Lecture 11

Linked Lists – Continued

When removing items from a linked list, there are multiple cases.

In the case that the value we are looking for is first item on list:

1. head = head.next();
2. size--;

For general case, head now points to the next item on the list, and the garbage collector will clean up the first item.

For the case of one list item, notice that head = head.next(); will set head to null.

In the case that the value is on the list, but is not first:

1. finger = head;
2. walk list until finger.next() != null or until finger.next().value().equals(value).
3. now finger points to value before the one to remove. To remove the desired item, just use finger.setNext(finger.next().next()) since finger points to the previous element of the list, we know that finger.next() is not null, hence we know that we can say finger.next().next() (which might be null).
4. size--;

In the case that the value is not on the list, the above approach works, but we have to recognize that the value was not found. Here, it helps to notice that the two terminating conditions for the while loop – finger.next() == null and finger.next().value().equals(value) are mutually exclusive. That is, they can’t both be true at the same time.

Variations on LinkedLists

One obvious variation on linked lists is to add some additional instance variables to make certain operations easier.

For example, we can add a “tail” reference that points to the last item on the list. Now, items can be added to the end of the list in constant time.

1. tail.setNext(new node)
2. tail = new node

The first step sets the next pointer of the Node that was the original last item on the list. The second step sets the value of the tail instance variable itself.
Without the tail reference, this took $O(n)$ time for the add.

Another variation on linked lists is known as the doubly linked list. In this variation, each Node has a value, and two references – one for the next Node in the list, one for the previous.

Doubly linked lists are useful when traversal in both directions is important. For example, your browser allows you to move back and forth through the web pages in your history. These could be stored in a doubly linked list so that only constant work is required to move either direction. This may also be a better solution than an ArrayList since size of linked list is more flexible.

Operations for doubly linked lists are similar to those for singly linked lists, except that slightly more work is required.

For example, when adding an element to the middle of the list (assuming, as before that we find the item before the location to add to):

1. `newNode.setPrev(finger)`
2. `newNode.setNext(finger.next())`
3. `finger.next().setPrev(newNode)`
4. `finger.setNext(newNode)`

Before, adding to the end of the list was identical. Now, due to the previous references, we have a small special case:

1. `newNode.setPrev(finger)`
2. `newNode.setNext(finger.next())`
3. `if(finger.next() != null)`
   `finger.next().setPrev(newNode)`
4. `finger.setNext(newNode)`

As before, we assume finger references the node previous to newNode, so adding to the front of the list is an entirely special case:

1. `newNode.setPrev(null)`
2. `newNode.setNext(head.next())`
3. `head.setPrev(newNode)`
4. `head = newNode`

Yet another variation of linked list is the circularly linked list. Instead of the last item on the list referencing null, it references the first item on the list. Such a list is handy for a buffer that is constantly reading in information, and the information is being read out of the buffer.

This is sometimes called a “producer-consumer” model. Keyboard input often works in this way. It is unreasonable for a program to look directly to the keyboard and wait for user keystrokes. Instead, user keystrokes fill a buffer, and these values are read concurrently when the program is active. For efficiency,
a reference to the next available slot in the buffer is kept, and the slots are filled in order. When the end of the buffer is reached, the reference wraps around to the front of the buffer, which is now hopefully free because it’s values have been read.

Circularly linked lists present an interesting problem. Before, we traversed the list by following `next` references until we reached the end of the list, indicated by `next()` returning `null`. Now, that will never happen. We can solve this problem by identifying when we have returned to the front of the list:

```java
if(head != null) {
    Node <String> finger = head;
    do {
        System.out.println(finger.value());
        finger = finger.next();
    while (finger.next() != head);
}
```

The `do-while` loop is used instead of a `while` loop to avoid the condition failing for a list of size 1.

**Skip Lists**

Skip lists use several linked lists referencing the elements to improve run time performance of some operations. A detailed description can be found on Wikipedia: