Task Management System Using Linked Lists

public class TaskManagementSystem {

private static class TaskNode {

Task task;

TaskNode next;

public TaskNode(Task task) {

this.task = task;

this.next = null;

}

}

public static class Task {

private final int taskId;

private final String taskName;

private String status; // Can be "Pending", "In Progress", or "Completed"

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

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

}

// Getters

public int getTaskId() { return taskId; }

public String getTaskName() { return taskName; }

public String getStatus() { return status; }

// Setter for status

public void setStatus(String status) {

this.status = status;

}

@Override

public String toString() {

return String.format("Task ID: %d | Name: %-20s | Status: %s",

taskId, taskName, status);

}

}

private TaskNode head;

private int size;

public TaskManagementSystem() {

head = null;

size = 0;

}

public void addTask(Task task) {

TaskNode newNode = new TaskNode(task);

if (head == null) {

head = newNode;

} else {

TaskNode current = head;

while (current.next != null) {

current = current.next;

}

current.next = newNode;

}

size++;

System.out.println("Added: " + task);

}

public void addTaskAtPosition(Task task, int position) {

if (position < 0 || position > size) {

throw new IndexOutOfBoundsException("Invalid position");

}

TaskNode newNode = new TaskNode(task);

if (position == 0) {

newNode.next = head;

head = newNode;

} else {

TaskNode current = head;

for (int i = 0; i < position - 1; i++) {

current = current.next;

}

newNode.next = current.next;

current.next = newNode;

}

size++;

System.out.println("Added at position " + position + ": " + task);

}

public Task searchTask(int taskId) {

TaskNode current = head;

while (current != null) {

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

return current.task;

}

current = current.next;

}

return null;

}

public boolean updateTaskStatus(int taskId, String newStatus) {

Task task = searchTask(taskId);

if (task != null) {

task.setStatus(newStatus);

return true;

}

return false;

}

public boolean deleteTask(int taskId) {

if (head == null) return false;

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

head = head.next;

size--;

System.out.println("Deleted task ID: " + taskId);

return true;

}

TaskNode current = head;

while (current.next != null) {

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

current.next = current.next.next;

size--;

System.out.println("Deleted task ID: " + taskId);

return true;

}

current = current.next;

}

return false;

}

public void displayAllTasks() {

System.out.println("\n=== Task List (" + size + " tasks) ===");

TaskNode current = head;

while (current != null) {

System.out.println(current.task);

current = current.next;

}

System.out.println("=== End of List ===");

}

public int getTaskCount() {

return size;

}

public static void main(String[] args) {

TaskManagementSystem tms = new TaskManagementSystem();

// Add sample tasks

tms.addTask(new Task(101, "Implement login", "Pending"));

tms.addTask(new Task(102, "Design database", "In Progress"));

tms.addTask(new Task(103, "Write documentation", "Pending"));

// Add task at specific position

tms.addTaskAtPosition(new Task(104, "Fix critical bug", "Pending"), 1);

// Display all tasks

tms.displayAllTasks();

// Search for a task

Task found = tms.searchTask(102);

System.out.println("\nFound task: " + found);

// Update task status

tms.updateTaskStatus(101, "Completed");

System.out.println("\nUpdated task 101 to Completed");

// Delete a task

tms.deleteTask(103);

// Display final list

tms.displayAllTasks();

}

}

1. Linked List Types

**Singly Linked List** (used in this implementation):

* Each node points only to the next node
* Memory efficient (only one pointer per node)
* Simpler implementation than doubly linked lists

**Doubly Linked List**:

* Each node has pointers to both next and previous nodes
* Allows traversal in both directions
* More flexible but uses more memory

2. Task Class

* Contains taskId, taskName, and status fields
* Includes getters and a setter for status updates
* Clean toString() method for display purposes

3. Core Operations

**Adding Tasks**:

* addTask(): Appends to end of list (O(n))
* addTaskAtPosition(): Inserts at specific position (O(n))

**Searching**:

* Linear search by task ID (O(n))

**Updating**:

* Status updates after finding the task (O(n))

**Deleting**:

* Finds and removes task by ID (O(n))

**Displaying**:

* Traverses entire list to print all tasks (O(n))

Time Complexity Analysis:

| **Operation** | **Time Complexity** | **Notes** |
| --- | --- | --- |
| Add at end | O(n) | Must traverse to list end |
| Add at position | O(n) | Worst case when adding near end |
| Search by ID | O(n) | Linear search required |
| Update status | O(n) | Requires finding task first |
| Delete by ID | O(n) | Search + pointer adjustment |
| Display all tasks | O(n) | Must visit each node |
| Get count | O(1) | Maintained with size variable |

Advantages of Linked Lists for Task Management

1. **Dynamic Size**: Grows and shrinks as tasks are added/removed
2. **Efficient Insertions**: No need to shift elements like in arrays
3. **Memory Efficiency**: Allocates memory only when needed
4. **No Pre-allocation**: Don't need to predict maximum size
5. **Implementation Simplicity**: Easy to model task sequences

When to Choose Linked Lists Over Arrays

* When tasks are frequently added/removed from middle of list
* When the number of tasks is unpredictable
* When memory usage needs to be optimized
* When you don't need random access by index

Output:

