

Lists and Linked Lists

The concept of a List as conceived for the language LISP has been a model for subsequent list data types. It was a simple abstraction from the familiar informal idea of a list as an open, linear structure in which items could be accessed, added or removed at any point within the structure. The practical motives for such a structure are twofold:

- ease of modification (editing);
- efficiency (cost of adding an item is constant and there is no wastage of memory).

Both these factors are lacking in the array and vector structures in Java. The cost is that in a List we do not have access to elements in constant time (as there is through the index of an array). There is a good deal of confusion in the literature, and in practice, about exactly what constitutes a list. The following specifications show some of this confusion as well as different degrees of abstraction.

Domain: A List is a bounded linear sequence of elements of a given type.

Operations:

Create(L: List, elementType)

pre: none

post: returns an empty list L prepared to hold elements of elementType

First (L: List, e : elementType)
pre: L is not empty
post: e holds the first element of L

Rest (L:List)
pre: L is not empty
post: post-L is the same as pre-L except that the first element of pre-L is removed.

Cons (L:List, e: elementType)
pre: none
post: post-L is the same as pre-L except that e is prepended to pre-L.

Empty (L:List)
pre: none
post: true if and only if L is empty

A convenient way to implement such a list is as linked pairs where the first element of a pair contains an element of the list and the second refers either to another pair, or to a special value indicating there are no more pairs. Such a pair is known as a *node*, and the special value is *null*. This leads to the next stage of implementation which is usually presented as an ADT Linked List with a specification as follows:

Domain: A Linked List is a set of nodes each containing an object and a reference to another node. The nodes and their references form a bounded linear structure. If the list is not empty one node is designated the *current* node.

Operations:

FindFirst(L: LinkedList)

pre: L is not empty

post: the first node is set as current node

FindNext(L: LinkedList)

pre: L is not empty and the last node is not the current node

post: the node following the pre-current node is made the post-current node

Retrieve (L: LinkedList, e: Object)

pre: L is not empty

post: e holds the object in the current node

Update (L:LinkedList, e: Object)

pre: L is not empty

post: current node holds the object e

Insert(L:LinkedList, e:Object)

pre: L is not full

post: a new node containing e is added as first node in L and current is reset to be the new node.

Delete(L:LinkedList)

pre: L is not empty

post: current node is removed from L and its predecessor and successor, if any, become each others predecessor and successor. If post-L is not empty current node is set as the first node.

IsEmpty (L : LinkedList) : boolean

pre:none

post: true if there are no nodes in L, otherwise false

IsFull(L :LinkedList): boolean

pre: none

post: true if the number of nodes in L has reached the bound, otherwise false

IsLast(L : LinkedList) : boolean

pre: L is not empty

post: true if the current node is the last node, otherwise false.

Initialise(L: LinkedList)

pre: none

post: L is set to empty (i.e. IsEmpty is true)

Note can group operations into types: basic ones which depend on the position of current (Insert, Delete, Update and Retrieve), ones which set current (FindFirst and FindNext), and ones for testing preconditions (like IsEmpty, IsFull, IsLast).

An interface declaration in Java allows the specification of a reference type without an implementation. It gives the specification of a set of methods in which the body of each method is replaced by a semi-colon. Interfaces themselves form an inheritance hierarchy but one which is outside the class hierarchy. For example in

Watt&Brown Ch.8 there is an informal outline of a List ADT, followed by a simplified version of the Java interface List. Note that although a List in Java is an 'indexed sequence of objects' and the get() and set() methods allow the user to access elements through their index, the access to the memory location of an element is not direct (as with an array). So the get() method needs to follow links until reaching the element with given index. (See Watt&Brown (W&B) section 8.6 or Barry Cornelius *Understanding Java*, Section 14.3 for more details.)

The general means of traversing a collection such as a list is through an iterator. This works as though the elements were threaded together in index order then the traversal occurs as we unthread them one at a time. W&B Sec.8.4 gives more details and examples of iterators. The iterator() method in the List interface returns an object of type Iterator. The Iterator interface has a sub-interface, suited for lists, ListIterator. It is this interface that has the methods next() and previous() fundamental to a doubly-linked list (DLL). In the Java API LinkedList implements the interface List and uses a DLL. A simpler implementation using a SLL is outlined in W&B Sec.8.6 and a useful summary of Lists in the Java API is in Sec.8.7. It is important to understand the basic principles of insertion and deletion of nodes in a SLL and a DLL. These are well-explained in W&B Ch.4 and many other books. Some examples are shown in the lectures but you also need to study these yourself. An example of typical code to implement a simple form of List (in

the form given by Bailey Ch.6) through an SLL is as follows:

```
public class SLLElement    {

    protected Object  data;
    //value stored here
    protected SLLElement  nextElement;
    // ref to next

    public SLLElement(Object v,
SLLElement next) {
    //post:  constructs a new element
    //with value v followed by next
        data = v;
        nextElement = next;
    }

    .....// missing constructor here

    public SLLElement next ()  {
        //post: returns reference to the
next value in the list

        return  nextElement;
    }

    public void setNext (SLLElement next)
    {
    //post: sets reference to new next
value
        nextElement = next ;
    }
}
```

```

.....//corresponding 'get' and 'set'
//methods for value in this node
}

```

Now we construct a new class which implements the simple (Bailey-style) interface List. The `implements` keyword allows a class to implement (or conform to) one or more interfaces.

```

public SLList implements List {

    protected int count;    //list size
    protected SLLElement head; //ref
to first element

    public SLList() {
//post: constructs an empty list

        head = null;
        count = 0;
    }

    public int size() {
//post: returns the number of elements
//in the list
        return count;
    }

    public boolean isEmpty () {
//post: returns true iff the list is
//empty

        return size() ==0;
    }
}

```

```

    public void addToHead (Object
value)    {
//post: adds value to beginning of
//list

        head = new SLLElement(value,
head);
        count ++;
    }

    public Object removeFromHead()    {
//pre: list is not empty
//post : removes and returns the
//value from beginning of list
        SLLElement temp = head;
        head = head.next;
        count --;
        return temp.value;
    }

```

Further details in W&B Ch.4, Bailey Ch. 6, Collins Ch.6 and Lab2.