Linked List with Tail

In this lesson, we will study another variation of linked lists called "Linked List with a Tail". We will also learn the benfits of using a tail pointer in both SLL and DLL. An implementation of DLL with a Tail will also be covered.

We'll cover the following

Introduction

Another variation of a basic linked list is a **Linked List with a Tail**. In this type of list, in addition to the *head* being the starting of the list, a *tail* is used as the pointer to the last node of the list. Both *SLL* and *DLL* can be implemented using a **tail pointer**.

Comparison between SLL with Tail and DLL with Tail

The benefit of using a tail pointer is seen in the *insertion* and *deletion* operations at the **end** of the list. Let's analyze the efficiency, in terms of time complexity, of both of these operations in SLL and DLL.

	Singly Linked List with Tail	**Doubly Linked List with Tail**

	Singly Linked List with Tail	Doubly Linked List with Tail
`insertAtTail()` or `insertAtEnd()`	Takes **constant time** i.e, `O(1)`	Takes **constant time** i.e `O(1)`
`deleteAtTail()` or `deleteAtEnd()`	Will take **linear time** i.e, `O(n)` because for deletion of a node, the previous element of that node should also be known.	Will take **constant time** i.e, `O(1)` , because DLL also has a pointer to the previous node!

From this comparison, we can see that the real advantage of using a *tail* pointer comes in the `deleteAtEnd` scenario while dealing with **Doubly Linked Lists** as the tail provides a more efficient implementation of this function.

Implementation of Doubly Linked List with Tail

In the code below, we have used a member variable called `tailNode` , which will point to the **last node** of the list. Initially, it will be equal to `null` .

```
        return headNode;
    }

    //getter for tailNode
    public Node getTailNode() {
        return tailNode;
    }

    //getter for size
    public int getSize() {
        return size;
    }

    //print list function
    public void printList() {
        if (isEmpty()) {
            System.out.println("List is Empty!");
            return;
        }
```

```
    }

        Node temp = headNode;
        System.out.print("List : null <- ");

        while (temp.nextNode != null) {
            System.out.print(temp.data.toString() + " <-> ");
            temp = temp.nextNode;
        }

        System.out.println(temp.data.toString() + " -> null");
    }
}
```

class for Doubly Linked list with Tail

Impact on Insertion

1) **`insertAtHead()`**

Insertion at head remains almost the same as in DLL *without* tail. The only difference is that if the element is inserted in an already **empty** linked list then, we have to update the `tailNode` as well.

2) **`insertAtEnd()`**

Insertion at the end is a *linear* operation in DLL *without* tail. However, in DLL *with* tail, it becomes a **constant** operation. We simply insert the new node as the `nextNode` of the `tailNode` and then update the `tailNode` to point to the new node, after insertion.

Let us take a look at the code for the operations mentioned above.

main.java

DoublyLinkedList.java

```
public class DoublyLinkedList<T> {

    //Node inner class for DLL
    public class Node {
        public T data;
        public Node nextNode;
        public Node prevNode;
    }

    //member variables
    public Node headNode;
    public Node tailNode;
    public int size;

    //constructor
```

```
    //constructor
    public DoublyLinkedList() {
        this.headNode = null;
        this.tailNode = null; //null initially
        this.size = 0;
    }

    //returns true if list is empty
    public boolean isEmpty() {
        if (headNode == null && tailNode == null) //checking tailNode to make sure
            return true;
        return false;
    }

    //getter for headNode
    public Node getHeadNode() {
        return headNode;
    }

    //getter for tailNode
    public Node getTailNode() {
        return tailNode;
    }

    //getter for size
    public int getSize() {
        return size;
    }

    //insert at start of the list
    public void insertAtHead(T data) {
        Node newNode = new Node();
        newNode.data = data;
        newNode.nextNode = this.headNode; //Linking newNode to head's nextNode
        newNode.prevNode = null; //it will be inserted at start so prevNode will be
        if (!isEmpty())
            headNode.prevNode = newNode;
        else
            tailNode = newNode;
        this.headNode = newNode;
        size++;
    }

    //insert at end of the list
    public void insertAtEnd(T data) {
        if (isEmpty()) { //if list is empty then insert at head
            insertAtHead(data);
            return;
        }
        //make a new node and assign it the value to be inserted
        Node newNode = new Node();
        newNode.data = data;
        newNode.nextNode = null; //it will be inserted at end so nextNode will be nu
        newNode.prevNode = tailNode; //newNode comes after tailNode so its prevNode
        tailNode.nextNode = newNode; //make newNode the nextNode of tailNode
        tailNode = newNode; //update tailNode to be the newNode
        size++;
    }
```

```
    }

    //print list function
    public void printList() {
        if (isEmpty()) {
            System.out.println("List is Empty!");
            return;
        }

        Node temp = headNode;
        System.out.print("List : null <- ");

        while (temp.nextNode != null) {
            System.out.print(temp.data.toString() + " <-> ");
            temp = temp.nextNode;
        }

        System.out.println(temp.data.toString() + " -> null");
    }
}
```

insertAtHead() and insertAtEnd()

Impact on Deletion

1) **`deleteAtHead()`**

Deletion at head remains almost the same as in DLL *without* tail. The only difference is that if the element to be deleted is the only element in the linked list then, we have to update the `tailNode` as `null` after deletion.

2) **`deleteAtTail()`**

Deletion at the tail (i.e, *the end*) is a *linear* operation in DLL *without* tail. However, in DLL *with* tail, it becomes a **constant** operation. We can use the same approach as in deletion at the start. Firstly, we access the last element of the list by the `tailNode`. Then we make the `prevNode` of `tailNode` equal to new `tailNode`. If the element being deleted was the only element in the list, that means we also have to assign `headNode` to the `null` value.

Let us take a look at the code for these operations.

main.java

DoublyLinkedList.java

DoublyLinkedList.java

```java
public class DoublyLinkedList<T> {

    //Node inner class for DLL
    public class Node {
        public T data;
        public Node nextNode;
        public Node prevNode;
    }

    //member variables
    public Node headNode;
    public Node tailNode;
    public int size;

    //constructor
    public DoublyLinkedList() {
        this.headNode = null;
        this.tailNode = null; //null initially
        this.size = 0;
    }

    //returns true if list is empty
    public boolean isEmpty() {
        if (headNode == null && tailNode == null) //checking tailNode to make sure
            return true;
        return false;
    }

    //getter for headNode
    public Node getHeadNode() {
        return headNode;
    }

    //getter for tailNode
    public Node getTailNode() {
        return tailNode;
    }

    //getter for size
    public int getSize() {
        return size;
    }

    //insert at start of the list
    public void insertAtHead(T data) {
        Node newNode = new Node();
        newNode.data = data;
        newNode.nextNode = this.headNode; //Linking newNode to head's nextNode
        newNode.prevNode = null; //it will be inserted at start so prevNode will be
        if (!isEmpty())
            headNode.prevNode = newNode;
        else
            tailNode = newNode;
        this.headNode = newNode;
        size++;
```

```
    }

    //insert at end of the list
    public void insertAtEnd(T data) {
        if (isEmpty()) { //if list is empty then insert at head
            insertAtHead(data);
            return;
        }
        //make a new node and assign it the value to be inserted
        Node newNode = new Node();
        newNode.data = data;
        newNode.nextNode = null; //it will be inserted at end so nextNode will be nul
        newNode.prevNode = tailNode; //newNode comes after tailNode so its prevNode w
        tailNode.nextNode = newNode; //make newNode the nextNode of tailNode
        tailNode = newNode; //update tailNode to be the newNode
        size++;
    }

    public void deleteAtHead() {
        if (isEmpty())
            return;

        headNode = headNode.nextNode;
        if(headNode == null)
            tailNode = null;
        else
            headNode.prevNode = null;
        size--;
    }

    public void deleteAtTail() {
        if (isEmpty())
            return;
        tailNode = tailNode.prevNode;
        if (tailNode == null)
            headNode = null;
        else
            tailNode.nextNode = null;
        size--;
    }

    //print list function
    public void printList() {
        if (isEmpty()) {
            System.out.println("List is Empty!");
            return;
        }

        Node temp = headNode;
        System.out.print("List : null <- ");

        while (temp.nextNode != null) {
            System.out.print(temp.data.toString() + " <-> ");
            temp = temp.nextNode;
        }

        System.out.println(temp.data.toString() + " -> null");
```

```
        }
    }
```

deleteAtHead() and deleteAtTail()

In this lesson, we covered another variation of Linked List known as “Linked List with Tail”. We also studied the implementation of DLL with a tail.

Let’s solve some more challenges related to Linked Lists in the upcoming lessons.