

# CS1Ah Lecture Note 21

## Linked Lists

Previous lectures have focussed on the use of arrays for holding ordered sequences of items. The class `Vector` was also introduced as a more flexible alternative to arrays. This lecture takes a look at an alternative data-structure for representing ordered sequences called *linked lists*.

The idea of a linked list is to introduce a *link* object to both hold each item in a sequence and to indicate how to find the next link using a reference.

### 21.1 Implementation of Singly-Linked Lists

We give here a simple example implementation.

First we introduce a class for links.

```
1  class SLink {
2      protected Object element;
3      protected SLink nextLink;
4
5      SLink(Object element, SLink nextLink) {
6          this.element = element;
7          this.nextLink = nextLink;
8      }
9  }
```

We choose to use `Object` for the type of the `element` field allowing any kind of object to be stored at each link.

Next we introduce a *handle* class `SLinkedList`, each of whose objects holds a reference to a linked list built from `SLink` objects. The idea is that users of linked lists will always refer to lists via these handle objects. The `SLinkedList` class also provides methods for manipulating the linked lists.

```
1  public class SLinkedList {
2      protected SLink firstLink = null;
3
4      public SLinkedList() {}
5  }
```

```

6   public boolean isEmpty() { return firstLink == null; }
7
8   public void addFirst(Object o) {
9       firstLink = new SLink(o, firstLink);
10  }
11
12  public Object getFirst() {
13      if (isEmpty()) return null;
14      else return firstLink.element;
15  }
16
17  public Object removeFirst() {
18      if (isEmpty()) return null;
19      Object result = firstLink.element;
20      firstLink = firstLink.nextLink;
21      return result;
22  }

```

## 21.2 An iterator for a singly-linked list

Java iterators conceptually point to *between* sequence items. To begin, they are *before* the first item, and the `next()` method shifts an iterator on one place and returns a reference to the item just crossed.

We introduce a new class `Iterator`

```

1   public class Iterator {
2       protected SLink linkToRight;
3
4       // Constructor
5       protected Iterator(SLink link) { linkToRight = link; }
6
7       // Iterator methods
8
9       public boolean hasNext() { return linkToRight != null; }
10
11      public Object next() {
12          if (hasNext()) {
13              Object result = linkToRight.element;
14              linkToRight = linkToRight.nextLink;
15              return result;
16          }
17          else
18              return null;
19      }
20  }

```

and we add to the `SLinkedList` class the method

```

1   public Iterator iterator() {
2       return new Iterator(firstLink);
3   }

```

## 21.3 Testing the singly-linked list implementation

We use a couple of methods of a class `Test` to exercise the above implementation. First we introduce a `printList` method which illustrates use of the list iterator.

```

1   public static void printList(SLinkedList s) {
2
3       System.out.print("[");
4
5       Iterator i = s.iterator();
6       if (i.hasNext()) System.out.print(i.next());
7
8       while (i.hasNext()) {
9           System.out.print(", " + i.next());
10      }
11      System.out.println("]");
12      return;
13  }

```

We set up the main method of `Test` to run through calls of the various methods of `SLinkedList` on an example list of `Integer` objects. Here is its output:

```

[]
isEmpty() == true
Adding 2 with addFirst()
[2]
isEmpty() == false
getFirst() == 2
Adding 3 with addFirst()
[3, 2]
isEmpty() == false
getFirst() == 3
Adding 5 with addFirst()
[5, 3, 2]
isEmpty() == false
getFirst() == 5
contains(4) == false
contains(5) == true
recContains(4) == false
recContains(5) == true
Calling removeFirst() till isEmpty()...
Calling removeFirst()

```

```
removeFirst() removed 5
[3, 2]
Calling removeFirst()
removeFirst() removed 3
[2]
Calling removeFirst()
removeFirst() removed 2
[]
```

## 21.4 Accessing the end of a singly-linked list

Abstractly, the behaviour of the list methods above should look familiar: effectively the main operations are those for a stack (for example, `addFirst()` is `push`, `removeFirst()` is `pop()`).

However accessing items, adding items to or removing items from the far end of a singly-linked list as implemented above is a much slower operation because we have to iterate down each link of the list in turn.

The access and add operations can be made much more efficient by adding to handle objects an extra reference to the last link of the linked list. This singly-linked list implementation could then be regarded as an efficient implementation of the *queue* abstract data-structure. However, this reference to the last link doesn't solve the problem of efficiently removing the last item, since there is no way of quickly updating this reference to the last link to refer to one link before the last.

## 21.5 Doubly-linked lists

A singly-linked list is so-called because each link has a single reference onto the next link. One can also design a linked-list class where each link also has a link to the previous link if any. Such lists are said to be *doubly-linked*.

The advantage of a doubly-linked list is that one can efficiently access, add and remove items from either end. In addition, one can set up iterators which can easily move in either direction along the sequence.

## 21.6 Adding and removing elements from linked lists

One great advantage of linked lists is — when compared to arrays — the relative ease and speed with which one can add new items to or remove existing items from the middle of a sequence, when one has in hand an iterator to the relevant sequence part.

For example, a linked list can be a good implementation for the sequence of characters in a simple text editor buffer. One would have an iterator corresponding to the current cursor position and the linked-list add and remove methods would support the basic operations of inserting new characters at the cursor as they are typed, or deleting characters in response to the 'delete' key being pressed.

*Paul Jackson, 26th November 2002.*