Sure! Below is how you can implement a **singly linked list** using **recursion** in **Java**. The implementation includes the following operations:

1. **Insertion at the end (recursive)**
2. **Display the list (recursive)**
3. **Search an element (recursive)**
4. **Delete a node (recursive)**
5. **Find the length of the list (recursive)**

Step 1: Define the Node Class

Each node contains:

- **data** (value of the node)
- **next** (pointer to the next node)

```java
class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}
```

Step 2: Define the LinkedList Class

The **LinkedList** class will contain methods that use recursion.

1. Insert at the End (Recursive)

```java
class LinkedList {
    private Node head;

    public LinkedList() {
        this.head = null;
    }

    // Recursive helper function to insert at the end
    private Node insertRecursive(Node node, int data) {
        if (node == null) {
            return new Node(data);
        }
        node.next = insertRecursive(node.next, data);
        return node;
```

/

```
    }

    // Public method to call the recursive function
    public void insert(int data) {
        head = insertRecursive(head, data);
    }
```

2. Print Linked List (Recursive)

```
    private void printRecursive(Node node) {
        if (node == null) {
            System.out.println("null");
            return;
        }
        System.out.print(node.data + " -> ");
        printRecursive(node.next);
    }

    public void printList() {
        printRecursive(head);
    }
```

3. Search for an Element (Recursive)

```
    private boolean searchRecursive(Node node, int key) {
        if (node == null) {
            return false;
        }
        if (node.data == key) {
            return true;
        }
        return searchRecursive(node.next, key);
    }

    public boolean search(int key) {
        return searchRecursive(head, key);
    }
```

4. Delete a Node (Recursive)

```
    private Node deleteRecursive(Node node, int key) {
        if (node == null) {
            return null;
```

/

```
        }
        if (node.data == key) {
            return node.next; // Remove current node
        }
        node.next = deleteRecursive(node.next, key);
        return node;
    }

    public void delete(int key) {
        head = deleteRecursive(head, key);
    }
```

---

5. Find Length of the Linked List (Recursive)

```
    private int lengthRecursive(Node node) {
        if (node == null) {
            return 0;
        }
        return 1 + lengthRecursive(node.next);
    }

    public int length() {
        return lengthRecursive(head);
    }
}
```

Step 3: Test the Implementation

```
public class LinkedListTest {
    public static void main(String[] args) {
        LinkedList ll = new LinkedList();

        // Insert elements recursively
        ll.insert(1);
        ll.insert(2);
        ll.insert(3);
        ll.insert(4);

        // Print the linked list
        ll.printList(); // Output: 1 -> 2 -> 3 -> 4 -> null

        // Search for an element
        System.out.println(ll.search(3)); // Output: true
        System.out.println(ll.search(5)); // Output: false

        // Delete an element
        ll.delete(2);
```

/

```
        ll.printList(); // Output: 1 -> 3 -> 4 -> null

        // Get the length of the list
        System.out.println(ll.length()); // Output: 3
    }
}
```

Key Takeaways

1. **Recursion Base Case**: Each recursive function has a stopping condition (when `node == null`).
2. **Recursive Calls**: Functions update the `next` pointer to maintain linkage.
3. **Efficiency Consideration**: While recursion simplifies logic, it may cause stack overflow for very long lists.

Would you like me to extend this to a **doubly linked list** as well? 🚀

/