

Linked List Variation : Doubly Linked List

In a standard linked list, you can only traverse the list in one direction. It may be useful however, to be able to go both forwards and backwards when traversing a linked list. In order to allow this, we need each Node to contain not one, but two references to Nodes - one to the next Node in the list and one to the previous Node in the list.

A double linked list can be implemented by editing code for a standard linked list. Consider the following DLLNode class:

```
// Arup Guha  
// 9/9/02  
// This class stores a node to be used in a doubly linked list.  
public class DLLNode {  
  
    public int data;  
    public DLLNode next;  
    public DLLNode prev;  
  
    public DLLNode (int x) {  
  
        data = x;  
        next = null;  
        prev = null;  
    }  
}
```

The only difference here compared to the Node class we used for a standard linked list is an extra reference prev. Also, consider how you would use inheritance to implement this class and a Doubly Linked List class.

Issues to consider for a DLL

Many of the issues with respect to coding a Doubly Linked List are exactly the same as dealing with a singly linked list. First and foremost, you need to make sure that you prevent Null Pointer Exceptions. You still end up iterating through the list forwards, and in fact, it's easiest to use the exact same method with the helper reference as the standard linked list class.

One difference is that in most cases, you end up having more "reference maintainance," since you have to maintain both forward and backwards references.

Let's develop the basics of what we need to do with each of the following situations:

1) Insertions

- a) Into an empty list**
- b) Into the front of a list**
- c) In the middle of a list**
- d) At the end of a list**

2) Deletions

- a) All situations where the value is not in the list**
- b) Deleting from the front of the list**
- c) Deleting from the middle of the list**
- d) Deleting from the end of the list**

```
// Arup Guha  
// 9/9/02  
// This class implements a doubly linked list, using a single  
// DLLNode object as an instance variable. The class supports  
// inserting and removing nodes. A sample test of the methods  
// is included in the main method, but this test is not  
// comprehensive.
```

```
import java.util.Random;
```

```
public class DLL {
```

```
    private DLLNode front;
```

```
// Initializes the DLL object to be empty.
```

```
    public DLL() {  
        front = null;  
    }
```

```
// Initializes the DLL object to reference the DLLNode passed  
// to the method.
```

```
    public DLL(DLLNode one) {  
        front = one;  
        front.prev = null;  
    }
```

```
    public boolean isEmpty() {  
        return (front == null);  
    }
```

```
    public void makeEmpty() {  
        front = null;  
    }
```

```

// Creates a node with the value x and inserts it into the DLL
// object.
public void insert(int x) {

    // System.out.println("insert "+x); // Good for debugging.

    // Create a node with the value x.
    DLLNode temp = new DLLNode(x);

    // Take care of the case where the DLL object is empty.
    if (front == null)
        front = temp;

    // Deal with the standard case.
    else {

        // Insertion into the front of the DLL object.
        if (front.data > x) {
            temp.next = front;
            front.prev = temp;
            front = temp;
        }

        else {

            // Set up helper reference to refer to the node that the
            // inserted node should be inserted after.
            DLLNode helper = front;
            while ((helper.next != null) && (helper.next.data < x))
                helper = helper.next;
        }
    }

```

```

// Insert into the end of the list.
if (helper.next == null) {
    temp.prev = helper;
    helper.next = temp;
}
// Insert into the middle of the list.
else {
    temp.next = helper.next;
    helper.next.prev = temp;
    temp.prev = helper;
    helper.next = temp;
}
}
}
}
}

```

```

// Removes a node storing the value x. If no such node exists,
// nothing is done and the method returns false. Otherwise,
// the node is removed and true is returned.
public boolean remove(int x) {

```

```

// System.out.println("remove "+x); // Good for debugging.

```

```

// Can't remove anything from an empty list.
if (front == null)
    return false;

```

```
// Remove the first element, if necessary.  
if (front.data == x) {  
  
    // Two cases: One node in the list, or more.  
    if (front.next == null)  
        front = null;  
    else {  
        front = front.next;  
        front.prev = null;  
    }  
    return true;  
}  
  
// Set up helper reference to refer to the node right before  
// the node to be deleted would be stored.  
DLLNode helper = front;  
while ((helper.next != null) && (helper.next.data < x))  
    helper = helper.next;  
  
// x is larger than the last node in the list.  
if (helper.next == null)  
    return false;
```

```
// x is contained in the list.  
if (helper.next.data == x) {  
  
    // x is stored in the last node of the list.  
    if (helper.next.next == null)  
        helper.next = null;  
  
    // at least one value is stored in the list after x.  
    else {  
        helper = helper.next;  
        helper.next.prev = helper.prev;  
        helper.prev.next = helper.next;  
    }  
    return true;  
}  
return false; // Case if x isn't in the list.  
}
```

```
// Prints out each item stored in the DLL object, in order.  
public void printlist() {  
  
    DLLNode temp = front;  
    while (temp != null) {  
        System.out.print(temp.data+" ");  
        temp = temp.next;  
    }  
    System.out.println();  
}
```

```
public static void main(String[] args) {  
  
    DLL list = new DLL();  
    Random ran = new Random();  
  
    // Randomly chooses to either insert or delete 30 elements  
    // from a newly created DLL object.  
    for (int i=0; i<30;i++) {  
        int temp = ran.nextInt()%8;  
        if (temp >= 0)  
            list.insert(temp);  
        else  
            list.remove(-temp);  
    }  
    list.printlist();  
  
}  
}
```

Linked List Variation: Circular Linked Lists

Although I am not going to show you any code for a circular linked list, the variations from the standard linked list are minor. The only difference between here and a standard linked list is that the next component of the last node in the list references the first node in the list. One of the main issues to keep track of here is making sure that you don't iterate through the list forever, (since no next reference in the entire list will ever be null.) And as always, Null Pointer Exceptions are important to watch out for.

Let's draw pictures to illustrate some of the major issues to deal with when considering implementing a circular linked list.

A quick note on running times

As you might imagine, none of these linked list variations gives you a significant advantage in time, (with respect to order notation.) Both searching and deleting are $O(n)$ operations in a standard linked list, and are also the same in both the doubly linked lists and the circularly linked lists. (Even when you can traverse backwards, in the best case scenario, you still have to go through 1/2 of the list, if you have references and both the front and end of the list.

Double Ended Queues

A double ended queue is a data structure that you can enqueue and dequeue from EITHER the front or the back. This abstract data type could be implemented with a doubly linked list. The class would contain two instance variables: one for the front of the DLL and one for the back. Here are the four standard operations the Double Ended Queue has to support:

- 1) insertFirst(e)**
- 2) insertLast(e)**
- 3) removeFirst(e)**
- 4) removeLast(e)**

A good exercise would be for you all to construct a DEQ class that contains two DLLNode instance variables and implement the four methods above.