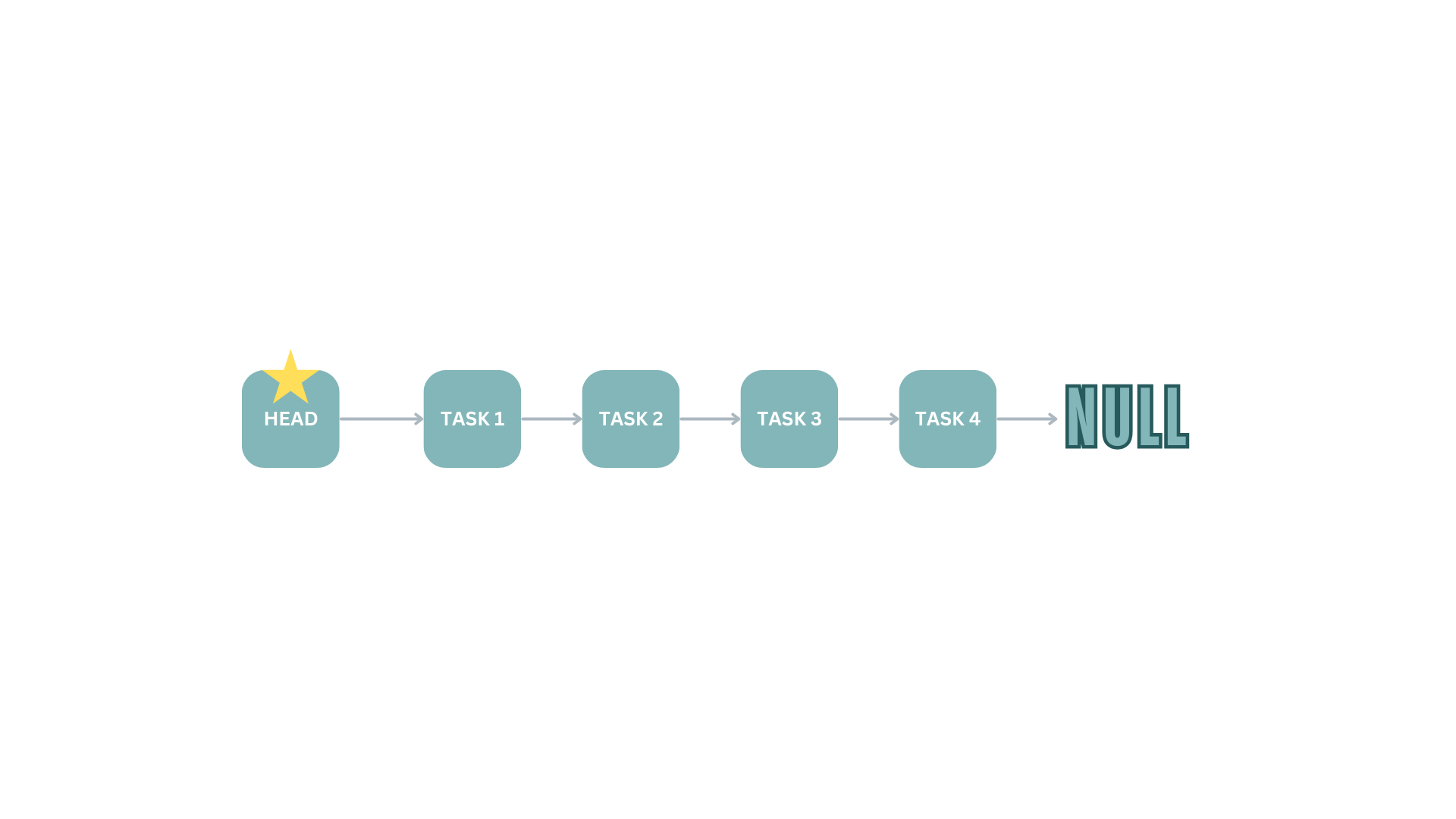
**Theories**

|  |  |
| --- | --- |
| Data Structure | method for arranging data to ensure effective use. Binary trees, linked lists, and arrays are examples of common data structures |
| Algorithm | series of detailed instructions for resolving a particular issue |
| Time Complexity | measurement of how long an algorithm takes to execute, based on how much data it is processing |
| Big O Notation | used to explain an algorithm's time complexity, it’s a notation used in mathematics to represent how a function limits when the argument tends towards a specific value or infinity |
| Recursion | method of programming where a function calls itself |

# Part 1

## Task 1 – Logical Level

1. **Explaining the ADT’s:**

* **Linked List:**

The singly linked list in the taskmanager package is designed to manage the tasks from the users.

**Attributes:**

1. It has got two private variables, `head` and `tail`, which point to the first and last nodes of the list.

**Methods:**

1. The `isEmpty` method checks whether the list is empty by confirming if the `head` is null.
2. The `addLast` method creates a new node for task where it will be added at the end of the list, if the list was empty this new node becomes both the head and the tail. If not, the method updates the current tail's `next` reference to point to the new node and then updates the tail to be the new node.
3. The `removeFirst` method removes the first node in the list. If the list is empty, it prints a message saying that the list is empty. If not, It will make the head to be the next node, but if the head then will be null then we know that the node we removed was the only node in the list so the list will be empty and the tail will be set to null.
4. The `getHead` method will return the current head of the list, allowing external access to the first node.
5. The `printLL` method goes across the entire list from the head to the tail, printing out each task in the list. I did this by using a while loop that iterates through the nodes, starting from the head and moving to each subsequent node through the `next` reference until it reaches the end of the list (null).

**Classes:**

1. The `Node` class has been written as a static inner class within `LinkedList`.

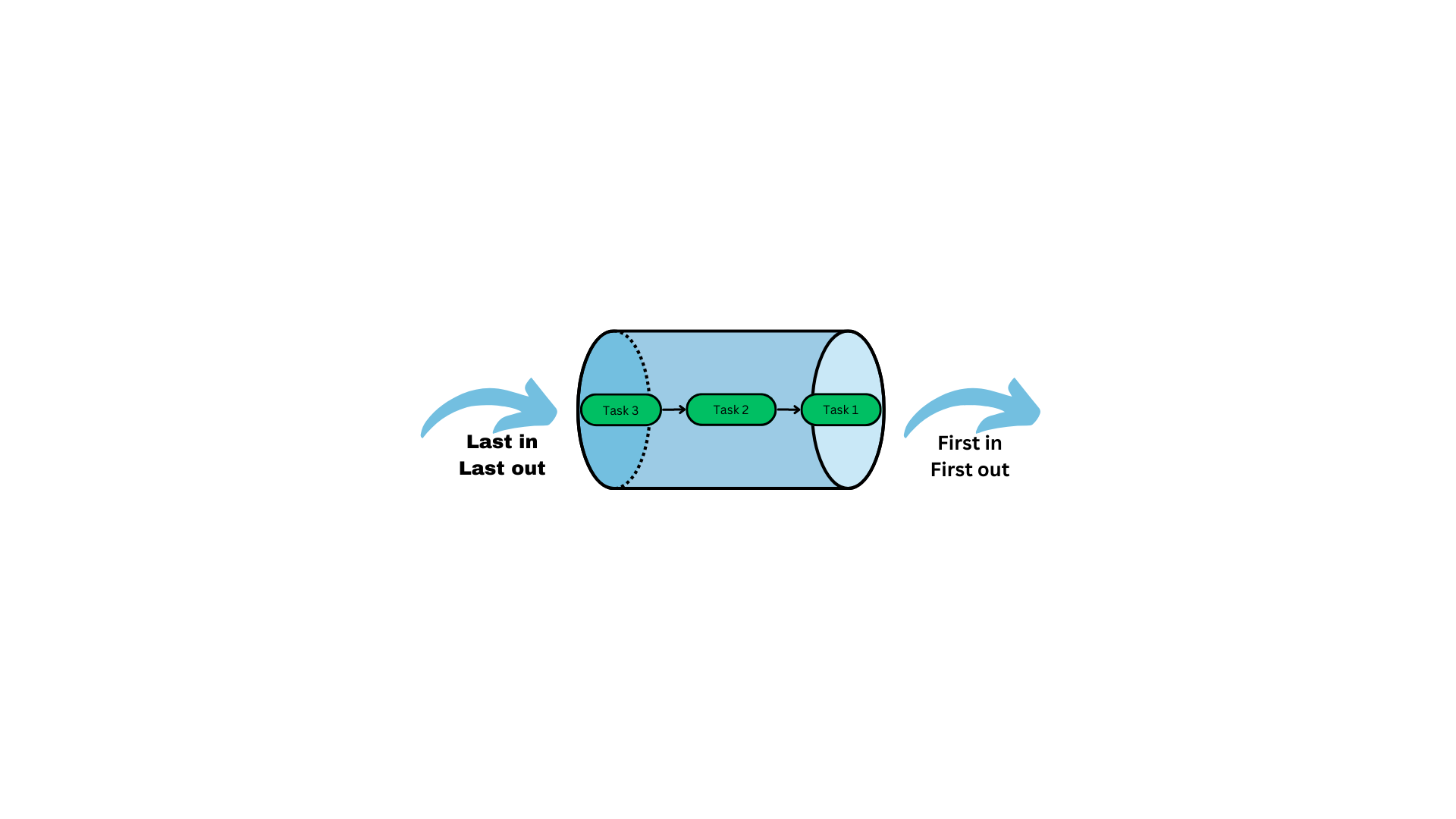
The `Task` object and it's reference to the node after it in the list are in each node. The constructor for `Node` initializes the `task` with the given `Task` object and sets the `next` reference to null.

**Class Diagram for LinkedList:**

|  |
| --- |
| LinkedList |
| - head: Node  - tail: Node |
| + isEmpty(): Boolean  + addLast(task: Task): void  + removeFirst(): void  + getHead(): Node  + printLL(): void |

|  |
| --- |
| Node |
| + task: Task  + Next: Node |
| + Node(task: Task) |

* **Queue:**



The Queue class in the taskManager1 package implements the Queue Abstract Data Type (ADT) using a linked list. The class encapsulates a LinkedList object called queueList, which stores the queues elements.

**Attributes:**

The creating from the Linkedlist a queueList that will store the task that have been marked as completed.

The constractar Queue() will initialise a new of LinkedList that called queueList and then it make sure that that queue is empty.

**Methods:**

1. This method (enqueue) adds a new task to the end of the queue by calling the LinkedList class's addLast method. This ensures that the queue follows the First-In, First-Out (FIFO).
2. This method (dequeue) removes the task at the front of the queue by calling the LinkedList class's removeFirst method. If the queue is empty, this operation wont do anything.
3. This method (getFront) will bring the head of the tasks, but first its going to check if the queue is empty, if yes it will null.
4. This Boolean method (isEmpty) will check if the queue is empty by calling the isEmpty method from LinkedList. So, if yes ‘Empty’ it will return true otherwise it will return false.
5. This method (printQueue) will call the PrintLL method from LikedList to print the queue tasks.

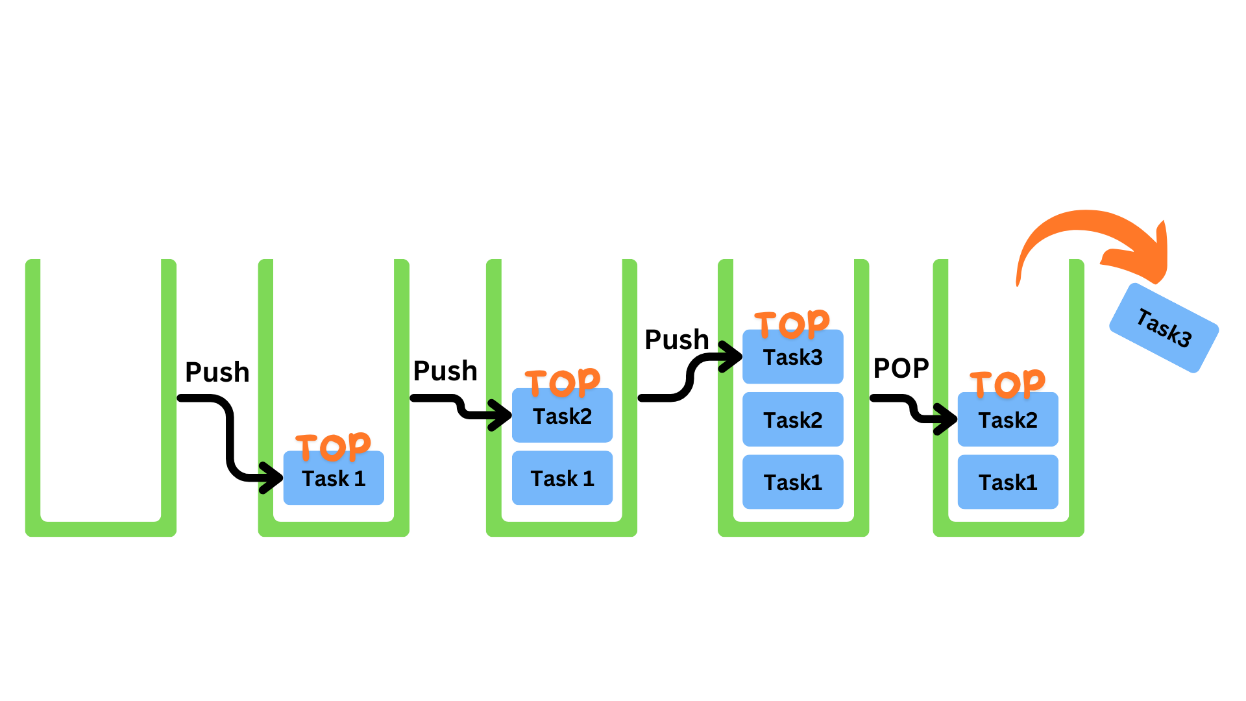
**Class Diagram for Queue:**

|  |
| --- |
| Queue |
| * queueList: LinkedList |
| + Queue()  + enqueue(task: Task): void  + dequeue(): void  + getFront() Task  + isEmpty(): boolean  + printQueue(): void |

|  |
| --- |
| LinkedList |
| - head: Node  - tail: Node |
| + isEmpty(): Boolean  + addLast(task: Task): void  + removeFirst(): void  + getHead(): Node  + printLL(): void |

|  |
| --- |
| Node |
| + task: Task  + Next: Node |
| + Node(task: Task) |

* **Stack:**

****

The IsUrgentStack class in the taskManager1 package is an implementation of a stack data structure that I designed to handle tasks that are marked as urgent. This stack works on the Last-In-First-Out (LIFO), where elements are added and removed from the top of the stack.

**Attributes:**

1. The private head variable from Node references to the top of the stack, but when the stack is empty the head will be null.

**Node static class:**

Attribute: Task 🡪 task that have the task data, Node 🡪 next that references to the next node.

Contractor: public Node 🡪 initializes the node with a given Task and sets the next reference to null.

Methods:

1. The public boolean isEmpty()will determines whether the stack is empty by making sure that head is null. and it will return true if the stack is empty, otherwise false.
2. The public void push(Task task) will push a new task to the top of the stack only if it is marked as urgent. It creates a new Node with the task, assigns the next reference to the current head, and updates the head to this new node.
3. The public Task pop() will delete and return the task at the top of the stack. If the stack is empty, it returns null and prints a message to indicate so. Otherwise, it saves the current head to a temporary variable, moves the head to the next node, and returns’ the task from the temporary node.
4. Public Task peek() returns the task at the top of the stack without removing it. If the stack is empty, it prints the message and returns null. If not so, it returns the task in the head node.

|  |
| --- |
| IsUrgentStack |
| -head: Node |
| + isEmpty(): Boolean  + push(task: Task): void  + pop(): Task  + Peek(): Task |

|  |
| --- |
| Node |
| + task: Task  + Next: Node |
| + Node(task: Task) |

### **What is a Stack?**

Data is stored and arranged in a data structure called a stack. It functions according to the Last-In, First-Out (LIFO) principle, which states that the item that was added most recently gets eliminated first.

Consider a stack of plates that you can add to and remove from the top. Similarly, a stack of tennis balls follows the same logic. The behavior of the stack is based on the idea of adding and removing objects from the same end.

**Stack ADT (Abstract Data Type)**

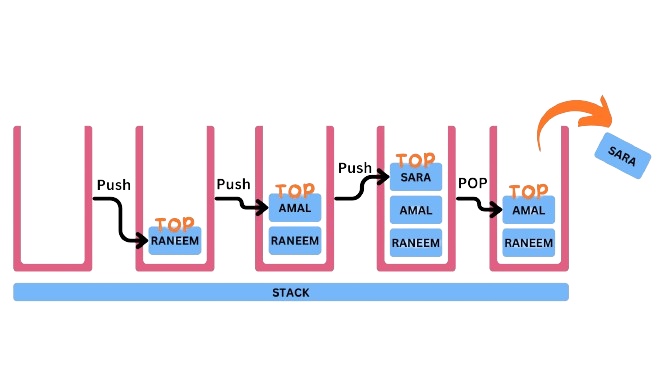
An insertion or deletion can only be done at one end of the list, referred to as the top, which is how the stack ADT is defined. The operations can be used in stack:

|  |  |
| --- | --- |
| Push | is to add an element to the top of the stack. |
| Pop | is to remove the element from the top of the stack. |
| Top | is to return the element at the top of the stack without removing it. |
| IsEmpty | is to check if the stack is empty. |

*And all these operations their complex is big o of 1, O(1).*

**Logical representing for the stack:**

Imagine a stack as a container that you can add and remove only from the top.



**Undo in Word Documents and Other Applications**

***Push***

When a user types a character, formats a text, or adds an image, the program logs this action and adds it to the stack of undo.

***Pop***

When the user presses the undo command (Ctrl+Z), the application pops the most recent action from the undo stack.

After reading the popped action, the application takes the appropriate action to undo it. For example, the undo operation will remove the character that was typed in the previous action from the document.

***Example in a Text Editor:***

When I enter the following characters with the following sequence

'A', 'B', 'C'

These actions will be added in the undo stack as:

Push ('A'), Push ('B'), Push ('C')

The stack now looks like 🡪 ['A', 'B', 'C']

***When the user undoes:***

Undo 1: Pop 'C' will be deleted

Undo 2: Pop 'B' will be deleted

Undo 3: Pop 'A' will be deleted

***Redo also uses stack:***

When an action is undone, it is pushed onto the redo stack. For instance, undoing the action 'C' would involve popping 'C' from the undo stack and pushing it onto the redo stack.

This allows the application to track action's that have been undone and potentially redo them later.

***Redo Process:***

When the user types the redo command (Ctrl+Y), the application pops the action from the redo stack and re-applies it.

**Real-Life Application: History of Visited Websites**

Stacks are commonly used in web browsers to manage the history of visited websites. Browsers can provide an efficient way for users to retrace their steps and move forward by utilizing two stacks—one for backward navigation and one for forward navigation. This implementation provides quick access to previous and next pages.

**How I used The Stack ADT in My Code:**

I developed an efficient structure for managing urgent tasks by implementing the IsUrgentStack class with the Stack ADT and a singly linked list. This implementation takes advantage of a stack for LIFO operation's, as well as a singly linked list for dynamic memory management and efficient element addition and removal, This combination ensures that the most recent urgent tasks are addressed first providing an effective method for prioritizing and managing urgent tasks.

## Task 2 - Application (user) level

### **Advantages of Encapsulation and Information Hiding When Using an ADT**

Encapsulation and information hiding are fundamental concepts in object-oriented programming. Information hiding refers to separating the description of how to use a class from the implementation detail's. The idea behind information hiding is that a programmer can use a class we create without having to understand how the methods are implemented. The programmer only need's to know what they provide to the class and what they receive in return. This minimizes the amount of information required to use a class, making it easier to apply our class's.

|  |  |
| --- | --- |
| Encapsulation | Information Hiding |
| An object-oriented principle that associates data with the functions that operate on it. | A design principle that hides implementation details and unintended data modifications. |
| Access modifiers are not strictly enforced, and they can be public, private, or protected. | Strictly applies the private access modifier. |
| A method for hiding information. | Helps with defensive programming. |

* **Encapsulation:** It’s the grouping of data and methods that work on that data into a single unit or class. It limits direct access to some of the object's, which can help prevent data interference and misuse. It protects the object's internal state, which can only be changed using its methods.

Advantages: The internal representation of an object is hidden from the outside by encapsulating its data. This protects the data from unintended changes. Like what I did in my code, in TaskManager class the task list and the size are encapsulated and the addTask method is the only way to add tasks, ensuring that tasks are added in a controlled manner, and in this way the **Encapsulation** will provide a protection for the data.

Also, it will divide the code into separate units and each is responsible for a specific function. This modularity makes the code more understandable, testable, and maintainable, in my code the Queue and IsUrgentStack classes provide functionality for managing completed and urgent tasks, respectively.

It will make easier to maintenance because it enables changes to the internal implementation without affecting the external interface. This simplifies code updates and maintenance. So, if I want to change how tasks are stored internally in the TaskManager, you can do so without affecting the methods that interact with it, such as addTask and markTaskCompleted.

* **Information Hiding:**

It entails hiding a classs’ internal implementation details from the outside world and exposing only what is required via a public interface. This reduces system complexity while increasing robustness by preventing external components from relying on potentially changing internal details.

Advantages: By concealing implementation details, the system's complexity will be reduced. The users of the class are only required to use the public interface, not the internal workings.

Users of the Queue class do not need to understand how task's are stored internally; instead, they can simply use the enqueue, dequeue, and isEmpty methods.

Information hiding enhances application security by preventing external code from accessing sensitive data directly.

The IsUrgentStack class's head attribute is hidden from external access, allowing only the provided methods to change the stack's internal state.

Information hiding enables developers to change the internal implementation without affecting external code that relies on the class. This increases the codebase's flexibility and adaptability to changes.

If I change the underlying data structure of the Queue class from a linked list to an array, the public interface will remain unchanged.

**In another way:**

Information Hiding: This entails hiding the internal details of a module in order to control access and interaction with those details from other parts' of the system.

Data Abstraction refers to the process of distinguishing a data types logical properties from its implementation details.

Abstraction is the process of developing a simplified model of a system that includes only the most important details from the viewer's perspective.The goal is to simplify a complex system and describe its core components in a clear and concise manner. This usually entails naming the components and explaining their functions. Higher levels of abstraction makes it easier to create data structures that are not dependent on specific solution's.

## Task 3 - Implementation (concrete) level

### 1.

|  |
| --- |
|  |

**Explaining of the Class Diagram \*\*Note: I changes the category class and it’s diagram is down with its explaining.**

**1. Main**

Attributes:

taskman: TaskManager: An instance of TaskManager used to manage tasks.

scanner: Scanner: A Scanner object for user input.

choice: int: To store the user's choice that he choose from the menu.

Methods:

main(String[] args): void: The main method is the most important point of the application. It initializes a TaskManager object and enters a loop to interact with the user, allowing them to add tasks, mark their tasks as completed, display tasks, view completed tasks, retrieve urgent tasks, see the sorted tasks by category, or they can exit the application.

**2. TaskManager**

Attributes:

head: Task: it is going to point to the first task in the linked list of tasks.

size: int: Keeps track of the number of tasks.

completedTasksQueue: Queue: A Queue to store completed tasks.

urgentTaskStack: IsUrgentStack: A stack to manage urgent tasks.

Methods:

TaskManager(): Constructor that initializes head to null, size to 0, and instantiates completedTasksQueue and urgentTaskStack.

addTask(String name, String dueDate, char isUrgent, String category): void: It is going to add a new task to the list. Confirm the due date, create a new task, and insert it in the correct position in the linked list based on the due date.

markTaskCompleted(String taskName): void: It will mark a task as completed by finding it in the list, removing it, and adding it to the completedTasksQueue.

displayTasksByDueDate(): void: Displays all tasks sorted by due date.

viewCompletedTasksHistory(): void: Displays the history of completed tasks from the completedTasksQueue.

retrieveUrgentTasks(): void: Displays all urgent tasks by popping them from the urgentTaskStack and temporarily storing them in a temporary stack to maintaen order.

sortTasksByCategory(): void: (Placeholder method) Intended to sort tasks by category, but the implementation is missing in the provided code.

compareDueDates(String date1, String date2): int: Helper method to compare two due date's.

**3. Task**

Attributes:

name: String: That is name of the task.

dueDate: String: The due date of the task on this --> YYYY/MM/DD format.

isUrgent: boolean: Show if the task is urgent; if yes 't' if no 'f'.

category: String: The category of the task.

isCompleted: boolean: Indicates if the task is completed and it will be completeed when the user mark it as completed.

next: Task: Points to the next task in the linked list.

Methods:

Task(String name, String dueDate, boolean isUrgent, String category): Constructor that initializes a task with given attributes.

toString(): String: Returns a string representation of the task.

**4. LinkedList**

Attributes:

head: Node: It would point to the first node in the linked list.

tail: Node: It will point to the last node in the linked list.

Methods:

isEmpty(): boolean: It is going to check if the list is empty.

addLast(Task task): void: It is going to add a task to the end of the list.

removeFirst(): void: It is going to remove the first task from the list.

getHead(): Node: Returns the head node.

printLL(): void: Print all tasks in the list.

Inner Class Node:

Attributes:

task: Task: The task contained in the node.

next: Node: Points to the next node in the list.

Constructor:

Node(Task task): Initializes a node with the given task's.

**5. Queue**

Attributes:

queueList: LinkedList: A linked list is used to store tasks in the queue.

Methods:

Queue(): It is a constructor that initializes the linked list.

enqueue(Task task): void: It is going to add a task to the end of the queue.

dequeue(): void: It will remove the task at the front of the queue.

getFront(): Task: Returns the task at the front of the queue without removing it.

isEmpty(): boolean: It is going to check if the queue is empty.

printQueue(): void: Prints all tasks in the queue.

**6. IsUrgentStack**

Attributes:

head: Node: It will point to the top node in the stack.

Methods:

isEmpty(): boolean: Check if the stack is empty.

push(Task task): void: Add an urgent task to the top of the stack's.

pop(): Task: will remove and return the task at the top of the stack.

peek(): Task: Return's the task at the top of the stack without removing it.

Inner Class Node:

Attributes:

task: Task: The task contained in the node.

next: Node: Points to the next node in the stack.

Constructor:

Node(Task task): Initializes a node with the given task.

**7. Category**

Category Class:

A node in a linked list of categories is represented by the inner class CategoryNode.

Every CategoryNode has a linked list of tasks connected to it as well as the name of the category even if it was upper case or lower case.

The Category class offers method's for grouping tasks by categories (displaytasksbytheirCategory) and adding tasks to categories (addCategoriestask).

Task Class: A task is represented by the Task class.

It has a taskName field that has the task name and a reference to the task that comes next in the list.

In a linked list of tasks connected to a category, every task is a node.

LinkedList Class: A singly linked list is implements simply in the LinkedList class.

A nested Node class that represents a node in the linked list is contained in it.

Methods to add a job to the end of the list (addLast), delete the first task from the list (removeFirst), obtain the head of the list (getHead), and print the list (printLL) are all provids by the LinkedList class.

Methods in Category Class Explained: Category(): The linked list's head is initialized to null by the constructor.

String category, String taskName, addTaskToCategoryNode: adds a task to the designeted field. The task is appended to the category if it already exists. If not, the work is added to a new category that is created.

Adds a job to the designeted category node (used internally by addCategoriestask) using the formula addTaskToCategoryNode(CategoryNode categoryNode, String taskName).

showTasksByCategory(): Shows each task arranged according to its category. It prints the name of the category and the tasks related to it after going through each category node.

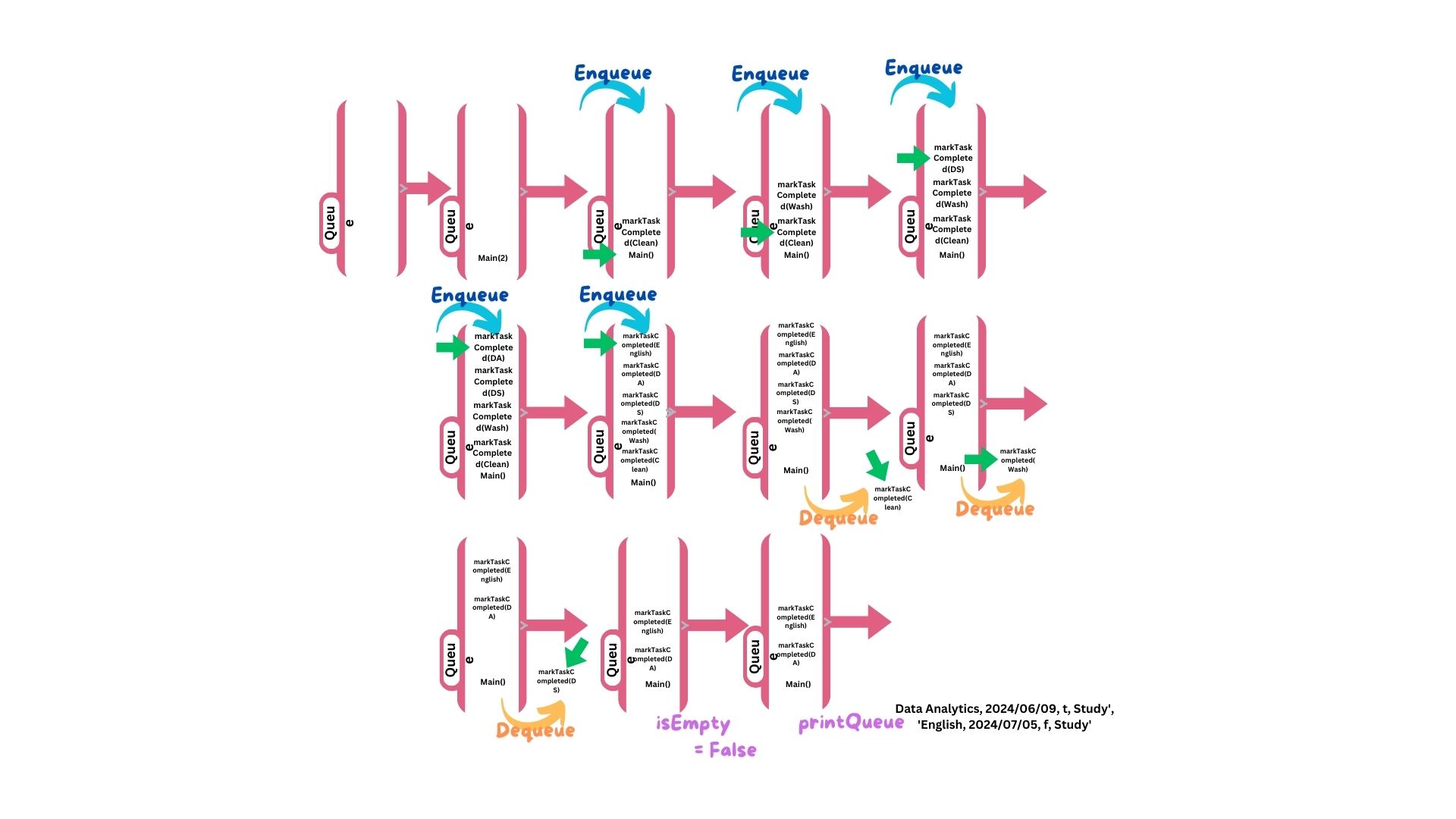
|  |
| --- |
| Category |
| -head: CategoryNode |
| + Category()  + addcategiriestasks(category: String,  taskName: String): void  + displaytasksbythierCategory(): void |

|  |
| --- |
| CategoryNode |
| - category: String  - taskHead: Task  - next: CategoryNode |
| + CategoryNode(category: String)  + addcategiriestasks(category: String,  taskName: String): void  + displaytasksbythierCategory(): void |
| + addTaskToCategory(category: String  taskName: String): void |

|  |
| --- |
| Task |
| - taskName: String  - next: Task |
| + Task(taskName: String) |

|  |
| --- |
| LinkedList |
| -head: Node |
| + Node(task: Task)  + addLast(task: Task): void  + removeFirst(): void + getHead(): Node  + printLL(): void |

### 2.

****

In the queue the first task goes in it’s the first task will go out, as you see in the enqueue I’m adding task to the queue ‘add first from the linked list’ and this happens when the user marks task as completed, but if the user want to remove the first task he is going to use the dequeue method and when the first task is deleted ‘Top’ the top or head is going to be the task next to it, and getFront is going to show us the first task in the queue which is the top ‘head’, the isEmpty method is going to say if the queue have task or not and then return a Boolean value, at the end you can print the task in the queue by the printQueue method and if there is no task to print it will say that there is no task to print.

### 3. Task and Category ADTs

Task: A single task with attributes like name, due date, urgency, category, and completion status. In a linked list, each task connect to the next task, making it easy to insert and traverse

Category: A category that includes a name and a linked list of tasks. This ensure that tasks are organized into their respective categories.

**Linked Lists**

Task LinkedList: Used in the TaskManager to maintain an ordered list of tasks sorted by due date. This ensure that tasks are sorted and easily retrieved.

Category LinkedList: Manages categories in the TaskManager, allowing the dynamic addition of new categories and tasks within them

**Queue and Stack ADTs**

Queue: Used to manage completed tasks. It employs the FIFO (First In, First Out) principle, making it easy to retrieve tasks in the order they were completed.

Stack: Used for urgent tasks. It uses the LIFO (Last In, First Out) principle, ensuring that the most recently marked urgent task is retrieved first.

And now going to the Algorithms

Insertion Algorithm: When a task is added to the task linked list, it is sort in order of due date.

Category Normalization: To handle case insensitivity, category names are normalized to lowercase, resulting in consistent categorization.

Advantages of Using Independent ADTs

Using independent ADTs (Abstract Data Types) in software development has several advantages, including improved overall design, maintainability, and functionality of the system. Here are three main **Benefit:**

Modularity and Reusability

Modularity: ADTs encourage modularity by enclosing data and operations in self-contained units. Each ADT (such as Task, Category, Queue, and Stack) has a well-defined interface and can be create, tested, and debugged independently. This modular makes it easier to understand and maintain the codebase.

Reusability: Once an ADT is implemented, it can be reused throughout the application or even in different projects. For example, the Queue and Stack implementations can be reused in other scenarios where similar data management is required.

Maintainability and Extendibility

Maintainability: ADTs improve maintainability by isolating changes. If the implementation of an ADT needs to be changed, the changes are limited to that ADT and do not affect the rest of the system. For example, changing how tasks are stored in the Task linked list does not affect on the other aspect of the task manager.

Extendibility: ADTs make it easier to expand the system. An ADT can be updated with new features or enhancements without changing the interface. For example, adding new methods to the TaskManager to support additional operations on tasks or categories is possible without affecting existing functionality.

Data Integrity and Abstraction

Data Integrity: ADTs ensure data integrity by restricting access to it. They provide operations that ensure data is manipulated in a consistent and valid way. For example, the Queue and Stack ADT''s ensure that tasks are correctly enqueued, dequeued, pushed, and popped, preserving the data structure's integrity.

Abstraction: ADTs provide a layer of abstraction that hides the implementation details from the user. so this abstraction enables developers to use ADTs without concern for their internal workings. For example, developers can use the TaskManager to add and retrieve tasks without having to understand the underlying linked list operations.

Trade-offs of Using Different Abstract Data Types (ADTs) in my Task Management System.

When designing a task management system, selecting the appropriate data structures for operations like search, insert, and remove is critical to achieving efficient performance. I compare the trade-offs of three ADTs which are Binary Search Trees (BST), Sorted Arrays, and Ordered Linked Lists.

**1. Binary Search Trees (BST)**

Operations:

Insert: O(log n) on average; O(n) in the worst case when the tree is unbalanced.

Search: O(log n) on average, O(n) in worst-case when the tree is unbalanced.

Remove: O(log n) on average and O(n) in the worst case when the tree is unbalanced.

Balanced Performance: When balanced, It will give efficient average-case performance across all operations.

Dynamic: BSTs enable dynamic insertion and deletion without changing the entire data structure.

Cons:

Complexity: Creating and maintaining balanced trees can be challenging and hard to make.

Worst-case Performance: If the tree becomes unbalanced, performance can have the complexity -->O(n).

It's Impact on my task management system:

BSTs can efficiently handle a diverse set of tasks with varying deadlines, allowing for quick insertion, search, and removal. Balancing mechanisms maintain optimal performance, but the complexity of implementation may be a disadvantage for my system.

**2. Sorted Arrays**

Operations:

Insert: It will be a big O(n) because it requires shifting elements to keep order.

It will be a big O(log n) for the binary search.

Remove: O(n) - requires shifting elements to keep order.

Pros:

Search Efficiency: Binary search permits fast lookups.

Memory Efficiency: Arrays have lower overhead than the linked structures.

Cons:

Insertion and Deletion Overhead: Both insertions and deletions require element shifting, which makes them expensive operation's.

Fixed Size: Static arrays require resizing or have a predetermined size, which can be inefficient in terms of memory usage.

Impact on Task Management System:

Sorted arrays are appropriate when the task list is relatively static, with few insertions and deletions. The search operation is efficient, but the high cost of insertion and removal makes this ADT unsuitable for highly dynamic environment's.

**3. Ordered Linked Lists**

Operations:

Insert: O(n) because it needs traversal to find the right position.

Search: O(n) when its linear search

Remove: O(n) because it is going to needs traversal to find the element.

Pros:

Linked lists can grow and shrink dynamically, eliminating the need for resizing.

Insertion Flexibility: Unlike arrays, insertion is simpler because no elements need to be shifted because it works with nodes.

Cons:

Linear search is less efficient than BSTs and sorted arrays for finding element's.

Memory Overhead: Each element requires additional memory to store the pointer/refernce to the next element.

**Impact on Task Management System:**

Ordered linked lists are better suited to handling frequent insertions and deletions than arrays. However, there linear search time makes them inefficient for large datasets.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ADT’s | Time complexity (Search) | Time complexity (Insert) | Time complexity (Remove) | Space complexity |
| BST | O(log n) on average  O(n) on worest | O(log n) on average  O(n) on worest | O(log n) on average  O(n) on worest | O(n) the tree structure |
| Array sorted | O(log n) | O(n) | O(n) | O(n) memory using |
| Linked List Ordered | O(n) | O(n) | O(n) | O(n) for pointers |

**Time Complexity:**

BST: balanced (O(log n) for search, insert, and remove), unbalanced (O(n)).

Sorted arrays: search (O(log n)), insert/remove (O(n)).

Ordered linked lists: Search (O(n)), Insert/Remove (O(n)).

Space Complexity:

BST takes O(n) to store n elements, plus overhead for tree structure.

Sorted Arrays: O(n) for storing n elements, with efficient memory usage

Ordered Linked Lists: O(n), plus overhead for pointers.

**And here I have some recommendations:**

If the data is large and dynamic, using BST will achieve balanced performance in dynamic scenarios.

Its better to use sorted arrays when the task list is relatively static and memory efficiency is critical.

Ordered linked lists should be used for frequent insertions and deletions where search performence is less critical.

**The Relevance of Abstract Data Types (ADTs) in Object-Oriented Programming**

Object-Oriented Design: ADTs, like classes and objects, contribute to the defintion of data types and operations. Classes serve as blueprints for object creation, while ADTs demonstrate how to perform code operations.

Encapsulation: Both ADTs and object-oriented programming rely on encapsulation, which involves combine data and operations. This is accomplished using classes, which store data and methods for later use. ADTs maintain data and operations while interacting with the application and code through procedures or functions.

Data Abstraction: ADTs and object-oriented programming both support data abstraction, which is hiding how data is stored while allowing users to interact with it through an interface. ADTs define a clear way to interact with data structures and their operations, which object-oriented programming accomplish by leveraging classes and objects.

**5.**

The fundamental component of object-oriented programming (OOP) are the abstract data types (ADTs), and they offer a framework for creating intricate software systems with readable, reusable, and maintainable code. Modularity, abstraction, and data integrity are encouraged by ADTs, which encapsulate data and related activities.

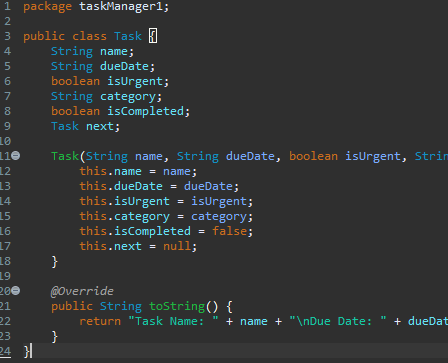
ADTs are usually implemented as classes in OOP, which specify the structure and behavior of data.

**1. Modularity and Encapsulation**

Encapsulation: Encapsulation is the process of combinings data and the operations performed on the data. ADTs assist in encapsulating an object's internal state and only exposing the operations that are required. This offers a clean interface for intaraction while concealing the underlying specific's.

Modularity: By establishing independent functional units, ADTs support modularity. Because each ADT is in charge of a distinct part of the system, it is simpler to maintain and grow.

And this example of the Task and Category classes in the task manager system contain the properties and functions associated with tasks and categoris, respectively.



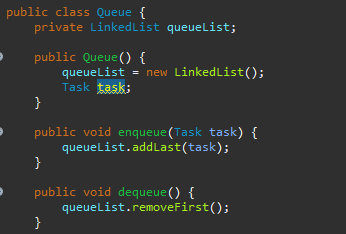
To make the task class can be used independently, it encapsulates the data related to a task and provides methods to interact with that data.

To make the task class can be used independently, it encapsulates the data related to a task and provides methods to interact with that data.

**2. Abstraction**

Abstraction simplifies complex systems when it allows developers to focus on high-level operations rather than the underlying detailes. ADTs add a layer of abstraction by defining operations that can be performed on data without revealing the implementation.

This exampl of the Queue and Stack classes in the task manager system is going to provide an abstraction for managing completed and urgent task;s, respectively.



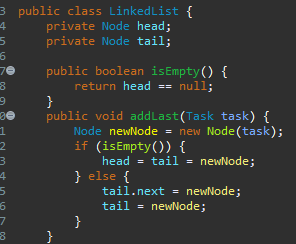
The Queue class is going to abstracts the operations of adding and removing tasks in a FIFO 'first in and first out' order without letting on how the tasks are stored internally.

3. Reusability and Maintainability

Reusability: ADTs promote reuse by providing a clear and consistent interface. Once implemented, an ADT can be reused in various parts of the application or even in other projects.

ADTs improve maintainability by isolating changes. Changes to an ADT's internal implemantation have no effect on the other parts of the system that use it.

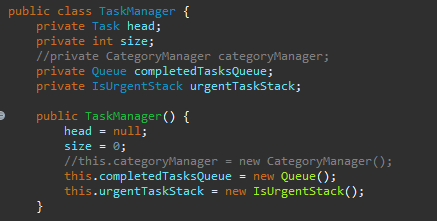
For example, the Queue and CategoryManager classes use the LinkedList class to manege task and category list's, respectively.



The LinkedList class provides a reusable implemantation of a linked list that can be used in many and diffferent parts of the system. This reusability reduces code duplication and enhances maintainability.  
4. Data Integrity

Data Integrity: ADTs protects data by controlling how it is accessed and modified. ADTs avoid invalid or inconsistent stats by limiting direct access to the internal state and providing controlled interaction methods.

And in this example the TaskManager class manages tasks and ensures that they are added, completed, and retrieved in a controled manner.



The TaskManager classs use's ADTs to manage tasks while when maintaining data integrity. The encapsulated methods for adding, completing, and retrieving tasks will make sure that the task data remains consistent and valid.

# Part 2

Selection Sort: Selection Sort is a basic sorting algorithm that operates by repeatedly choosing the element from the unsorted part of the array that is the least (or maximum, depending on the sorting order) and starting it at the beginning. Sorted and unsorted subarrays are kept up to date by the algorithm. It replaces the initial element of the unsorted subarray with the minimum element it finds in the subarray. This process will be reapeted until all the array is sorted.

Time Complexity: For any selection sort, the time complexity is going to be the big O of n \* n --> O(n^2), where n is the number of entries in the array. This is due to the fact that the approach uses two nested loops: one to locate the minimal element and another to go through the array.

Merge Sort: This divid technique is going to splits the input array to two halfs, sort each half recursively, and then combine the sorted halves to create a single sorted array. The array is divided in half periodically until there is only one element in each subarray. At that point, the subarrays will be merged back together while preserving the sorted order.

Complexity of Time:

The time complexity always of the Merge Sort is O(n log n) and that is because the dividing, where n is the array's element count. This is because the technique takes O(log n) time to split the array in half recursively until each subarray has just one element. After that, it takes O(n) time to combine the subarrays.

1. Comparing Sorting Algorithms Experimentally:

Execution:

*Iam going to* use an improved form of Selection Sort besides Merge Sort for this purpose. Reducing the amount of comparisons and swaps, the improved Selection Sort method determines the minimum and maximum values in each iteration and swaps them into the appropriate placss.

For every sorting algorithm, I will write functions and test them on arrays with sizes of 100,000 and 1,000,000. I will also create three different types of arrays: sorted, random, and reverse sorted.

Time Measurement: I am going to utilize the proper timing routines provided by the programming language I’m using to determine how long it takes for each algorithm to sort the arrays. As instructed, well measure the time in milliseconds or microseconds' and note the outcome.

**In the merge sort:**

Time = N (num of operations)/operation per second

operation per second = 1/ 10^-10 = 10^10

Time = 10^10log10^10/10^10

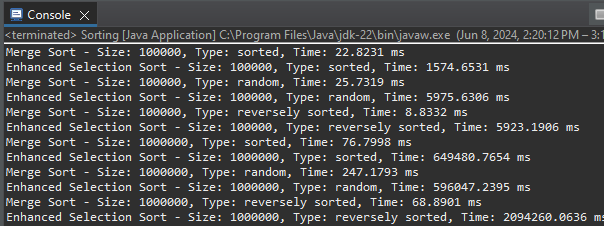
Time = log10^10 = 10log10 = 33.219 sec

**For the selection sorting:**

Time = 10^10^2/10^10

Time = 10^20/10^10

So the time is going to equal --> 10^10 sec = 316.8876



**Visualization and Analysis:**

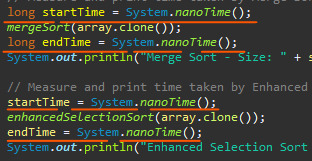
I will plot the results in a chart where the x-axis represents the size of the array and if the array in 'Sorted, Random, or Reversed' and the Y-axis represents the time spent in sorting, after sorting each array type and size using both algorithms. For every kind of array, we'll make a different chart, and I will talk about the insights of these plots.

This plot shows the time spend in the merge and enhanced selection sort where the array size = 10^5, the Merge sort as we see in the right side of the plot takes less time that the enhanced selection takes.

This plot shows the time spend in the merge and enhanced selection sort where the array size = 10^6, the Merge sort as we see in the right side of the plot takes less time that the enhanced selection takes for all the arrays types ‘'Sorted, Random, or Reversed'’.

This plot is for the merge sort only that shows the time each size of the array take for all the types where the reversed type was the best performance in time.

This plot is for the enhanced selection sort only that shows the time each size of the array take for all the types where the reversed type was the best performance in time for both 10^5 and 10^6.



### 2. **Memory Stack Frames in Recursive Merge Sort**

Each recursive call to mergeSort creates a stack frame that includes:

Parameter's passed to the function (array, l, r).

Local variables (m).

Return address, it will indicat where the function should return after execution.

Recursive Calls and Stack Frames

For the following array: [[38, 27, 43, 3, 9, 82]]

Initial Call:

mergeSort(array, 0, 5)

Stack Frame: mergeSort(array, 0, 5)

First Level of Recursion:

l = 0, r = 5, m = 2

Stack Frame: mergeSort(array, 0, 2) and mergeSort(array, 3, 5)

Second Leval of Recursion (Left Half):

mergeSort(array, 0, 2)

l = 0, r = 2, m = 1

Stack Frame: mergeSort(array, 0, 1) and mergeSort(array, 2, 2)

Third Level of Recursion (Left-Left Half):

mergeSort(array, 0, 1)

l = 0, r = 1, m = 0

Stack Frame: mergeSort(array, 0, 0) and mergeSort(array, 1, 1)

Base Case Reached:

mergeSort(array, 0, 0) and mergeSort(array, 1, 1)

Both reach base case, return immediately, and their stack frames are popped off the stack.

Merging Left-Left Half:

merge(array, 0, 0, 1)

The stack frame for merge is created and then destroyed after merging.

Array becomes [27, 38, 43, 3, 9, 82] after merging the left-left half.

Returning to Second Level (Left-Right Half):

mergeSort(array, 2, 2)

Reaches base case, returns immediately, and the stack frame is popped off the stack.

Merging Left Half:

merge(array, 0, 1, 2)

The stack frame for merge is created and then destroyed after merging.

Second Level of Recursion (Right Half):

mergeSort(array, 3, 5)

l = 3, r = 5, m = 4

Stack Frame: mergeSort(array, 3, 4) and mergeSort(array, 5, 5)

Third Level of Recursion (Right-Left Half):

mergeSort(array, 3, 4)

l = 3, r = 4, m = 3

Stack Frame: mergeSort(array, 3, 3) and mergeSort(array, 4, 4)

Base Case Reached:

mergeSort(array, 3, 3) and mergeSort(array, 4, 4)

Both reach base case, return immediately, and their stack frames are popped off the stack.

Merging Right-Left Half:

merge(array, 3, 3, 4)

The stack frame for merge is created and then destroyed after merging.

Returning to Second Level (Right-Right Half):

mergeSort(array, 5, 5)

Reaches base case, returns immediately, and the stack frame is popped off the stack.

Merging Right Half:

merge(array, 3, 4, 5)

The stack frame for merge is created and then destroyed after merging.

[27, 38, 43, 3, 9, 82]

Final Merge:

merge(array, 0, 2, 5)

The stack frame for merge is created and then destroyed after merging.

[3, 9, 27, 38, 43, 82] after the final merge.

**Operations Performed by the Memory Stack**

Creation of Stack Frames:

Each time mergeSort is called, a new stack frame is created that includes the parameters passed (array, l, r), local variables (e.g., m), and the return address.

Stack Frame Lifecycle:

The stack frame is created when a function call is made and destroeed when the function returns. For recursive functions, multiple stack frames are created as the function calls itself with different parameters.

Parameter Passing:

Each stack frame includes the parameters passed to the function, ensuring that each call works with its segment's of the array.

Local Variables:

Each stack frame is going to include space for local variables as like m, which are used within the function.

Return Address:

Each stack frame includes the return address, so the function knows' where to return control after execution.

Merging:

After reaching the base case in it, stack frames start to unwind, and the merge function is called to merge sorted subarrays. This creates additional stack frames that are also managed by the memory stack.

Function Calls: When a function is called in a program, the current execation state, including the program counter and local variabls, is pushed to the stack. This is commonly referred to as a "stack frame" or "activation record" for the function call.

### **3. Big-O Notation and Asymptotic Analysis**

Here is the mathematical notation that describs the upper bound of an algorithms times' complexity. It gives an asymptotic analysis of performance, focusing on how the runtime scale with the sizze of the input. Common Big-O notations are:

O(1): Constant time, O(log n): Logarithmic time, O(n): Linear time, O(n log n):Log-linear time, O(n^2): Quadratic time, O(2^n):Exponential time

Asymptotic Analysis

Selection Sort:

Best Case: O(n^2)--> Even if the array is already sorted, it is going to still performs O(n^2) comparisons.

Worst Case: O(n^2)--> In the worst case in the array, the algorithm performs O(n^2) comparisons and swaps.

Average Case: O(n^2)--> On average, Selection Sort performs O(n^2) comparisons.

Merge Sort:

Best Case: O(n log n)--> The algorithm is going to always divides the array and merges it, to be result in O(n log n) time complexity.

Worst Case: O(n log n)--> Merge Sort consistently performs at O(n log n) regardles of the input order.

Average Case: O(n log n) --> The average time complexity remains O(n log n).

Comparison of Efficiency

Selection Sort is less efficient for larger arrays due to its quadratic time complexity, making it suitable only for small datasets or educational purposes, so its better to use it with smaller ones.

Merge Sort is more efficient for larger datasets due to its logarithmic factor in time complexety, making it suitable for larger arrays where performance is critical.

### **4. 4. Alternative Methods for Measuring Efficiency**

Alternative Methods

Apart from experimental analysis, we can measure the efficiency of algorithms using the following methods,

The Amortized Analysis method computes the average time for every operation over a sequence of operations, ensuring that even if some operations are expensive, the overall cost remains low.

Profiling and Benchmarking:

Profiling entail's using tools to assess the performance of various components of an algorithm, identifying bottlenecks and optimizing them.

Benchmarking is the process of running an algorithm with specific input sizes and measuring its execution time in order to compare it to other algorithms.

Examples and Illastrations

Amortized Analysis Example:

Consider a dynamic array (like an ArrayList) that doubles its size when full. Although resizing is costly (O(n)), it happens infrequently, leading to an average time complexity of O(1) per insertion operation.

Profiling Example:

Using a tool like gprof in C or the cProfile module in Python, we can measure the time spent in each function. This helps identify slow functions that need optimization.

Applicability and Limitations

Amortized Analysis:

Applicability: Suitable for data structures where a few expensive operations are offset by a lot of cheap ones.

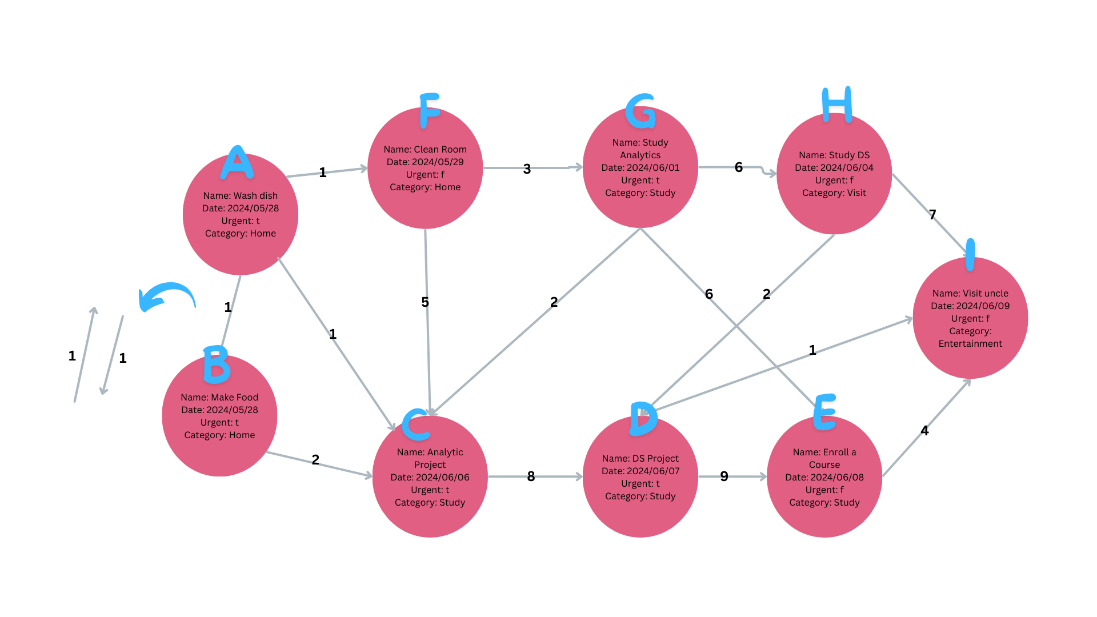
Limitations: May not reflect worst-case scenarios and can be complex to analyze.

Profiling and Benchmarking:

Applicability: Useful for practical performance measurement and optimization in real-world applications.

Limitations: Dependent on the specific hardware and software environment, which might not generalize well to other contexts.

# Part 3



*Dijkstra's Algorithm*

1. **Initialization**:

Source: Task A

Distance: {A: 0, B: ∞, C: ∞, D: ∞, E: ∞, F: ∞, G: ∞, H: ∞, I: ∞}

Previous: {A: None, B: None, C: None, D: None, E: None, F: None, G: None, H: None, I: None}

1. **Iteration**:

**(0, A)** 🡪 B: Distance = 1 (Previous: A), C: Distance = 1 (Previous: A), F: Distance = 1 (Previous: A)

**(1, B) 🡪 Nothing**

**(1, C) 🡪 D:** Distance = 8+1 (Previous: C)

**(1, F) 🡪 G:** Distance = 3+1 (Previous: F)

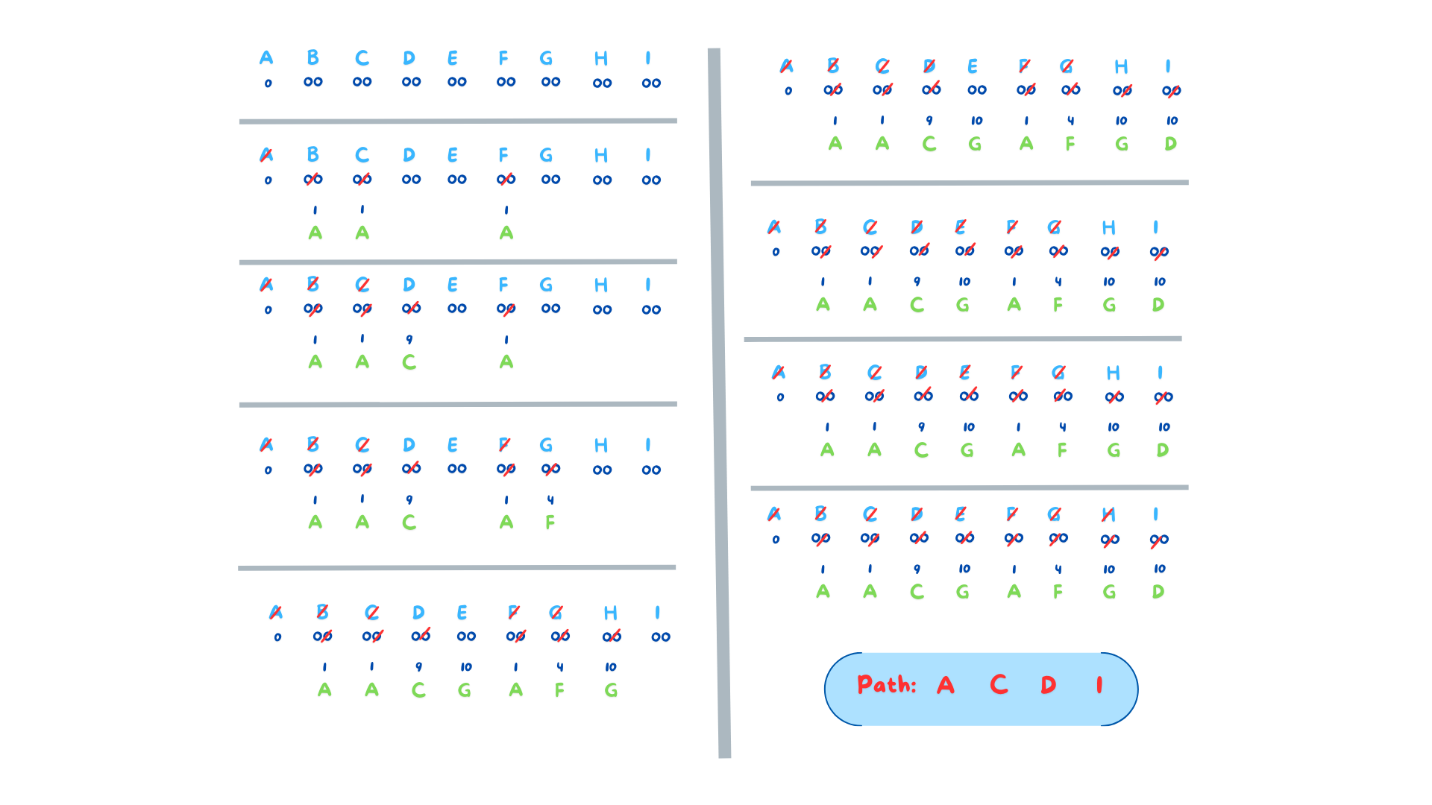
**(4, G) 🡪 E:** Distance = 4+6 (Previous: G), **H:** Distance = 4+6 (Previous: F)

**(9, D) 🡪 I:** Distance = 9+1 (Previous: D)

**(10, E) 🡪** Nothing

**(10, H) 🡪** Nothing

**{A:0, B:1, C:1, D:9, E:10, F:1, G:4, H:10, I:10}**



### Complexity Evaluation

#### Dijkstra's Algorithm:

* **Time Complexity:** O(V^2) for the adjacency matrix representation, where V is the number of vertices. Each vertex is visited once, and each edge we walk on once.
* **Space Complexity:** O(V^2) for storing the adjacency matrix.

#### Bellman-Ford Algorithm with Negative Weight Edge

1. **Initialization:**
   * **Source:** Task A
   * **Distance:** {A: 0, B: ∞, C: ∞, D: ∞, E: ∞, F: ∞, G: ∞, H: ∞, I: ∞}
   * **Previous:** {A: None, B: None, C: None, D: None, E: None, F: None, G: None, H: None, I: None}
2. **Iteration:**
   * **(1st Iteration):**
     + **(0, A) → B, C, F:**
       - Distance to B: 1 (Previous: A)
       - Distance to C: 1 (Previous: A)
       - Distance to F: 1 (Previous: A)
     + **(1, B) → None:**
       - No further updates.
     + **(1, C) → D:**
       - Distance to D: 8 + 1 (Previous: C)
     + **(1, F) → G:**
       - Distance to G: 3 + 1 (Previous: F)
     + **(4, G) → E, H:**
       - Distance to E: -3 + 4 (Previous: G)
       - Distance to H: 4 + 6 (Previous: G)
     + **(9, D) → I:**
       - Distance to I: 9 + 1 (Previous: D)
     + **(10, E) → None:**
       - No further updates.
     + **(10, H) → None:**
       - No further updates.
3. **Final Result:**
   * **Shortest Path:**
     + {A:0, B:1, C:1, D:9, E:-3, F:1, G:4, H:10, I:10}

**The Dijkstra step by step:**

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

### Complexity Evaluation

#### Bellman-Ford Algorithm:

* **Time Complexity:** O(V \* E) in the worst case, where V is the number of vertices and E is the number of edges. Each edge is relaxed V - 1 times.
* **Space Complexity:** O(V) for storing the distances and predecessors arrays.

[The Bellman step by step is down](#_Bellman_Step_by)

**References**

 EnjoyAlgorithms, 2023. Application of stack data structure in programming. [online] Available at: <https://www.enjoyalgorithms.com/blog/application-of-stack-data-structure-in-programming> [Accessed 8 June 2024].

 GeeksforGeeks, 2023. Dijkstra’s shortest path algorithm | Greedy Algo-7. [online] Available at: <https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/> [Accessed 8 June 2024].

 W3Schools, 2023. Dijkstra’s Algorithm. [online] Available at: <https://www.w3schools.com/dsa/dsa_algo_graphs_dijkstra.php> [Accessed 8 June 2024].

 Learn Loner, 2023. Encapsulation and Information Hiding. [online] Available at: <https://learnloner.com/encapsulation-and-information-hiding/> [Accessed 8 June 2024].

 EnableGeek, 2023. Encapsulation and Information Hiding in Object-Oriented Programming. [online] Available at: <https://www.enablegeek.com/blog/encapsulation-and-information-hiding-in-object-oriented-programming/> [Accessed 8 June 2024].

 TutorialsPoint, 2023. Difference between data hiding and encapsulation. [online] Available at: <https://www.tutorialspoint.com/difference-between-data-hiding-and-encapsulation> [Accessed 8 June 2024].

 Programiz, 2023. Dijkstra’s Algorithm. [online] Available at: <https://www.programiz.com/dsa/dijkstra-algorithm> [Accessed 8 June 2024].

#### **Bellman** Step by Step

