Lecture 7: Doubly Linked Lists and Unit Testing

Exercises

Here we have included the `SinglyLinkedList` from last time to help you with the following questions.

Lecture6/SinglyLinkedList.java

```java
import java.util.NoSuchElementException;

public class SinglyLinkedList<T> {

    // Static inner class
    public static class Node<S> {
        private Node<S> next;
        private S value;
        public Node(Node<S> n, S v) {
            next = n;
            value = v;
        }
        public S getValue() { return value; }
        public Node<S> getNext() { return next; }
    }

    private Node<T> first; // Sometimes called head
    private Node<T> last; // Sometimes called tail
    private int size;

    public SinglyLinkedList() {}  
    public boolean isEmpty() { return first == null; }  
    public int size() { return size; }  
    public Node<T> getFirst() { return first; }  
    public Node<T> getLast() { return last; }  
    private Node<T> addToEmpty(T t) {
        Node<T> node = new Node<T>(null, t);
        first = node;
        last = node;
        size++;
        return node;
    }
    public Node<T> addLast(T t) {
        if (isEmpty()) return addToEmpty(t);
        Node<T> node = new Node<T>(null, t);
        last.next = node;
        last = node;
    }
}
```
last = node;
size++;
return node;
}

public Node<T> addFirst(T t) {
    if (isEmpty()) return addToEmpty(t);
    Node<T> node = new Node<T>(first,t);
    first = node;
    size++;
    return node;
}

public T removeFirst() {
    if (isEmpty()) throw new NoSuchElementException();
    Node<T> node = first;
    if (first == last) last = null; //Case when size = 1
    first = first.next;
    size--;
    node.next = null;
    return node.value;
}

public Node<T> getNode(int i) {
    if (i < 0 || i >= size())
        throw new IndexOutOfBoundsException();
    Node<T> curr = first;
    for (int j = 0; j < i; ++j) curr = curr.next;
    return curr;
}

public Node<T> addAfter(Node<T> listNode, T v) {
    boolean updateLast = listNode == last;
    Node<T> newNode = new Node<T>(listNode.next,v);
    listNode.next = newNode;
    if (updateLast) last = newNode;
    size++;
    return newNode;
}

public T removeAfter(Node<T> listNode) {
    if (listNode.next == null)
        throw new NoSuchElementException();
    Node<T> node = listNode.next;
    boolean updateLast = node == last;
    listNode.next = node.next;
    if (updateLast) last = listNode;
    size--;
}
node.next = null;
return node.value;
}

1. Write code that will print out each element of a SinglyLinkedList.

2. (⋆) Implement an equals method for SinglyLinkedList. It should return true if the lists store the same values in the same order. Values should be compared using equals (i.e., not ==).

3. (⋆⋆) Famous interview problem: Write a reverse method for the above SinglyLinkedList that runs in \( \Theta(n) \) time and \( \Theta(1) \) memory (not including the already existing list nodes).

Solutions

1. We first consider the following code:

   ```java
   public static void print(SinglyLinkedList<?> sll) {
   for (int i = 0; i < sll.size(); ++i)
       System.out.println(sll.getNode(i).getValue());
   }
   }
   ```

   The <?> means we have a generic type parameter but don’t know what it is. The problem with the above method is its runtime is \( \Theta(n^2) \). Instead we do the following:

   ```java
   public static void print(SinglyLinkedList<?> sll) {
   SinglyLinkedList.Node<?> curr = sll.getFirst();
   while (curr != null) {
       System.out.println(curr);
       curr = curr.next;
   }
   }
   ```

2. Below we have an implementation of equals:

   ```java
   public boolean equals(Object o) {
   if (o instanceof SinglyLinkedList<?>) {
   SinglyLinkedList<?> sll = (SinglyLinkedList<?>)o;
   if (sll.size() != size) return false;
   Node<T> curr = first;
   Node<?> ocurr = sll.first;
   while (curr != null) {
       if (curr.value == null)
   ```
if (ocurr.value != null) return false;
}
else {
    if (!curr.value.equals(ocurr.value)) return false;
    curr = curr.next;
    ocurr = ocurr.next;
}
return true;
}
else return false;
}

Above we cannot say o instanceof SinglyLinkedList<T> since the T doesn’t exist at runtime. In other words, there is no way to determine at runtime if the other SinglyLinkedList has the same generic type parameter as the current one. If there is a type incompatibility, the equals method of curr.value will catch it. Note that we properly handle the case that the list contains some nulls that the user entered intentionally.

3. To solve this in one pass over the list, we need to take each node and point it back at the node before it. Since we don’t have pointers backward we always maintain references to the current node and the one before it as we iterate (a technique sometimes called piggybacked pointers). Each step we fix our references and advance both pointers. Draw a picture here if you want to better sort out the logic. Code follows:

    public void reverse()
    {
        last = first;
        Node<T> beforeCurr = null;
        Node<T> curr = first;
        while (curr != null)
        {
            Node<T> aftCurr = curr.next;
            curr.next = beforeCurr;
            beforeCurr = curr;
            curr = aftCurr;
        }
        first = beforeCurr;
    }

Doubly Linked Lists

Here are some of the limitations of singly linked lists:
1. Getting the \( k \)th element requires iterating over the whole list (\( \Theta(n) \) worst-case).

2. Cannot remove the last element efficiently.

3. Cannot iterate through the list in reverse order.

4. Cannot delete a node efficiently without a reference to the node before it.

By paying the price of an extra reference per node (that we must store and update) we can improve on all of these (for getting the \( k \)th element, we will only gain a factor of 2 in the worst-case). The idea is to store a prev reference in each node that points to the previous node in the list. This gives us more flexibility when moving around the list.

**Sentinels**

We could proceed to design a doubly linked list that mimics our singly linked list: maintain head and tail references, have ends pointing at null, etc. Instead we will use a trick that will greatly simplify our code at the price of a single node. Our simpler code will also have far fewer if-statements which can lead to performance gains. The trick is to create a single dummy node called a sentinel. Even when our doubly linked list is empty it will still contain the sentinel node. The sentinel node has the following properties:

1. When the list is empty the following is true:
   \[
   \text{sentinel} == \text{sentinel.next} && \text{sentinel} == \text{sentinel.prev}
   \]

2. Suppose the list is not empty. Let \( \text{firstNode} \) denote the first node and \( \text{lastNode} \) the last. Then the following are all true:
   \[
   \begin{align*}
   \text{sentinel} &= \text{firstNode.prev} \\
   \text{sentinel.next} &= \text{firstNode} \\
   \text{sentinel} &= \text{lastNode.next} \\
   \text{sentinel.prev} &= \text{lastNode}
   \end{align*}
   \]

By building the sentinel node into our doubly linked list we will never have to deal with null references and pesky NullPointerExceptions. Furthermore, we will not have to store and maintain references to the first and last node since they can be found quickly through the sentinel. Recall that some of the trickiest code in the singly linked list was correctly updating the first and last references on each operation.

**Doubly Linked List Exercises**

1. Suppose a doubly linked list (as described above) is empty and we add a node with value 1. Explain how this is done.
2. Suppose a doubly linked list (as described above) has the values 1, 2, 3, 4 in that order.
   (a) Explain what happens when we delete the node with value 2.
   (b) Explain what happens when we append a 5 to the end of the list.

3. Suppose a doubly linked list is sorted. How long does it take to find a value \( v \) in the list?

4. (⋆⋆⋆) Suppose you have a list of 1000000 numbers that you want to iterate over. Will it be faster if we used a dynamic array or a doubly linked list?

**Doubly Linked List Solutions**

1. A new node is created containing the value 1 with both its prev and next references pointed at the sentinel. Both of the prev and next references of the sentinel will point at the new node.

2. (a) The next pointer of the node with value 1 will be repointed at the node with value 3. The prev pointer of the node with value 3 will be repointed at the node with value 1.
   (b) A new node with value 5 is created with next reference pointing at the sentinel, and previous reference pointing at the node with value 4. The prev pointer of the sentinel is repointed at the new node. The next pointer of the node with value 4 is repointed at the new node.

3. The worst case is still \( \Theta(n) \). Just getting access to any node/value in the middle of the list requires \( \Theta(n) \) work.

4. The ArrayList should be measurably faster. There are several reasons for this which we will quickly mention below. The basic idea is that memory accesses are more expensive than CPU operations. Understanding how much data must be loaded from memory can sometimes be the most important factor when optimizing a program. The terms used below may be unfamiliar to you, but I will discuss them in class and you can look them up online if you want to learn more.

   (a) The ArrayList is stored in contiguous memory (i.e., consecutive addresses in memory), so each cache line pulled from memory will contain the data you need to access. Furthermore, the cache prefetching done by the memory system will correctly load the next data you will be accessing.
   (b) The doubly linked list node has 2 references and some object overhead for each value stored. Thus a node object may be more than 3 times larger than the space used for the value reference. Since iterating over the linked list will need to pull at least 3 times more data from memory, it will run more slowly.
This also addresses the following related question: “If the RAM (i.e., memory) on my machine is so large, why should I care about saving space?” There is some truth to this thought. Programmers these days definitely spend less time optimizing their memory usage than before, when memory was scarce. That said, even though RAM is huge, caches are much smaller. For example, one of the biggest factors that a good linear algebra library from a bad one is how it optimizes cache utilization.

Lab 3 Preparation: JUnit and import static

In today’s lab we will be writing tests for a simple set data structure. To do this we will be using the Java library JUnit. A typical work flow for JUnit is to first create a class named ClassNameTests that will contain the tests for your class ClassName. Each method in ClassNameTests will verify that a particular assumption is true of ClassName. JUnit requires all tests to be instance methods (i.e., not static). Furthermore, each of your test methods will run in a freshly constructed object, so you can do common setup in the constructor (there is another way, but we won’t discuss it here). It is convention to start each test method with the word test. For example, testConstructor or testAddSingleElement. All of your test methods must be annotated with the annotation @Test. This is how you tell the JUnit library that this is a test method, and that it should be used when running your tests. To use @Test we must include import org.junit.Test; in our import statements.

Within your tests you will want to verify that certain assumptions hold true. To do this we will use the assert statements that come with JUnit. For example, suppose you are testing the constructor of your data structure, and you want to verify that a newly constructed object has size 0. You could write the following test:

```java
@Test
public void testConstructor() {
    BoundedIntSet bis = new BoundedIntSet();
    Assert.assertEquals("Size of newly constructed set", 0, bis.size());
    Assert.assertEquals(0, bis.size()); //Same as above without message
    Assert.assertFalse("0 in newly constructed set", bis.contains(0));
    Assert.assertFalse(bis.contains(0)); //Same as above without message
}
```

where Assert is a class in the package org.junit. Since all of these helpful assert methods are static methods of class Assert, we will have to type Assert.whatever many times. To ease this burden we can use import static. By typing

```java
import static org.junit.Assert.*;
```

Java will let us just write assertEquals(...) instead of Assert.assertEquals(...). Another example where this could be useful is if you use many of the static functions from the Math class in your file. Then you could put

```java
import static java.lang.Math.*;
```

and write abs(x) instead of Math.abs(x).
Sometimes your test will want to verify that a particular type of exception is thrown. This can be done with a try-catch and asserts. A shorter method is to use the following syntax:

```java
@Test(expected=IndexOutOfBoundsException.class)
public void testAddNegativeException() {
    set.add(-1);
}
```

Above `set.add(-1)` is supposed to throw an `IndexOutOfBoundsException` and the test above will pass if and only if it does.