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.

```java
public class StudentRecord {

    class Node {
        int rollNumber;
        String name;
        int age;
        double grade;

        Node next;

        Node(int rollNumber, String name, int age, double grade) {
            this.rollNumber = rollNumber;
            this.name = name;
            this.age = age;
            this.grade = grade;
            this.next = null;
        }
    }

    static Node head = null;
    static Node tail = null;

    int size = 0;

    public void addFirst(int rollNumber, String name, int age, double grade) {
        if (head == null) {
            head = new Node(rollNumber, name, age, grade);
            tail = head;
        } else {
            Node newNode = new Node(rollNumber, name, age, grade);
            newNode.next = head;
            head = newNode;
        }

        size++;
    }

    public void addLast(int rollNumber, String name, int age, double grade) {
        if (tail == null) {
            head = new Node(rollNumber, name, age, grade);
            tail = head;
        } else {
            tail.next = new Node(rollNumber, name, age, grade);
```

```
            tail = tail.next;
        }

        size++;
    }

    //Add node at ith position
    public void addNode(int i, int rollNumber, String name, int age, double
grade) {
        if (i > size + 1 || i < 1) {
            System.out.println("Provide correct position from 1 to " + (size +
1));
            return;
        }
        if (i == 1) {
            Node newNode = new Node(rollNumber, name, age, grade);
            newNode.next = head;
            head = newNode;
            if (tail == null) {
                tail = head;
            }
        }else if (i == size + 1) {
            Node newNode = new Node(rollNumber, name, age, grade);
            tail.next = newNode;
            tail = newNode;
        } else {
            int j = 1;
            Node temp = head;
            while (i-1 != j) {
                temp = temp.next;
                j++;
            }
            Node temp2 = temp.next;
            temp.next = new Node(rollNumber, name, age, grade);
            temp.next.next = temp2;
        }
        System.out.println("New node added successfully!");
        size++;
        return;
    }
```

```
    public void deleteRecord(int rollNumber) {
        Node temp = head;
        Node prev = null;
        while (temp != null) {
            if (temp.rollNumber == rollNumber) {
                if (prev == null) {
                    head = head.next;
                    if (head == null) {
                        tail = null;
                    }
                } else {
                    if (temp == tail) {
                        prev.next = null;
                        tail = prev;
                    }
                    prev.next = prev.next.next;
                }
                size--;
                System.out.println("Record Deleted Successfully!");
                return;
            }

            prev = temp;
            temp = temp.next;

        }
        System.out.println("Record not found!");
    }

    public Node searchRecord(int rollNumber) {
        Node temp = head;
        while (temp != null) {
            if (temp.rollNumber == rollNumber) {
                System.out.println("Record Found!\nHere are the details:");
                System.out.println("RollNumber: " + temp.rollNumber);
                System.out.println("Name: " + temp.name);
                System.out.println("Age: " + temp.age);
                System.out.println("Grade: " + temp.grade);
                return temp;
            }
            temp = temp.next;
        }
```

```
        System.out.println("Record not found!");
        return null;
    }

    public void displayAllRecords() {
        if (head == null) {
            System.out.println("No record found!");
            return;
        }
        Node temp = head;
        int i = 0;
        while (temp != null) {
            System.out.println("Record-" + i++);
            System.out.println("RollNumber: " + temp.rollNumber);
            System.out.println("Name: " + temp.name);
            System.out.println("Age: " + temp.age);
            System.out.println("Grade: " + temp.grade);
            temp = temp.next;
        }
    }

    public void updateGrade(int rollNumber, double grade) {
        Node temp = head;
        while (temp != null) {
            if (temp.rollNumber == rollNumber) {
                temp.grade = grade;
                System.out.println("Grade updated successfully!");
                return;
            }
            temp = temp.next;
        }
        System.out.println("Record not found!");
    }

}
```

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.

```
public class MovieManagementSystem {

    class Node {

        String title;
        String director;
        int year;
        double rating;

        Node next;
        Node prev;

        Node(String title, String director, int year, double rating) {
            this.title = title;
            this.director = director;
            this.year = year;
            this.rating = rating;
            this.next = null;
            this.prev = null;
        }
    }
```

```
    Node head = null;
    Node tail = null;

    int size = 0;

    public void addFirst(String title, String director, int year, double
rating) {
        if (head == null) {
            head = new Node(title, director, year, rating);
            tail = head;
        } else {
            Node newNode = new Node(title, director, year, rating);
            newNode.next = head;
            head.prev = newNode;
            head = newNode;
        }

        size++;
    }

    public void addLast(String title, String director, int year, double rating)
{
        if (tail == null) {
            head = new Node(title, director, year, rating);
            tail = head;
        } else {
            tail.next = new Node(title, director, year, rating);
            tail.next.prev = tail;
            tail = tail.next;
        }

        size++;
    }

    // Add node at ith position
    public void addNode(int i, String title, String director, int year, double
rating) {
        if (i > size + 1 || i < 1) {
            System.out.println("Provide correct position from 1 to " + (size +
1));
            return;
        }
        if (i == 1) {
```

```
            Node newNode = new Node(title, director, year, rating);
            newNode.next = head;
            head.prev = newNode;
            head = newNode;
            if (tail == null) {
                tail = head;
            }
        } else if (i == size + 1) {
            Node newNode = new Node(title, director, year, rating);
            tail.next = newNode;
            newNode.prev = tail;
            tail = newNode;
        } else {
            int j = 1;
            Node temp = head;
            while (i - 1 > j) {
                temp = temp.next;
                j++;
            }
            Node temp2 = temp.next;
            temp.next = new Node(title, director, year, rating);
            temp.next.prev = temp;
            temp.next.next = temp2;
            temp2.prev = temp.next;
        }
        System.out.println("New record added successfully!");
        size++;
        return;
    }

    public void deleteRecord(String title) {
        Node temp = head;
        Node prev = null;
        while (temp != null) {
            if (temp.title.equals(title)) {
                if (temp == head) {
                    head = head.next;
                    if (head == null) {
                        tail = null;
                    } else {
                        head.prev = null;
                    }
                } else if (temp == tail) {
```

```
                    prev.next = null;
                    tail = prev;
                } else {
                    prev.next.next.prev = prev;
                    prev.next = prev.next.next;
                }
                size--;
                System.out.println("Record Deleted Successfully!");
                return;
            }

            prev = temp;
            temp = temp.next;
        }


        System.out.println("Record not found!");
    }

    public Node searchRecord(String director , double rating) {
        Node temp = head;
        while (temp != null) {
            if (temp.director == director || temp.rating == rating) {
                System.out.println("A record found");
                return temp;
            }
            temp = temp.next;
        }
        System.out.println("Record not found!");
        return null;
    }

    public void displayAllRecords() {
        if (head == null) {
            System.out.println("No record found!");
            return;
        }

        System.out.println("Forward Order:\n");
        Node temp = head;
        int i = 1;
        while (temp != null) {
            System.out.println("Record-" + i++);
```

```
            System.out.println("Movie Title: " + temp.title);
            System.out.println("Director: " + temp.director);
            System.out.println("Year of release: " + temp.year);
            System.out.println("Rating: " + temp.rating);
            System.out.println();
            temp = temp.next;
        }
        System.out.println("\n\nReverse Order:\n");
        temp = tail;
        i = size;
        while (temp != null) {
            System.out.println("Record-" + i--);
            System.out.println("Movie Title: " + temp.title);
            System.out.println("Director: " + temp.director);
            System.out.println("Year of release: " + temp.year);
            System.out.println("Rating: " + temp.rating);
            System.out.println();
            temp = temp.prev;
        }
    }

    public void updateGrade(String title, double rating) {
        Node temp = head;
        while (temp != null) {
            if (temp.title.equals(title)) {
                temp.rating = rating;
                System.out.println("Rating updated successfully!");
                return;
            }
            temp = temp.next;
        }
        System.out.println("Record not found!");
    }

}
```

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.

```
public class TaskScheduler {

    class Node {
        int taskID;
        String taskName;
        String priority;
        String dueDate;

        Node next;

        Node(int taskID, String taskName, String priority, String dueDate) {
            this.taskID = taskID;
            this.taskName = taskName;
            this.priority = priority;
            this.dueDate = dueDate;
            this.next = this;
        }
    }
```

```
    static Node head = null;
    static Node tail = null;

    int size = 0;

    public void addFirst(int taskID, String taskName, String priority, String
dueDate) {
        if (head == null) {
            head = new Node(taskID, taskName, priority, dueDate);
            tail = head;
        } else {
            Node newNode = new Node(taskID, taskName, priority, dueDate);
            newNode.next = head;
            head = newNode;
            tail.next = head;
        }

        size++;
    }

    public void addLast(int taskID, String taskName, String priority, String
dueDate) {
        if (tail == null) {
            head = new Node(taskID, taskName, priority, dueDate);
            tail = head;
        } else {
            tail.next = new Node(taskID, taskName, priority, dueDate);
            tail = tail.next;
            tail.next = head;
        }

        size++;
    }

    // Add node at ith position
    public void addNode(int i, int taskID, String taskName, String priority,
String dueDate) {
        if (i > size + 1 || i < 1) {
            System.out.println("Provide correct position from 1 to " + (size +
1));
            return;
        }
        if (i == 1) {
```

```
            Node newNode = new Node(taskID, taskName, priority, dueDate);
            newNode.next = head;
            head = newNode;
            if (tail == null) {
                tail = head;
            }
            tail.next = head;
        } else if (i == size + 1) {
            Node newNode = new Node(taskID, taskName, priority, dueDate);
            tail.next = newNode;
            tail = newNode;
            tail.next = head;
        } else {
            int j = 1;
            Node temp = head;
            while (i - 1 != j) {
                temp = temp.next;
                j++;
            }
            Node temp2 = temp.next;
            temp.next = new Node(taskID, taskName, priority, dueDate);
            temp.next.next = temp2;
        }
        System.out.println("New node added successfully!");
        size++;
        return;
    }

    public void deleteRecord(int taskID) {
        Node temp = head;
        Node prev = null;
        do{
            if (temp.taskID == taskID) {
                if (temp == head) {
                    head = head.next;
                    if (head == null) {
                        tail = null;
                    }
                    tail.next = head;
                } else if (temp == tail) {
                    prev.next = head;
                    tail = prev;
                } else {
```

```
                prev.next = prev.next.next;
            }
            size--;
            System.out.println("Record Deleted Successfully!");
            return;
        }

        prev = temp;
        temp = temp.next;

    }
    while (temp != head);
    System.out.println("Record not found!");
}

public Node searchRecord(int taskID) {
    Node temp = head;
    do {
        if (temp.taskID == taskID) {
            System.out.println("Record Found!");
            return temp;
        }
        temp = temp.next;
    }
    while (temp != head);
    System.out.println("Record not found!");
    return null;
}

public void displayAllRecords() {
    if (head == null) {
        System.out.println("No record found!");
        return;
    }
    Node temp = head;
    int i = 0;
    do {
        System.out.println("Record-" + i++);
        System.out.println("RollNumber: " + temp.taskID);
        System.out.println("Name: " + temp.taskName);
        System.out.println("Age: " + temp.priority);
        System.out.println("Grade: " + temp.dueDate);
        temp = temp.next;
```

```
        }
        while (temp != head);
    }

    public void updateGrade(int taskID, String dueDate) {
        Node temp = head;
        do{
            if (temp.taskID == taskID) {
                temp.dueDate = dueDate;
                System.out.println("Due-date updated successfully!");
                return;
            }
            temp = temp.next;
        }
        while (temp != head);
        System.out.println("Record not found!");
    }

}
```

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.

```
public class InventoryManagementSystem {

    class Node {
        int itemID;
        String name;
        int quantity;
        double price;

        Node next;

        Node(int itemID, String name, int quantity, double price) {
            this.itemID = itemID;
            this.name = name;
            this.quantity = quantity;
            this.price = price;
            this.next = null;
        }
    }

    Node head = null;
    Node tail = null;
```

```
    int size = 0;

    public void addFirst(int itemID, String name, int quantity, double price) {
        if (head == null) {
            head = new Node(itemID, name, quantity, price);
            tail = head;
        } else {
            Node newNode = new Node(itemID, name, quantity, price);
            newNode.next = head;
            head = newNode;
        }

        size++;
    }

    public void addLast(int itemID, String name, int quantity, double price) {
        if (tail == null) {
            head = new Node(itemID, name, quantity, price);
            tail = head;
        } else {
            tail.next = new Node(itemID, name, quantity, price);
            tail = tail.next;
        }

        size++;
    }

    // Add node at ith position
    public void addNode(int i, int itemID, String name, int quantity, double
price) {
        if (i > size + 1 || i < 1) {
            System.out.println("Provide correct position from 1 to " + (size +
1));
            return;
        }
        if (i == 1) {
            Node newNode = new Node(itemID, name, quantity, price);
            newNode.next = head;
            head = newNode;
            if (tail == null) {
                tail = head;
            }
```

```
        } else if (i == size + 1) {
            Node newNode = new Node(itemID, name, quantity, price);
            tail.next = newNode;
            tail = newNode;
        } else {
            int j = 1;
            Node temp = head;
            while (i - 1 != j) {
                temp = temp.next;
                j++;
            }
            Node temp2 = temp.next;
            temp.next = new Node(itemID, name, quantity, price);
            temp.next.next = temp2;
        }
        System.out.println("New item added successfully!");
        size++;
        return;
    }

    public void removeItem(int itemID) {
        Node temp = head;
        Node prev = null;
        while (temp != null) {
            if (temp.itemID == itemID) {
                if (prev == null) {
                    head = head.next;
                    if (head == null) {
                        tail = null;
                    }
                } else {
                    if (temp == tail) {
                        prev.next = null;
                        tail = prev;
                    }
                    prev.next = prev.next.next;
                }
                size--;
                System.out.println("Item Removed Successfully!");
                return;
            }

            prev = temp;
```

```
            temp = temp.next;

        }
        System.out.println("Item not found!");
    }

    public void updateQuantity(int itemID, int quantity) {
        Node temp = head;
        while (temp != null) {
            if (temp.itemID == itemID) {
                temp.quantity = quantity;
                System.out.println("Quantity updated successfully!");
                return;
            }
            temp = temp.next;
        }
        System.out.println("Item not found!");
    }

    public Node searchRecord(int itemID, String name) {
        Node temp = head;
        while (temp != null) {
            if (temp.itemID == itemID || temp.name.equals(name)) {
                System.out.println("Record Found!");
                return temp;
            }
            temp = temp.next;
        }
        System.out.println("Record not found!");
        return null;
    }

    public void displayInventoryValue() {
        if (head == null) {
            System.out.println("No item found!");
            return;
        }
        Node temp = head;
        double totVal = 0;
        while (temp != null) {
            totVal += (temp.price * temp.quantity);
            temp = temp.next;}
        System.out.println("Total value of inventory is " + totVal);
```

```
    }
     public Node sortPrice(Node head) {
        if (head == null || head.next == null) {
            return head;
        }

        Node slow = head;
        Node fast = head;
        while (fast.next != null && fast.next.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }

        Node mid = slow.next;
        slow.next = null;
        Node left = sortPrice(head);
        Node right = sortPrice(mid);

        Node newHead = new Node(-1,"-1",-1,-1);
        Node temp = newHead;
        Node temp1 = left;
        Node temp2 = right;
        while (temp1 != null && temp2 != null) {
            if (temp1.price < temp2.price) {
                temp.next = temp1;
                temp1 = temp1.next;
            } else {
                temp.next = temp2;
                temp2 = temp2.next;
            }
            temp = temp.next;
        }

        if (temp1 != null) {
            temp.next = temp1;
        } else {
            temp.next = temp2;
        }

        this.head = newHead.next;
        return head;
}}
```

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.

```
public class LibraryManagementSystem {

    class Node {

        String title;
        String author;
        int bookID;
        String genre;
        boolean availability;

        Node next;
        Node prev;

        Node(String title, String author, int bookID, String genre, boolean
availability) {
            this.title = title;
            this.author = author;
            this.bookID = bookID;
            this.genre = genre;
            this.availability = true;
            this.next = null;
```

```
            this.prev = null;
        }
    }

    Node head = null;
    Node tail = null;

    int size = 0;

    public void addFirst(String title, String author, int bookID, String genre,
boolean availability) {
        if (head == null) {
            head = new Node(title, author, bookID, genre, availability);
            tail = head;
        } else {
            Node newNode = new Node(title, author, bookID, genre,
availability);
            newNode.next = head;
            head.prev = newNode;
            head = newNode;
        }

        size++;
    }

    public void addLast(String title, String author, int bookID, String genre,
boolean availability) {
        if (tail == null) {
            head = new Node(title, author, bookID, genre, availability);
            tail = head;
        } else {
            tail.next = new Node(title, author, bookID, genre, availability);
            tail.next.prev = tail;
            tail = tail.next;
        }

        size++;
    }

    // Add node at ith position
    public void addNode(int i, String title, String author, int bookID, String
genre, boolean availability) {
        if (i > size + 1 || i < 1) {
```

```
            System.out.println("Provide correct position from 1 to " + (size +
1));
            return;
        }
        if (i == 1) {
            Node newNode = new Node(title, author, bookID, genre,
availability);
            newNode.next = head;
            head.prev = newNode;
            head = newNode;
            if (tail == null) {
                tail = head;
            }
        } else if (i == size + 1) {
            Node newNode = new Node(title, author, bookID, genre,
availability);
            tail.next = newNode;
            newNode.prev = tail;
            tail = newNode;
        } else {
            int j = 1;
            Node temp = head;
            while (i - 1 > j) {
                temp = temp.next;
                j++;
            }
            Node temp2 = temp.next;
            temp.next = new Node(title, author, bookID, genre, availability);
            temp.next.prev = temp;
            temp.next.next = temp2;
            temp2.prev = temp.next;
        }
        System.out.println("New book added successfully!");
        size++;
        return;
    }

    public void removeBook(int bookID) {
        Node temp = head;
        Node prev = null;
        while (temp != null) {
            if (temp.bookID == bookID) {
                if (temp == head) {
```

```
                head = head.next;
                if (head == null) {
                    tail = null;
                } else {
                    head.prev = null;
                }
            } else if (temp == tail) {
                prev.next = null;
                tail = prev;
            } else {
                prev.next.next.prev = prev;
                prev.next = prev.next.next;
            }
            size--;
            System.out.println("Book Removed Successfully!");
            return;
        }

        prev = temp;
        temp = temp.next;
    }

    System.out.println("Book not found!");
}

public Node searchBook(String author, String title) {
    Node temp = head;
    while (temp != null) {
        if (temp.author.equals(author) || temp.title.equals(title)) {
            System.out.println("A record found");
            return temp;
        }
        temp = temp.next;
    }
    System.out.println("Book not found!");
    return null;
}

public void changeAvailability(Node n) {
    n.availability = !n.availability;
}

public void displayAllBooks() {
```

```
        if (head == null) {
            System.out.println("No Book found!");
            return;
        }

        System.out.println("Forward Order:\n");
        Node temp = head;
        int i = 1;
        while (temp != null) {
            System.out.println("Record-" + i++);
            System.out.println("Book Title: " + temp.title);
            System.out.println("Author: " + temp.author);
            System.out.println("bookID: " + temp.bookID);
            System.out.println("Genre: " + temp.genre);
            System.out.println("Availability: " + temp.availability);
            System.out.println();
            temp = temp.next;
        }
        System.out.println("\n\nReverse Order:\n");
        temp = tail;
        i = size;
        while (temp != null) {
            System.out.println("Record-" + i--);
            System.out.println("Book Title: " + temp.title);
            System.out.println("Author: " + temp.author);
            System.out.println("bookID: " + temp.bookID);
            System.out.println("Genre: " + temp.genre);
            System.out.println("Availability: " + temp.availability);
            System.out.println();
            temp = temp.prev;
        }
    }

    public int totalBooks() {
        return this.size;
    }

}
```

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.

```
import java.util.Scanner;

class Process {
    int id, burstTime, remainingTime;
    Process next;

    public Process(int id, int burstTime) {
        this.id = id;
        this.burstTime = burstTime;
        this.remainingTime = burstTime;
        this.next = null;
    }
}

class CircularLinkedList {
    private Process head = null, tail = null;

    // Add a new process at the end
    public void addProcess(int id, int burstTime) {
        Process newProcess = new Process(id, burstTime);
        if (head == null) {
```

```
            head = newProcess;
            tail = newProcess;
            newProcess.next = head; // Circular link
        } else {
            tail.next = newProcess;
            tail = newProcess;
            tail.next = head; // Maintain circular link
        }
    }

    // Remove a process by ID
    public void removeProcess(int id) {
        if (head == null)
            return;

        Process curr = head, prev = null;

        // Find the process to delete
        do {
            if (curr.id == id) {
                if (curr == head && curr == tail) { // Only one process
                    head = tail = null;
                } else {
                    if (curr == head)
                        head = head.next;
                    if (curr == tail)
                        tail = prev;
                    if (prev != null)
                        prev.next = curr.next;
                }
                return;
            }
            prev = curr;
            curr = curr.next;
        } while (curr != head);
    }

    // Check if the list is empty
    public boolean isEmpty() {
        return head == null;
    }

    // Display the process queue
```

```
    public void displayProcesses() {
        if (head == null) {
            System.out.println("No processes in the queue.");
            return;
        }
        Process temp = head;
        System.out.print("Processes in queue: ");
        do {
            System.out.print("[P" + temp.id + " | Remaining: " +
temp.remainingTime + "ms] -> ");
            temp = temp.next;
        } while (temp != head);
        System.out.println();
    }

    // Simulate Round Robin Scheduling
    public void roundRobinScheduling(int timeQuantum) {
        if (head == null)
            return;

        int totalProcesses = 0;
        Process temp = head;
        do {
            totalProcesses++;
            temp = temp.next;
        } while (temp != head);

        int[] waitingTime = new int[totalProcesses];
        int[] turnaroundTime = new int[totalProcesses];
        int currentTime = 0;

        while (!isEmpty()) {
            temp = head;
            do {
                if (temp.remainingTime > 0) {
                    int executeTime = Math.min(timeQuantum,
temp.remainingTime);
                    temp.remainingTime -= executeTime;
                    currentTime += executeTime;

                    if (temp.remainingTime == 0) {
                        turnaroundTime[temp.id - 1] = currentTime;
                        removeProcess(temp.id);
```

```
                    }
                }
                displayProcesses();
                temp = temp.next;
            } while (temp != head && !isEmpty());
        }

        // Calculate Waiting Time
        for (int i = 0; i < totalProcesses; i++) {
            waitingTime[i] = turnaroundTime[i] - (i + 1) * timeQuantum;
        }

        // Display Average Waiting & Turnaround Time
        int totalWaitingTime = 0, totalTurnaroundTime = 0;
        for (int i = 0; i < totalProcesses; i++) {
            totalWaitingTime += waitingTime[i];
            totalTurnaroundTime += turnaroundTime[i];
        }

        System.out.println("\nAverage Waiting Time: " + (double)
totalWaitingTime / totalProcesses);
        System.out.println("Average Turnaround Time: " + (double)
totalTurnaroundTime / totalProcesses);
    }
}

public class RoundRobin {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        CircularLinkedList processQueue = new CircularLinkedList();

        System.out.print("Enter the time quantum: ");
        int timeQuantum = scanner.nextInt();

        System.out.print("Enter the number of processes: ");
        int n = scanner.nextInt();

        for (int i = 1; i <= n; i++) {
            System.out.print("Enter burst time for Process " + i + ": ");
            int burstTime = scanner.nextInt();
            processQueue.addProcess(i, burstTime);
        }
```

```
        System.out.println("\nStarting Round Robin Scheduling...");
        processQueue.roundRobinScheduling(timeQuantum);

        scanner.close();
    }
}
```

---

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.

```
import java.util.*;

public class SocialMedia {

    class Node {

        int userID;
        String name;
        int age;
```

```
        List<Node> friendList;

        Node next;

        Node(int userID, String name, int age ) {
            this.userID = userID;
            this.name = name;
            this.age = age;
            this.friendList = new ArrayList<>();
            this.next = null;
        }
    }

    Node head = null;
    Node tail = null;

    int size = 0;


    public void addUser(int userID, String name, int age ) {
        if (tail == null) {
            head = new Node(userID, name, age);
            tail = head;
        } else {
            tail.next = new Node(userID, name, age);
            tail = tail.next;
        }

        size++;
    }

    public void addConnection(int userID1, int userID2) {
        Node temp = head;
        Node friend1 = null;
        Node friend2 = null;
        while (temp != null) {
            if (temp.userID == userID1) {
                friend1 = temp;
            } else if (temp.userID == userID2) {
                friend2 = temp;
            }
            temp = temp.next;
        }
```

```
        if (friend1 == null) {
            System.out.println("User with user id " + userID1 + " not found.");
            return;
        }
        if (friend2 == null) {
            System.out.println("User with user id " + userID2 + " not found.");
            return;
        }

        friend1.friendList.add(friend2);
        friend2.friendList.add(friend1);

    }

    public void removeConnection(int userID1, int userID2) {
        Node temp = head;
        Node friend1 = null;
        Node friend2 = null;
        while (temp != null) {
            if (temp.userID == userID1) {
                friend1 = temp;
            } else if (temp.userID == userID2) {
                friend2 = temp;
            }
            temp = temp.next;
        }

        if (friend1 == null) {
            System.out.println("User with user id " + userID1 + " not found.");
            return;
        }
        if (friend2 == null) {
            System.out.println("User with user id " + userID2 + " not found.");
            return;
        }

        friend1.friendList.remove(friend2);
        friend2.friendList.remove(friend1);
    }


    public void mutualFriends(int userID1, int userID2) {
```

```
        Node temp = head;
        Node friend1 = null;
        Node friend2 = null;
        while (temp != null) {
            if (temp.userID == userID1) {
                friend1 = temp;
            } else if (temp.userID == userID2) {
                friend2 = temp;
            }
            temp = temp.next;
        }

        if (friend1 == null) {
            System.out.println("User with user id " + userID1 + " not found.");
            return;
        }
        if (friend2 == null) {
            System.out.println("User with user id " + userID2 + " not found.");
            return;
        }

        for (int i = 0; i < friend1.friendList.size(); i++) {
            for (int j = 0; j < friend2.friendList.size(); j++) {
                if (friend1.friendList.get(i) == friend2.friendList.get(j)) {
                    Node f = friend2.friendList.get(j);
                    System.out.println("Friend Name: " + f.name);
                    System.out.println("Friend ID: " + f.userID);
                    System.out.println();
                    break;
                }

            }
        }
    }

    public void displayFriends(int userID) {
        Node temp = head;
        Node user = null;
        while (temp != null) {
            if (temp.userID == userID) {
                user = temp;
                break;
            }
```

```
            temp = temp.next;
        }

        if (user == null) {
            System.out.println("User with user id " + userID + " not found.");
            return;
        }

        for (int i = 0; i < user.friendList.size(); i++) {
            Node f = user.friendList.get(i);
            System.out.println("Name: " + f.name);
            System.out.println("User ID: " + f.userID);
            System.out.println();
        }

    }

    public Node searchRecord(int userID, String name) {
        Node temp = head;
        while (temp != null) {
            if (temp.userID == userID || temp.name.equals(name)) {
                System.out.println("Record Found!");
                return temp;
            }
            temp = temp.next;
        }
        System.out.println("Record not found!");
        return null;
    }


    public void countFriends() {
        Node temp = head;
        while (temp != null) {
            System.out.println(temp.userID);
            System.out.println(temp.name);
            System.out.println("Number of Friends: " + temp.friendList.size());
            temp = temp.next;
        }
    }
}
```

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.

```
public class UndoRedoTextEditor {

    int limit = 0;

    UndoRedoTextEditor(int limit) {
        this.limit = limit;
    }

    class Node {

        String text;

        Node next;
        Node prev;

        Node(String text) {
            this.text = text;
            this.next = null;
            this.prev = null;
        }
```

```
    }

    Node head = null;
    Node tail = null;
    Node current = null;
    int size = 0;


    public void addState(String text) {
        if (tail == null) {
            head = new Node(text);
            tail = head;
            current = head;
        } else {
            tail.next = new Node(text);
            tail.next.prev = tail;
            tail = tail.next;
            current = tail;
        }

        size++;
        if (size > limit) {
            head = head.next;
            size--;
        }
    }


    public void undo() {
        if (current.prev != null) {
            current = current.prev;
        } else {
            System.out.println("No state found!");
        }
    }


    public void redo() {
        if (current.next != null) {
            current = current.next;
        } else {
            System.out.println("No record found!");
        }
```

```
    }


    public void displayCurrentState() {
        System.out.println(current.text);
    }

}
```

---

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.

```
public class TicketReservationSystem {

    class Node {
        int ticketID;
        String customerName;
        String movieName;
```

```
        int seatNumber;
        String bookingTime;

        Node next;

        Node(int ticketID, String customerName, String movieName, String
bookingTime, int seatNumber) {
            this.ticketID = ticketID;
            this.customerName = customerName;
            this.movieName = movieName;
            this.bookingTime = bookingTime;
            this.seatNumber = seatNumber;
            this.next = this;
        }
    }

    static Node head = null;
    static Node tail = null;

    int size = 0;


    public void addNewTicket(int ticketID, String customerName, String
movieName, String bookingTime, int seatNumber) {
        if (tail == null) {
            head = new Node(ticketID, customerName, movieName, bookingTime,
seatNumber);
            tail = head;
        } else {
            tail.next = new Node(ticketID, customerName, movieName,
bookingTime, seatNumber);
            tail = tail.next;
            tail.next = head;
        }

        size++;
    }


    public void removeTicket(int ticketID) {
        Node temp = head;
        Node prev = null;
        do {
```

```
            if (temp.ticketID == ticketID) {
                if (temp == head) {
                    head = head.next;
                    if (head == null) {
                        tail = null;
                    }
                    tail.next = head;
                } else if (temp == tail) {
                    prev.next = head;
                    tail = prev;
                } else {
                    prev.next = prev.next.next;
                }
                size--;
                System.out.println("Ticket Removed Successfully!");
                return;
            }

            prev = temp;
            temp = temp.next;

        } while (temp != head);
        System.out.println("Ticket not found!");
    }


    public void displayAllRecords() {
        if (head == null) {
            System.out.println("No Ticket found!");
            return;
        }
        Node temp = head;
        int i = 0;
        do {
            System.out.println("Record-" + i++);
            System.out.println("TicketID: " + temp.ticketID);
            System.out.println("Customer Name: " + temp.customerName);
            System.out.println("Movie Name: " + temp.movieName);
            System.out.println("Booking Time: " + temp.bookingTime);
            System.out.println("Seat Number: " + temp.seatNumber);
            temp = temp.next;
        } while (temp != head);
    }
```

```
    public Node searchRecord(String customerName, String movieName) {
        Node temp = head;
        do {
            if (temp.customerName.equals(customerName) ||
temp.movieName.equals(movieName)) {
                System.out.println("Ticket Found!");
                return temp;
            }
            temp = temp.next;
        } while (temp != head);
        System.out.println("Ticket not found!");
        return null;
    }

    public int totalBookedTickets() {
        return this.size;
    }
}
```