**Exercise 5: Task Management System**

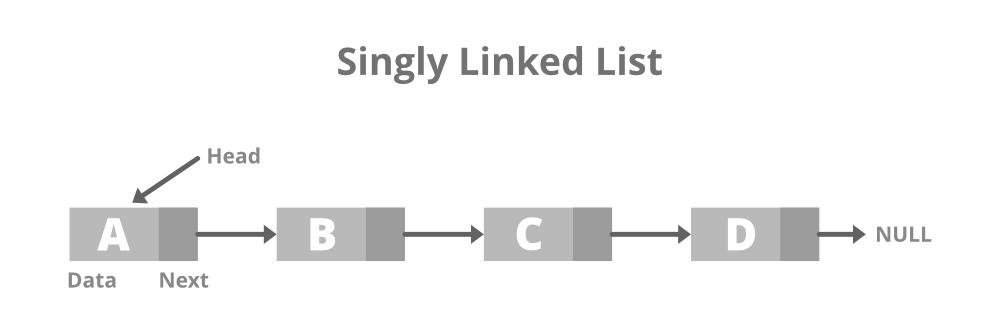
1. **Understand Linked Lists:**

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

Ans. **Singly Linked List**

It is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type.

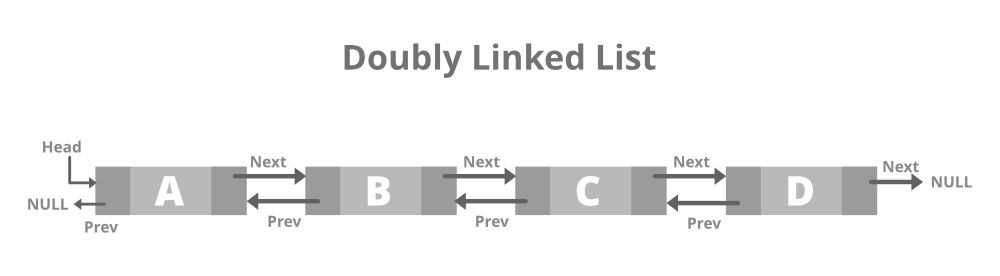
The node contains a pointer to the next node means that the node stores the address of the next node in the sequence. A single linked list allows the traversal of data only in one way.



**Doubly Linked List**

A doubly linked list or a two-way linked list is a more complex type of linked list that contains a pointer to the next as well as the previous node in sequence.

Therefore, it contains three parts of data, a pointer to the next node, and a pointer to the previous node. This would enable us to traverse the list in the backward direction as well.



1. **Setup:**
   * Create a class **Task** with attributes like **taskId**, **taskName**, and **status**.
2. **Implementation:**
   * Implement a singly linked list to manage tasks.
   * Implement methods to **add**, **search**, **traverse**, and **delete** tasks in the linked list.

Code:

package Algorithm\_DataStructures.Exercise5.Task\_Management\_System;

class Task {

    int taskId;

    String taskName;

    String status;

    Task next;

    public Task(int taskId, String taskName, String status) {

        this.taskId = taskId;

        this.taskName = taskName;

        this.status = status;

        this.next = null;

    }

}

class TaskManager {

    Task head;

    public void addTask(int taskId, String taskName, String status) {

        Task newTask = new Task(taskId, taskName, status);

        if (head == null) {

            head = newTask;

        } else {

            Task current = head;

            while (current.next != null) {

                current = current.next;

            }

            current.next = newTask;

        }

        System.out.println("Task added successfully.");

    }

    public Task searchTask(int taskId) {

        Task current = head;

        while (current != null) {

            if (current.taskId == taskId) {

                return current;

            }

            current = current.next;

        }

        return null;

    }

    public void traverseTasks() {

        Task current = head;

        while (current != null) {

            System.out.println("Task ID: " + current.taskId + ", Task Name: " + current.taskName + ", Status: " + current.status);

            current = current.next;

        }

    }

    public void deleteTask(int taskId) {

        if (head == null) {

            System.out.println("Task list is empty.");

            return;

        }

        if (head.taskId == taskId) {

            head = head.next;

            System.out.println("Task deleted successfully.");

            return;

        }

        Task current = head;

        Task prev = null;

        while (current != null && current.taskId != taskId) {

            prev = current;

            current = current.next;

        }

        if (current == null) {

            System.out.println("Task with ID " + taskId + " not found.");

        } else {

            prev.next = current.next;

            System.out.println("Task deleted successfully.");

        }

    }

}

public class TMS {

    public static void main(String[] args) {

        TaskManager taskManager = new TaskManager();

        taskManager.addTask(1, "Task 1", "Pending");

        taskManager.addTask(2, "Task 2", "Completed");

        taskManager.addTask(3, "Task 3", "In Progress");

        System.out.println("\nTraversing all tasks:");

        taskManager.traverseTasks();

        System.out.println("\nSearch for task with ID 2:");

        Task foundTask = taskManager.searchTask(2);

        if (foundTask != null) {

            System.out.println("Task found: Task ID - " + foundTask.taskId + ", Task Name - " + foundTask.taskName + ", Status - " + foundTask.status);

        } else {

            System.out.println("Task not found.");

        }

        System.out.println("\nDeleting task with ID 2:");

        taskManager.deleteTask(2);

        System.out.println("\nTraversing all tasks after deletion:");

        taskManager.traverseTasks();

    }

}

4. **Analysis:**

* Analyze the time complexity of each operation.

Ans. **Time Complexity Analysis:**

1. **Add Task Operation:**
   * Time Complexity: O(n)
   * Adding a task at the end of the linked list requires traversing the list to find the last node, resulting in a time complexity of O(n) where n is the number of tasks in the list.
2. **Search Task Operation:**
   * Time Complexity: O(n)
   * Searching for a task in the linked list requires traversing the list from the head to the target task, resulting in a time complexity of O(n) where n is the number of tasks in the list.
3. **Traverse Tasks Operation:**
   * Time Complexity: O(n)
   * Traversing all tasks in the linked list involves visiting each node once, resulting in a time complexity of O(n) where n is the number of tasks in the list.
4. **Delete Task Operation:**
   * Time Complexity: O(n)
   * Deleting a task involves searching for the task in the list (O(n)) and potentially updating the references of neighbouring nodes, resulting in a time complexity of O(n) where n is the number of tasks in the list.

* Discuss the advantages of linked lists over arrays for dynamic data.

Ans. **Advantages of Linked Lists Over Arrays for Dynamic Data:**

1. **Dynamic Size:**
   * Linked lists can dynamically grow and shrink in size without the need for resizing or copying elements, unlike arrays which have a fixed size once created.
2. **Insertions and Deletions:**
   * Insertions and deletions in a linked list have a time complexity of O(1) if performed at the beginning or end of the list, and O(n) in the worst case for insertions/deletions at arbitrary positions. This contrasts with arrays where insertions and deletions can be O(n) due to shifting elements.
3. **Memory Management:**
   * Linked lists do not require contiguous memory allocation, allowing for efficient memory usage and avoiding memory fragmentation issues that can occur with arrays.
4. **Ease of Manipulation:**
   * Linked lists make it easier to insert or delete elements in the middle of the list without needing to shift subsequent elements, making them more suitable for scenarios where frequent data manipulation is required.
5. **Efficient for Large Datasets:**
   * For large datasets where frequent insertions and deletions are common, linked lists can provide better performance compared to arrays due to their dynamic nature and efficient insertion/deletion operations.