**Linked Lists** in Java: Singly, Doubly and Circular Linked Lists

1. Introduction

Linked lists are fundamental data structures used to implement dynamic collections. Unlike arrays, linked lists are not stored contiguously in memory, allowing efficient insertion and deletion of elements.

2. Singly Linked List

A singly linked list consists of nodes where each node has:

- A data field to store the value.

- A reference (or link) to the next node in the sequence.

2.1 Node Class for Singly Linked List

class Node {

int data;

Node next;

Node(int data) {

this.data = data;

this.next = null;

}

}

2.2 Singly Linked List Class

class SinglyLinkedList {

private Node head;

// Method to insert a new node at the beginning

public void insertAtBeginning(int data) {

Node newNode = new Node(data);

newNode.next = head;

head = newNode;

}

// Method to insert a new node at the end

public void insertAtEnd(int data) {

Node newNode = new Node(data);

if (head == null) {

head = newNode;

} else {

Node current = head;

while (current.next != null) {

current = current.next;

}

current.next = newNode;

}

}

// Method to delete a node by value

public void deleteByValue(int data) {

if (head == null) return;

if (head.data == data) {

head = head.next;

return;

}

Node current = head;

while (current.next != null && current.next.data != data) {

current = current.next;

}

if (current.next != null) {

current.next = current.next.next;

}

}

// Method to display the linked list

public void display() {

Node current = head;

while (current != null) {

System.out.print(current.data + " -> ");

current = current.next;

}

System.out.println("null");

}

}

3. Doubly Linked List

A doubly linked list consists of nodes where each node has:

- A data field to store the value.

- A reference to the next node in the sequence.

- A reference to the previous node in the sequence.

3.1 Node Class for Doubly Linked List

class DoublyNode {

int data;

DoublyNode next;

DoublyNode prev;

DoublyNode(int data) {

this.data = data;

this.next = null;

this.prev = null;

}

}

3.2 Doubly Linked List Class

class DoublyLinkedList {

private DoublyNode head;

// Method to insert a new node at the beginning

public void insertAtBeginning(int data) {

DoublyNode newNode = new DoublyNode(data);

if (head != null) {

head.prev = newNode;

}

newNode.next = head;

head = newNode;

}

// Method to insert a new node at the end

public void insertAtEnd(int data) {

DoublyNode newNode = new DoublyNode(data);

if (head == null) {

head = newNode;

} else {

DoublyNode current = head;

while (current.next != null) {

current = current.next;

}

current.next = newNode;

newNode.prev = current;

}

}

// Method to delete a node by value

public void deleteByValue(int data) {

if (head == null) return;

if (head.data == data) {

head = head.next;

if (head != null) {

head.prev = null;

}

return;

}

DoublyNode current = head;

while (current != null && current.data != data) {

current = current.next;

}

if (current != null) {

if (current.next != null) {

current.next.prev = current.prev;

}

if (current.prev != null) {

current.prev.next = current.next;

}

}

}

// Method to display the linked list in forward direction

public void displayForward() {

DoublyNode current = head;

while (current != null) {

System.out.print(current.data + " <-> ");

current = current.next;

}

System.out.println("null");

}

// Method to display the linked list in backward direction

public void displayBackward() {

if (head == null) return;

DoublyNode current = head;

while (current.next != null) {

current = current.next;

}

while (current != null) {

System.out.print(current.data + " <-> ");

current = current.prev;

}

System.out.println("null");

}

}

4. Mathematical Context

4.1 Complexity Analysis

- Insertion at Beginning: O(1)

- Insertion at End: O(n) for singly linked list, O(1) if tail reference is maintained in doubly linked list.

- Deletion by Value: O(n)

- Traversal: O(n)

4.2 Use Cases

- Singly Linked List: Suitable for applications with more frequent insertion/deletion at the beginning or end.

- Doubly Linked List: Suitable when bidirectional traversal is needed or more frequent deletions from the middle.

5. Conclusion

Linked lists provide a flexible way to manage collections of data. While singly linked lists are simpler and use less memory, doubly linked lists offer more functionality at the cost of additional space. Understanding their operations and performance implications is crucial for choosing the right type for your application.

6. Circular Linked List

Circular linked lists are a variation of linked lists where the last node points to the first node, forming a circle. This structure can be implemented as either a singly circular linked list or a doubly circular linked list.

6.1 Singly Circular Linked List

In a singly circular linked list, each node has a reference to the next node, and the last node points back to the first node.

# 6.1.1 Node Class for Singly Circular Linked List

class CircularNode {

int data;

CircularNode next;

CircularNode(int data) {

this.data = data;

this.next = null;

}

}

# 6.1.2 Singly Circular Linked List Class

class SinglyCircularLinkedList {

private CircularNode head;

// Method to insert a new node at the beginning

public void insertAtBeginning(int data) {

CircularNode newNode = new CircularNode(data);

if (head == null) {

head = newNode;

head.next = head;

} else {

CircularNode current = head;

while (current.next != head) {

current = current.next;

}

newNode.next = head;

current.next = newNode;

head = newNode;

}

}

// Method to insert a new node at the end

public void insertAtEnd(int data) {

CircularNode newNode = new CircularNode(data);

if (head == null) {

head = newNode;

head.next = head;

} else {

CircularNode current = head;

while (current.next != head) {

current = current.next;

}

current.next = newNode;

newNode.next = head;

}

}

// Method to delete a node by value

public void deleteByValue(int data) {

if (head == null) return;

if (head.data == data) {

if (head.next == head) {

head = null;

} else {

CircularNode current = head;

while (current.next != head) {

current = current.next;

}

current.next = head.next;

head = head.next;

}

return;

}

CircularNode current = head;

while (current.next != head && current.next.data != data) {

current = current.next;

}

if (current.next != head) {

current.next = current.next.next;

}

}

// Method to display the linked list

public void display() {

if (head == null) return;

CircularNode current = head;

do {

System.out.print(current.data + " -> ");

current = current.next;

} while (current != head);

System.out.println("(head)");

}

}

6.2 Doubly Circular Linked List

In a doubly circular linked list, each node has a reference to both the next and previous nodes, forming a circle.

# 6.2.1 Node Class for Doubly Circular Linked List

class DoublyCircularNode {

int data;

DoublyCircularNode next;

DoublyCircularNode prev;

DoublyCircularNode(int data) {

this.data = data;

this.next = null;

this.prev = null;

}

}

# 6.2.2 Doubly Circular Linked List Class

class DoublyCircularLinkedList {

private DoublyCircularNode head;

// Method to insert a new node at the beginning

public void insertAtBeginning(int data) {

DoublyCircularNode newNode = new DoublyCircularNode(data);

if (head == null) {

head = newNode;

head.next = head;

head.prev = head;

} else {

DoublyCircularNode last = head.prev;

newNode.next = head;

newNode.prev = last;

last.next = newNode;

head.prev = newNode;

head = newNode;

}

}

// Method to insert a new node at the end

public void insertAtEnd(int data) {

DoublyCircularNode newNode = new DoublyCircularNode(data);

if (head == null) {

head = newNode;

head.next = head;

head.prev = head;

} else {

DoublyCircularNode last = head.prev;

newNode.next = head;

newNode.prev = last;

last.next = newNode;

head.prev = newNode;

}

}

// Method to delete a node by value

public void deleteByValue(int data) {

if (head == null) return;

if (head.data == data) {

if (head.next == head) {

head = null;

} else {

DoublyCircularNode last = head.prev;

head = head.next;

head.prev = last;

last.next = head;

}

return;

}

DoublyCircularNode current = head;

while (current.next != head && current.data != data) {

current = current.next;

}

if (current != head) {

current.prev.next = current.next;

current.next.prev = current.prev;

}

}

// Method to display the linked list in forward direction

public void displayForward() {

if (head == null) return;

DoublyCircularNode current = head;

do {

System.out.print(current.data + " <-> ");

current = current.next;

} while (current != head);

System.out.println("(head)");

}

// Method to display the linked list in backward direction

public void displayBackward() {

if (head == null) return;

DoublyCircularNode last = head.prev;

DoublyCircularNode current = last;

do {

System.out.print(current.data + " <-> ");

current = current.prev;

} while (current != last);

System.out.println("(head)");

}

}

7. Mathematical Context

7.1 Complexity Analysis

- Insertion at Beginning: O(1)

- Insertion at End: O(n) for singly circular linked list, O(1) if tail reference is maintained in doubly circular linked list.

- Deletion by Value: O(n)

- Traversal: O(n)

7.2 Use Cases

- Singly Circular Linked List: Useful in applications where cyclic traversal is required, such as round-robin scheduling.

- Doubly Circular Linked List: Suitable for applications that require both forward and backward traversal, such as in navigation systems.

8. Conclusion

Circular linked lists offer the benefit of cyclic traversal, making them suitable for certain applications like buffering systems and round-robin scheduling. Understanding the properties and operations of both singly and doubly circular linked lists is essential for leveraging their full potential in various scenarios.