Double Linked Lists (DLL) Rate Topic: ★★★★★ 1 Votes

karabasf
Posted 04 April 2012 - 03:45 PM

Level: Intermediate
Assumed knowledge: before you can start this tutorial, you're supposed to have knowledge of:

- Interfaces
- Basic Java operations
- Exceptions
- Generics
- Basic knowledge of Object Oriented Programming (Inheritance)

So what will we be doing in this tutorial? We're going to use a double linked list in order to implement a Queue. As Java does not comes with a Double Linked List, we will program our own. Therefore, this tutorial is split up in two parts, the first part (which is this tutorial) will cover the double linked list, while the second part will cover the Queue.

Let's start with the definition of a double linked list. Consider a linked list:

```
Head ---> Node ---> .... Node ----> Tail --->
NULL
```

A linked list is very powerful when we perform head operations: we add and/or remove the first element of the linked list. However, if we want to remove or add an element at the tail, this might be a problem, as it will take O(n) time in order to traverse all the elements and to delete the tail.

Now, consider the Double Linked List (DLL from this point on)

```
NULL <---> Head <---> Node <---> .... Node
<----> Tail <---> NULL
```

Now, we have a node defining the head and the tail (the begin and the end) of our list. And because they are double referenced (pointing back and forth) this means we can easily add or remove items from the head and the tail, without worrying too much about the performance.

But before we start, what are advantages of lists over arrays?
Although arrays are quite powerful, as it takes O(1) time to add, remove, or to get an element, we need to know the size of the array beforehand. We could increase the size of the array each time (it's definetly an option), but considering the flexibility of a DLL and the fact that I don't need to worry about the size... Well, it's certainly an advantage.

So let's get started. One important difference between a DLL and an array is that a DLL makes use of nodes. As Java does not have any predefined node, we have to program our own. Basic functions of a node are:

- getElement returns the element which the DNode is storing
- getNext The next DNode object (subsequent DNode)
- getPrev The previous DNode object
- setNext Method to set the next DNode
- setPrev Method to set the previous DNode
- toString A string representation of the DNode

Now we know what a DNode must do, we can start programming it. Lets start with the constructor. Notice that one constructor suffices, as we want to set the left and right DNode when we instantiate it and we also want to set the data which it should hold.

Java Tutorials

- Creating a Calculator using JFrame
- Getting Started with JavaFX 2
- Bound Properties in Java.
- Random Movie/File Selector
- Rock paper scissors with classes and methods
- Scenarios for Bit-Wise Operators
- How to Approach Swing for Beginners
- Priority Queues- Arrays
- Double Linked Lists (DLL)
- Queues- Arrays and Linked Lists
- 206 More Java Tutorials...

Reference Sheets

General Discussion

- Caffeine Lounge
- Corner Cubicle
- Student Campus
- Software Development
- Industry News
- Introduce Yourself
- Nightmare.In.Code

Programming Help

- C and C++
- VB.NET
- Java
- C#
- ASP.NET
- .NET Framework
- VB6
- PHP
- Ruby
- Python
- ColdFusion
- Databases
- Other Languages
- Game Development
- Mobile Development
- 52 Weeks Of Code

Web Development

- Web Development
- HTML & CSS
- JavaScript
- Graphic Design
- Flash & ActionScript
- Blogging
- SEO & Advertising
- Web Servers & Hosting
- Site Check

```java
public class DNode<E> {
    DNode<E> prevNode;        //Pointer to the previous DNode
    DNode<E> nextNode;        //Pointer to the next DNode
    E element;              //The element which the DNode should hold/store

    //Default constructor of the DNode to set a node element with a value and pointers to the next and previous node
    public DNode(E element, DNode<E> prevNode, DNode<E> nextNode){
        this.prevNode = prevNode;
        this.nextNode = nextNode;
        this.element = element;
    }
}
```

Now the constructor is defined, we can easily implement the other methods. If you know some basic OOP, this shouldn't be too hard for you. This results in:

```java
//Get the element which is stored in the DNode
public E getElement(){
    return this.element;
}

//Set the element in the DNode
public void setElement(E newElement){
    this.element = newElement;
}

//Get the previous DNode
public DNode<E> getPrev(){
    return prevNode;
}

//Set the previous DNode
public void setPrev(DNode<E> newPrev){
    this.prevNode = newPrev;
}

//Same methods for the next node.
//These are almost the same as the methods for the previous node, so try it out yourself
```

As a final step, add a toString() method. As I only care about the stored element in the DNode, I choose to display the stored element.

```java
//String representation of the DNode
//Customizable
public String toString(){
    if (element == null)
        return null;
    else
        return element.toString();
}
```

Now that we programmed our DNode, we can start with our DLL (You should know what it means by now). Just like the DNode, define the operations of the DLL, what should it do? Luckily, we know what it needs to do. The operations for this tutorial are:

- addFirst Add an element after the head of the DLL (recall that the head stores a null entry)
- addLast Add an element before the tail of the DLL (recall that the tail stores a null entry)
- removeFirst Remove the element after the head of the DLL

Java Snippets

- Fibonacci series
- Bubble Sort
- Insertion Sort
- Using FileWriter to write text to a file
- TCP/IP Client and Server
- Classic Tic Tac Toe
- Fibonacci Sequence
- allow only number input in JTextField
- DecimalFormat
- Drawing a shape
- 654 More Java Snippets...

DIC Chatroom

Join our IRC Chat

Bye Bye Ads

DIC++

Monthly Drawing

DEPARTMENT OF EDUCATION, TRAINING AND EMPLOY

Acceptable use polic

Restricted - Insti

Access to "http://www.f because it matches the "' restricted sites.

(optional for a DLL)
- removeLast Remove the element before the tail of the DLL (optional for a DLL)
- size The amount of elements in the DLL
- getFirst Get the first element after the head of the DLL
- getLast Get the element before the tail of the DLL
- getNext Get the next node from the reference node
- getPrev get the previous node from the reference node
- addBefore Add an element before an user specified position (which is a DNode)
- addAfter Add an element after an user specified position (which is a DNode)
- remove Remove an entry from the DLL
- toString A string representation of the DLL

Several methods and definitions exist for the aforementioned operations. Furthermore, a DLL is not only restricted to these operations, you might also define a hasNext(DNode<E> v), a hasPrev(DNode<E> v) or maybe even a toArray() method. We will not consider those operations for this tutorial, so the implementation of those is left to the reader.

So, now we get our operations, lets start with the constructor. We need three variables, one representing the head, one the tail and finally a variable to keep track of the size of the DLL. When we instantiate a DLL, it's empty, so this result in the following situation:

```
NULL <---> Head <---> Tail <---> NULL
```

The head is referring to a null element (as it does not have a previous node) and to the tail. On the other side, the tail is referred to the head and to a null element (as it does not have a next node)

This results in:

```
public class DoubleLinkedList<E>{
    private DNode<E> head;    //A DNode denoting the head of the DLL
    private DNode<E> tail;    //A DNode denoting the tail of the DLL
    private int size;       //The size of the DLL

    //Default constructor. Constructs an Empty DLL
    public DLList(){
        head = new DNode<E>(null, null, null);
        tail = new DNode<E>(null, null, null);
        size = 0;

        //Set the references
        head.setNext(tail);
        tail.setPrev(head);
    }
}
```

Before we dive straight in the "hardcore" part of the DLL, we need to define the getNext(DNode<E> refNode) and getPrev(DNode<E> refNode). The reason why we need this, is to check if the previous node and the nextNode is not null (which occurs when you use the header and the tail as a reference node). First, this is because it makes no sense to traverse the DLL further than the head and the tail. Second, we certainly don't want NullPointerExceptions when performing other operations, so this check is needed.

```
//Return the next DNode based on the refnode
private DNode<E> getNext(DNode<E> referenceNode) throws BoundaryViolationException{
    if(referenceNode == tail)
        throw new BoundaryViolationException("Reference node cannot be equal to the tail");

    return referenceNode.getNext();
}

//Return the previous DNode based on the refnode
private DNode<E> getPrev(DNode<E> referenceNode) throws BoundaryViolationException{
    if(referenceNode == head)
        throw new BoundaryViolationException("Reference node cannot be equal to the head");

    return referenceNode.getPrev();
}
```

Note that I used a self-defined exception called the BoundaryViolation exception. This extends the RuntimeException and looks like:

```
public class BoundaryViolationException  extends RuntimeException {
    public BoundaryViolationException (String message) {
        super (message);
    }
}
```

Now, lets consider two algorithms for the addBefore and the addAfter method

```
//This looks like:
    newNode
      |
      v
Node1<---->Node2<---> .... <--->

addBefore(DNode<E> Node2, E newElement)
    prevNode = Node2.getPrev        //represents Node1

    //Start setting the references
    DNode<E> newNode = new DNode<E>(newElement, prevNode, Node2);       //The previous node of the new node is Node1, while the next node is Node2
    prevNode.setNext(newNode);      //Node1 should now refer to the newNode instead of Node2
    Node2.setPrev(newNode);         //Node2 should now refer to the newNode instead of Node1
    size++;
```

```
//This looks like:
    newNode
      |
      v
Node1<---->Node2<---> .... <--->

addAfter(DNode<E> Node1, E newElement)
    nextNode = Node1.getNext        //represents Node2

    //Start setting the references
    DNode<E> newNode = new DNode<E>(newElement, Node1, nextNode);       //The previous node of the new node is Node1, while the next node is Node2
    Node1.setNext(newNode);         //Node1 should now refer to the newNode instead of Node2
    nextNode.setPrev(newNode);      //Node2 should now refer to the newNode instead of Node1
    size++;
```

Now put this in Java code:

```
//Add an element before the refnode
private void addBefore(DNode<E> referenceNode, E element){
    DNode<E> prevNode = this.getPrev(referenceNode);

    //Start setting the reference
    DNode<E> newNode = new DNode<E>(element, prevNode, referenceNode);
    referenceNode.setPrev(newNode);
    prevNode.setNext(newNode);
    size++;
}

//Add an element after the refnode
private void addAfter(DNode<E> referenceNode, E element){
    DNode<E> nextNode = this.getNext(referenceNode);

    //Start setting the reference
    DNode<E> newNode = new DNode<E>(element, referenceNode, nextNode);
    referenceNode.setNext(newNode);
    nextNode.setPrev(newNode);
    size++;
}
```

Note that due to our specification of the DLList, we cannot add an element on basis of a DNode... We have to retrieve a DNode from the DLList before we can call those methods from the main. So, why bother putting these in this tutorial? These two operations are for illustrative purposes. For example, you could use these methods to add an element at an user specified position, which is based on a value or an (integer) position in the DLList. However, this is outside the scope of this tutorial, so the reader is free to implement his own method.
Also note that for this implementation of the DLL I made these methods private: they're only meant for internal use (that is within the class of DLL).

So, what if we want to add at the front of the DLL and at the rear? That's quite easy. If we use the addBefore method to add at the rear and addBefore at the front, we're getting there. This is because we can feed these methods with our sentinel DNodes!

```
//Add an element at the front of the DLL
public void addFirst(E element){
    //Call the addAfter method to add an element after the head
    this.addAfter(head, element);
}

//Add an element at the rear of the DLL
public void addLast(E element){
    //Call the addBefore method to add an element before the tail
    this.addBefore(tail, element);
}
```

Next to the add methods, we want to remove an element from the DLL. Consider this algoritm:

```
    removeNode
```

```
        ^
        |
Node1 <--->|      |<---> Node2

remove(removeNode)
    //Get the reference
    Node1 = removeNode.prev();
    Node2 = removeNode.next();

    //Start setting the reference
    removeNode.setNext(null);
    removeNode.setPrev(null);   //The removeNode
is dereferenced from the DLL
    Node1.setNext(Node2);
    Node2.setPrev(Node1);
```

In Java:

```
//Method to remove an element
private E remove(DNode<E> removeNode){
    DNode<E> prevNode =
this.getPrev(removeNode); //Node1
    DNode<E> nextNode =
this.getNext(removeNode); //Node2

    //Dereference the node which is removed
    removeNode.setPrev(null);
    removeNode.setNext(null);

    //Set the other references
    prevNode.setNext(nextNode);
    nextNode.setPrev(prevNode);

    size--;
    //return the stored element of the
removeNode
    return removeNode.getElement();
}
```

Using the same reasoning as for addFirst() and addLast(), we can use the remove method to removeLast() and removeFirst(). In Java, this looks like:

```
//Method to remove the last element of the DLL
public E removeLast() throws EmptyListException{
    //Check if the DLL list is empty
    if(this.isEmpty())
        throw new EmptyListException("Double
Linked list is empty, cannot remove element");

    //Else, get the Node before the tail
    DNode<E> tempNode = this.getPrev(tail);

    //Remove tempNode
    return this.remove(tempNode);
}

//Method to remove the first element of the DLL
public E removeFirst() throws
EmptyListException{
    //Check if the DLL list is empty
    if(this.isEmpty())
        throw new EmptyListException("Double
Linked list is empty, cannot remove element");

    //Else, get the Node after the head
    DNode<E> tempNode = this.getNext(head);

    //Remove tempNode
    return this.remove(tempNode);
}
```

Note that I defined a custom Exception called EmptyListException. This is basically an extension of RuntimeException. This looks like:

```
public class EmptyListException extends
RuntimeException {
    public EmptyListException (String message){
        super (message);
    }
}
```

Also, the isEmpty() is fairly easy to construct. When is something empty? Exactly, when it does not contains any elements (size == 0). I leave the implemetation of this method up to the reader.

As one of the final steps, by using these two methods getPrev and getNext, we can define two wrappers to get our first and the last element of the DLL (note, get, not remove). So, how do we do this? We simply pass our tail and head as an argument. In Java:

```
//Method to get the first element in the DLL
public E getFirst(){
    //Check if the DLL list is empty
    if(this.isEmpty())
        throw new EmptyListException("Double
Linked list is empty, cannot retrieve first
element");

    //Get the DNode after the head
    DNode<E> temp = this.getNext(head);

    //return the element it was storing
    return temp.getElement();
}

//Method to get the last element in the DLL
public E getLast(){
    //Check if the DLL list is empty
    if(this.isEmpty())
        throw new EmptyListException("Double
Linked list is empty, cannot retrieve last
element");

    //Get the DNode after the head
    DNode<E> temp = this.getPrev(tail);

    //return the element it was storing
    return temp.getElement();
}
```

The only thing which remains is the size() and the toString() method. The size() method is quite straightforward (which again, is a task for you), while the toString method requires some iteration. We only want to print the DNodes which are not equal to the head and the tail. Thus:

```
//A string representation of the DLL
public String toString(){
    String s = "[";

    DNode<E> probe = head.getNext();
    while(probe!= tail){
        s += probe + " ";
        probe = probe.getNext();
    }

    s += "]";

    return s;
}
```

Your final DLL should look something like this:

Spoiler Show

Note that although this implementation of the DLL contains a considerable amount of code, it's far from complete. An user cannot turn the DLL into an array, cannot manually traverse the elements of the DLL, and so on. However, this tutorial shows the basics of a DLL, if I were to show every possible method... Well, lets say that this would be a very long tutorial.
Another consideration is the DNode class. This is located outside the DLL, meaning that in the public staic void main(String[] args) a DNode object could be instantiated. This is not something you would like to have.

Therefore, although the DLL is quite limited at this time, you should now able to extend it further. This tutorial treated the following:

- How to make a DNode class, nodes for the DLL
- How to make a constructor for the DLL
- How to add and remove elements from the DLL

Now you get the basics of a DLL, you're now able (or at least a bit more aware) of how to program your own DLL.

I hope this tutorial was helpful for you ^^

References:
Data Structures & Algorithms in Java by Michael T. Goodrich and Roberto Tamassia

Page 1 of 1

Related Java Topics beta

- Double Linked Lists
- Queues- Arrays And Linked Lists Tutorial
- Doubts About Linked Lists!
- Clear A Double Linked List
- Linked Lists
- Java Data Structures Resource Thread
- LinkedList Iterator

Issue

Method To Meld Doubly Linked Lists

Priority Queues And Linked Lists

Multiplying Polynomials Using Linked Lists (solved)