



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
 - Comparison between SLL with Tail and DLL with Tail
- Implementation of Doubly Linked List with Tail
 - Impact on Insertion
 - 1) insertAtHead()
 - 2) insertAtEnd()
 - Impact on Deletion
 - 1) deleteAtHead()
 - 2) deleteAtTail()

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**.

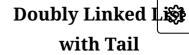
Comparison between SLL with Tail and DLL with Tail

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





<pre>insertAtTail() Or insertAtEnd()</pre>	Takes constant time i.e, 0(1)	Takes constant time i.e
<pre>deleteAtTail() Or deleteAtEnd()</pre>	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, 0(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.

```
31
             return headNode;
        }
32
33
34
        //getter for tailNode
        public Node getTailNode() {
35
             return tailNode;
36
37
        }
38
39
        //getter for size
        public int getSize() {
40
41
             return size;
42
        }
43
44
        //print list function
        public void printList() {
45
             if (isEmpty()) {
46
                 System.out.println("List is Empty!");
47
48
                 return;
49
             }
```

```
50
51
            Node temp = headNode;
             System.out.print("List : null <- ");</pre>
52
53
54
            while (temp.nextNode != null) {
55
                 System.out.print(temp.data.toString() + " <-> ");
56
                 temp = temp.nextNode;
57
             }
58
59
            System.out.println(temp.data.toString() + " -> null");
60
        }
61
```

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> {
 1
 2
 3
         //Node inner class for DLL
 4
         public class Node {
 5
             public T data;
             public Node nextNode;
 6
 7
             public Node prevNode;
 8
         }
 9
10
         //member variables
         public Node headNode;
11
         public Node tailNode;
12
13
         public int size;
14
         //-----
```

```
ΤD
        //constructor
        public DoublyLinkedList() {
16
17
            this.headNode = null;
            this.tailNode = null; //null initially
18
19
            this.size = 0;
20
        }
21
        //returns true if list is empty
22
        public boolean isEmpty() {
23
24
            if (headNode == null && tailNode == null) //checking tailNode to make sure
25
                 return true;
26
            return false;
27
        }
28
29
        //getter for headNode
30
        public Node getHeadNode() {
31
            return headNode;
32
        }
33
34
        //getter for tailNode
35
        public Node getTailNode() {
            return tailNode;
36
37
        }
38
39
        //getter for size
        public int getSize() {
40
41
             return size;
42
        }
43
44
        //insert at start of the list
45
        public void insertAtHead(T data) {
46
            Node newNode = new Node();
47
            newNode.data = data;
48
            newNode.nextNode = this.headNode; //Linking newNode to head's nextNode
49
            newNode.prevNode = null; //it will be inserted at start so prevNode will be
50
            if (!isEmpty())
                headNode.prevNode = newNode;
51
52
            else
53
                tailNode = newNode;
            this.headNode = newNode;
54
55
            size++;
56
        }
57
        //insert at end of the list
58
59
        public void insertAtEnd(T data) {
            if (isEmpty()) { //if list is empty then insert at head
60
61
                 insertAtHead(data);
62
                 return;
            }
63
            //make a new node and assign it the value to be inserted
64
            Node newNode = new Node();
65
            newNode.data = data;
66
            newNode.nextNode = null; //it will be inserted at end so nextNode will be nul
67
            newNode.prevNode = tailNode; //newNode comes after tailNode so its prevNode √
68
            tailNode.nextNode = newNode; //make newNode the nextNode of tailNode
69
70
            tailNode = newNode; //update tailNode to be the newNode
71
            size++;
```

```
}
73
74
         //print list function
75
         public void printList() {
76
             if (isEmpty()) {
77
                  System.out.println("List is Empty!");
78
79
             }
80
81
             Node temp = headNode;
82
             System.out.print("List : null <- ");</pre>
83
84
             while (temp.nextNode != null) {
                  System.out.print(temp.data.toString() + " <-> ");
85
                  temp = temp.nextNode;
86
87
             }
88
             System.out.println(temp.data.toString() + " -> null");
89
90
         }
91
    }
\triangleright
```

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.

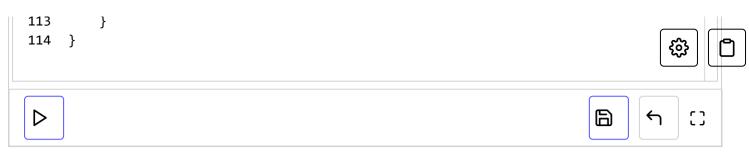
Let us take a look at the code for these operations.

```
(33)
```

```
· | | C
```

```
public class DoublyLinkedList<T> {
 1
 2
 3
        //Node inner class for DLL
 4
        public class Node {
 5
            public T data;
 6
            public Node nextNode;
 7
            public Node prevNode;
 8
        }
 9
        //member variables
10
11
        public Node headNode;
        public Node tailNode;
12
13
        public int size;
14
15
        //constructor
        public DoublyLinkedList() {
16
17
            this.headNode = null;
            this.tailNode = null; //null initially
18
19
            this.size = 0;
20
        }
21
22
        //returns true if list is empty
23
        public boolean isEmpty() {
24
            if (headNode == null && tailNode == null) //checking tailNode to make sure
25
                 return true;
26
            return false;
        }
27
28
29
        //getter for headNode
        public Node getHeadNode() {
30
31
            return headNode;
32
        }
33
34
        //getter for tailNode
35
        public Node getTailNode() {
            return tailNode;
36
37
        }
38
39
        //getter for size
40
        public int getSize() {
41
            return size;
42
        }
43
44
        //insert at start of the list
        public void insertAtHead(T data) {
45
46
            Node newNode = new Node();
47
            newNode.data = data;
            newNode.nextNode = this.headNode; //Linking newNode to head's nextNode
48
49
            newNode.prevNode = null; //it will be inserted at start so prevNode will be
50
            if (!isEmpty())
                 headNode.prevNode = newNode;
51
52
            else
53
                 tailNode = newNode;
54
            this.headNode = newNode;
55
            size++;
```

```
56
         }
 57
 58
         //insert at end of the list
 59
         public void insertAtEnd(T data) {
 60
             if (isEmpty()) { //if list is empty then insert at head
 61
                  insertAtHead(data);
                  return;
 62
 63
             }
             //make a new node and assign it the value to be inserted
64
65
             Node newNode = new Node();
 66
             newNode.data = data;
 67
             newNode.nextNode = null; //it will be inserted at end so nextNode will be nul
             newNode.prevNode = tailNode; //newNode comes after tailNode so its prevNode |
68
             tailNode.nextNode = newNode; //make newNode the nextNode of tailNode
 69
 70
             tailNode = newNode; //update tailNode to be the newNode
             size++;
71
         }
 72
 73
 74
         public void deleteAtHead() {
 75
             if (isEmpty())
 76
                  return;
 77
             headNode = headNode.nextNode;
 78
 79
             if(headNode == null)
 80
                  tailNode = null;
81
             else
82
                  headNode.prevNode = null;
83
             size--;
         }
 84
 85
 86
         public void deleteAtTail() {
 87
             if (isEmpty())
 88
                  return;
             tailNode = tailNode.prevNode;
 89
90
             if (tailNode == null)
91
                  headNode = null;
92
             else
93
                  tailNode.nextNode = null;
94
             size--;
95
         }
96
97
         //print list function
         public void printList() {
98
 99
             if (isEmpty()) {
100
                  System.out.println("List is Empty!");
101
                  return;
102
             }
103
104
             Node temp = headNode;
105
             System.out.print("List : null <- ");</pre>
106
107
             while (temp.nextNode != null) {
                  System.out.print(temp.data.toString() + " <-> ");
108
109
                  temp = temp.nextNode;
110
             }
111
             System.out.println(temp.data.toString() + " -> null");
112
```



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.

