1. **Understand Linked-list:**

Explain the different types of linked lists (Singly Linked List, Doubly Linked List)

* **Singly Linked List**:

1. Structure: Consists of nodes, each with a value and a pointer to the next node.
2. Traversal: Can only move in one direction, from the head to the end.
3. Insertion/Deletion: Easier to implement, especially at the head; harder at the end or middle.
4. Memory: Uses less memory compared to doubly linked lists since it stores only one pointer per node.
5. Use Cases: Efficient for stack implementations and simple list operations.

* **Doubly Linked List:**

1. Structure: Nodes have two pointers, one to the next node and one to the previous node.
2. Traversal: Allows movement in both directions (forward and backward).
3. Insertion/Deletion: Easier to insert or delete nodes from both ends or middle.
4. Memory: Uses more memory because of the extra pointer for each node.
5. Use Cases: Ideal for complex list operations where bidirectional traversal is needed.
6. **Setup:**
7. *Created a Task class to represent individual tasks:*

Attributes include taskId (int), taskName (String), and status (String). Added a next attribute of type Task for linking to the next task in the list. Implemented a constructor to initialize these attributes.

1. *Developed a TaskList class to manage the linked list of tasks:*

Includes a private head attribute of type Task to mark the start of the list.

Contains a Scanner object for user input.

1. *Implemented core methods in Task List:*

**isTaskIdExists():** Checks for duplicate task IDs.

**addTask():** Adds a new task to the list, ensuring unique task IDs.

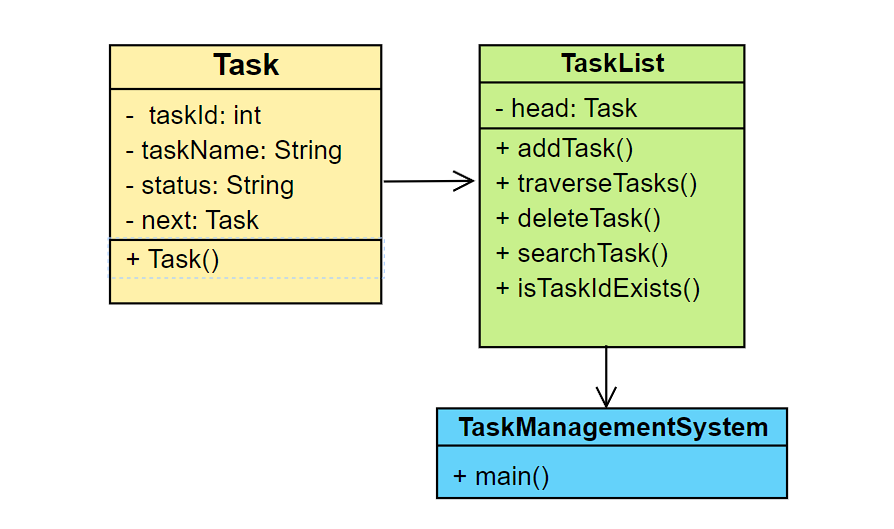
**traverseTasks():** Displays all tasks in the list.

**deleteTask():** Removes a task based on its ID.

1. *Created a TaskManagementSystem class with the main method:*

* Initializes a TaskList object.
* Implements a menu-driven interface using a do-while loop and switch-case structure.
* Handles user input for various operations on the task list.

This setup provides a foundation for a dynamic, user-interactive task management system using a singly linked list data structure. CopyRetryClaude does not have the ability to run the code it generates yet. Claude can make mistakes. Please double-check responses.

1. **Implementation:**

Implement a singly linked list to manage tasks:

* The singly linked list is implemented using the Task class as nodes and the TaskList class to manage the list.
* Each Task object contains a reference (*next*) to the next task in the list, forming a chain of tasks.
* The TaskList class maintains a head reference to the first task in the list.

Implement methods to add, search, traverse, and delete tasks in the linked list:

* Add method (*addTask()*):
* Creates a new *Task* object with user-provided details.
* If the list is empty, sets the new task as the head.
* Otherwise, traverses to the end of the list and adds the new task there.
* Ensures unique task IDs by checking for duplicates before adding.
* Search method (implicit in other methods):
* Traverses the list from the head, comparing task IDs.
* Used in *isTaskIdExists()* to check for duplicate IDs.
* Can be easily extended to a standalone search method if needed.

3. Traverse method (*traverseTasks()*):

* Starts from the head of the list.
* Iterates through each task, printing its details.
* Continues until it reaches the end of the list (null reference).

4. Delete method (*deleteTask()*):

* Asks the user for the task ID to delete.
* Handles special case if the task to delete is the head.
* Otherwise, traverses the list to find the task.
* Updates the *next* reference of the previous task to skip the deleted task.
* Provides appropriate feedback on successful deletion or if the task is not found.

Here is the github repo of the code -

1. **Analysis:**

* Time Complexity of Recursive Algorithms:

If we were to implement the operations recursively, here's how the time complexities would look:

* *Adding a task (recursively):* O(n), where n is the number of tasks in the list.
* *Searching for a task (recursively):* O(n) in the worst case.
* *Traversing the list (recursively):* O(n).
* *Deleting a task (recursively):* O(n) in the worst case.

In all these cases, the recursive function would call itself n times in the worst case, where n is the number of elements in the list.

* Optimization of Recursive Solutions:
* *Tail Recursion:* Convert the recursive functions to tail-recursive form. This allows some compilers to optimize the recursion into iteration, reducing stack usage.
* *Memoization*: For operations that might repeat calculations (not typically in simple linked list operations, but useful in more complex recursive algorithms), store results of expensive function calls and return the cached result when the same inputs occur again.
* *Limit Recursion Depth:* Implement a hybrid approach where recursion is used up to a certain depth, then switches to iteration for the rest of the operation.

By applying these optimizations, we can mitigate the risks of stack overflow for large lists and improve the overall efficiency of recursive implementations in a task management system. However, for simple linked list operations, iterative approaches (as used in our implementation) are often more efficient and easier to reason about than recursive ones.