CodInClub

Powered By BridgeLabz

Best Practices for Data Structure - LinkedList

1. **Head & Tail Management**: Always maintain the `head` (and `tail` in doubly and circular lists) to avoid traversing the entire list when accessing the first or last elements.
2. **Null Checks**: Before performing operations like deletion or traversal, check if the list is empty to prevent errors.
3. **Efficient Insertion/Deletion**: Insert at the beginning or end for O(1) time complexity. For operations in the middle, ensure proper pointer updates to maintain list integrity.
4. **Memory Management**: Properly nullify pointers (`next`, `prev`) when deleting nodes to prevent memory leaks, especially in languages without garbage collection.
5. **Boundary Handling**: Carefully handle edge cases like inserting/deleting at the head, tail, or middle of the list, ensuring correct pointer updates.
6. **Avoid Infinite Loops** (Circular Lists): Implement conditions to stop traversal after one complete cycle to avoid infinite loops.
7. **Modular Code**: Break operations into small, reusable functions for better readability and maintainability.
8. **Keep Code Simple**: Focus on clarity over complexity. Avoid unnecessary traversals and complex logic unless required for your use case.

1. Singly Linked List: Student Record Management

Problem Statement: Create a program to manage student records using a singly linked list. Each node will store information about a student, including their `Roll Number`, `Name`, `Age`, and `Grade`. Implement the following operations:

1. Add a new student record at the beginning, end, or at a specific position.
2. Delete a student record by `Roll Number`.
3. Search for a student record by `Roll Number`.
4. Display all student records.
5. Update a student's grade based on their `Roll Number`.

Hint:

- Use a singly linked list where each node contains student information and a pointer to the next node.
- The head of the list will represent the first student, and the last node’s next pointer will be `null`.
- Update the `next` pointers when inserting or deleting nodes.

---

2. Doubly Linked List: Movie Management System

Problem Statement: Implement a movie management system using a doubly linked list. Each node will represent a movie and contain `Movie Title`, `Director`, `Year of Release`, and `Rating`. Implement the following functionalities:

1. Add a movie record at the beginning, end, or at a specific position.
2. Remove a movie record by `Movie Title`.
3. Search for a movie record by `Director` or `Rating`.
4. Display all movie records in both forward and reverse order.
5. Update a movie's `Rating` based on the `Movie Title`.

Hint:

- Use a doubly linked list where each node has two pointers: one pointing to the next node and the other to the previous node.
- Maintain pointers to both the head and tail for easier insertion and deletion at both ends.
- For reverse display, start from the tail and traverse backward using the `prev` pointers.

---

3. Circular Linked List: Task Scheduler

Problem Statement: Create a task scheduler using a circular linked list. Each node in the list represents a task with `Task ID`, `Task Name`, `Priority`, and `Due Date`. Implement the following functionalities:

1. Add a task at the beginning, end, or at a specific position in the circular list.
2. Remove a task by `Task ID`.
3. View the current task and move to the next task in the circular list.
4. Display all tasks in the list starting from the head node.
5. Search for a task by `Priority`.

Hint:

- Use a circular linked list where the last node's `next` pointer points back to the first node, creating a circular structure.
- Ensure that the list loops when traversed from the head node, so tasks can be revisited in a circular manner.

- When deleting or adding tasks, maintain the circular nature by updating the appropriate `next` pointers.

4. Singly Linked List: Inventory Management System

Problem Statement: Design an inventory management system using a singly linked list where each node stores information about an item such as `Item Name`, `Item ID`, `Quantity`, and `Price`. Implement the following functionalities:

1. Add an item at the beginning, end, or at a specific position.
2. Remove an item based on `Item ID`.
3. Update the quantity of an item by `Item ID`.
4. Search for an item based on `Item ID` or `Item Name`.
5. Calculate and display the total value of inventory (Sum of `Price * Quantity` for each item).
6. Sort the inventory based on `Item Name` or `Price` in ascending or descending order.

Hint:

- Use a singly linked list where each node represents an item in the inventory.
- Implement sorting using an appropriate algorithm (e.g., merge sort) on the linked list.
- For total value calculation, traverse through the list and sum up `Quantity * Price` for each item.

---

5. Doubly Linked List: Library Management System

Problem Statement: Design a library management system using a doubly linked list. Each node represents a book and contains the following attributes: `Book Title`, `Author`, `Genre`, `Book ID`, and `Availability Status`. Implement the following functionalities:

1. Add a new book at the beginning, end, or at a specific position.
2. Remove a book by `Book ID`.
3. Search for a book by `Book Title` or `Author`.
4. Update a book's `Availability Status`.
5. Display all books in forward and reverse order.
6. Count the total number of books in the library.

Hint:

- Use a doubly linked list with two pointers (`next` and `prev`) in each node to facilitate traversal in both directions.
- Ensure that when removing a book, both the `next` and `prev` pointers are correctly updated.
- Displaying in reverse order will require traversal from the last node using `prev` pointers.

---

6. Circular Linked List: Round Robin Scheduling Algorithm

Problem Statement: Implement a round-robin CPU scheduling algorithm using a circular linked list. Each node will represent a process and contain `Process ID`, `Burst Time`, and `Priority`. Implement the following functionalities:

1. Add a new process at the end of the circular list.
2. Remove a process by `Process ID` after its execution.
3. Simulate the scheduling of processes in a round-robin manner with a fixed time quantum.
4. Display the list of processes in the circular queue after each round.
5. Calculate and display the average waiting time and turn-around time for all processes.

Hint:

- Use a circular linked list to represent a queue of processes.
- Each process executes for a fixed time quantum, and then control moves to the next process in the circular list.
- Maintain the current node as the process being executed, and after each round, update the list to simulate execution.

---

7. Singly Linked List: Social Media Friend Connections

Problem Statement: Create a system to manage social media friend connections using a singly linked list. Each node represents a user with `User ID`, `Name`, `Age`, and `List of Friend IDs`. Implement the following operations:

1. Add a friend connection between two users.
2. Remove a friend connection.
3. Find mutual friends between two users.

4. Display all friends of a specific user.
5. Search for a user by `Name` or `User ID`.
6. Count the number of friends for each user.

Hint:

- Use a singly linked list where each node contains a list of friends (which can be another linked list or array of `Friend IDs`).
- For mutual friends, traverse both lists and compare the `Friend IDs`.
- The `List of Friend IDs` for each user can be implemented as a nested linked list or array.

---

8. Doubly Linked List: Undo/Redo Functionality for Text Editor

Problem Statement: Design an undo/redo functionality for a text editor using a doubly linked list. Each node represents a state of the text content (e.g., after typing a word or performing a command). Implement the following:

1. Add a new text state at the end of the list every time the user types or performs an action.
2. Implement the undo functionality (revert to the previous state).
3. Implement the redo functionality (revert back to the next state after undo).
4. Display the current state of the text.
5. Limit the undo/redo history to a fixed size (e.g., last 10 states).

Hint:

- Use a doubly linked list where each node represents a state of the text.
- The `next` pointer will represent the forward history (redo), and the `prev` pointer will represent the backward history (undo).
- Keep track of the current state and adjust the `next` and `prev` pointers for undo/redo operations.

---

9. Circular Linked List: Online Ticket Reservation System

Problem Statement: Design an online ticket reservation system using a circular linked list, where each node represents a booked ticket. Each node will store the following information:

`Ticket ID`, `Customer Name`, `Movie Name`, `Seat Number`, and `Booking Time`. Implement the following functionalities:

1. Add a new ticket reservation at the end of the circular list.
2. Remove a ticket by `Ticket ID`.
3. Display the current tickets in the list.
4. Search for a ticket by `Customer Name` or `Movie Name`.
5. Calculate the total number of booked tickets.

Hint:

- Use a circular linked list to represent the ticket reservations, with the last node's `next` pointer pointing to the first node.
- When removing a ticket, update the circular pointers accordingly.
- For displaying all tickets, traverse the list starting from the first node, looping back after reaching the last node.