

Student Name: Leen Shareef Saleh

Student ID: 22110090

Instructor Name: Dr. Ahmad Bataineh

Course Name: Data Structures & Algorithms

Topic: Final Report Submission of Implementing and developing Application of Task management system

In completion of the requirements for passing the Data Structures and Algorithms course

By Leen Shareef Saleh

Spring Semester 2023/2024

Part 1 - Task 1 – Logical Level

1. Create a design for the valid operations that can be performed on different structures; outline the functionalities and the interactions withing the system.





|  |  |  |
| --- | --- | --- |
| Data Structures names | Methods | Descriptions |
| LinkedList | insertAtBeginning(task) | Inserts a task at the beginning of the linked list |
| insertAtEnd(task) | Inserts a task at the end of the linked list |
| insertInOrder(tas) | Inserts a task in the linked list in order based on some criteria (due date) |
| deleteAtBeginning() | Deletes the task at the beginning of the linked list. |
| deleteAtEnd() | Deletes the task at the end of the linked list. |
| deleteAtIndex(index int) | Deletes the task at the specified index in the linked list. |
| get(index: int): Tasks | Retrieves the task at the specified index in the linked list. |
| contains(task): boolean | Checks if the linked list contains the specified task. |
| size(): int | Returns the number of tasks in the linked list. |
| printList(): void | Prints all the tasks in the linked list. |
| LinkedListStack | isEmpty(): Boolean | Checks if the stack is empty. |
| stacksize(): int | Returns the number of tasks in the stack. |
| push(task: Tasks) | Adds a task to the top of the stack. |
| pop(): Tasks | Removes and returns the task from the top of the stack. |
| peek(): Tasks | Returns the task at the top of the stack without removing it |
| pop(taskId: int) | Removes and returns the task with the specified ID from the stack. |
| QueueLinkedList | isEmpty(): boolean | Checks if the queue is empty |
| size(): int | Returns the number of tasks in the queue. |
| enqueue(task) | Adds a task to the rear of the queue. |
| dequeue(): Tasks | Removes and returns the task from the front of the queue. |
| printQueue(): void | Prints all the tasks in the queue. |
| ArrayList | insert(element: Tasks) | Adds a task to the array list. |
| insertAt(element: Tasks, position: int) | Inserts a task at the specified position in the array list. |
| replace(element: Tasks, position: int) | Replaces the task at the specified position in the array list with a new task. |
| read(index: int): Tasks | Retrieves the task at the specified index in the array list. |
| remove(index: int) | Removes the task at the specified index in the array list. |
| isEmpty(): boolean | Checks if the array list is empty. |
| resize(): void | Increases the capacity of the array list when it is full. |
| printList(): void | Prints all the tasks in the array list. |
| size(): int | Returns the number of tasks in the array list. |

The Task Management System is designed to handle tasks efficiently using various data structures like linked lists, stacks, and queues. The system offers several functionalities: adding tasks, printing all tasks, marking tasks as completed, viewing completed tasks, viewing urgent tasks, and showing tasks based on categories. Users can interact with the system through a main class that handles user input and communicates with the Task Management System.

The main class prompts users to enter task details, such as ID, due date, urgency, and category when adding a task. It then sends this information to the Task Management System. The system organizes tasks in a linked list and, if a task is urgent, also stores it in a stack. All tasks are printed sorted by their due dates, and the system can mark tasks as completed, moving them to a completed tasks queue. Additionally, the system can show urgent tasks and categorize tasks into groups A, B, and C.

The Task Management System relies on a few key data structures. The linked list manages the tasks, allowing for easy insertion, deletion, and retrieval. The stack is used to manage urgent tasks, while the queue manages completed tasks. The system also includes an array list for dynamic storage and a selection sorting algorithm for sorting tasks based on categories.

Tasks are represented by a task class that includes attributes like ID, due date, urgency, category, and completion status. The system uses a node class to store these tasks within the linked list, stack, and queue. The selection sorting algorithm helps sort tasks by categories using a simple selection sort method.

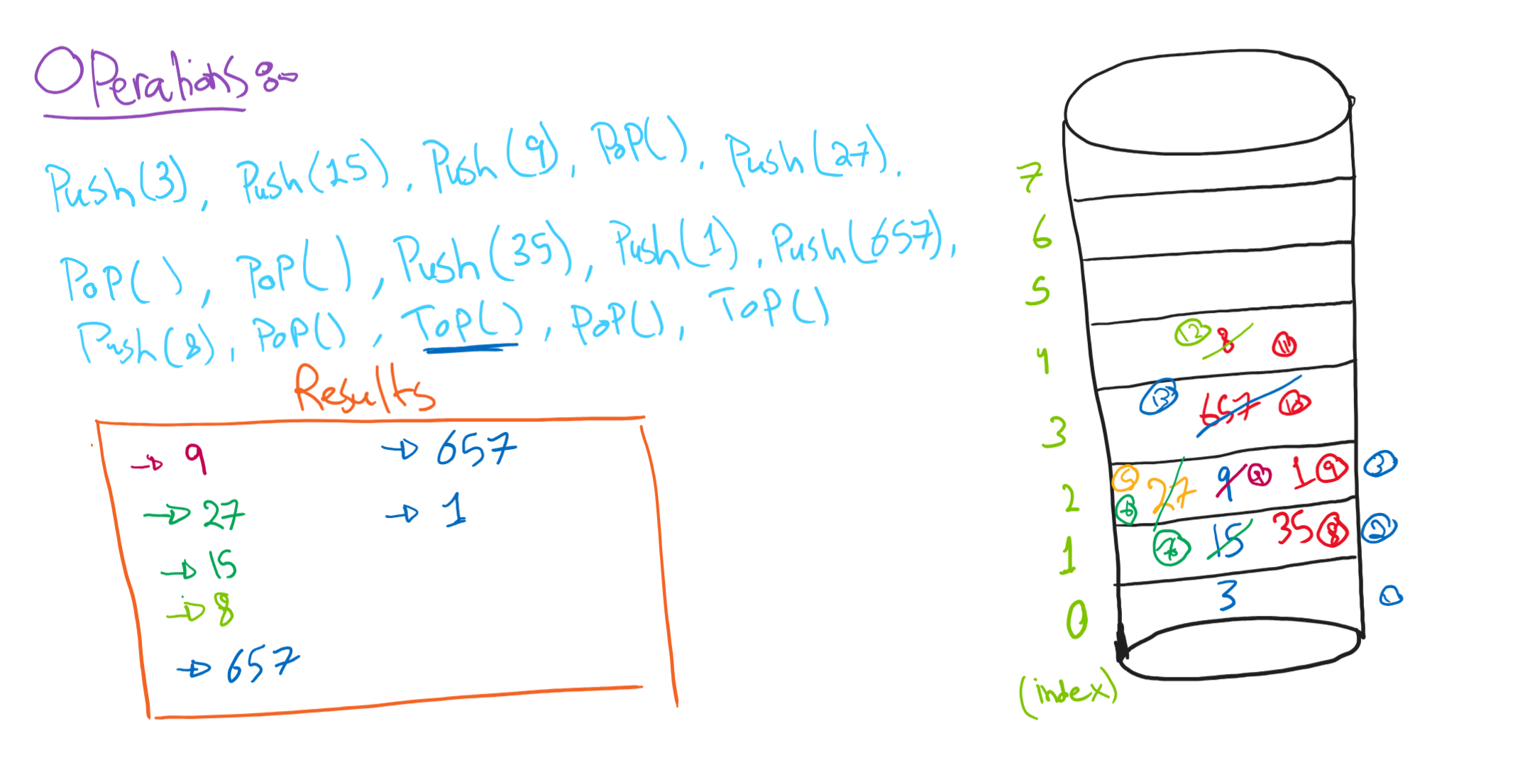
Interactions between classes are straightforward. The main class interacts with the Task Management System to perform various tasks, while the Task Management System manages the data structures. Tasks are stored within nodes of these data structures and manipulated as needed. By utilizing these functionalities and interactions, the Task Management System ensures efficient handling, prioritization, and categorization of tasks.

1. Specify the meaning and components of a software stack data type in the conceptual structure.

Stack is a Data Structure in which we store and organize data, Implementation for Last In First Out (LIFO), so an element is added or removed based on its placement order in the stack. If we want to remove the first element we have inserted we must first remove all element that were added after it. So the insertion or the deletion is done only from the top of the stack which is the last element.[[1]](#footnote-1)

**Operations:**

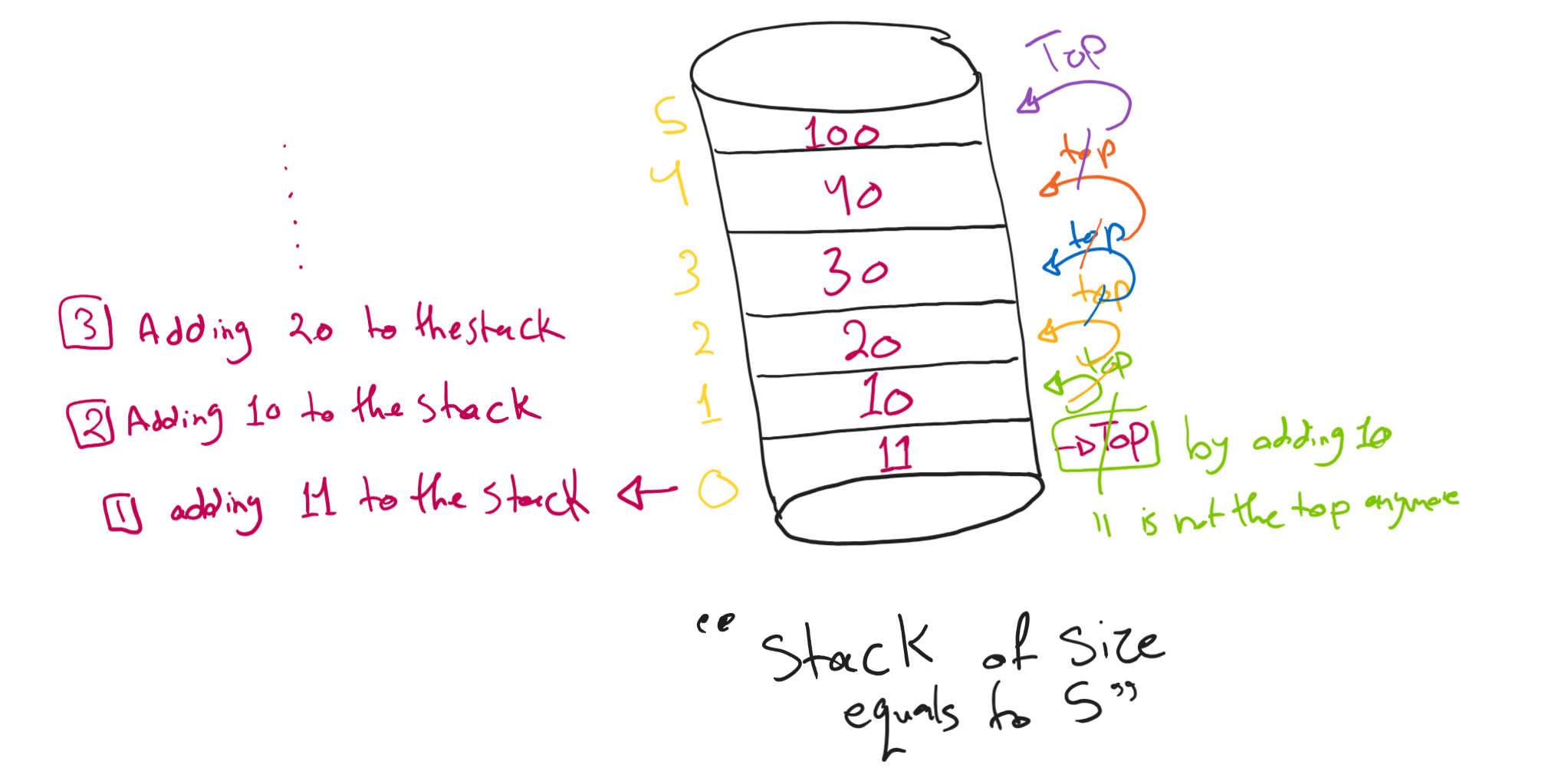
1. Push(x): adding an element to the end of a stack. (by index, no iteration)
2. Pop(): returning the top with deleting it.
3. Top (): returning the top. (peek)
4. isEmpty(): check whether the stack is empty or not.
5. isFull(): checking whether the stack is full or not.



Note: for isEmpty or isFull, the output will be True or False based on the Stack itself.

For the Tima complexity, since all the operations above are by index (accessed by index, no iteration) it has a constant time, O(1).

**For changing the value of Top (the end) each time we add an element:**

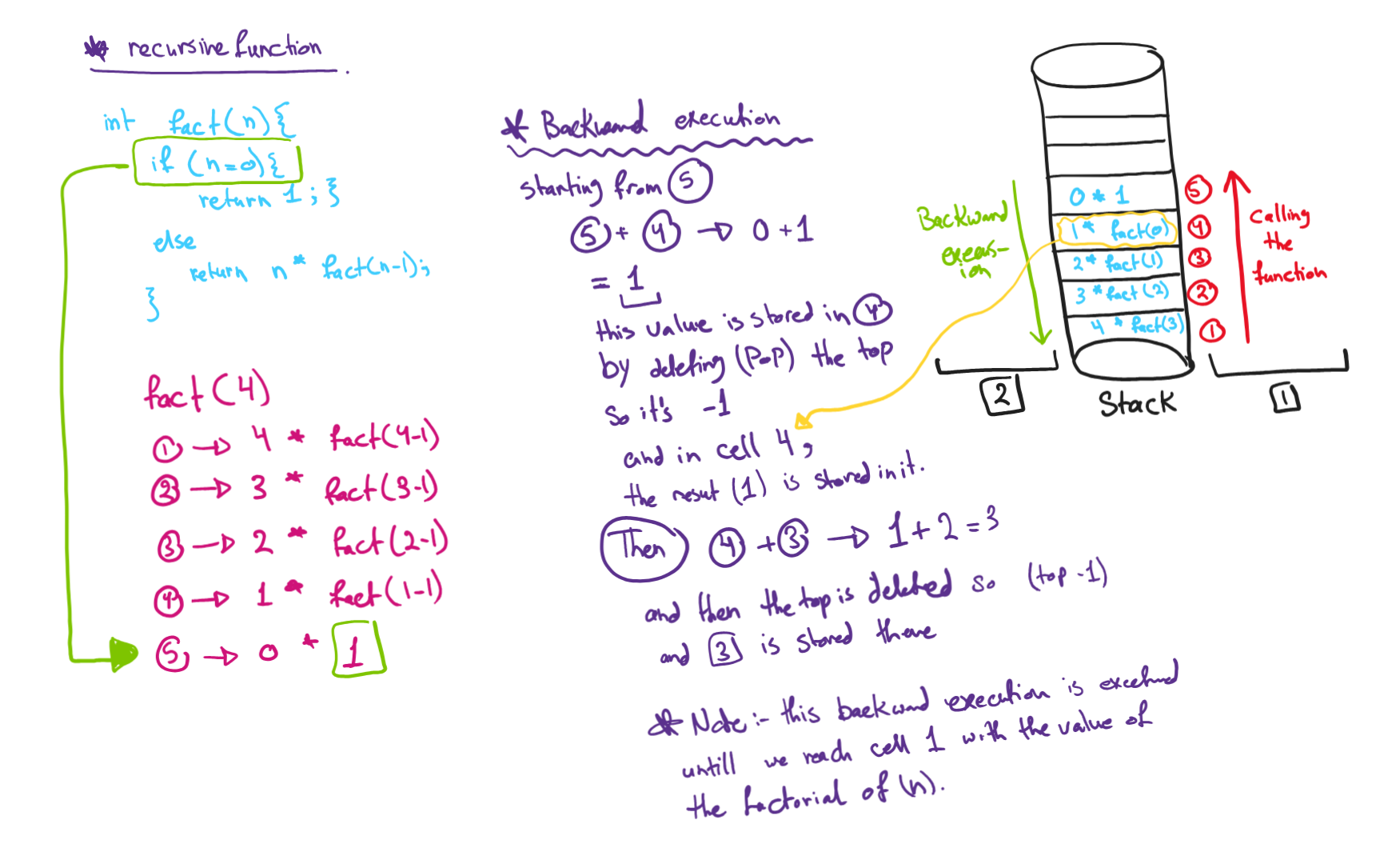


* As demonstrated above, the top is incremented by 1 each time we are adding an element to the stack.
* If we are removing elements then the top will be decreased by 1 each time we remove an element.
* Top referees to the idea of returning the last element and displaying it.

**Applications:**

1. Recursion: Stacks are used in recursive functions.

In this function, I have two executions (function call stack, return of function)



1. Undo/Redo operations: in Microsoft word application, when we type then delete the application saves the last thing that was deleted in an undo Stack pressing undo (Ctrl + z) moves the last thing saved to the redo Stack.[[2]](#footnote-2)
2. Browser History: when we are browsing the internet, our browser keeps tracking the web pages that we have visited, this tracking (history) helps in moving back and forth for the pages that I have seen.[[3]](#footnote-3)

For bringing back a page (Ctrl + Shift + T)

For deleting a page (Ctrl + W)

**Implementing of the Stack:**

We can implement the stack using Array List or Linked List.

*Stack with Array-List*

Array List 🡪 consecutive items (accessing by index) and for its operations [ Top(), isEmpty(), is Full(), push(x), pop()] the time complexity is O(1), constant.

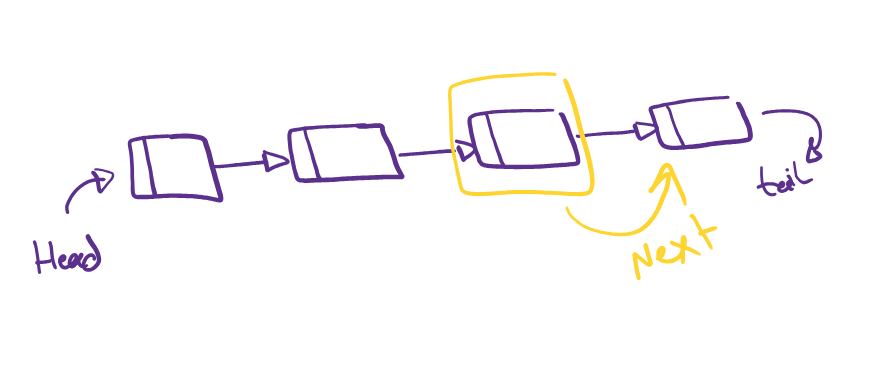
But, what if the Array is Full? We can create a new array with a higher size than the full one, and then we copy all the elements to the new one.

Well, this is not practical at all the time complexity will be O(n), because we are copying each element to the new array.

So instead of this non-practical process, we use Linked-List.

*Stack with Linked-List*

No consecutive items, no indexes, and the cells are connected through out pointers



Here in linked list, we have no index so for accessing we need to start from head and walk through each element in the stack, so I have iterations, this effects the time complexity by having it O(n).

For the time complexity:

* For all [push(x), pop(), peek()] 🡪 O(1) as we are not doing iterations.
* For isEmpty() 🡪 O(n), where n is the size of the stack, (by doing iterations)

Using linked-list is really helpful because you can add or remove from the stack without the need to set a fixed size.

Task 2 - Application (user) level

We use the ADT to solve a problem. When working at this level we only need to know how to create instances of the ADT and invoke its operations.

1. Referring to the system description in task1.1 and its structure specifications, examine the advantages of encapsulation and information hiding when using an ADT in the main method. support your answers with examples.[[4]](#footnote-4)

Encapsulation is somehow same as (in capsule), in java all the setting of variables of classes are inside a method. **encapsulation is simply bundling data with operations that act on the data.**

**In General – advantages of using encapsulation and information hiding:**

1. **Modularity: Encapsulation lets us hide how a class does its job. This means we can tweak how the class works without messing up other parts of our program. It helps keep different parts of our code separate and makes it simpler to make changes later on. Which makes the usability of the code more efficiently.**
2. **Code organization: Encapsulation helps us organize our code neatly, with clear lines between different parts. This makes the code easier to read and find our way around, which means it's simpler to keep everything running smoothly. It also makes it easier for team members to work together and understand each other's code.**
3. **Versioning and Maintenance: When we need to tweak a class, encapsulation lets us adjust how it works inside without changing how it interacts with the outside world. This makes it easier to manage different versions of the code and keep everything running smoothly, as we can make changes without messing up other parts of our program.**

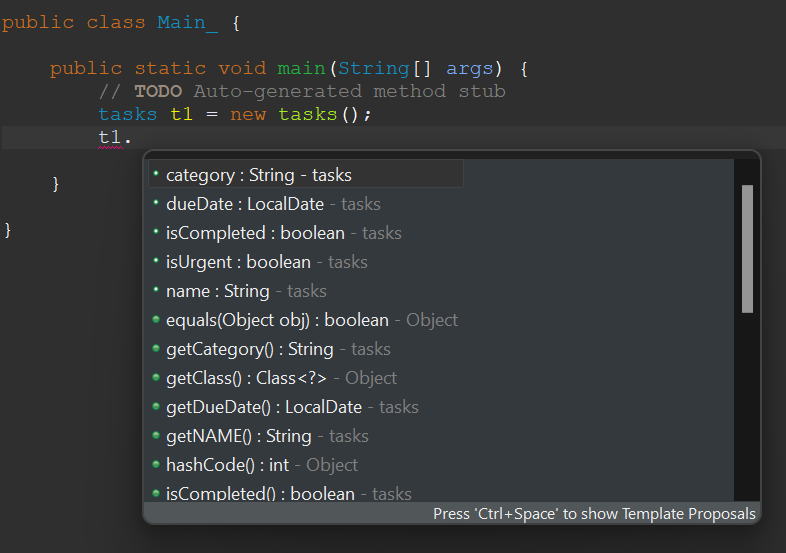
**For the code (Task management system) examples**

***Without encapsulation***

**A screen shot of a computer program

Description automatically generated**

here we have public modifier, which make it accessible from any external classes. Using Main class to set the details of the task can be done by creating a new object of task then filling its details using.



This is not a good practice in OOP, we want to call methods that do this for us, instead of typing “= this”

And one crucial point that we do not want the data to be visible for external classes

*With encapsulation (using getters and setters)*

Our goal is to improve the design, We can bundle the data and the methods operating on it into a single class.

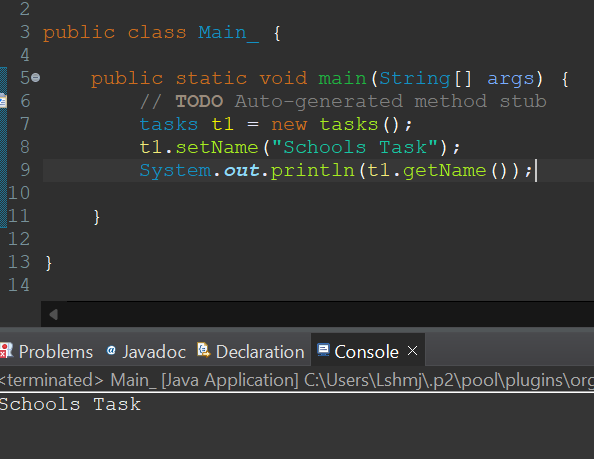
A screen shot of a computer program

Description automatically generated

Now setting getters and setters make the code modular, users can easily call GetName and setName without having much knowledge about its implementation.

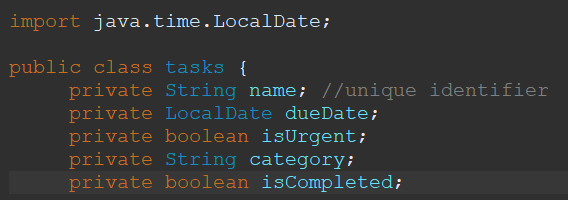
Encapsulation is a key idea in object-oriented programming where we keep data (attributes) and methods (functions) together in one place, called a class. It hides the internal details of the class, allowing only controlled access to the data through special methods (getters and setters), which keeps the data safe from unwanted changes.

Using getters and setters in the main



And we can even make encapsulation more strict by using information hiding.

*Information hiding*



**Encapsulation**

Encapsulation is the concept of bundling data and methods that operate on the data within a single unit, usually a class. This approach provides several benefits. Firstly, it offers data protection by keeping data (variables) private and only exposing necessary methods (such as getters and setters), thereby preventing external code from directly modifying the internal state of an object and reducing the chance of accidental data corruption. Secondly, it enhances modularity by creating self-contained modules where each class has its own data and behavior, making the code easier to understand and maintain. Thirdly, encapsulation facilitates ease of maintenance. When internal implementation changes are needed, they can be made without affecting the code that uses the class, provided the public interface (methods) remains unchanged. This results in fewer bugs and less effort when updating or improving the code.

**Information Hiding**

Information hiding is a principle where the internal details of a class are concealed from the outside world, exposing only the necessary parts of the object. This principle offers several advantages. It reduces complexity by hiding the complex implementation details, allowing users of the class to interact with it through a simple interface without needing to understand its internal workings. Improved security is another benefit, as sensitive data is protected from unintended interference. For example, if a class handles sensitive information like passwords or account details, hiding this information prevents unauthorized access and manipulation. Additionally, information hiding enhances flexibility. Since the implementation details are concealed, internal changes can be made without affecting the classes that depend on it, making the code more adaptable and easier to refactor.

**Advantages in the Main Method**

When using an Abstract Data Type (ADT) in the main method, encapsulation and information hiding provide significant advantages. Encapsulation simplifies usage by allowing the main method to interact with the class through its public methods, without needing to know the internal details of how data is managed. This abstraction helps in reducing errors, as the main method cannot directly change private variables, preventing accidental mistakes such as setting an invalid state directly. Overall, encapsulation and information hiding make the code more robust, maintainable, and easier to understand.

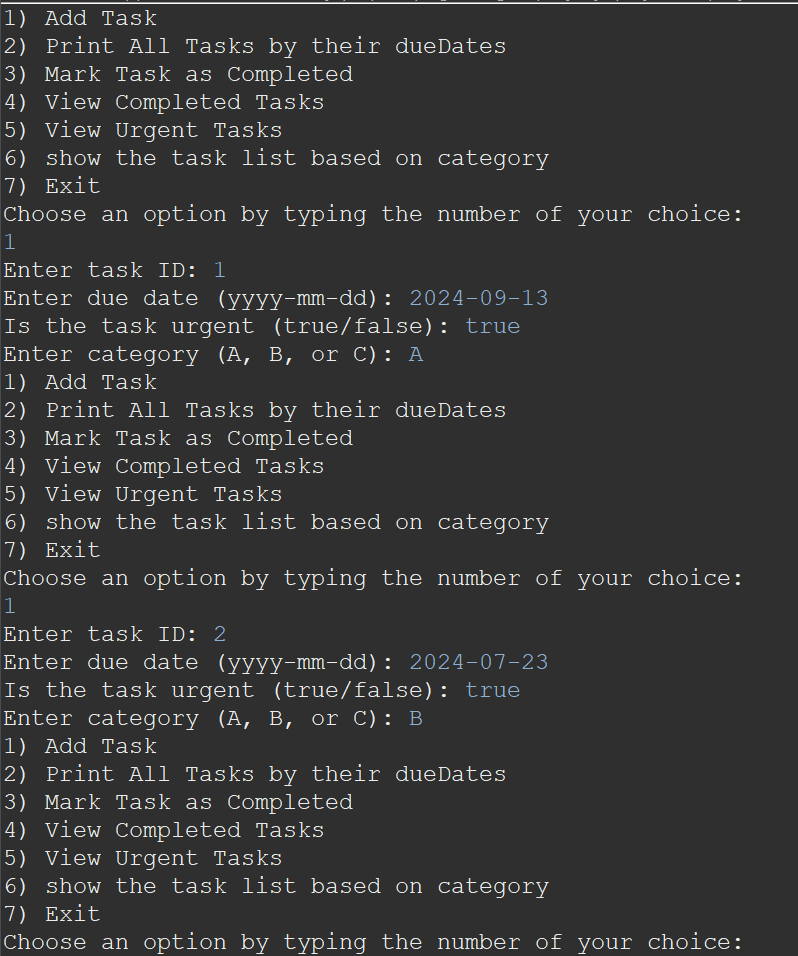
Information hiding is a programming principle that stops direct changes to a class's data by setting strict rules for accessing and modifying it. It keeps design details hidden from the user, especially parts that might change. When combined with encapsulation, information hiding helps create modular and easy-to-manage code.

In summary, encapsulation packages data and methods together, while information hiding makes sure only the necessary details are visible from outside. Together, they help create strong and easy-to-maintain software systems.

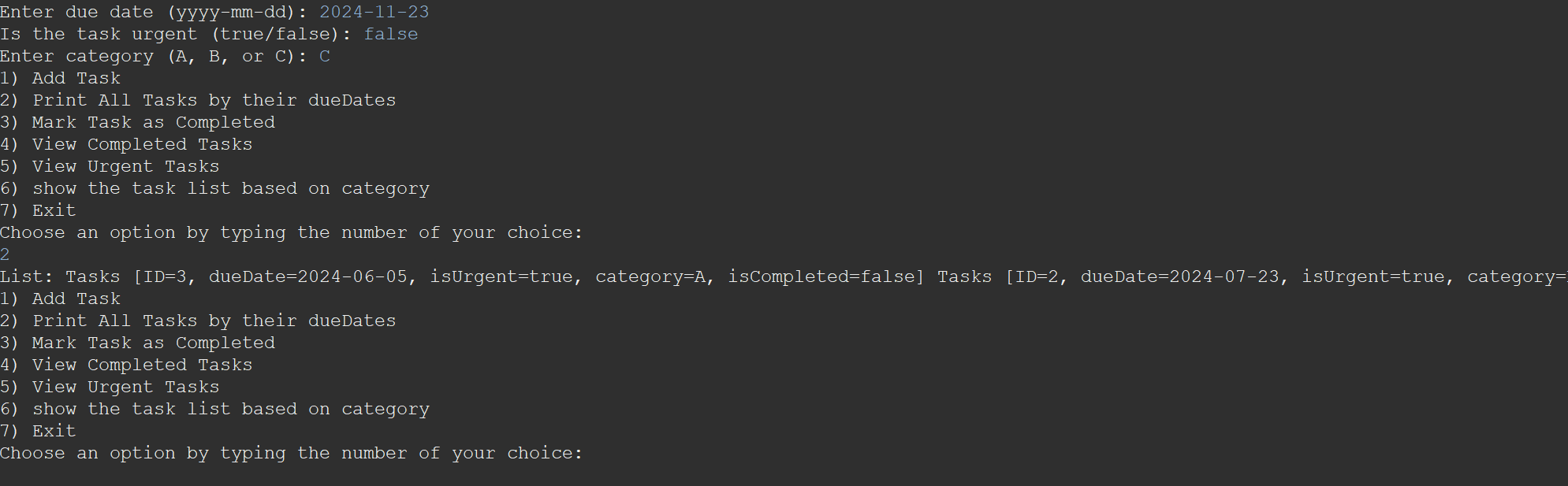
Task 3 - Implementation (concrete) level

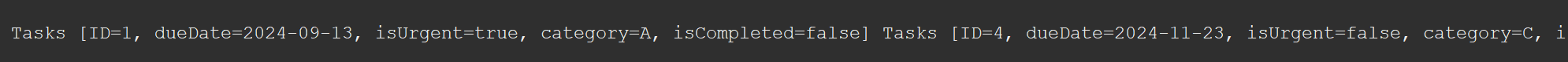
1. Implement the designed solution in task 1 for the presented problem using an executable programming language including error handling mechanisms. Run a testing of the application (executable main method) and document the outcomes.

**Adding tasks to the systems**

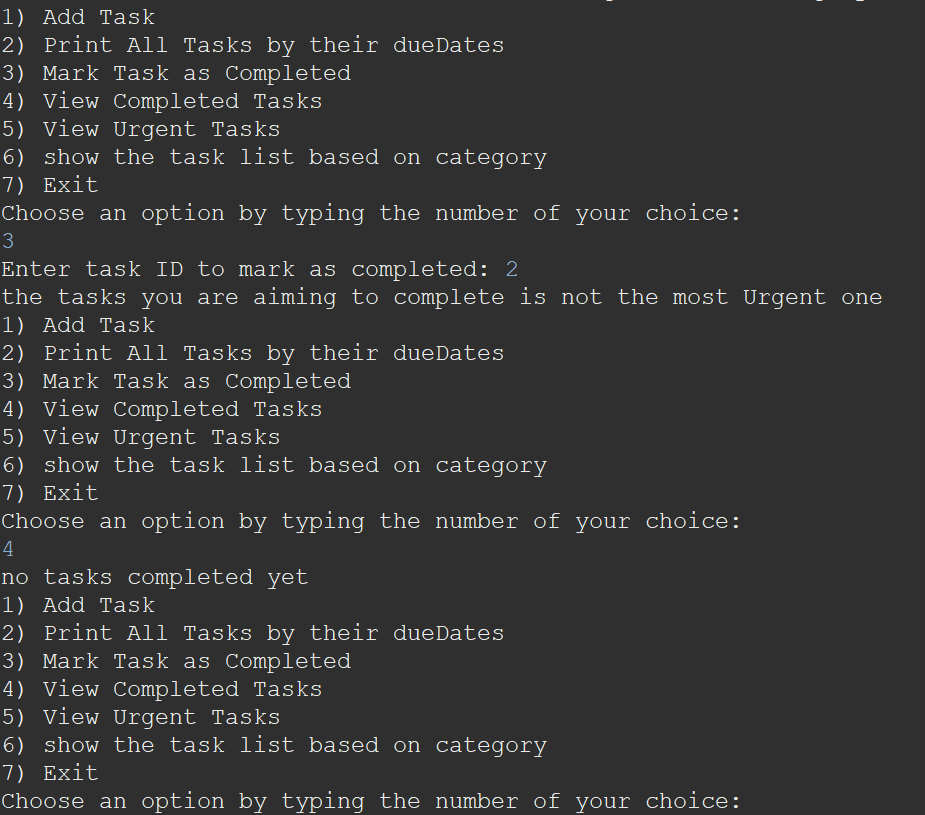


**Printing all tasks by due dates**

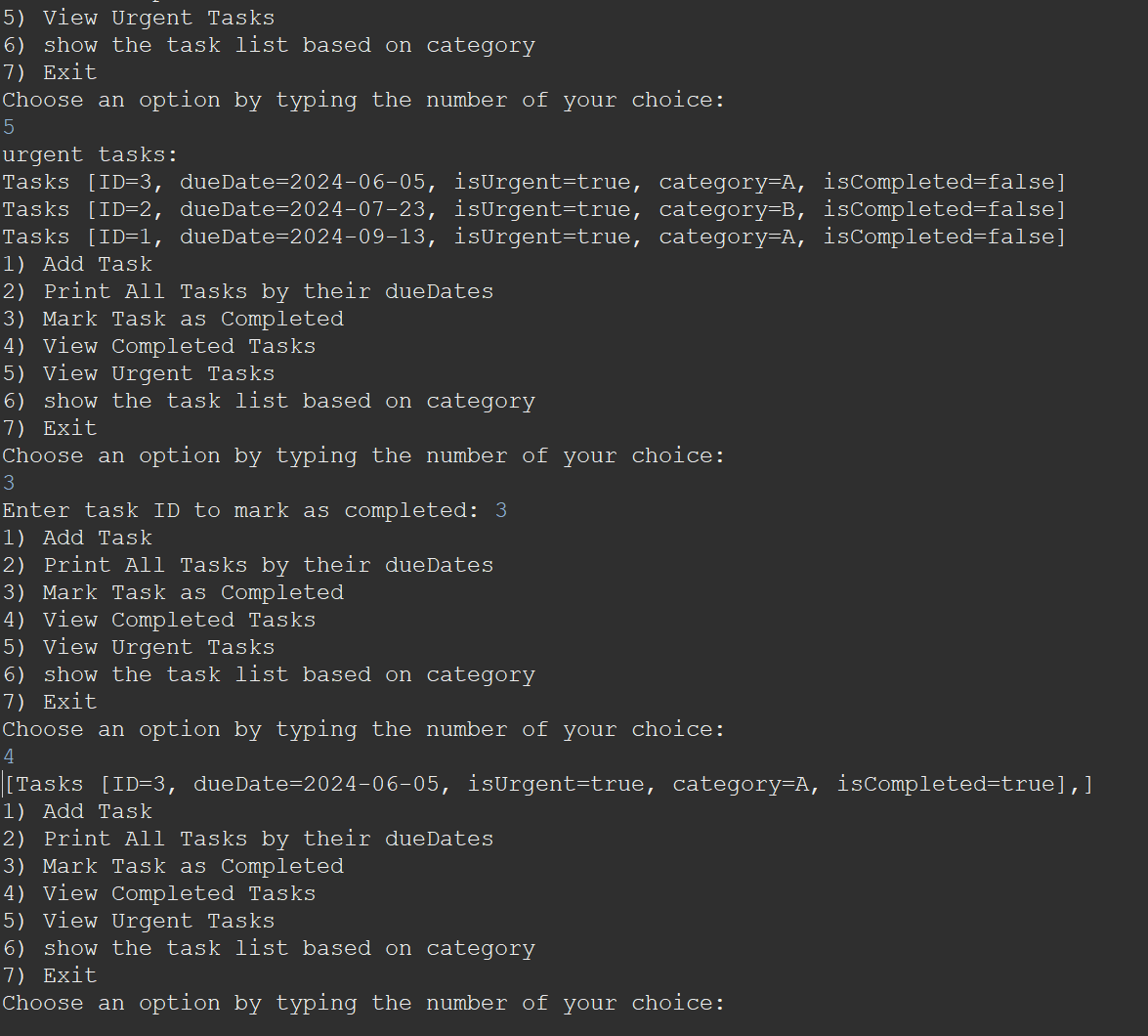




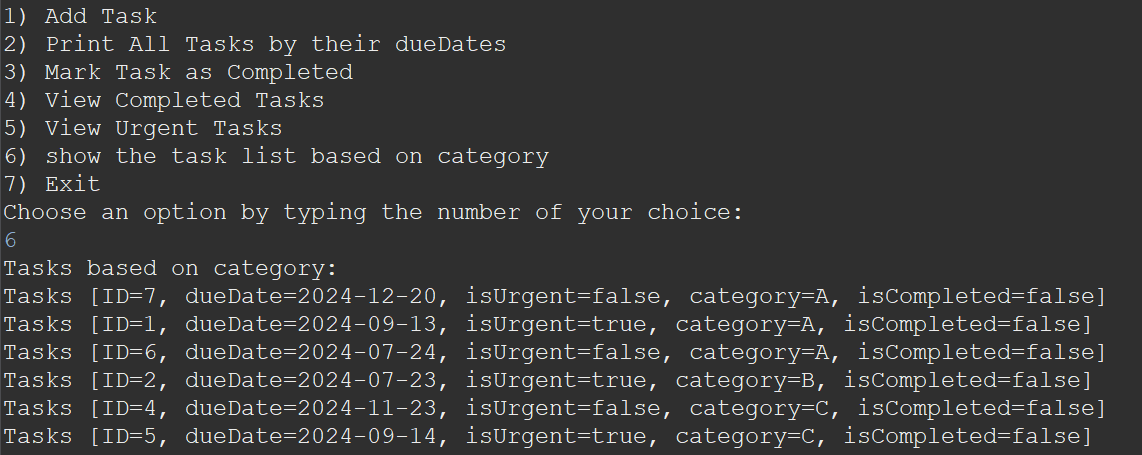
**Mark tasks as completed**



**Printing the urgent tasks to get the most recent added task to make it as completed**



**Show tasks based on categories**

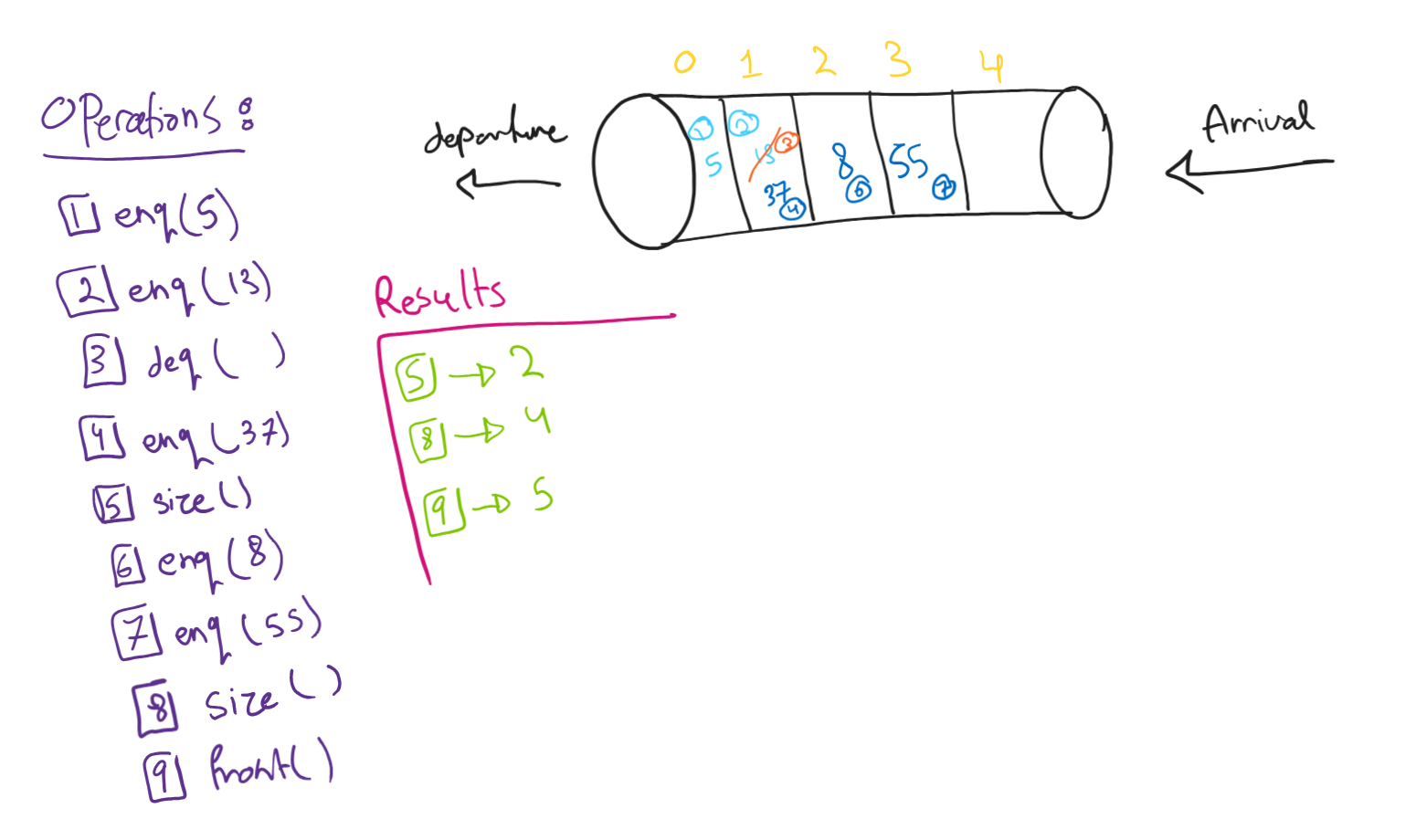


1. Referring to your implementation of FIFO queue, and the test cases in the experiment, provide an illustrate of at least 4 operations of queue data structure.

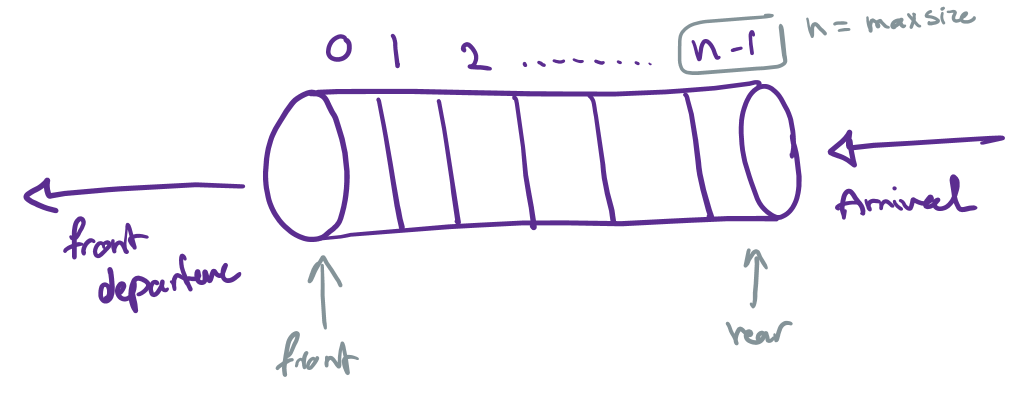
A Queue is a Data Structure that follows the concept of FIFO, first in first out, that means the first element to be added is the first element to be removed. [[5]](#footnote-5)

**Queue Operations:**

1. EnQueue: adding an element to the rear of the Queue.
2. DeQueue: Deleting an element form the front of the Queue (first element in a Queue).
3. IsEmpty: Checks if the Queue is empty or not.
4. isFull: Checks if the Queue is full or not
5. Front: returning the first element of the Queue.
6. Size: returning how many element are in the Queue.



**Design:**



* We first need two variables, front and rear, front points on the first element and the rear points on the last element.
* When the Queue is empty, front and rear point at the same value which is -1.

**Tracing for the operations with focusing on front and rear**

A whiteboard with colorful writing

Description automatically generated

We have now two special cases:

* If the Queue is empty, then when performing dequeue will raise an error.
* If the Queue is full, and we perform enqueue will raise an error.

A whiteboard with writing on it

Description automatically generated



**Applications of Queue:**

1. Priority Queues: Priority queues are queues where items are removed based on their priority level, not just the order they arrived. This is useful when some tasks need to be done before others. For example, in a hospital, patients with serious conditions are treated before those with less urgent needs. In computing, priority queues are often built using structures like heaps to quickly access the highest-priority item.
2. Task Scheduling:[[6]](#footnote-6)

A screenshot of a computer

Description automatically generated

Task scheduling uses queues to organize and execute tasks in a specific order. scheduling makes sure resources are used efficiently and tasks get done on time. For example, in a printer queue, print jobs are processed in the order they arrive, so all tasks eventually get completed.

1. Data Transformation in network communications: Queues help manage data as it's sent and received over a network. Data is broken into small packets, which are lined up in a queue to be sent. On the other side, packets are queued up again to be put back together in the right order. This makes sure data is sent and received smoothly and correctly, even if packets arrive at different times.

**Time complexity of Queue: [[7]](#footnote-7)**

|  |  |  |
| --- | --- | --- |
| Queue/operations | Array List | Linked List |
| **Enqueue**, inserting an element to the end of the Queue | O(1), because single element in inserted at the last position which is done in a constant time. | O(1), new node is created and the pointer of the rear is incremented by 1. |
| **Dequeue**, removing an element from the front of the queue. | O(1), the front pointer is incremented by 1. | O(1), only first node is deleted and the front is incremented by 1. |
| **Peek(),** this operation return the element that is in front of the queue. | O(1), returning the element in the front of a queue. | O(1), returning the element in the front of a queue. |
| **isFull** | O(1), check if rear = max size -1, no iterations | It has dynamic size |
| **isEmpty** | O(1), front = -1 and rear = 1 | O(1), check if front = null or -1 |
| **Size(),** getting how many elements are in the Queue | O(1), rear – front +1, mathematical operation | O(1), rear – front +1, mathematical operation |
| **Front(),** returning the front element | O(1) | O(1) |

1. Referring to the scenario and the implemented solution, demonstrate how the ADTs/and Algorithms collaborate to address the problem at hand. How you think that using independent ADTs improves software development over different aspects; evaluate at least three benefits.

The Task Management System implemented uses various Abstract Data Types (ADTs) and algorithms to manage tasks efficiently. This system allows users to add tasks, view tasks based on due dates and categories, mark tasks as completed, and manage urgent tasks separately. The following ADTs are used in the implementation (each data structures plays a crucial role to handle the problem at hand):

* **LinkedList**
* **LinkedListStack**
* **QueueLinkedList**
* **ArrayList**

*The classes that exist in my system:*

1. **TaskManagementSystem Class**:

* Manages tasks using a combination of ADTs.
* Uses a linked list to store all tasks in order of their due dates.
* Uses a stack to keep track of urgent tasks.
* Uses a queue to manage completed tasks.

1. **Tasks Class**:

* Represents a task with attributes like ID, due date, urgency, category, and completion status.

1. **LinkedList Class**:

* Stores tasks and maintains their order based on due dates.
* Supports insertion, deletion, and retrieval operations.

1. **LinkedListStack Class**:

* Manages urgent tasks using a stack structure.
* Supports push and pop operations.

1. **QueueLinkedList Class**:

* Manages completed tasks using a queue structure (Maintains the order in which tasks were completed)
* Supports enqueue and dequeue operations.

1. **ArrayList**:

* Provides a dynamic array for task storage.
* Allows random access to tasks.
* Facilitates operations like insertion, deletion, and resizing.

**Collaboration in the Task Management Application**

**Insertion Sort by Due Date**

To manage tasks efficiently, the application uses an insertion sort algorithm to keep tasks sorted by their due dates. This is implemented within a LinkedList. When a new task is added, the system traverses the list to find the correct position based on the due date and inserts the task there. This ensures that the tasks are always in chronological order, making it easy to manage deadlines.

**How the System Works Together**

**Adding a Task**

When a new task is added to the system, it is first inserted into the LinkedList using the insertion sort algorithm. This keeps the tasks sorted by their due dates. If the task is marked as urgent, it is also added to a stack (LinkedListStack) to ensure that urgent tasks can be accessed quickly. Additionally, the task is inserted into the BST according to its category, allowing for efficient category-based retrieval.

**Marking a Task as Completed**

When a task is completed, it is first located within the LinkedList. If it is an urgent task, it is also removed from the LinkedListStack. The completed task is then enqueued into a QueueLinkedList, which keeps track of all completed tasks in the order they were finished. This process ensures that both ongoing and completed tasks are managed effectively, and urgent tasks are given priority.

**Viewing Tasks by Category**

To view tasks based on their categories, the system retrieves them from the BST using an in-order traversal. This type of traversal ensures that tasks are printed in a sorted order according to their categories. By organizing tasks this way, users can easily find and review tasks grouped by their respective categories, improving task management and overview.

*Adding a task to the system*

1. The user inputs the task details, Then a Tasks object is created with the provided details.

A whiteboard with blue writing

Description automatically generated

The addTask method of TaskManagementSystem calls the insertInOrder method of the LinkedList. The insertInOrder method finds the correct position in the list based on the due date and inserts the new task using isafter(), isbefore(). And if the task is urgent it will be pushed to a stack of urgent tasks.

For the application implemented, the scenario where it is need to manage a series of tasks that require different processing orders: some tasks need to be processed in the order they arrive, while others need to be processed in a last-in-first-out (LIFO) manner. To handle these tasks effectively, we implement a solution using Linked Lists, Linked List Stacks, and Linked List Queues.

1. **Linked List**

* **Structure**: A linked list is a linear collection of nodes where each node points to the next node. This structure allows dynamic memory allocation, efficient insertion, and deletion operations. With a time complexity of O(1).
* **Usage**: In our scenario, a linked list can serve as the underlying structure for both stacks and queues, providing the flexibility to manage the data dynamically. Which deals with the time complexity perfectly.

Here, we have added 4 tasks to the linked list calling it, when one of the tasks are completed for example task 2, there will be no shifting to all other tasks, its just the pointer (.next)

A diagram of a diagram

Description automatically generated

 **Linked List Stack**

* **Structure**: A stack implemented using a linked list (Linked List Stack) maintains the LIFO order. The top element is always at the head of the linked list.
* **Algorithms**:
* **Push**: Add an element at the head of the list.
* **Pop**: Remove and return the element from the head of the list.
* **Peek**: Retrieve the element at the head without removing it.
* **Usage**: Ideal for scenarios where the most recently added tasks need to be processed first, such as undo operations in text editors or syntax parsing in compilers.

In our application, the most recent added urgent task, must be the first to be completed, so with the concept of LIFO the stack is the aimed data structures for this part.

**Linked List Queue**

* **Structure**: A queue implemented using a linked list (Linked List Queue) maintains the FIFO order. Elements are added at the tail and removed from the head.
* **Algorithms**:
* **Enqueue**: Add an element at the tail of the list.
* **Dequeue**: Remove and return the element from the head of the list.
* **Peek**: Retrieve the element at the head without removing it.
* **Usage**: Suitable for scenarios where tasks need to be processed in the order they arrive, such as print job management or task scheduling in operating systems.

A drawing of a diagram

Description automatically generated

**Benefits of using ADTs:**

 **Modularity and Reusability**

Independent ADTs encapsulate data and operations in a single unit. This modularity makes it easy to reuse ADTs across different projects. Developers can use the same stack or queue implementation in various applications without rewriting code, saving time and effort.

 **Maintainability and Scalability**

ADTs provide a clear interface for data manipulation. Changes to the internal implementation do not affect the external interface. Enhancing or modifying an ADT's implementation becomes easier without impacting the rest of the system. This approach supports scalability and long-term maintenance.

 **Abstraction and Flexibility**

ADTs abstract the underlying data structures, allowing developers to focus on higher-level logic without worrying about low-level details. This abstraction leads to cleaner code and easier debugging. Developers can switch between different implementations (e.g., array-based vs. linked list-based) without altering the core logic.

Abstract Data Types (ADTs) encapsulate data and operations into a single unit, making it easier to manage and modify data structures. They provide abstraction, so users don't need to know the implementation details, simplifying programming and reducing errors. ADTs can be implemented using different data structures, offering flexibility and adaptability to changing needs. By controlling access to data, ADTs ensure data integrity and prevent unauthorized modifications. They also support modularity, allowing ADTs to be combined to create more complex structures, enhancing flexibility and organization in programming. Overall, ADTs are a powerful tool for efficient data management.

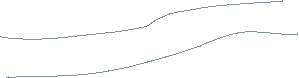
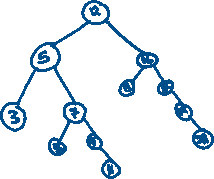
1. To implement the necessary functionalities in the task management system, you may opt to utilize existing Abstract Data Types (ADTs) or devise new ones. However, it's crucial to analyze the trade offs in terms of time and space complexities when considering ADTs such as Binary Search Trees (BST), Sorted Arrays, and Ordered Linked Lists for operations like search, insert, and remove. Interpret the trade-offs through various implementations, outlining the comparative analysis of each ADT, the impact on the overall performance of the task management system.

|  |  |  |  |
| --- | --- | --- | --- |
|  | BST | Ordered Linked List | Sorted Array (selection sort) |
| Time complexity | **Search**: O(log n)  **Insert**: O(log n)  **Remove**: O(log n) | **Search**: O(n), since we will iterate over the linked list, it might be O(1) if the element was located in the head.  **Insert:** O(n)  **Remove:** O(n) | **Search**: O(n)  **Insert**: O(n) as elements need to be shifted to maintain order.  **Remove**: O(n) as elements need to be shifted to maintain order. |
| Space complexity | O(n) for storing n elements. | O(n) for storing n elements. | O(n) for storing n elements. |

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

A BST is a hierarchical data structure where each node has at most two children, referred to as the left child and the right child. For each node, all elements in the left subtree are less than the node, and all elements in the right subtree are greater. It Provides efficient search, insertion, and removal operations on average.For a BST the high for the tree is the number of levels it contains.

High = log(n+1) – 1 🡪 time complexity: O(log n)



The length of the bath depends on the high of the tree, and when searching for an element, we move by one exact path.

1. **Sorted Array**

Ordered linked lists are useful for maintaining order with frequent updates, making them suitable for scenarios where tasks need to be inserted and deleted frequently while maintaining order. They offer memory flexibility since they can grow dynamically and don’t require contiguous memory blocks. However, search operations are linear (O(n)), which makes them less suitable for large datasets where fast search is required. Each node also incurs additional memory overhead for pointers, increasing overall memory usage. They are ideal for dynamic task lists where the task list changes frequently, and maintaining an ordered list is necessary, as well as environments with frequent insertions and deletions, requiring flexible memory management.

1. **Ordered Linked List**

An ordered linked list maintains elements in sorted order using nodes that contain data and a reference to the next node. Simple to implement. Dynamic in nature; elements can be easily inserted and removed without shifting.

Considering the trade-offs and the requirements of the task management system, the recommended approaches are to use a Balanced Binary Search Tree and an Ordered Linked List. A self-balancing BST is preferred due to its efficient search, insert, and remove operations, offering O(log n) time complexity, which ensures balanced performance for dynamic task management, handling frequent additions, deletions, and lookups efficiently. Alternatively, an Ordered Linked List is a suitable choice for moderate-sized datasets, providing a simple implementation that maintains sorted order naturally. This approach is adequate for managing tasks based on due dates and categories, especially when the number of tasks is not excessively large.

1. Discuss the foundational role of ADTs in Object Oriented Programming, support your answer with examples from your code.

Data structures and Object-Oriented Programming (OOP) are basic ideas in computer science that make each other better and more useful. To understand how they work together well, let's look at their main ideas and how they help each other.

**Data Structures and their roles**

Data structures are important for organizing and storing data in a way that makes it easy to use. Examples include arrays, linked lists, stacks, queues, trees, and graphs. Each type is designed for different tasks, and choosing the right one helps make programs run faster and more efficiently.

**Object-Oriented Programming (OOP)**

OOP is a way of programming based on "objects," which are like mini-programs that hold data (attributes) and actions (methods). The key idea of OOP:

* Encapsulation: Keeping data and methods that use the data together in one place.
* Inheritance: Creating new objects based on existing ones.
* Polymorphism: Using objects in different ways based on their data type or class.
* Abstraction: Hiding complex details to make things simpler.



**How Data Structures Enhance OOP?**

1. Encapsulation And Information Hiding: data structures encapsulate the data and provide methods to work with these data. By this we make sure that the data is accessed with well-defined interfaces, so no unauthorized people could access it. For example, a Queue class hides how the queue works inside and provides methods like enqueue() and dequeue() for adding and removing items. This way, users don't need to know how it works inside.
2. Modularity and reusability: When data structures are made as classes, they become easy to manage and use in different programs. Each data structure can be a separate class with its own methods and properties. For example, a LinkedList class can be used in many different programs without changing how it works inside, making it easy to reuse code and reduce repetition.



1. Inheritance and polymorphism: OOP lets you create new data structures based on existing ones. A base class can be extended to make more specific types. For example, a Collection class can be the base for Queue, Stack, and List classes. Polymorphism lets you use these classes in a flexible way, allowing for easier updates and expansions.

**How OOP Enhances Data Structures**

1. Clear Structure and organization and code readability: OOP makes it easier to create and understand data structures by using classes and objects. This leads to a clear and organized way to implement them. For example, having a Node class for a linked list makes the structure and connections within the data clear and easy to manage.

A screen shot of a computer program

Description automatically generated

1. Improved Maintainability: Encapsulation in OOP helps keep data structures easy to maintain. You can change the internal workings of a data structure without affecting the rest of the code, as long as the interface stays the same. For instance, if you switch from using an array to a linked list inside a Dynamic Array class, you don't need to change the methods that use this class, as long as those methods work the same way on the outside.
2. Enhanced Collaboration: OOP helps developers work together better by defining clear interfaces and keeping implementation details hidden. This allows different developers to work on different parts of a project without causing conflicts. For example, one developer can work on creating the Queue class while another developer works on the program that uses the Queue class, with fewer issues due to dependencies.
3. Abstraction: OOP principles allow for abstraction, where complex data structures can be represented with simple interfaces. This makes it easier to work with complex data without needing to understand the underlying implementation. For example, an interface for a List can provide methods like add(), remove(), and get(), abstracting the underlying data structure (array, linked list, etc.).

**Symbiotic relationship**

In conclusion, data structures and OOP work together and make each other better. Data structures help organize and manage data efficiently, while OOP provides a clear and structured way to implement these data structures. This makes the code easier to maintain and reuse. When combined, they help build strong, scalable, and easy-to-maintain software systems. This relationship is especially important in complex applications where efficient data structures are needed for good performance, and OOP principles keep the implementation organized and modular.

Part 2 - Algorithms Performance and Efficiency.

Imagine you intend to arrange the task list in ascending order based on their due dates. For simplified implementation and efficient handling with sorting algorithms, let's assume you're provided with an array of integers denoting days from 1 to 365. In this representation, lower numbers signify closer due dates.

1. Experimentally compare two sorting algorithms: Merge Sort and the enhanced version of the Selection Sort with the following requirements:

**Visualize the results in chart (size vs time)**

A graph of different colored bars

Description automatically generated with medium confidence

A graph of multiple orange bars

Description automatically generated with medium confidence

**Conclude insights and discuss the findings in the light of each algorithm complexity and different array sizes and types**

|  |  |  |  |
| --- | --- | --- | --- |
| Array Size | Array Type | Merge Sort Time (ms) | Enhanced Selection Sort Time (ms) |
| 100,000 | Sorted | 35 | 13375 |
| 100,000 | Random | 30 | 13373 |
| 100,000 | Reversed | 13 | 15468 |
| 1,000,000 | Sorted | 160 | 1028996 |
| 1,000,000 | Random | 220 | 932938 |
| 1,000,000 | Reversed | 103 | 1411612 |

We've taken a close look at Merge Sort and Enhanced Selection Sort to see how they handle different array sizes and types. Here's what we found:

**Merge Sort vs Enhanced Selection Sort**

Merge Sort consistently performed better than Enhanced Selection Sort in all our tests. Even when the arrays were already sorted, Merge Sort still came out on top, leaving Enhanced Selection Sort far behind with slower times.

**Different Array Types, Different Results**

Merge Sort didn't seem to care much about whether the array was sorted, random, or reversed. It handled them all pretty well, sorting them efficiently. Enhanced Selection Sort struggled more with sorted and reversed arrays, showing its weakness when the data wasn't completely random.

**Size Matters**

As we expected, both algorithms took longer to sort larger arrays. Merge Sort's time increased, but it stayed manageable because of its smart approach. Enhanced Selection Sort, though, really struggled with bigger arrays, taking a lot longer to finish sorting them.

**Practical Considerations**

For everyday tasks with smaller or somewhat jumbled data, Merge Sort is a great choice. It's reliable and gets the job done quickly. But if you're dealing with massive datasets or data that's already somewhat sorted, Enhanced Selection Sort isn't the best option. Its performance drops off a lot in those situations.

Our experiments confirmed what we expected: Merge Sort is the go-to sorting algorithm for most situations. Its efficiency and reliability make it stand out. Enhanced Selection Sort, on the other hand, has its strengths but falls short when things get big or orderly. So, when you're picking a sorting algorithm, Merge Sort wins hands down for its consistent performance and effectiveness.

**Time Complexity**:

* Merge Sort consistently outperforms the enhanced Selection Sort for large arrays due to its O(nlogn) complexity.
* Enhanced Selection Sort's O(n^2) complexity becomes significantly disadvantageous as the array size increases.

**Array Types**:

* Merge Sort performs similarly across different array types due to its divide-and-conquer approach.
* Enhanced Selection Sort shows minor variations but is generally slower due to its inherent quadratic complexity.

**Calculate/predict the time required by each algorithm to sort 1010 input, given a CPU that executes an operation in 10-10 seconds.**

1. **Merge Sort:**

Merge Sort is a widely used sorting algorithm known for its efficiency. It has a time complexity of O(n log n), making it suitable for large datasets.

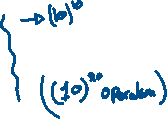


*For the above, considering the CPU speed of 10^-10 seconds per operation:*

Time required ≈ 230.258 billion operations \* 10^-10 seconds/operation ≈ 23.0258 seconds

**2. Enhanced Selection Sort:**

Enhanced Selection Sort is a variation of the Selection Sort algorithm, which repeatedly finds the minimum element from the unsorted portion and swaps it with the first element. Despite its simplicity, Enhanced Selection Sort has a time complexity of O(n^2), making it less efficient for large datasets compared to Merge Sort. For an input size of 1010 elements, the number of operations can be estimated as (1010)^2. Given the CPU execution time, the estimated time for Enhanced Selection Sort is approximately 10^10 seconds.



So, the predicted time required by each algorithm to sort 10^10 input on a CPU with a speed of 10^-10 seconds per operation would be approximately:

* **Merge Sort**: 23.0258 seconds
* **Enhanced Selection Sort**: 10^10 seconds (which is impractically long)

**Comparison and Analysis:**

Merge Sort demonstrates superior performance over Enhanced Selection Sort for large input sizes. Its divide-and-conquer approach results in faster sorting times, making it a preferred choice for handling large datasets efficiently. On the other hand, Enhanced Selection Sort, with its quadratic time complexity, is less efficient for sorting large datasets and may not be suitable for practical applications. This comparison highlights the importance of choosing an efficient sorting algorithm based on the size of the dataset.

1. Considering the recursive implementation of Merge Sort algorithm in the previous step, Determine how the memory stack frames are used when invoking the recursive function with an array of size 6 filled randomly. Determine the operations performed by a memory stack in detail. (Report)

A whiteboard with writing on it

Description automatically generated

1. Discuss how Big-O notation can be used to assess the efficiency of an algorithm. What is the asymptotic analysis of the two sorting algorithms implemented in the previous task. Compare the efficiency of both sorting algorithms based on this analysis. (Report)

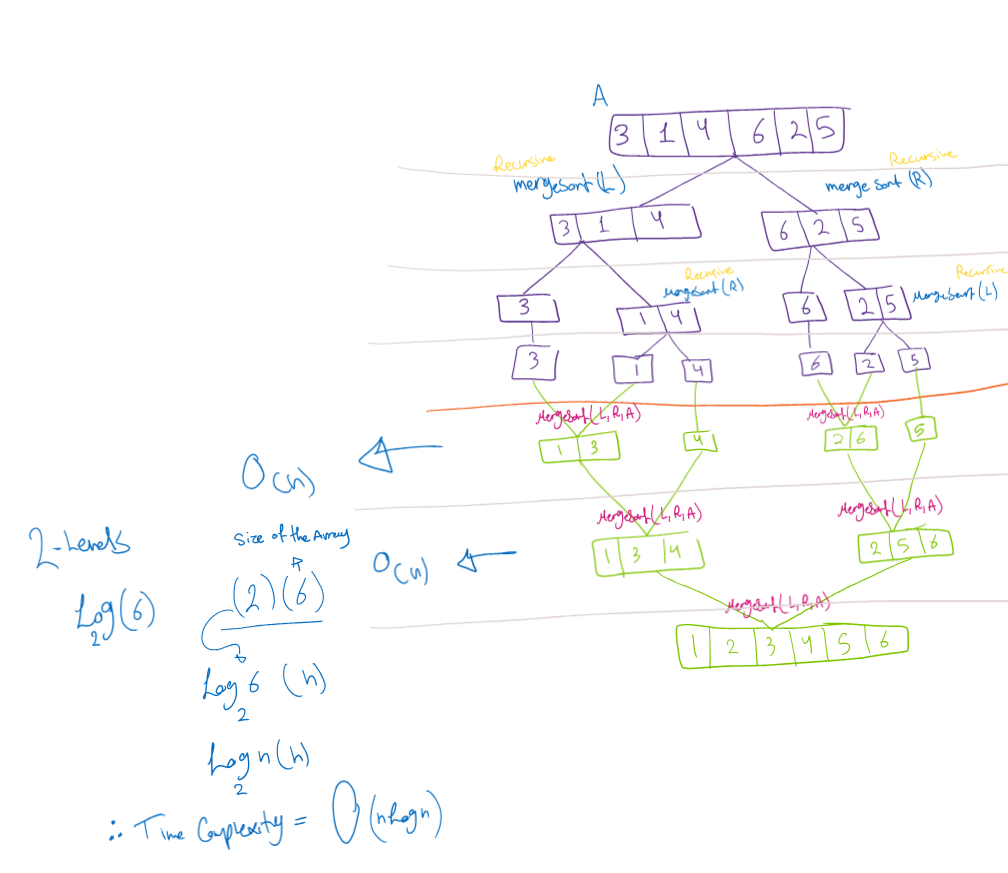
Big-O notation is a mathematical concept used to describe the performance or complexity of an algorithm. Specifically, it provides an upper bound on the time or space complexity of an algorithm as the input size grows. This notation helps in comparing the efficiency of different algorithms by focusing on their growth rates rather than exact execution times, which can vary based on hardware and implementation details.

**Key Concepts of Big-O Notation:**

1. **Time Complexity**: Indicates how the running time of an algorithm increases with the size of the input.
2. **Space Complexity**: Indicates how the memory usage of an algorithm increases with the size of the input.
3. **Asymptotic Behavior**: Big-O notation describes the behavior of an algorithm as the input size approaches highlighting the most significant factors affecting performance.

**Merge Sort**

* **Time Complexity**: O(n log n)
  + **Explanation**: Merge Sort divides the input array into two halves, recursively sorts them, and then merges the sorted halves. The division process takes O(log n) steps, and each merge operation takes O(n) time, resulting in an overall time complexity of O(n log n).



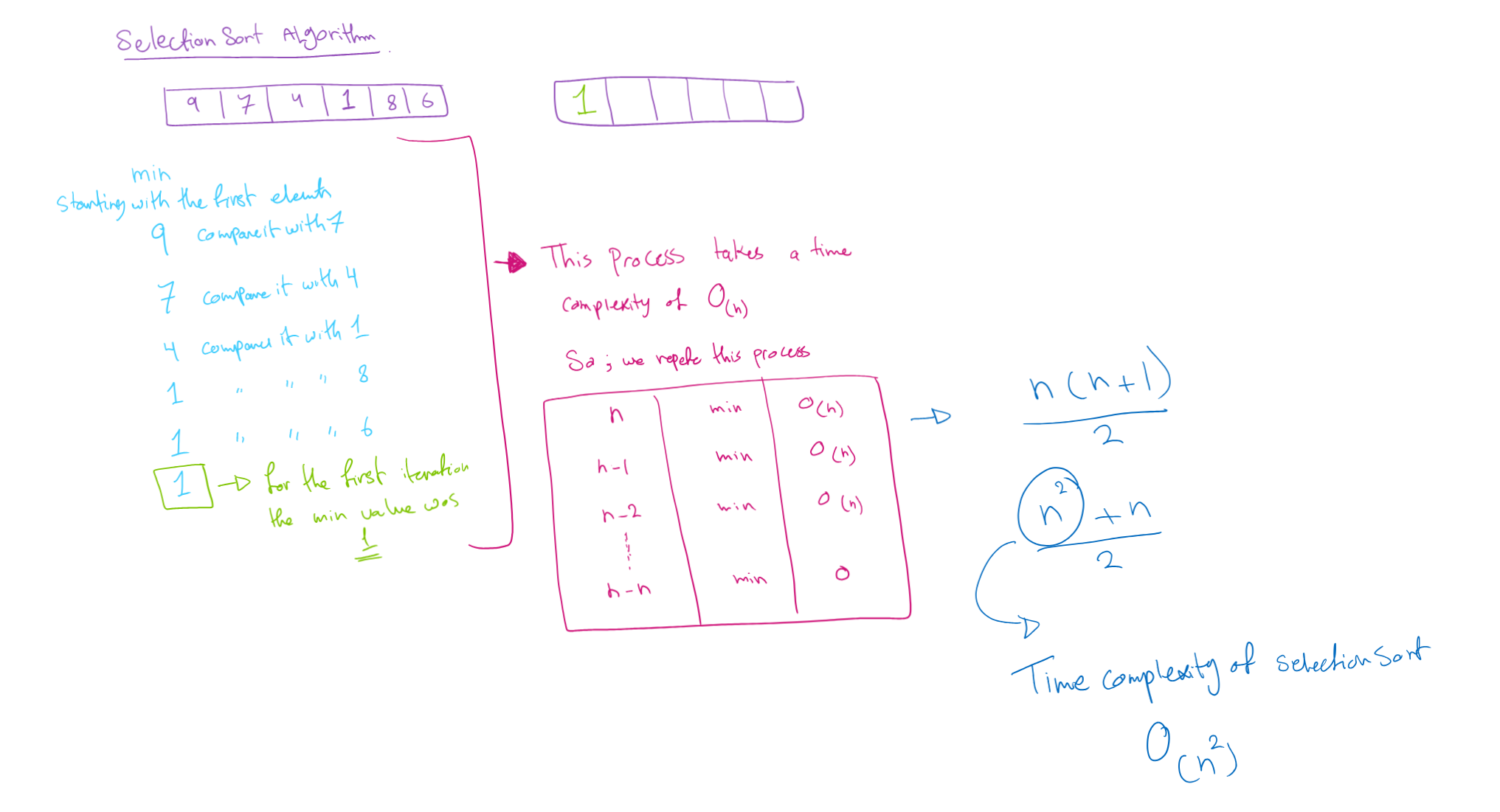
* **Space Complexity**: O(nlogn)
  + **Explanation**: Merge Sort requires additional space to store the two halves of the array during the merge process, leading to a space complexity of O(nlogn);

**Log n** 🡪 for the number of times the algorithm is dividing the array

**N 🡪** for the number of elements in the array

**Enhanced Selection Sort**

* **Time Complexity**: O(n^2)
  + **Explanation**: The enhanced version of Selection Sort finds both the minimum and maximum values in each iteration and places them in their correct positions. Despite this enhancement, the algorithm still requires O(n) iterations, and each iteration involves O(n) comparisons, leading to a time complexity of O(n^2).



* **Space Complexity**: O(n)
  + **Explanation**: In Enhanced Selection Sort one array is created, and we are swapping, so the space complexity is the number of elements in the array, it depends on the size of the array.

**Comparing the Efficiency of Merge Sort and Enhanced Selection Sort**

Based on the asymptotic analysis, we can compare the efficiency of both sorting algorithms:

**Time Complexity Comparison**

* **Merge Sort (O(n log n))**: This time complexity indicates that Merge Sort is significantly more efficient for large input sizes compared to Enhanced Selection Sort. making Merge Sort more scalable as the input size increases.
* **Enhanced Selection Sort (O(n^2))**: The quadratic time complexity indicates that Enhanced Selection Sort is less efficient for large input sizes. The performance deteriorates rapidly as the input size grows due to the nested loop structure, making it unsuitable for large datasets.

**Space Complexity Comparison**

* **Merge Sort (O(nLogn))**: Merge Sort requires additional memory proportional to the input size.
* **Enhanced Selection Sort (O(n))**: In Enhanced Selection Sort one array is created, and we are swapping, so the space complexity is the number of elements in the array, it depends on the size of the array.

**Practical Implications**

* **Large Datasets**: For large datasets, Merge Sort is the better choice due to its superior time complexity. It will handle large input sizes more efficiently and complete the sorting process faster than Enhanced Selection Sort.
* **Memory Constraints**: In scenarios where memory usage is a concern, Enhanced Selection Sort might be preferred due to its constant space complexity. However, this comes at the cost of significantly longer execution times for large inputs.

In conclusion, Big-O notation provides a framework for assessing and comparing the efficiency of algorithms by focusing on their growth rates. The asymptotic analysis reveals that Merge Sort, with its O(n log n) time complexity, is generally more efficient for sorting large arrays compared to Enhanced Selection Sort, which has an O(n^2) time complexity. However, the choice of algorithm can also depend on specific constraints such as available memory.

1. In the above task, you have conducted the experimental method of measuring the performance of an algorithm, determine two alternative methods can be used measure efficiency, illustrate your answer with an example. (Report)

Asymptotic analysis is a mathematical approach to analyze the performance of an algorithm by focusing on its growth rate as the input size increases. This analysis uses notations such as Big O, Big Omega, and Theta to describe the upper bound, lower bound, and tight bound of the algorithm's complexity, respectively.

**(Big O – upper bound and Omega – lower bound)**

When having an f(x), O(f(x)) 🡪 we call it **upper bound** (upper bound tells me what is the limit of my function) *Example: 3x^2 + x + 10, here the 3x^2 is the largest term in this function.*

If we have a function of 3n+5, can we say that 3n+5 =< 8n : n>= 1?

Testing: N = 1 🡪 does 3+5 =< 8? Yes

N = 2 🡪 does 6+5 =< 8 ? Yes

And till infinity the function is valid, we are aiming to find the limitations; one of the limitations is that the function is less than 8n.

Lower and upper bound theory is a mathematical concept used to find the smallest and largest possible values for a quantity under certain constraints or conditions. This concept is often applied in optimization problems where the goal is to determine the maximum or minimum value of a function given specific constraints. Mathematically, the lower bound of a set of numbers is the smallest number in that set, while the upper bound is the largest number.

Lower and upper bound theory also helps in determining the range of possible values for a variable. For instance, if we know that a variable must be between 0 and 1, its lower bound is 0 and its upper bound is 1. Lower and upper bound theory is crucial in mathematical analysis, optimization, and decision-making as it helps determine the range of possible values for a quantity and identify the optimal value within that range.

The lower and upper bound theory is particularly important in understanding the efficiency of algorithms. The lower bound, denoted by the asymptotic notation Big Omega , represents the minimum time complexity that any algorithm solving a common problem can achieve for random input. the upper bound, represented by the Big Oh (O) notation, signifies the maximum running time an algorithm can have. According to the lower bound theory, no algorithm can have a time complexity lower than the lower bound for random inputs. Every algorithm must take at least the lower bound, and L(n) is the minimum of all possible algorithms' maximum complexities.

The lower bound is essential because it helps compare the actual complexity of an algorithm. If the lower bound and the actual complexity are of the same order, the algorithm is considered optimal. An optimal algorithm has its upper bound equal to its lower bound. Merge Sort is a classic example of such an optimal algorithm. finding the lower bound of an algorithm is vital in ensuring its efficiency and optimality.

**Omega example**

f(n)=3n+5

We want to determine if, f(n) >= 3n for n>=1

* When n=1, 3(1)+5 = 8 , and 8 >=3
* When n=2, 3(2) +5, and 11 >= 3

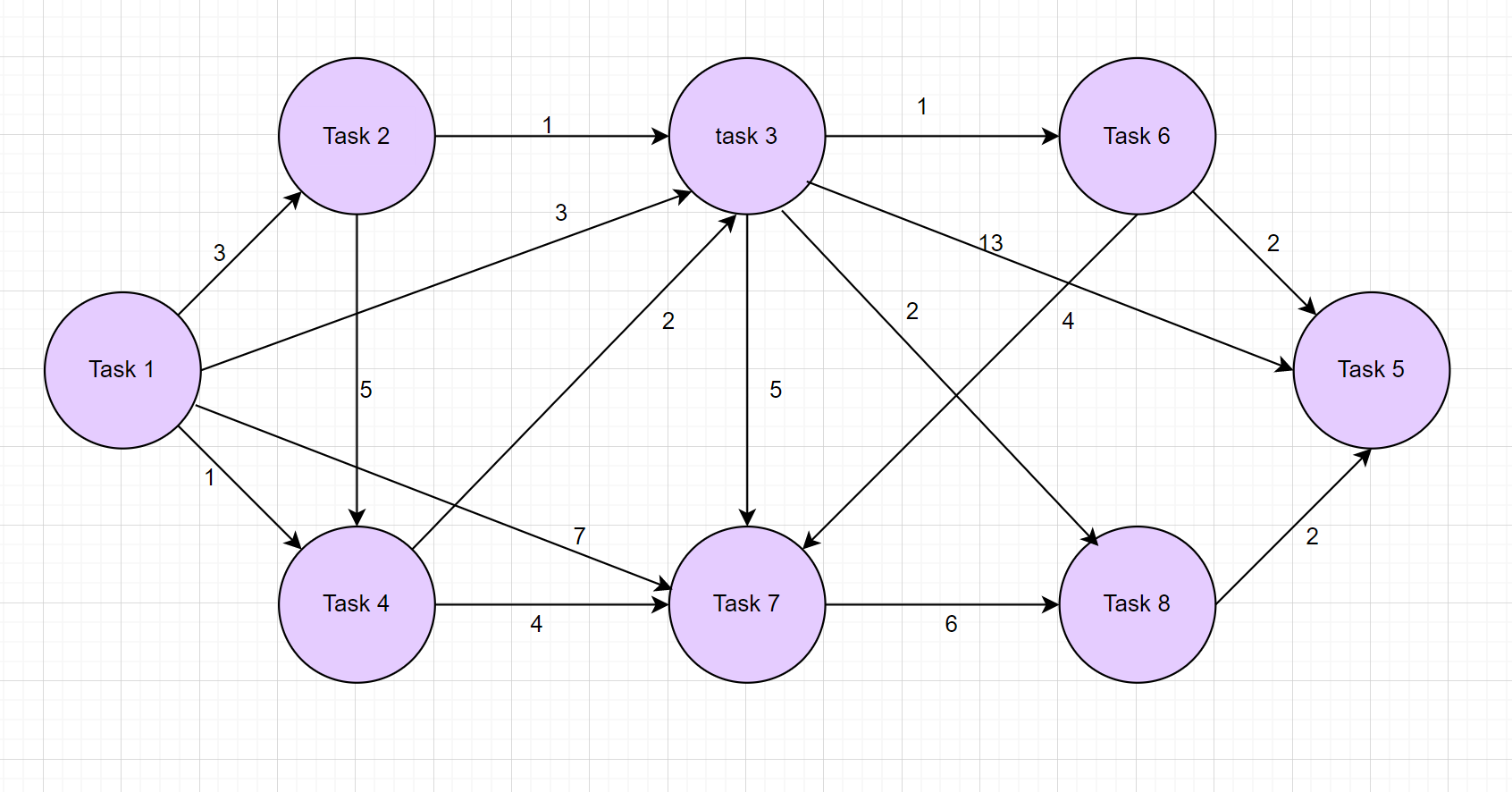
For n>=1, the function 3n+5 is always greater than or equal to 3n.

**(Theta Notation – average bound):** Imagine we want to know how fast a sorting algorithm is. We can use something called Theta notation. This is like putting a speed limit on how fast an algorithm can go. It helps us understand if an algorithm is fast or slow. For example, looking at the Insertion Sort. Sometimes, it can sort things really quickly (like when things are almost in order). But when things are a mess, it can take a long time. Theta notation helps us see both sides of this. We can say: Insertion Sort is fast when things are close to being sorted (Theta(n)), but slow when they're not (Theta(n^2)).

Part 3 - Adding a Recommendation Feature.

Graph construction: To implement this feature, you are required to model the data as a graph, where vertices represent tasks and edges represent effort required to move from one task to another. Negative could indicate various meanings, such as, loss of effort or transitioning from a less urgent task to a more urgent one. Result Reading: The shortest path in the graph indicates the sequence of tasks to be completed within the minimum effort.

* 1. Model tasks as graph with the following requirements:
* Number of vertices at least 8
* Number of edges [15-30] Consider directed edges.
* Consider one single source vertex.
* Ensure a path exists from the source to each other vertex.

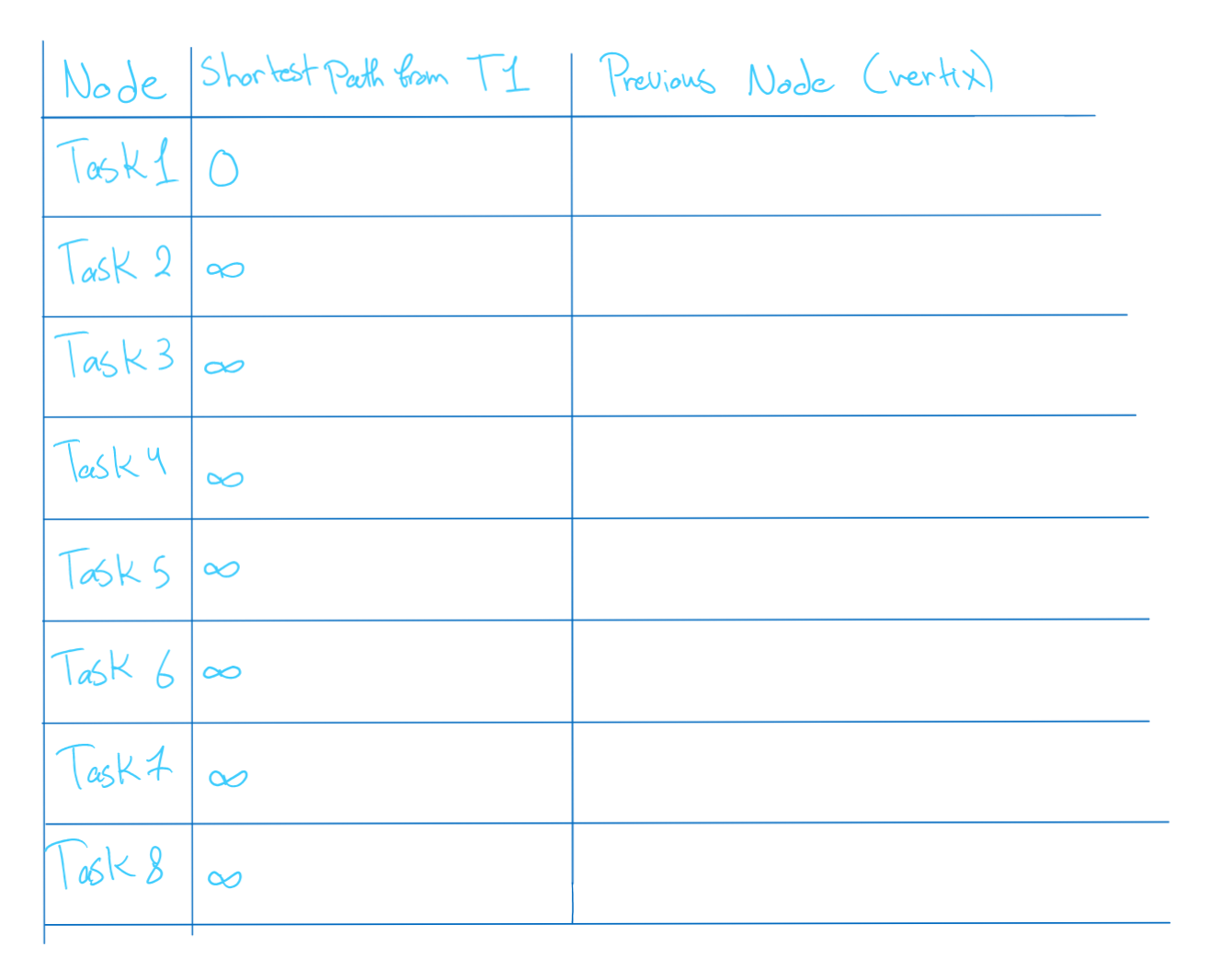


* 1. Provide a detailed step-by-step tracing of both shortest path algorithms (Dijkstra and Bellman-Ford) starting from a single source. For Bellman-Ford algorithm, update at least one edge in the graph to be negative before applying the algorithm.

A diagram of a network

Description automatically generated

**Dijkstra Tracing The Algorithm (**node to start from is Task 1**)**



*Starting with Task 1*

A diagram of a network

Description automatically generated

A white sheet of paper with blue writing

Description automatically generated

Visiting Task 4

A diagram of a diagram

Description automatically generated

A white sheet of paper with blue writing

Description automatically generated

Visiting task 2 – nothing changed for the table

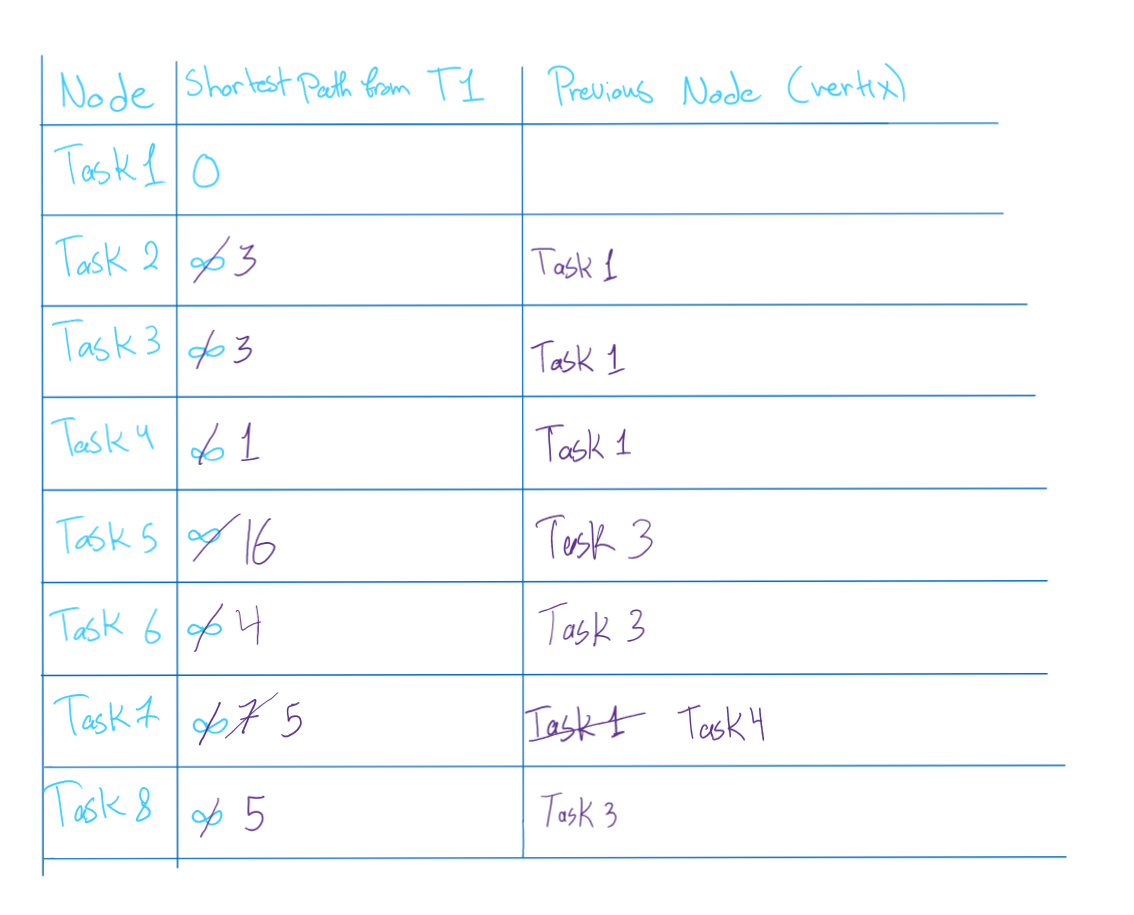
A diagram of a network

Description automatically generated

Visiting task 3

A diagram of a network

Description automatically generated



Visiting task 6

A diagram of a diagram

Description automatically generated



Visiting task 7 – nothing changed for the table

A diagram of a network

Description automatically generated

Visiting 8 - nothing changed for the

A diagram of a network

Description automatically generated

*Bellman-Ford algorithm*

A diagram of a diagram

Description automatically generated

A grid of numbers with black dots

Description automatically generated

* 1. Provide an implementation of both algorithms according to your constructed graphs and critically evaluate the complexity of each one of them (Code + Report).

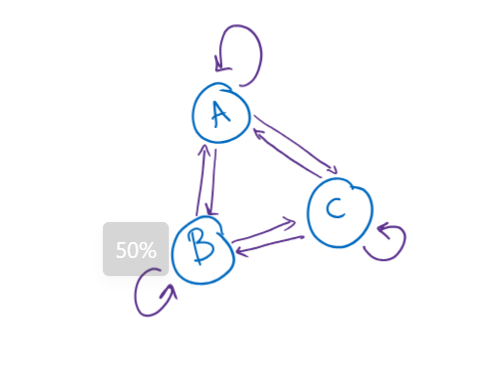
Time complexity of Dijkstra Algorithm

To get the shortest path, the algorithm passed through all the edges in the graph, so the time complexity depends on the number of edges

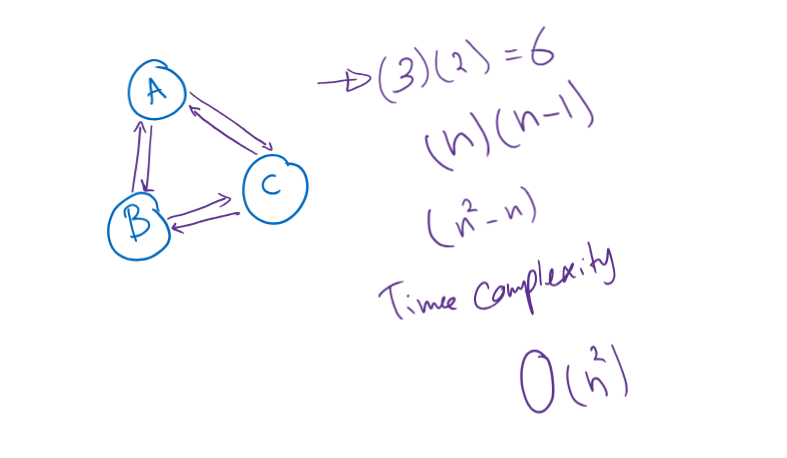
O(|Edges|)

The worst case for any Graph is that each vertex has a number of edges in which each vertex is connected directly to all other vertices. The number of edges is V\*V

So that time complexity is O(v^2) 🡪 this case including the loop to the vertex itself.



If loops are not allowed as edges, then each node is connected to the number of nodes - 1



The time complexity still remains the same O(V^2)

Time complexity of *Bellman-Ford algorithm*

The number of iterations applied are the number of (vertices -1), and it passed through all edges.

So the time complexity of it is the number of edges multiplied by the number of nodes -1

* + O(|V|\*(n-1))
  + O(|V|\*n)
* The worst case is having V^2 \*V, having the worst case V^3

The Bellman-Ford algorithm needs to look at each edge of the graph many times. Specifically, it looks at each edge once for each vertex in the graph, minus one. So, the total number of operations is (|V|\*(n-1)). meaning the time it takes to run the algorithm depends on both the number of vertices and the number of edges.

*space complexity for Both Algorithms*

Like Bellman-Ford, Dijkstra's algorithm also needs to store information about each vertex and each edge, including distances and the priority queue. Therefore, the space required is proportional to the number of vertices plus the number of edges, resulting in a space complexity of (O(V + E))

Referencesssss

Wondershare Edraw (2021). *UML Tutorial: How to Draw UML Class Diagram*. [online] YouTube. Available at:

https://www.youtube.com/watch?v=ao1ESgIy2Ws&t=196s&pp=ygULdW1sIGRpYWdyYW0%3D [Accessed 7 Jun. 2024].

GeeksforGeeks. (2024). *Stack Data Structure*. [online] Available at: https://www.geeksforgeeks.org/stack-data-structure/#:~:text=A%20Stack%20is%20a%20linear%20data%20structure%20that [Accessed 7 Jun. 2024].

‌Stack Overflow. (n.d.). *Abstraction VS Information Hiding VS Encapsulation*. [online] Available at: <https://stackoverflow.com/questions/24626/abstraction-vs-information-hiding-vs-encapsulation>.

GeeksforGeeks. (2024). *How to Implement a FIFO (First-In-First-Out) Queue in Java?* [online] Available at: https://www.geeksforgeeks.org/how-to-implement-a-fifo-first-in-first-out-queue-in-java/ [Accessed 7 Jun. 2024].

Stack Overflow. (n.d.). *FIFO based Queue implementations?* [online] Available at: https://stackoverflow.com/questions/10213707/fifo-based-queue-implementations [Accessed 7 Jun. 2024].

Datta, S. (2021). *What is Abstract Data Type? | Baeldung on Computer Science*. [online] www.baeldung.com. Available at: <https://www.baeldung.com/cs/adt>.

Alaa, A. (n.d.). *Week 7: Binary Search Trees, Sets (ADT) and Maps (ADT)*. [online] Tutorials for SBME Students. Available at: https://sbme-tutorials.github.io/2019/data-structures/notes/7\_week7.html [Accessed 7 Jun. 2024].

1. [5.Stack\_V2.pdf (htu.edu.jo)](https://elearning.htu.edu.jo/pluginfile.php/100873/mod_resource/content/1/5.Stack_V2.pdf) [↑](#footnote-ref-1)
2. [Implement Undo and Redo features of a Text Editor - GeeksforGeeks](https://www.geeksforgeeks.org/implement-undo-and-redo-features-of-a-text-editor/) [↑](#footnote-ref-2)
3. [How Browsers Use Stacks for Navigation | CodingDrills](https://www.codingdrills.com/tutorial/stack-data-structure/browsers-stacks-navigation) [↑](#footnote-ref-3)
4. [Difference Between Information Hiding and Encapsulation | Baeldung](https://www.baeldung.com/java-information-hiding-vs-encapsulation) [↑](#footnote-ref-4)
5. [6.Queue\_V2.pdf (htu.edu.jo)](https://elearning.htu.edu.jo/pluginfile.php/100874/mod_resource/content/1/6.Queue_V2.pdf) [↑](#footnote-ref-5)
6. [Queue Basics | Codewhoop](https://www.codewhoop.com/queue/basics.html) [↑](#footnote-ref-6)
7. [Time and Space Complexity Analysis of Queue operations - GeeksforGeeks](https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-queue-operations/) [↑](#footnote-ref-7)