Notes 18: Searching in Arrays

To be added.

Notes 19: Linked Lists

Arrays work great for keeping items ordered by some ordering attribute, for example names or x-coordinates, as long as you are willing to pay the sorting cost. But what if you want to keep elements ordered but don’t want to pay the sorting cost? One solution is linked lists.

We assume the reader is familiar with linked lists. If you desire a review, check out the following excellent material about pointers and linked lists at the Stanford CS site:

- http://cslibrary.stanford.edu/104/
- http://cslibrary.stanford.edu/106/
- http://cslibrary.stanford.edu/102/
- http://cslibrary.stanford.edu/103/
- http://cslibrary.stanford.edu/105/

Note, the first link is for some claymation videos illustrating pointer concepts. No joke! Between your past experience and the above, you should be able to handle pointers and linked lists in some other language such as C, C++, or Java.

Actionscript does not provide the formal concept of pointers, but you can create self-referential data structures. Consider the following code for a listNode:

```
listNode.as:

class listNode
{
    private var next:listNode ;
    private var data:Object ;
    private var key ;

    function listNode(newData:Object, newKey) {
        next = null ;
        data = newData ;
        key = newKey ;
    }

    public function print() {
```
trace("-------- Inside listNode::print() --------")
trace("key = " + key);
trace("next = " + next);
trace("Data equals: ");
data.print(); // trace(data);
}

public function getNext(nextNode:listNode):listNode {
    return(next);
}

public function getKey() { return(this.key) ; }
public function getData():Object { return(this.data) ; }

public function setNext(nextNode:listNode) {
    next = nextNode ;
}

In each node we have the following members:

    data: the generic data object being stored
    key: the value used for the list ordering (usually a Number or String)
    next: The next listNode in the list. This is how you link together nodes.

The above gives you a solid foundation for building linked lists. Note, I have chosen to
make data of type “object”. This increases flexibility at the expense of strong type
checking. You could instead make a list where the data is of type Number, String, Date,
or some class you have defined such as our past Person or MovingMC examples.
Likewise, the type of key is not specified. This is “sloppy”, but lets you use either
Numbers or Strings for keys.

So, how do we use this class to create a list? Consider the following code that links
together an UNORDERED list:

**a_selfUse_listNode_1.fla:**

    var listHead:listNode ;
    listHead = null ;

    var tempNode:listNode ;

    tempNode = new listNode("bob",3) ;
listHead = tempNode;

tempNode = new listNode("sue",5);
tempNode.setNext(listHead);
listHead = tempNode;

tempNode = new listNode("Mary",1);
tempNode.setNext(listHead);
listHead = tempNode;

var currentNode:listNode;
currentNode = listHead;
while (currentNode != null) {
    currentNode.print();
    currentNode = currentNode.getNext();
}

The above creates a list of three nodes. The nodes will be ordered in the opposite order of how they were inserted. Do you see why? The last part of the code prints out the contents of the list. A variable named currentNode is created and set to listHead. Because currentNode is set point at existing nodes, we will call it a “pointer”. The idea is that it “points” at other nodes. We will move currentNode down the list pointing at one node after another. While currentNode does not equal null, the code calls the node.print() method and then advances currentNode to the next list element.

One of our goals was to have an ORDERED list. We need to write code that makes sure the nodes are inserted into the list in order. Consider the following code:

\textit{a\_selfUse\_listNode\_2.fla}

var listHead:listNode;
listHead = null;

var tempNode:listNode;

// first insert 3 nodes such that they happen to be
// in order by inserting at the front of the list each time.
tempNode = new listNode("Mary",5);
listHead = tempNode;

tempNode = new listNode("bob",3);
tempNode.setNext(listHead);
listHead = tempNode;

tempNode = new listNode("sue",1);
tempNode.setNext(listHead) ;
listHead = tempNode ;

// now add code to insert in order
var aft:listNode = null ;
var fore:listNode = listHead ;

tempNode = new listNode("Tom",4) ;
while (fore.getKey() < tempNode.getKey()){
    aft = fore ;
    fore = fore.getNext() ;
}
// at this point aft is pointing to the node BEFORE where
// we want to insert
tempNode.setNext(fore) ;
aft.setNext(tempNode) ;

var currentNode:listNode ;
currentNode = listHead ;
while (currentNode != null) {
    currentNode.print() ;
    currentNode = currentNode.getNext() ;
}

The first three inserts are just inserted as in the previous example, at the front of the list, but the key values are chosen so that after these inserts the three nodes are in sorted order. The next hunk of code inserts a new node, with data = “Tom” and key = 4, into the list in sorted order. It does this using “aft” and “fore” pointers. The idea is the “fore” leads, and “aft” follows, hence the names. Like currentNode in the previous example, aft and fore are assigned to existing nodes rather than nodes created for them. Thus, we can think of them as “pointing” to existing nodes and will call them pointers. Pointers aft and fore are used to find the correct place in the list to insert the new node. We want to find the place where fore points at the node that will come after the new node, and aft points at the node that will come before the new node. We do this by comparing fore.getKey() with the tempNode.getKey() and stop advance fore once fore.getKey() exceeds
tempNode.getKey(). When this point is reached a visualization of the state would be as in the drawing above.

The above works, but what if the key of the node we are inserting is smaller than anything in the list? What if it is larger? What if the list is empty? We need to handle each of the following conditions:

- empty list
- insert first in the list
- insert last in the list
- insert in middle (neither first or last)

Rather than write code to manage lists every single time, it makes much more sense to create a linked list class with well defined methods. The following file contains a linked list class:

`linkedList.as`

It includes the following public methods:

- `insertFront(data,key)` inserts the data,key as the first node in the list
- `insertEnd(data,key)` inserts data/key as the last node of the list
- `insert(data,key)` inserts data/key in order based on key
- `removeNode(key)` removes the first node with key value = argument
- `removeAll()` removes all nodes
- `print()` prints out the list

For examples on how to use this, look at

`singleListDriver.fla`

which just includes:

`listDriver.as`

In summary, to use this list class, you need each of the following files:

- `listNode.as`
- `linkedList.as`
- `listDriver.as`
- `singleListDriver.fla`

Now we have everything we need to maintain linked lists.
Exercises

1. Add a public method to linkedList(): removeFirstNode() that removes the first node in the list
2. Add a public method to linkedList(): removeLastNode() that removes the last node in the list
3. Add a public method to linkedList(): removeEveryOtherNode() that removes every other node starting with the first node.
4. Create a doublyLinkedList class. This class should have members next and back, where back points to the previous node in the list. The class should also have a tail as well as a head. Include all the same methods as for linkedList(). Also add removeFirstNode() and removeLastNode(). The later is now easier to implement.