CMPSCI 187: Programming With Data Structures

Lecture #8: A Linked StringLog Implementation
David Mix Barrington
21 September 2012

A Linked StringLog Implementation

- The Linked List Idea
- The LLStringNode Class
- Traversing a Linked List
- Inserting and Deleting in a Linked List
- The Data for the LinkedStringLog Class
- The Methods for the LinkedStringLog Class

The Linked List Idea

- The basic idea of a **linked list** is that we have a series of **nodes**, each one containing a data item. Each node has a **link** (a pointer) to the node that follows it in the sequence. If we know the **head** of the series, we can follow successive links to reach any of the nodes in it. The last node has a `null` pointer, because there is no node following it.

- The great advantage of a linked list is that we allocate memory to it only as needed. When we need a new node, it can lie anywhere in the computer's memory, as long as we have a pointer to find it.

- The disadvantage of a linked list is that we do not have **random access** to the nodes as we would in an array. If we want the 117'th node in the list, we have to find it by following 117 links from the head. We may keep another pointer to find the **tail** of the list quickly, but that doesn't help find a node in the middle.

The LLStringNode Class

- We need two things from a single node in a list of strings: to store a string, and to store a link to the next string. Of course, in Java the latter is expressed by saying that the node *has-a* node as a component.

- The only methods needed by the `LLSTringNode` class are a constructor, the **getters** `getInfo` and `getLink,` and the two **setters** `setInfo` and `setLink.`

```
public class LLSTringNode {
   private String info;
   private LLStringNode link;

   public LLStringNode(String info) {
      this.info = info;
      link = null;}
   // two getters and two setters
}
```

Traversing a Linked List

- Many operations on a linked list operate by **traversing** it, which means doing something to each node in order.

- If we are **searching** for something, we keep looking at nodes until we either find what we want or reach the tail of the list and give up.

- Here is the basic pseudocode for a traversal that might stop in the middle. Note that since this code is not in the `LLStringNode` class, we have to use a getter to follow the link out of a node.

```
LLSTringNode curr = head;
done = false;
while (!done && curr != null) {
   process node (may set “done”);
   curr = curr.getLink( );
deal with result;
```

Inserting Into a Linked List

- Inserting a new node into a list means adjusting as many links as necessary to put the new node in the right place. How hard this is depends on where the new node is to go.

- Inserting at the head is simple: we make the new node the new head, and set the new node's link to the old head.

- Inserting at the tail is easy once we have a pointer to the tail: we set the old tail's link to the new node, keep the new node's link as `null`, and update the list's tail pointer if we are keeping one.

- Inserting in the middle is the most complicated. If node x points to node y and we insert node z between them, we need x's link to be z and z's to be y.

Deleting From a Linked List

- The difficulty of deleting a node from a list depends on where the node is in the list, and what information we start with.
- To delete the head node, we just set the head to be the old head's link.
- To delete the tail node, we set its predecessor's link to `null`. That is, we do that once we have found the predecessor, which requires a traversal even if we start with a pointer to the tail node. (A **doubly linked list** would help here.)
- To delete a node in the middle, we need to reset the predecessor's link to the deleted node's link. Again, this requires finding the predecessor which in general requires a traversal.

The Data for the LinkedStringLog Class

- We need data structures to store the log itself (the series of strings) and to store the name. These will be `protected` to give access to subclasses.
- The name can just be stored as a `String`.
- To store the log as a linked list, we need a pointer to the head node of the list. We *say* that "`log`" is a node, but it isn't just a node because through the pointers it gives us access to all the nodes.
- DJW choose not to keep an `int` variable for the size of the list. This would make the size method easy but require us to update the size for inserts.

```
public class LinkedStringLog implements StringLogInterface{
   protected LLStringNode log;
   protected String name;
```

Constructors for LinkedStringLog

- Since a linked list may grow to have as many nodes as we need, we don't have to fool with a capacity measure for our StringLog. Thus we need only one constructor.

- If the head of the list is `null`, there are no nodes in the list, which is as it should be for a newly constructed StringLog. The traversal process will work fine on such an empty list -- we will fall out of the `while` loop immediately. (But we have to make sure that anything we try to do with the *node* named "`log`" is **guarded** by a test to make sure that such a node exists.)

```
public LinkedStringLog (String name){
   log = null;
   this.name = name;}
```

Transformers for LinkedStringLog

- The `clear` operation works just like the constructor, setting `log` to `null`.
- Since we don't care which node goes where, we can insert a new node at the beginning.
- Should we worry about whether `log` is `null` before the insert? No, because if it is we just transfer the null pointer to the link of the new node, which is now the only node in the list.

```
public void clear( ){
   log = null;}

public void insert (String element) {
   LLStringNode newNode = new LLStringNode (element);
   newNode.setLink (log);
   log = newNode;}
```

Observers for LinkedStringLog

- Again `getName` is easy, and this time `isFull` is even easier because it always returns false. We still need the method because it is required by the interface.

- Without keeping track of the size, we need a traversal to count the nodes in the list. Note that the use of `node` inside the loop is guarded.

```
public String getName( ){
   return name;}
public boolean isFull( ){
   return false;}
public int size( ){
   int count = 0;
   LLStringNode node = log;
   while (node != null){
      count++;
      node = node.getLink( );}
   return count;}
```

The `contains` and `toString` methods

- Each of these is a traversal, replacing the loops in `ArrayStringLog`.

```
public boolean contains (String element){
   LLStringNode node = log;
   while (node != null) {
      if (element.equalsIgnoreCase(node.getInfo()))
         return true;
      else node = node.getLink( );}
   return false;}

public String toString( ){
   String logString = "Log: " + name + "\n\n";
   LLStringNode node = log; int count = 0;
   while (node != null){
      count++;
      logString += count + ". " + node.getInfo( ) + "\n";
      node = node.getLink( );}
   return logString;}
```