Continuing with Linked Lists

Recall the LLNode class:

public class LLNode<T> {

private T element;

private LLNode<T> next;

public LLNode(T element, LLNode<T> next) {

this.element = element;

this.next = next;

}

public T getElement() {

return element;

}

public LLNode<T> getNext() {

return next;

}

public void setNext(LLNode<T> next) {

this.next = next;

}

}

Now, let us create the linked list:

+---+---+ +---+---+

list -> | 1 | \*-+--> | 2 | \*-+--x

+---+---+ +---+---+

LLNode<Double> list = new LLNode<Double>(1, new LLNode<Double>(2, null));

A final example:

Place 1.5 between 1 and 2 in the above list.

We must be careful to do things in the right order, or we will break the list.

1) Create a new node for 1.5.

2) Move the next pointer for the 1.5 node to point to the 2 node.

3) Move the next pointer for the 1 node to point to the 1.5 node.

If we did step 3 before step 2, we would lose all access to the 2 node.

What is the name of the next pointer of 1? list.next

To do step 2, we need to change the new's next pointer the list.next, i.e. what the first box is pointing at.

To do step 3, we need to change list.next to point to the new box.

We can do this in one line of carefully organized code!

Step 1) LLNode<Double> newNode = new LLNode<Double>(1.5, null);

Step 2) newNode.setNext(list.getNext());

Step 3) list.setNext(newNode);

Or in one line:

list.setNext(new LLNode<Double>(1.5, list.getNext());

Abstract Data Types

An abstract data type is a data structure that guarantees certain behaviors to the user, but it keeps its implementation details hidden.

By hiding the implementation details, we are able to change them if we discover a better way to do things without breaking the code that uses the

data structure. We also can prevent code that uses the data structure from accidentally breaking the data structure.

Examples:

Strings are an ADT. You are guaranteed certain behavior such as accessing a character from a location and appending strings, but you are not told how

they are implemented (though they are probably implemented as an array of chars).

JFrames are an ADT. You are guaranteed certain behavior such as changing its size, displaying it on the screen, but you do not know exactly how the

Java Swing developers chose to implement the window.

We will create the LinkedList as an abstract data type. The list is going to be a list of LLNodes, but we will keep the details away from the users

of the LinkedList so that code that uses the LinkedList can not accidentally break the list.

Creating a LinkedList class

A LinkedList will store a list of LLNodes. The only field we need is to store the node that is the first node of the list. We call this the "front" or "head" of the linked list.

There are several ways we can implement the linked list. We decided that if a list is empty, its front should be a null pointer, to indicate that there are

no nodes in the list. Another option would be to create a special node that acts as a "caboose" to the linked list. No implementation technique is wrong

as long as the list works properly. If we correctly create the LinkedList as an abstract data type, we should be able to switch between implementations and

code that uses the LinkedList class will still operate exactly the same.

Note that the LinkedList will need to use a generic to specify the type that will be stored in the list, and we want each LLNode in the list to use the same generic.

public class LinkedList<T> {

private LLNode<T> head;

public LinkedList() {

head = null;

}

Now, let use crete a method to add an element to the front of the list.

public void addToFront(T element) {

head = new LLNode<T>(element, head);

}

If we want to allow polymorphism and extending for this class, we should use getter/setter methods for front, but we don't want to make them public because that would let

any code using the LinkedList (rather than any code that "is" a LinkedList) to need to understand how the head of the linked list works.

Having code outside the LinkedList know about LLNodes would violate the "keep implementation details hidden" nature of an abstract data type.

The solution is to make the getter/setter methods "protected". Recall that protected means it can be used in this class or any class that extends this class. (Why would private not work?)

public void addToFront(T element) {

setFront(new LLNode<T>(element, getFront());

}

How about removing from the front? We set the head of the list to be the node after the current head of the list.

We also need to save the element at the head so we can return it after making the change to the linked list.

public T removeFromFront() {

T save = getFront().getElement();

setFront(getFront().getNext());

return T;

}

What if the list is empty? Then getFront().getElement() will return a NullPointerException.

It is not a good idea to have this method throw a NullPointerException because that is not very descriptive.

A programmer that is using our class will not understand what is null. The real reason for the error is that there are no

elements in the list. A hunt through pre-defined Java exceptions gives NoSuchElementException which more closely matches the cause of the error.

public T removeFromFront() {

if (isEmpty())

throw new NoSuchElementException();

else {

T save = getFront().getElement();

setFront(getFront().getNext());

return T;

}

}

public boolean isEmpty() {

return getFront() == null;

}

Linked Lists and Loops

Looping through a linked list has its own special form.

We need a "node pointer" to point to remember where we are in the list.

We also need to decide if we should end pointing to the last node of the list, or after the last node of the list.

Example, create a method inside LinkedList that returns the number of elements stored in the list.

public int length() {

int count = 0;

LLNode<T> nodeptr = front;

while (nodeptr != null) {

count++;

nodeptr = nodeptr.getNext();

}

return count;

}

Notice the behavior of the above code: we start the nodeptr at the first node of the list, and each time through the loop,

we increment the node pointer to the next node of the list.

An example using recursion.

Let us create a method lengthFromHere inside the LLNode class that returns the length of the list -after- this node.

The method will use the recursion technique. This is a simple way to come up with an algorithm, but it is not the most

efficient to run.

To come up with the recursive algorithm, you just need to consider two cases: what is the answer if this is the last node of the list,

and what is the answer if it is not?

public int lengthFromHere() {

if (this.getNext() == null) // there are no nodes after this one

return 0;

else // otherwise, ask the next node what the length after it is, and add 1

return getNext().lengthFromHere() + 1;

}

Recursion is a very simple way to code, and it is efficient in some languages, but unfortunately it is not efficient in Java.

In Java, a stack frame is placed on the stack with each method call. If our linked list is really long, the recursion will place so many

frames on the stack that it will use up the memory.

Still, recursion is good if you know that the length of the recursion will not be too long. So, it is a good technique to master so you have

it as a possibility when faced with a tough problem.