**Exercise 5: Task Management System**

**Scenario:**

You are developing a task management system where tasks need to be added, deleted, and traversed efficiently.

**Understanding Linked Lists**

**➤ Singly Linked List**

* Each node contains **data** and a **pointer to the next node**.
* Traversal is **one-way**, from the head node to the end (null).
* Simple to implement, and memory-efficient.

**➤ Doubly Linked List**

* Each node contains **data**, a pointer to the **next node**, and a pointer to the **previous node**.
* Enables **bi-directional traversal**.
* Slightly more memory and logic overhead, but useful for complex tasks like reverse traversals.

**Setup:-**

**Task.java :-**

public class Task {

int taskId;

String taskName;

String status;

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

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

}

public void display() {

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

}

}

**Implementation :-**

class Node {

Task task;

Node next;

public Node(Task task) {

this.task = task;

this.next = null;

}

}

**TaskManager.java :-**

public class TaskManager {

private Node head;

// Add a task to the end

public void addTask(Task task) {

Node newNode = new Node(task);

if (head == null) {

head = newNode;

} else {

Node current = head;

while (current.next != null) {

current = current.next;

}

current.next = newNode;

}

}

// Search task by ID

public Task searchTask(int taskId) {

Node current = head;

while (current != null) {

if (current.task.taskId == taskId) {

return current.task;

}

current = current.next;

}

return null;

}

// Delete task by ID

public void deleteTask(int taskId) {

if (head == null) return;

if (head.task.taskId == taskId) {

head = head.next;

return;

}

Node current = head;

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

current = current.next;

}

if (current.next != null) {

current.next = current.next.next;

}

}

// Traverse and display all tasks

public void displayTasks() {

Node current = head;

while (current != null) {

current.task.display();

current = current.next;

}

}

}

**Main.java :-**

public class Main {

public static void main(String[] args) {

TaskManager manager = new TaskManager();

manager.addTask(new Task(1, "Design UI", "Pending"));

manager.addTask(new Task(2, "Write Backend", "In Progress"));

manager.addTask(new Task(3, "Test Module", "Pending"));

System.out.println("All Tasks:");

manager.displayTasks();

System.out.println("\nSearching for Task ID 2:");

Task task = manager.searchTask(2);

if (task != null) task.display();

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

System.out.println("\nDeleting Task ID 2...");

manager.deleteTask(2);

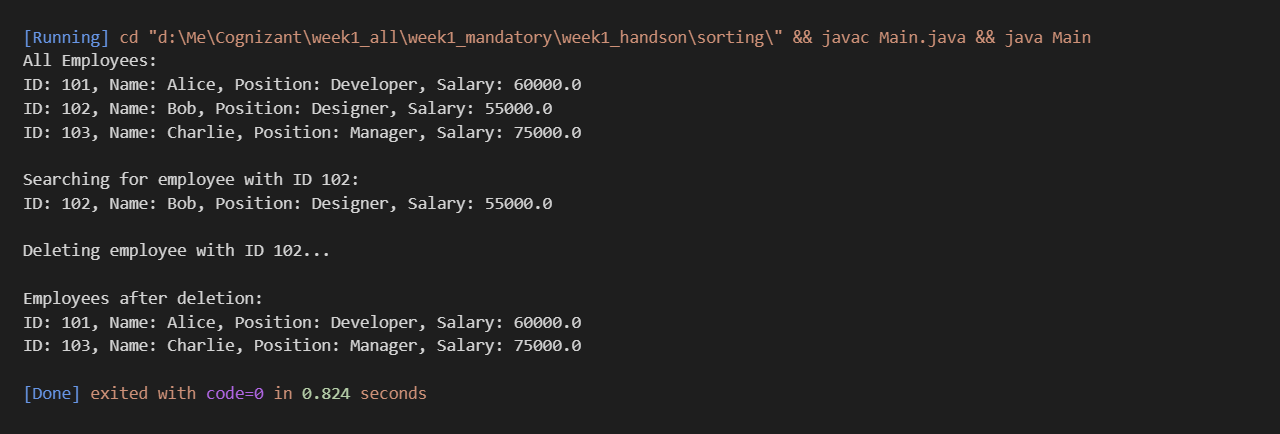
System.out.println("\nTasks after deletion:");

manager.displayTasks();

}

}

**Output :-**

****

**Time Complexity Analysis of Linked List Operations**

Here’s a breakdown of common operations in a **singly linked list**, particularly in the context of a task management system:

| **Operation** | **Time Complexity** | **Explanation** |
| --- | --- | --- |
| **Add Task** | O(n) | We traverse to the end of the list to add the task, which takes linear time. |
| **Search Task** | O(n) | Searching requires traversing from the head node until the match is found. |
| **Delete Task** | O(n) | We must find the node and its predecessor before deletion. |
| **Traverse Tasks** | O(n) | Each node is visited exactly once to display or process it. |

**Advantages of Linked Lists Over Arrays for Dynamic Data**

**1. Dynamic Size Allocation**

* **Linked Lists:** Size is dynamic; memory is allocated as new elements are added.
* **Arrays:** Fixed size once declared; resizing requires creating a new array and copying elements.

**Advantage:** In scenarios where the number of elements changes frequently (like adding/removing tasks), linked lists are more flexible.

**2. Efficient Insertion and Deletion**

* **Linked Lists:** Insertion and deletion (especially at the beginning) are **O(1)** operations if node references are known.
* **Arrays:** Require shifting elements for insertion or deletion, making them **O(n)** operations.

**Advantage:** Linked lists avoid overhead of shifting elements, leading to better performance in frequent updates.

**3. Memory Usage Efficiency**

* **Linked Lists:** No need to reserve unused memory ahead of time.
* **Arrays:** Often over-provisioned to accommodate future growth, leading to wasted memory.

**Advantage:** Linked lists are memory-efficient in cases of unpredictable or sparse data.

**Limitations of Linked Lists Compared to Arrays**

| **Criteria** | **Linked List** | **Array** |
| --- | --- | --- |
| **Random Access** | Not supported (O(n)) | Supported (O(1) via index) |
| **Cache Locality** | Poor (nodes may be scattered in memory) | Excellent (contiguous memory blocks) |
| **Memory Overhead** | Extra space for next pointer | No pointer overhead |