Last class I discussed how an array can be used to represent a list, and how various list operations can be implemented using arrays. Today I will finish up that discussion, and then move on to a different implementation of lists.

**Java ArrayList<T> class, where T is a generic type**

Java has an `ArrayList` class that implements the various List methods that I mentioned earlier, along with other methods. This class uses an array as its underlying data structure. The methods of this class enforce that the array is represented a list, namely if there are `size` elements in the array, then these are at slots 0, \ldots, `size` – 1. You should check out what these methods are for the `ArrayList` class in the Java API:

https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

Whenever you construct an `ArrayList` object, you need to specify the type of the elements in the array. You can think of this as a parameter that you pass to the constructor. In Java, the syntax for specifying the type uses `< >` brackets. For example, if we want to declare an `ArrayList` of objects that are of type `Shape`, we use the following syntax:

```
ArrayList<Shape> shapes = new ArrayList<Shape>();
```

If you look at the Java API, you’ll see that the class is `ArrayList<E>` where `E` denotes a generic type. We will see many examples of generic types in this course.

Although the `ArrayList` class implements a list using an underlying array, you do not index the elements of the array using the `a[ ]` syntax. Instead you access an array element using a `get` or `set` or other methods. There is an array `a[ ]` (or some other name) ”under the hood” but as the client of this class you don’t have access to it. You can only access the array with the getter or setter or one of the other methods.

Because the `ArrayList` class uses an array as its underlying data structure, if you add or remove an elements to/from the front of your list, this operation will take time proportional to `size` (the number of elements in the list) since all the other elements needs to shift position in the array by 1. This can be slow. Thus, although arrays allow you to get and set values very quickly (as I discussed last class), they can be slow for adding and removing because of this shifting property. Let’s look at another list data structure, called linked lists, which partially avoid this problem.

**Singly Linked lists**

Define a list node to be an object which contains:

- an element of a list (or a reference to an element)
- a reference to the next node in the list.

In Java, we can define a node class as follows:

```java
class SNode<T>{
    T element;
    SNode<T> next;
}
```

last updated: 27th Sept, 2016

1

lecture notes ©Michael Langer
where \( T \) is the generic type of the object in the list, e.g. \texttt{Shape} or \texttt{Student} or some predefined Java class like \texttt{Integer}.

To construct a "linked list" of these nodes, it is common to define a first node and a last node. We do so by defining variables \texttt{head} and \texttt{tail}:

```java
SNode<T> head;
SNode<T> tail;
}
```

which reference the first and last node in a linked list, respectively. If there is only one node in the list, then \texttt{head} and \texttt{tail} would point to the same node.

The big win for linked lists over array lists is that linked lists allow you to add an element or remove an element at the front of the list in a constant amount of time (whereas array lists require \( O(N) \) operations to add or remove at the front of a list of \( N \) elements). Let’s look the basic algorithm. Here I will use pseudocode, rather than Java.

\textbf{addFirst( e ) and removeFirst()}

We begin with an algorithm for adding an element \( e \) at the front of a list.

```java
addFirst( e ){
    construct new node
    newNode.element = e
    if (head == null){ // list is empty
        head = newNode
        tail = head
    }
    else{
        newNode.next = head
        head = newNode
    }
    size++ // if we have a size variable too
}
```

Notice that the order of the two instructions in the 'else' block matters! Also notice that we have considered the case that the list is empty. This special case ("edge case") will arise sometimes. Whenever you write methods, ask yourself what are the edge cases and make sure you test for them! I may omit the edges cases, sometimes intentionally, sometimes unintentionally. Don’t hesitate to ask if notice one is missing.
Next, here is an algorithm for removing an element at the front of the list.

```java
removeFirst(){
// test for edge cases omitted (empty list or list of size 1)
    tmp = head
    head = head.next
    tmp.next = null
    size = size - 1
    return tmp.element
}
```

Notice how we have used `tmp` here. We could have just had one instruction (`head = head.next`) but then the old first node in the list would still be pointing to the new first node in the list, even though the old first node isn’t part of the list. (This might not be a problem. But it isn’t clean, so I haven’t allowed it.)

Also note that the `removeFirst()` method ignores certain cases. For example, if there is only one element in the list, then removing the first means that we are also removing the last. In that case, we should set the `tail` reference to `null`. These are all details that are important in the real implementation.

**addLast( e ) and removeLast()**

Let’s now discuss two more methods that are commonly defined for linked lists, namely adding or removing an element at the back of the list. This requires manipulating the `tail` reference. Adding a node at the tail can be done in a small number of steps.

```java
addLast( e ){
    construct newNode
    newNode.element = e
    tail.next = newNode
    tail = tail.next
    size = size + 1  // optional
}
```

Removing the last node from a list is more complicated, however. It requires many steps. The reason is that you need to modify the `next` reference of the node that comes before the tail node which you want to remove. But you have no way to directly access the node that comes before `tail`, and so you have to find this node by searching from the front of the list.

The algorithm begins by checking if the list has just one element. If it does, then the last node is the first node and this element is removed. (I do not return the element below. That code could be added.) Otherwise, it scans the list for the element that comes before the last element.
removeLast()
{
    if (head == tail)
        head = null
        tail = null
    }
    else{
        tmp = head
        while (tmp.next != tail){
            tmp = tmp.next
        }
        tmp.next = null
        tail = tmp
    }
    size = size-1  // optional variable
}

This method requires about size steps. This is significantly more expensive than what we had with an array implementation, where we had a constant cost in removing the last element from a list. We will come back to this comparison at the end of the lecture, when we compare arrays with singly and doubly linked lists.

**Time Complexity**

The table below compares the time complexity for adding/removing an element from the head/tail of an array or linked list that has size N.

<table>
<thead>
<tr>
<th></th>
<th>array list</th>
<th>singly linked list</th>
</tr>
</thead>
<tbody>
<tr>
<td>addFirst( e )</td>
<td>O(N)</td>
<td>O(1)</td>
</tr>
<tr>
<td>removeFirst()</td>
<td>O(N)</td>
<td>O(1)</td>
</tr>
<tr>
<td>addLast( e )</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>removeLast()</td>
<td>O(1)</td>
<td>O(N)</td>
</tr>
</tbody>
</table>

The main problem with the singly linked list is that it is slow to remove the last element. We will around this next lecture, when we discuss the doubly linked list. The basic idea there is to add a prev reference to each node. This will allow us to quickly index the element that comes before the tail, just by asking for tail.prev.

**Singly Linked list class**

So far I have defined nodes for the linked list, and variables that point to the head and tail. We need to bundle these into a class:
class SLinkedList<T>{
    SNode<T> head;
    SNode<T> tail;
    int size;
}

As I discuss in the slides, suppose we have a linked list with four nodes. How many objects do we have in total? We have the four SNode objects. We have the four elements (objects of type Shape, say) that are referenced by these nodes. We also have the SLinkedList object which has the head and tail references. So this is 9 objects in total.

That’s it for singly linked lists! Next class we’ll do doubly linked lists.