CPSC 250L Lab 12
Introduction to Data Structures: Linked Lists

Fall 2017

1 Introduction

This lab will introduce one of the most elementary (and important) data structures, the linked list. You should sit, draw a linked list, and reason out a rough solution before attempting to code each method. This is called *designing* your solution. This planning stage will let you code more effectively and help you understand what your code is doing.

2 Exercises

Fork and clone the `cpsc250l-lab12` repository from the `cpsc250-students` group on Gitlab. In this exercise, commit and push after the successful implementation of each method. In other words, you must commit at least 3 times in this lab.

Additionally, an explanation of `Node.java` and linked lists is included in Section 4.

Exercise 1

Create a class called `LinkedListMethods2.java`. Do **NOT** edit `Node.java`. **Furthermore, the ONLY imports you are allowed to use are the following.**

1. import java.io.File;
2. import java.io.PrintWriter;
3. import java.io.IOException;
4. import java.util.Scanner;

Moreover, you are NOT allowed to use any Java collection (e.g ArrayList) nor any methods related to them. Implement the following methods.

1. public static Node<String> createListFromInput(Scanner in)

   This method returns a new linked list created from the lines in in. Specifically, each *line* in in should go in its own Node and the nodes should be linked in the order of appearance in in.

2. public static boolean isWord(Node<Character> head, Scanner dictionary)

   This method receives a linked list of characters representing a word to look up in a dictionary, and a Scanner with all of the words in a dictionary. The goal of the method is to indicate whether the word to look up is in the given dictionary (or not). To achieve this feat, create a String from the characters in the list in order, and then read words from the dictionary to see if the word from the list matches one of the words in the dictionary. The method returns true if there is a match and false otherwise. For example, if our list is c->a->t (spelling the word "cat") and our dictionary is "the cat chased mice", then this method would return true.

3. public static void reverse(Node<Integer> head, File output)

   This method prints the reversal of the list starting at head to output. Moreover, it should print one integer per line. You may delete nodes from this list if you need to do so.

   HINT: Its probably easier to create a new list that is the reverse of the original and then just traverse that.

Exercise 1 Complete

Run:

```
git add .
git commit -m "Completed exercise 1"
git push origin master
```

---

3 Common Mistakes

The solutions to some common mistakes are as follows.

1. The last `Node` in a linked list always has the `next` field set to `null`.
2. Be sure to check for `null` references!

4 Tutorial

4.1 `Node.java`

Before talking about how to do things with linked lists, we must first understand the structure of it. A linked list is a chain of *nodes*, each of which contain a *value* and a *reference to the next element* in the list. This chain continues until we cannot find a next element. In this lab, we use linked lists that are made from `Node` objects (from the `Node.java` file packaged with the repository).

The `Node` class has two fields in it, a *final field* of type `T` called `value` and a reference to the next node in the list, called `next`. To access the value of a `Node<T> node`, you type `node.value`. This will let you *read* the value of `node`, but you will not be able to write back to it as it is a `final` field. To access the next node, you type `node.next`. This field is changeable since you need to be able to edit it in order to create and modify lists. Generally, we set these fields to `private` and make you call getter and setter methods in order to retrieve or edit their values however in order to reduce the complexity of the assignment we allow you to access them directly.

To create a `Node`, you have two options

- `Node(T _value)` which initializes the `value` field to `_value` and `next` to `null` (this is what you will use most of the time),
- and `Node(T _value, Node<T> _next)` which initializes `value` to `_value` and `next` to `_next`.

4.2 Linked Lists

As we said earlier, a linked list is a chain of `Node` objects with a *head* and a *tail*. The head of a linked list is the first node in the list, while the tail is the last node of the list. In this lab, we say a `Node` is the tail of a list if its `next` field is set to `null`. When we are given a linked list, we typically only store a reference to the head of the list, appropriately named `head`. As a structure, a linked list starting at `head` is shown in Figure 1.

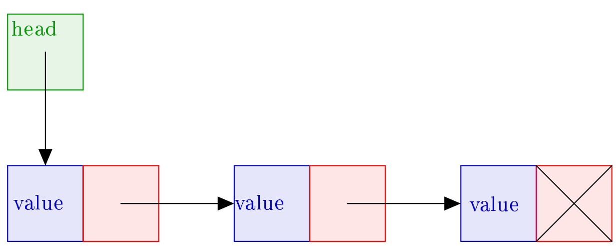

Figure 1: A diagram of a Linked List

To create a linked list, all we need to do is create a single `Node` object. If we want to add to the list, we make another `Node` object and set the `next` field of the tail of the list to the new `Node`. For example, the code below makes a linked list with integers between `start` and `end` inclusively. We give an example output in Figure 2.

Code:

```
public static makeListOfRange(int start, int end) {
    if (end < start)
        return null;
    Node<Integer> head    = new Node<Integer>(new Integer(start));
    Node<Integer> current = head;
    Node<Integer> next    = null;
    for (int i = start + 1 ; i <= end ; i++) {
        next         = new Node<Integer>(new Integer(i));
        current.next = next;
        current      = current.next;
    }
    return head;
}
```

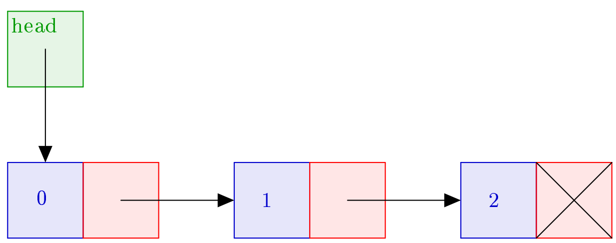

Figure 2: Output of `makeListOfRange(0,2)`

Now that we can create a linked list, the most important thing to learn to do is to *traverse* the list, or read through the list node by node. To do so, we set `Node<T> current = head` and loop while `current` is not `null`. In each iteration, we set `current = current.next` in order to progress through the chain. Generally speaking, a traversal looks like the code below.

Code:

```
Node<T> current = head;
while(current != null) {
    // Do something
    current = current.next;
}
```

For example, lets say we want to print a linked list to standard output. To do so, we traverse the list while printing each `Node`'s `value` field. The code to do so is given below.

Code:

```
public static printList(Node<String> head) {
    Node<String> current = head;
    while (current != null) {
        System.out.println(current.value);
        current = current.next;
    }
}
```

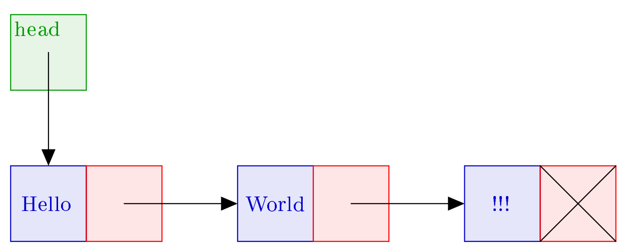

Figure 3: Example input to `printList`

If the list in Figure 3 is passed as input to `printList`, it would print the following.

Code:

```
Hello
World
!!!
```

The last thing we cover is how to *delete* a node from the list. In order to remove a `Node` from the list, we simply change its predecessor's `next` field to the current `Node`'s next field using `predecessor.next = current.next`. Typically, we also set `current.next` to `null` for security reasons. Because Java has a *garbage collector*, the JVM will automatically free the memory taken up by the removed `Node`. The following code will delete the first node in a linked list that has a value of `val`.

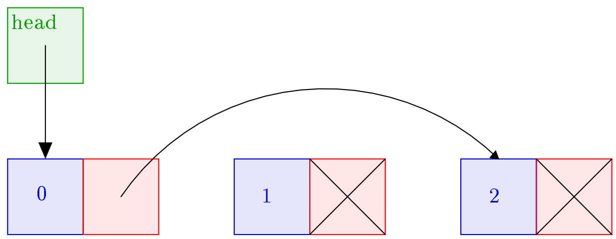

Figure 4: Output of `deleteNode` with Figure 1 and 1 as input

Code:

```
public static Node<Integer> deleteNode(Node<Integer> head, Integer val) {
    if (head == null)
        return null;
    // Special case if head has the desired value
    else if (head.value.equals(val)) {
        Node<Integer> out = head.next;
        head.next = null;
        return out;
    }
    // Find first instance of value
    Node<Integer> current  = head;
    Node<Integer> previous = null;
    while (current != null && !current.value.equals(val)) {
        previous = current;
        current  = current.next;
    }
    // Delete first instance of value, if it exists
    if (current != null) {
        previous.next = current.next;
        current.next  = null;
    }
    return head;
}
```

If we suppose the list shown in Figure 1 is the input, then `deleteNode(head, 1)` will output the list in Figure 4.

It is important to note that in our `deleteNode` method, we needed to check for when `head` is `null` and that we needed to handle the special case in which the `head` contains the desired value. Special cases for the head and tail of a list will be a common theme when dealing with them, so there is a decent chance that logic that works on the middle nodes will not work verbatim on the head and tail.