CmpSci 187: Programming with Data Structures
Spring 2015

Lecture #5

John Ridgway

February 10, 2015

Administrivia

- No discussion: homework posted instead
- First exam — Feb 19 at 7:00pm in Marcus 131.
- Grades on Moodle
- Assignment 02 due Thursday — due at 8:30 a.m.; we don't accept late or incorrectly-formatted assignments, so make sure you have it done and submitted in advance. Note that you that *must use linked lists, not arrays* to solve the linked list portion of the assignment.

1 Testing

Software Testing

- The book talks about building tests manually; we'll use JUnit.
- Identifying test cases.
- Test plans.
- Testing ADT implementations.

Writing a JUnit Test Suite — Preamble

```
import static org.junit.Assert.*;
import org.junit.Test;
import org.junit.Before;
public class TicTacToeTest {
  private TicTacToeGame drawGame;

  /**
   * Runs before each test.
   */
```

```
@Before
public void before() {
  drawGame = new TicTacToe(3);
  for (int space : new int[] {0, 1, 2}) {
    drawGame.move(space);
  }
}
```

Writing a JUnit Test

```
@Test
public void testGetWinnerDraw() {
  assertEquals("", drawGame.getWinner());
}

@Test
public void testGetCurrentPlayerDraw() {
  assertEquals("", drawGame.getCurrentPlayer());
}

@Test
public void testIsValidMoveDraw() {
  for (int i=1; i<10; i++) {
    assertFalse(drawGame.isValidMove(i));
  }
}
```

JUnit available tests

- `assertEquals(msg, expected, actual)`
- `assertFalse(msg, actual)` (and `assertTrue`, `assertNotNull`, `assertNull`)
- `assertSame(msg, object1, object2)` (and `assertNotSame`)

Testing Clicker Exercise

```
private Dog cardie;
private Dog duncan;
@Before
public void before() {
  cardie = new Dog();
  duncan = cardie;
}
@Test
public testSetAge() {
  cardie.setAge(6);
  duncan.setAge(4);
  assertEquals("", 6,
      cardie.getAge());
}
```

Will the test `testSetAge`

A. Pass? or

B. Fail?

2 Introduction to Linked Lists

Linked Lists

- Last class we implemented `StringLog` using arrays. This class we will do so using simple *linked lists*.
- Linked lists are a series of *node* objects, each containing one item and one link (a reference) to the next node. The `Chain` class we saw previously was a sketch of a linked list.
- We keep a reference to the *head* of the list. From it, we can iterate through each node in list to get to the end. (We can also keep pointers to other spots; the *tail* is often a good candidate, for reasons that will become clear later.)

Linked Lists vs Arrays

- Arrays require that you (1) know the amount of memory you need before you start (otherwise you have to copy into a new, bigger array), and (2) allocate it all in advance (potentially wasting space). Linked lists only create new objects (and thus only use memory) as items are added.
- But, linked lists do not support random access. That is, we can't do an lookup for a specific index in the list, like we can in an array. Instead, we have to start at the head and follow the pointers all the way down the list until we find the item or the end of the list.

Linked List Big-O Clicker Question

Suppose a linked list contains n nodes. What are the big-O bounds on returning the first element in the list, and on returning the last element in the list? (There is no tail reference for this list.)

A. $O(1)$ for each

B. $O(n)$ for the first, $O(1)$ for the last

C. $O(1)$ for the first, $O(n)$ for the last

D. $O(n)$ for the first, $O(n^2)$ for the last

E. $O(n)$ for each

Building a Linked List for StringLog

- We need to store a string, and store a link to the next string. A node can do this, and it *has-a* string and *has-a* next string reference.
- Thus DJW define an `LLStringNode` class with a constructor and two member variables `info` and `link`, each of which has a getter and setter.

- Note I may sometimes slip and call the `link` reference the `next` reference. I'll explain why later.

`LLStringNode`

```
public class LLStringNode {
  String info;
  LLStringNode link;
    public LLStringNode(String info) {
    this.info = info;
    this.link = null;
  }
  public String getInfo() {
    return info;
  }
  public LLStringNode getLink() {
    return link;
  }
  public void setInfo(String i) {
    info = i;
  }
  public void setLink(LLStringNode lk) {
    link = lk;
  }
}
```

Traversing a Linked List

- Many operations on a linked list require *traversal*: doing something to or with each node in the list, perhaps stopping early.
- For example, *searching* (that is, to write `contains()` or the like) requires us to look at each node until we find what we want or reach the end of the list.

Inserting Nodes Into Linked Lists

- *Inserting* means adding a node to the list. Difficulty varies depending upon where we want to insert.
- Inserting at head is straightforward. Make a new node, set its link to the old head, update head.
- Inserting at tail is easy *if we have a tail reference*. Create a new node. Point the old tail node's link at it. Update tail to the new node.
- Inserting in the middle (between `x` and `y`) is harder. Create the new node `n`. Note `x`'s old link. Set `x`'s new link to `n`. Set `n`'s new link to `x`'s old link (`y`).

Linked List Insert Clicker Question 1

After running the following, what are the contents of the list starting from `head`?

```
LLStringNode head = new LLStringNode("x");
head.setLink(new LLStringNode("y"));
```

A. `"x"`

B. `"y"`

C. `"x", "y"`

D. `"y", "x"`

E. it's empty

Linked List Insert Clicker Question 2

After running the following, what are the contents of the list starting from `head`?

```
LLStringNode head = new LLStringNode("x");
head.setLink(new LLStringNode("y"));
LLStringNode tmp = head;
head = new LLStringNode("z");
head.setLink(tmp);
```

A. `"x", "y", "z"`

B. `"y", "z", "x"`

C. `"z"`

D. `"z", "x", "y"`

E. `"y", "x", "z"`