Linked Lists and Loops

Here is another loop example with linked lists. Inside the LinkedList class, let us write a method addToEnd.

This is almost the same as the loop we wrote for length(), but the stopping condition is different. Can you see why?

public void addToEnd(T element) {

LLNode<T> nodeptr = getFront();

while (nodeptr.getNext() != null) // when loop exits, nodeptr will point to the last node of the list

nodeptr = nodeptr.getNext();

nodeptr.setNext(new LLNode<T>(element, null));

}

There is one case where this method will crash with a NullPointerException: if the list is empty, getFront() will return null, and nodeptr.getNext() will

throw a NullPointerException.

The error is caused by the list having no elements, so we have to make a special case for that.

Here is the first suggested technique, BUT IT HAS A SERIOUS PROBLEM!

public void addToEnd(T element) {

if (length() == 0())

addToFront(element);

else {

LLNode<T> nodeptr = getFront();

while (nodeptr.getNext() != null) {

nodeptr = nodeptr.getNext();

} // when loop exits, nodeptr will point to the last node of the list

nodeptr.setNext(new LLNode<T>(element, null));

}

}

What is wrong? The length() method is not a one step method in linked lists like it is in arrays! To compute the length, we have to run through the entire

linked list counting the nodes. If we are doing a large number of addToEnd calls on a long list, the program could take a long time!

MORAL: Although certain methods exist for linked lists (and others for arrays), it does not mean it is a good idea to use them. You must always consider

how long certain methods will take to run and choose the more efficient methods for your program.

public void addToEnd(T element) {

if (isEmpty())

addToFront(element);

else {

LLNode<T> nodeptr = getFront();

while (nodeptr.getNext() != null) {

nodeptr = nodeptr.getNext();

} // when loop exits, nodeptr will point to the last node of the list

nodeptr.setNext(new LLNode<T>(element, null));

}

}

The Iterable interface

We can try to think of everything that a program that uses our LinkedList will want to do, but that is impossible. Instead, other programs will need to write their own

loops to run through the linked list.

This creates a problem: to run through the linked list requires having a node pointer, but if our linked list is to be a abstract data type, we do not want to require outside

classes to have to deal with the linked list implementation details.

Java provides a pair of interfaces that let us provide a means for other code to loop through our linked list while still hiding the implementation details. These interfaces are Iterable and Iterator.

The Iterable interface is used to indicate that we can create an Iterator for the class, and an Iterator is an object that lets us run through the contents of the class.

Every class that is an abstract data type and can store multiple elements should implement the Iterable interface.

The Iterable interface has 1 method:

Iterator<T> iterator() -> returns an object that implements the Iterator interface, and has the same generic as the Iterable class.

The Iterator interface has 3 methods:

boolean hasNext() -> returns true if there are more elements in the list

T next() -> returns the next element in the list, and "iterates" so the next time we call next(), we get the next element of the list

void remove() -> removes the last returned element from the list. This method is optional so we can avoid implementing it by throwing an appropriate exception.

These Iterator methods generalize the main components of a loop. If we have an Iterator for the abstract data type, we can loop through the abstract data type's contents by calling the

hasNext() and next() methods of the Iterator. Thus, code using the Iterator does not need to know about the implementation details of the abstract data type.

Recall how we write a loop for a LinkedList:

LinkedList<String> list = new LinkedList<String>();

... code adding items to the String ...

LLNode<String> nodeptr = list.getFront(); // assuming getFront is public!

while (nodeptr != null) {

System.out.println(nodeptr.getElement()); // print each element

nodeptr = nodeptr.getNext();

}

Here is the same thing with the iterator:

LinkedList<String> list = new LinkedList<String>();

... code adding items to the String ...

Iterator<String> iterator = list.iterator(); // no knowledge of LLNode is needed!

while (iterator.hasNext()) {

System.out.println(iterator.next()); // a simpler loop!

}

To write the Iterator class, we need to go back to our loop using LLNodes. The key parts of the loop are:

1) initalization: We have to set up the loop. Here the code is "LLNode<T> nodeptr = getFront()"

2) condition: We need to indicate when loop should continue and when it should stop: "nodeptr != null"

3) retrieval: We need to get the next element from the list "nodeptr.getElement()"

4) increment: We need to move to the next element in the list "nodeptr = nodeptr.getNext()"

Now, we need to create the Iterator for LinkedList.

First, LinkedList must implement the Iterable interface. Notice that it takes a generic. We will use the same generic that is stored in the LinkedList.

public class LinkedList<T> implements Iterable<T> {

Now, the Iterable interface requires the iterator method that returns an object of type Iterator<T>. We will create a class called ListIterator that implements

this type and return an instance of that. However, since we are overriding the iterator method, we must be careful about the return type.

@Override

public Iterator<T> iterator() {

// this code will be filled in once we define an iterator for the linked list

}

Now, we need to create a class that implements the Iterator interface. We need to create the 3 methods: hasNext, next, and remove (but the last

method does not need to do anything). To get the iterator to work, we need to look back at the original way we ran through the LinkedList.

Note that the hasNext() method replaced the condition: nodeptr != null

Note the next() method replaced two lines, the expression: nodeptr.getElement()

and the increment: nodeptr = nodeptr.getNext();

Finally, we have to start with nodeptr = getFront() or the loop will not work. Where should we put this? In a constructor! It needs to be called when we are setting up the loop/iterator.

public ListIterator<T> implements Iterator<T> {

private LLNode<T> nodeptr;

public ListIterator(LLNode<T> firstNode) {

nodeptr = firstNode;

}

public boolean hasNext() {

return nodeptr != null;

}

public T next() {

T element = nodeptr.getElement();

nodeptr = nodeptr.getNext();

return element;

}

However, the API for Iterator says that we need to throw a NoSuchElementException if next() is called when there are no more elements in the linked list.

You can either add an if statement to the next method or place a try/catch block in the next method to catch the NullPointerException. Then we can throw the appropriate exception.

Finally, since we now know how to code the iterator() method of LinkedList:

@Override

public Iterator<T> iterator() {

return new ListIterator(this.getFront());

}