Singly Linked List in Java - Enhanced Notes

This document provides a **clean, structured, and industry-relevant implementation** of a Singly Linked List (SLL) in Java. It includes **real-world analogies**, use cases, and **developer-friendly comments** to bridge theory and practical coding.

What is a Singly Linked List?

A **Singly Linked List** is a **dynamic linear data structure** where each element (node) contains: - `data`: the actual value - `next`: a reference to the next node

Only the head of the list is directly accessible. Traversal is always forward, from one node to the next.

Real-World Analogy:

Imagine a **train** where each coach is linked to the next one. You can move forward coach by coach, but there's no direct access to the last one.

Industry Use Cases:

- **Task schedulers** (OS job queues)
- **Playlists** (music, video)
- **Undo functionality** (in editors)
- **Network packet queues**

Node Class

```java
/**
 * Represents a node in a singly linked list.
 */
class Node {
    int data;        // The value held by the node
    Node next;       // Reference to the next node

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

SinglyLinkedList Class - Core API

```java
/**
 * Implements basic operations of a Singly Linked List.
 */
public class SinglyLinkedList {
    Node head;  // Pointer to the first node in the list

    public SinglyLinkedList() {
        this.head = null;
    }
```

Core Operations

1. Insert at Beginning

```java
    public void insertAtBeginning(int data) {
        Node newNode = new Node(data);
        newNode.next = head;
        head = newNode;
    }
```

2. Insert at End

```java
    public void insertAtEnd(int data) {
        Node newNode = new Node(data);

        if (head == null) {
            head = newNode;
            return;
        }

        Node current = head;
        while (current.next != null) {
            current = current.next;
        }

        current.next = newNode;
    }
```

3. Insert at Specific Position

```java
    public void insertAtPosition(int data, int position) {
        if (position <= 0) {
            System.out.println("Invalid position");
            return;
```

```java
        }

        if (position == 1) {
            insertAtBeginning(data);
            return;
        }

        Node current = head;
        int currentPosition = 1;

        while (current != null && currentPosition < position - 1) {
            current = current.next;
            currentPosition++;
        }

        if (current == null) {
            System.out.println("Invalid position, fewer nodes in list");
            return;
        }

        Node newNode = new Node(data);
        newNode.next = current.next;
        current.next = newNode;
    }
```

4. **Delete at Beginning**

```java
    public void deleteAtBeginning() {
        if (head == null) {
            System.out.println("List is empty");
            return;
        }
        head = head.next;
    }
```

5. **Delete at End**

```java
    public void deleteAtEnd() {
        if (head == null) {
            System.out.println("List is empty");
            return;
        }

        if (head.next == null) {
            head = null;
            return;
```

```
    }

    Node current = head;
    while (current.next.next != null) {
        current = current.next;
    }

    current.next = null;
}
```

6. Delete at Specific Position

```
public void deleteAtPosition(int position) {
    if (position <= 0) {
        System.out.println("Invalid position");
        return;
    }

    if (position == 1) {
        deleteAtBeginning();
        return;
    }

    Node current = head;
    int currentPosition = 1;

    while (current != null && currentPosition < position - 1) {
        current = current.next;
        currentPosition++;
    }

    if (current == null || current.next == null) {
        System.out.println("Invalid position, fewer nodes in list");
        return;
    }

    current.next = current.next.next;
}
```

7. Search for a Key

```
public void search(int key) {
    if (head == null) {
        System.out.println("List is empty");
        return;
    }
```

```java
        Node current = head;
        while (current != null) {
            if (current.data == key) {
                System.out.println("Key found");
                return;
            }
            current = current.next;
        }

        System.out.println("Key not found");
    }
```

8. Display List

```java
    public void printList() {
        if (head == null) {
            System.out.println("List is empty");
            return;
        }

        Node current = head;
        while (current != null) {
            System.out.print(current.data + " --> ");
            current = current.next;
        }
        System.out.println("None");
    }
}
```

Driver Code Example

```java
public class Main {
    public static void main(String[] args) {
        SinglyLinkedList list = new SinglyLinkedList();

        list.insertAtPosition(10, -1);   // Invalid
        list.deleteAtPosition(-1);       // Invalid

        list.insertAtPosition(10, 1);    // Insert at head
        list.deleteAtPosition(1);        // Delete head

        list.insertAtEnd(10);
        list.insertAtEnd(20);
        list.insertAtEnd(30);
```

```java
        list.insertAtPosition(15, 2);
        list.insertAtPosition(40, 4);
        list.insertAtPosition(100, 10);  // Invalid

        list.printList();  // Print the current list
    }
}
```

Tips for Developers

- Always validate indices
- Watch out for null pointer cases
- Write helper/debug methods for better testing

Conclusion

Singly Linked Lists in Java offer flexibility with dynamic memory and are crucial for real-time applications like job queues, editor history, and memory management.

Doubly Linked List in Java - Enhanced Notes

This guide presents a clean and professional implementation of a **Doubly Linked List (DLL)** in Java, suitable for academic and industry learners. It follows a structure similar to the Singly Linked List documentation, with a focus on practical insights and robust code.

What is a Doubly Linked List?

A **Doubly Linked List** is a dynamic data structure where each node has three fields: - `data`: the actual value - `prev`: reference to the previous node - `next`: reference to the next node

This allows traversal in **both forward and backward directions**.

Real-World Analogy:

Think of a **two-way metro line**. You can move forward or reverse from any station (node) because each one knows its next and previous stops.

Where It's Used:

- **Undo/Redo functionality** in editors
- **Navigation systems** with forward/back history
- **Music/Video Players** for bidirectional playlist control
- **Complex data manipulation systems**

Node Class

```java
/**
 * Represents a node in a doubly linked list.
 */
class Node {
    int data;
    Node prev;
    Node next;

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

DoublyLinkedList Class - Core API

```java
/**
 * Implements core operations of a Doubly Linked List.
 */
public class DoublyLinkedList {
    Node head;

    public DoublyLinkedList() {
        this.head = null;
    }
```

Core Operations

1. Insert at Beginning

```java
public void insertAtBeginning(int data) {
    Node newNode = new Node(data);
    if (head != null) {
        newNode.next = head;
        head.prev = newNode;
    }
    head = newNode;
}
```

2. Insert at End

```java
public void insertAtEnd(int data) {
    Node newNode = new Node(data);
    if (head == null) {
        head = newNode;
        return;
    }

    Node current = head;
    while (current.next != null) {
        current = current.next;
    }

    current.next = newNode;
    newNode.prev = current;
}
```

3. Insert at Position

```java
public void insertAtPosition(int data, int position) {
    if (position <= 0) {
        System.out.println("Invalid position");
        return;
    }

    if (position == 1) {
        insertAtBeginning(data);
        return;
    }

    Node current = head;
    int currentPosition = 1;

    while (current != null && currentPosition < position - 1) {
        current = current.next;
        currentPosition++;
    }

    if (current == null) {
        System.out.println("Invalid position");
        return;
    }

    Node newNode = new Node(data);
    newNode.next = current.next;
    newNode.prev = current;

    if (current.next != null) {
        current.next.prev = newNode;
    }

    current.next = newNode;
}
```

4. Delete at Beginning

```java
public void deleteAtBeginning() {
    if (head == null) {
        System.out.println("List is empty");
        return;
    }

    head = head.next;
```

```
        if (head != null) {
            head.prev = null;
        }
    }
```

5. Delete at End

```
public void deleteAtEnd() {
    if (head == null) {
        System.out.println("List is empty");
        return;
    }

    if (head.next == null) {
        head = null;
        return;
    }

    Node current = head;
    while (current.next != null) {
        current = current.next;
    }

    current.prev.next = null;
}
```

6. Delete at Position

```
public void deleteAtPosition(int position) {
    if (position <= 0 || head == null) {
        System.out.println("Invalid position or list is empty");
        return;
    }

    if (position == 1) {
        deleteAtBeginning();
        return;
    }

    Node current = head;
    int currentPosition = 1;

    while (current != null && currentPosition < position) {
        current = current.next;
        currentPosition++;
    }
```

```java
        if (current == null) {
            System.out.println("Invalid position");
            return;
        }

        if (current.prev != null) {
            current.prev.next = current.next;
        }

        if (current.next != null) {
            current.next.prev = current.prev;
        }
    }
```

7. Display List (Forward)

```java
    public void printForward() {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " <-> ");
            current = current.next;
        }
        System.out.println("None");
    }
```

8. Display List (Backward)

```java
    public void printBackward() {
        if (head == null) {
            System.out.println("List is empty");
            return;
        }

        Node current = head;
        while (current.next != null) {
            current = current.next;
        }

        while (current != null) {
            System.out.print(current.data + " <-> ");
            current = current.prev;
        }
        System.out.println("None");
    }
}
```

Driver Code Example

```
public class Main {
    public static void main(String[] args) {
        DoublyLinkedList list = new DoublyLinkedList();

        list.insertAtEnd(10);
        list.insertAtEnd(20);
        list.insertAtEnd(30);
        list.insertAtPosition(15, 2);
        list.insertAtBeginning(5);
        list.deleteAtPosition(3);

        list.printForward();   // Forward display
        list.printBackward();  // Backward display
    }
}
```

Developer Notes

- Prefer DLL when two-way navigation is required
- Handle edge cases in deletion (especially head/tail)
- Always check for `null` before accessing `.next` or `.prev`

Conclusion

Doubly Linked Lists provide **bidirectional navigation**, making them more versatile than singly linked lists for many applications. They're foundational to **tree**, **graph**, and **navigation-based data structures**.