I am happy that my introduction to data structures didn’t scare you away! The first structure that I will be covering is the Doubly Linked List which is one of the most used and simplest data structures that implements the list abstract data type.

**What is a doubly Linked list?**

I introduced the List interface when explaining type parameterization in the intro (e.g. the <T> generic type in the interface definition). As a recap, this is merely an object-oriented syntax that allows us to instantiate an object of any class implementing this interface with any type we choose, thus improving code reusability. If this isn’t making sense I have a more detailed description HERE.

interface List<T> {

void add(T x);

T remove(int i);

boolean remove(T x);

T get(int i);

boolean contains(T x);

int size();

default boolean isEmpty() {

return size() == 0;

}

}

A doubly linked list is a structure or class that implements all of the methods from the list interface. We call it a doubly linked list because we may traverse the list in both forward and reverse. We do this using a Node inner class. This Node inner class contains 3 member variables: a piece of data, and a pointer to two other Nodes, one being the next node in the list and the other being the previous node.

class Node {  
 T data;  
 Node next, prev;  
  
 Node(T data) {  
 this(data, null, null);  
 }  
  
 Node(T data, Node prev, Node next) {  
 this.data = data;  
 this.prev = prev;  
 this.next = next;  
 }  
}

A simple conceptualization of the doubly linked list is to think of it as a chain. When the list is empty we have a head Link or Node. Since the list is empty the head node doesn’t contain any data, and it doesn’t have any other Links/Nodes to point to. As we add Links/Nodes to the list we link them to both the next chain link as well as the head link forming a circular chain.

There are two key features to notice about this implementation of the doubly linked list. 1) This is a circularly linked list, meaning that the end of the list is linked to the front of the list. 2) The head node will never contain any data, it will always be null. We add information to the doubly linked list by adding nodes to the structure. The head node always contains null for data because this is how we recognize the head node.

**What are it’s operations?**

As I mentioned before we are implementing all of the methods from the List ADT. Some of these methods are pretty self-explanatory such as the size() and isEmpty() methods. There are some other methods that aren’t trivial, but do not need to be described in great detail. get(int i) and contains(T x) fall into this category.

For the get method if index provided is within the bounds of the list size (e.g. i >= 0 and I < size()) the method follows the next Node pointers to the Node at the i-th index. If the index i is out of bounds it throws an exception. The remove function that takes an integer index as a parameter works very similar to this get method.

public T get(int i) {  
 if (i < 0 || i >= size())  
 throw new IndexOutOfBoundsException();  
 Node p = head.next;  
 while (i > 0) {  
 p = p.next;  
 i--;  
 }  
 return p.data;  
}

Similarly, the contains method takes a piece of data and follows the next Node pointers through the list to see if it finds a piece of data in the list that matches the input. If it finds one it returns true, if we reach the head node Node know we have made it to the end of the list and we return false.

public boolean contains(T x) {  
 Node p = head.next;  
 while (p != head) {  
 if (p.data.equals(x))  
 return true;  
 p = p.next;  
 }  
 return false;  
}

Note: For those of you who are not familiar with pointers they are simply a reference to another object. They “point” if you will, to another object of that type. In our case the prev and next node pointers point to the node before the current node or the node succeeding the current node.

Let’s move on to methods that modify the list, because we have to continually update pointers for these operations we will go into them in more detail. The last thing we want is a Nullpointer exception, or even worse in c/c++ the dreaded Segmentation Fault!

When adding an element to the list we must first create a new Node with the given data. Then we need to point its pointers to the appropriate nodes. This new Node will point it’s next pointer to the head node since it is the new end of the list, and it will point it’s previous pointer to the current last node in the list (e.g. the head.prev node). Finally, we have to update the head pointers to account for the new node. To do this we set the head pointer to the last Node head.prev equal to the current previous node’s next element. Which we have already updated to the new last node. Don’t worry about the modCount variable right now, that will be addressed in my Iterators tutorial coming soon! This is all done in the last line of the add function. I know this sounds confusing so don’t be discouraged, I’ve created some diagrams for you to look at below!

public void add(T x) {  
 modCount++;  
 n++;  
 Node p = new Node(x, head.prev, head);  
 head.prev = head.prev.next = p;  
}

When removing a specific element from the list we search for a node who’s data matches the input until one of two conditions are met. Either we have made a mistake and we have null pointers, hence the pointer p == null. Or we have reached the head node again and the p.data == null. If either of these conditions are met before we find a matching item we return false and don’t remove a node because we didn’t find a node to remove. If we find a matching node then we need to update the pointers and decrease the member variable denoting the size of the list. To update the pointers to remove the node we must point the next node in succession’s previous pointer (p.next.prev) to the previous node (p.prev), and point the previous node’s next pointer (p.prev.next) to the next node in succession (p.next).

public boolean remove(T x) {  
 //assert x != null;  
  
 Node p = head.next;  
  
 // p.data == null as break condition bc we still add a block with null data  
 while(p != null && p.data != null) {  
 if(p.data.equals(x)) {  
 //remove  
 p.next.prev = p.prev;  
 p.prev.next = p.next;  
  
 n--;  
 modCount++;  
  
  
 return true;  
 }  
 p = p.next;  
 }  
 return false;  
}

**What are the advantages and disadvantages?**

**Analysis:**

Operation Array dll

Add O(N) O(1)

Remove O(N) O(N)

Remove - O(1)

given node

get O(1) O(N)

contains O(N) O(N)