Lecture 8: Doubly Linked Lists

Sam McCauley
Data Structures, Spring 2026

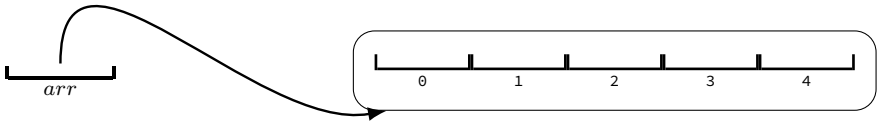
More on Memory and References

Let's discuss some important takeaways from memory and references.

Arrays and Strings. Arrays are stored like objects: the variable stores their address, rather than storing the values explicitly.

This is why arrays use the new keyword. The new keyword sets up the actual array slots; the array variable itself is just a reference.

```java
int[] arr = new int[5];
```



Strings in Java are handled unusually: long story short, you should assume that they also store a reference. However, they do not use the new keyword, and their back-end behavior can be more complicated than what we've described. As we'll mention below, this is particularly important when comparing `Strings`: you should use `.equals()` rather than == for `Strings` in Java.

`this` Keyword. In Java, the `this` keyword gives a reference to the current object.

Now that we have a better understanding of how things are stored in Java, let's revisit the following example.

```java
public class Student{
    String name;
    int graduationYear;

    public Student(String name, int graduationYear) {
        //this.name is instance variable; name is parameter
        this.name = name;
        this.graduationYear = graduationYear;
    }
}
```

When we write `this.name`, what we mean is: "follow the reference for `this`, and find the data for `name` in that `Student` object". That is, you can view `this.name` as giving "directions" to the instance variable version of `name`.

We also used `this` in Lab 2. Our `onePlayerGame` method in the `CoinStrip` class called `ai.findBestMove(this);`. Looking at `OptimalAI.java`, we can see a method `findBestMove(CoinStrip game)`. So when we called `ai.findBestMove(this)` from within our `CoinStrip` method, we passed the current game—the object whose code we are currently running—as the argument to this method.

Multiple References to the Same Object. References are just like "giving directions"—so it's fine for multiple references to point to the same object. This can be powerful, but can also lead to problems if done accidentally. Let's look at some examples of how it works so that we're familiar in case it comes up.

We'll use a `Student` to practice.

```
1 Student s1 = new Student("Freida",2005);
2 Student s2 = s1;  //s2 and s1 point to the same place
3 System.out.println(s2.getGraduationYear()); //prints 2005
4 System.out.println(s1.getGraduationYear()); //prints 2005
5 s2.setGraduationYear(1997);
6 System.out.println(s1.getGraduationYear()); //prints 1997
7 System.out.println(s2.getGraduationYear()); //prints 1997
```

If two variables reference the same data, then any operation on one will be an operation on the other.

It works differently when we have two different objects.

```
1 Student s1 = new Student("Victoria", 2005);
2 Student s2 = new Student("Victoria", 2005);  //s2 points to a new
     student with the same data
3 System.out.println(s2.getGraduationYear()); //prints 2005
4 System.out.println(s1.getGraduationYear()); //prints 2005
5 s2.setGraduationYear(1997);
6 System.out.println(s1.getGraduationYear()); //prints 2005
7 System.out.println(s2.getGraduationYear()); //prints 1997
```

Testing Equality of Objects and .equals() method. When you test equality of two objects, you usually do not want to use `==`. This only tests if the *references* are the same—in other words, it tests if they point to the same object. Usually, you do not care about references; you care about the data stored inside the object.

Next lecture, we will learn about a method called `.equals()` which is meant to compare the actual value of objects. You should use this method to compare any objects.

It is particularly important to remember to use `.equals()` for `Strings` because == will *usually*, but not always, work.

toString() method. In Java, there is a special method, `public String toString()`. If you implement this method, then it will automatically be called when an object is cast to a `String`.

Most notably, this method is called when we call `System.out.println()`. For an example, see the code for `LinkedListInt.java` and `DoublyLinkedListInt.java`.

Doubly Linked List

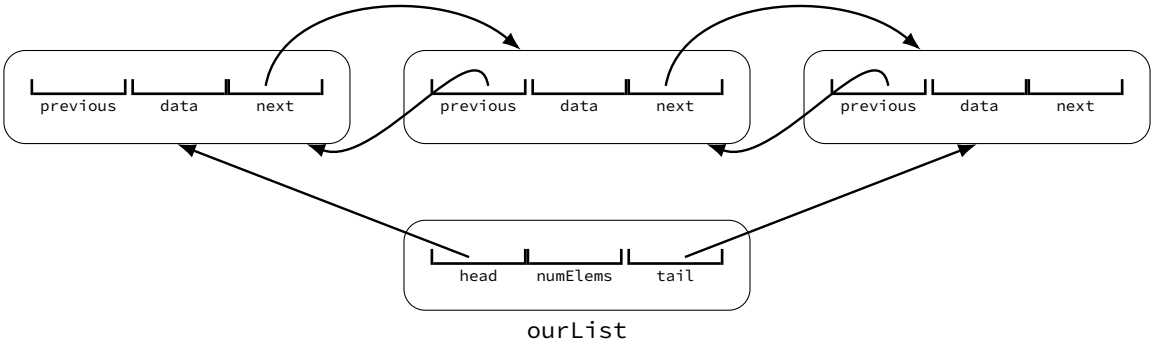
Let's look at one kind of data structure: the Doubly Linked List. The `LinkedListInt` class we went over recently is a "Singly Linked List."

Let's briefly motivate this data structure before we start; we'll discuss in more detail momentarily. There was a subtle inefficiency in the `LinkedListInt` class: it's impossible to go *backwards*. If we are in the 107th node of the list, and we want to find the 106th, the only option is to start from the `head` node and call `getNext()` 106 times.

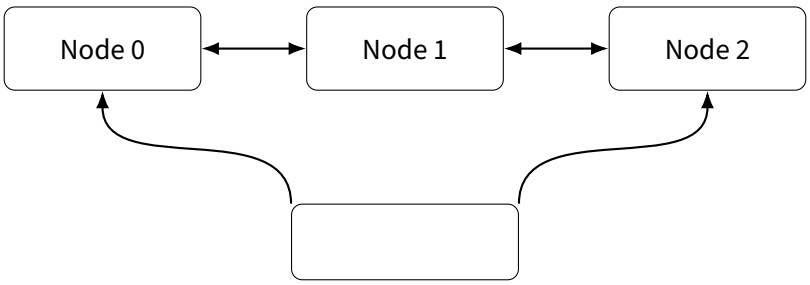
This means that adding to the end of the list (or near the end of the list) is very slow; removing elements near the end of the list is slow as well.

In the Doubly Linked List, we'll store references in both directions. Each node keeps track of both the next node, and the previous node. We'll have to do some extra bookkeeping to keep track of both of these nodes, but in exchange, some operations (most notably adding to the end of the list) will be considerably faster.

Doubly Linked Lists are perhaps more common than Singly Linked Lists (though both are used extensively). For example, the `LinkedList` library in Java uses a Doubly Linked List.



This diagram is getting a little bit busy. Oftentimes, doubly linked lists are drawn without the instance variables explicitly listed; something like this.



Changes to the Node. We need a new class to store our nodes that can store both the previous and next node. We'll call this class `DLLNodeInt` (the DLL is short for "Doubly Linked List").

The instance variables of the new class will look like this. The methods are skipped for space. In short, the class will start with all methods of the `NodeInt` class; we'll also add a getter called `getPrevious()` and a setter called `setPrevious()`.

```
1 public class DLLNodeInt {
2     private int data;
3     private DLLNodeInt next;
4     private DLLNodeInt previous;
```

Changes to the Doubly Linked List Class. In the `LinkedListInt` class we stored a reference to the first node of the list, which we called `head`.

In a Doubly Linked List, we will store two pointers, to the first and last node of the list. We'll call them `head` and `tail`.

```
1 public class DoublyLinkedListInt {
2     private DLLNodeInt head;
3     private DLLNodeInt tail;
4     private int numElems;
```

Note that if there is only one node in the linked list, `head` and `tail` are the same node (since the first and last node of the linked list is the same).

As in `LinkedListInt`, the `previous` node for `head` is `null`; likewise, the `next` node of `tail` is `null`.

DoublyLinkedListInt Methods. Many of the methods of `DoublyLinkedListInt` are the same as `LinkedListInt` or very similar. Let's highlight some of the main differences.

To add an element to the end of the list `add(int newElement)`, we have some good news: we do not need to traverse the list to find the last node. We have a `tail` pointer directly to it.

Then, we need to modify the pointers. If the list already has at least 1 element, then `tail` is not `null`, so we need to make its `next` value reference our new node. If the list has no elements, our new node is both the head and tail of the list.

```
1  // add newElement to the end of the list
2  public void add(int newElement) {
3      DLLNodeInt newNode = new DLLNodeInt(newElement);
4      newNode.setPrevious(tail);
5      if(numElems > 0) {
6          tail.setNext(newNode);
7      } else {
8          head = newNode;
9      }
10     tail = newNode;
11     numElems++;
12 }
```

Perhaps the most significant difference is the `getNode(int index)` method. This is a private method that finds the node at position `index` of the list. For our Singly Linked List, `getNode` always traversed the list from the beginning. For us, we either traverse from the beginning or from the end; whichever the node we're searching for is closer to.

```
1  //returns the node at index in the list
2  //assumes that index satisfies 0 <= index < numElems
3  private DLLNodeInt getNode(int index) {
4      if(index >= size()/2) {
5          DLLNodeInt current = tail; //node size() - 1
6          for (int skips = size() - 1; skips > index; skips--) {
7              // current is now at node "skips" in the list
8              current = current.getPrevious();
9          }
10         return current;
11     }
12     else {
13         DLLNodeInt current = head;
14         for (int skips = 0; skips < index; skips++) {
15             // current is now at node "skips" in the list
16             current = current.getNext();
17         }
18         return current;
19     }
20 }
```

Comparing Efficiency

Let's compare the tradeoffs of ArrayListInt and LinkedListInt in more detail.

What We Mean By Efficiency. We'll learn notation to talk about efficiency rigorously soon. For now, we are just looking at broad strokes.

In particular, you should focus on the following question: does a given method *traverse the entire data structure*? If one method needs to potentially traverse millions of elements, and another doesn't, the one that avoids this expensive traversal is likely much faster.

Let's look at some examples of efficiency for methods we've already seen.

get() and set() When we call `get()` and `set()` on an ArrayListInt, we can access a given slot in the array directly. No traversal is required.

When we call `get()` and `set()` on a LinkedListInt, we need to start with the `head` node, go to the next node, then the next, and so on. If we call `get(size() - 1)`, we need to traverse *every* node in the linked list.

When we call `get()` and `set()` on a DoublyLinkedListInt, the situation is largely the same: we start with the `head` or `tail`, and traverse the list until reaching the desired index. We may need to traverse up to half the nodes in the doubly linked list

For this reason, `get()` and `set()` are much faster in an ArrayListInt than a LinkedListInt or a DoublyLinkedListInt if the number of elements in the list is large.

add() When we call `add()` on an ArrayListInt, we first ensure that the capacity is large enough. This may involve traversing the entire array, copying all elements into a larger array. This can be slow, but the good news is we don't need to do this very often.

When we call `add()` on a LinkedListInt, we first use getNode() to find the last node in the linked list. This necessitates traversing the entire data structure.

When we call `add()` on a DoublyLinkedListInt, we use getNode() to find the last node in the linked list. This does not require traversing the list: the method returns the `tail` node right away. The rest of the `add()` method is just setting references.

Let's compare these. `LinkedListInt` needs to traverse the whole list every time. `ArrayListInt` needs to traverse the list sometimes, but not always—it's often faster, but not always. `DoublyLinkedListInt` never needs to traverse the list; it's the best of the three.