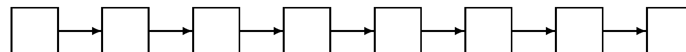
Chapter 12

Linked List Stacks, Queues and Deques

In Chapter 1 we briefly described the idea of a linked list. Recall that the concept is like a chain. Each link in the chain is tied to the next, but to get from the beginning of the chain to the end you must traverse every link.



Linked lists are often termed a *dynamic* data structure, as opposed to the more *static* array data type. The term denotes the observation that a linked list can grow and shrink as elements are added and removed from the collection. An array, on the other hand, is a single block of memory, and the number of elements is fixed when it is created and cannot be changed. This limitation is only slightly mitigated with the Vector data structure. With a Vector it is still true that the size changes only infrequently as elements are added and removed.

The dynamic nature of lists is a property that cuts both ways. While it means that lists are extremely flexible, it also means that they perform more memory operations, as link values are constantly being allocated and deleted as elements are added and removed from the collection. The cost of memory operations must be carefully balanced against the advantages of flexibility. We will see this balance in, for example, a comparison between the use of a vector or a list in the implementation of a stack (Section 12.2).

Linked lists are also often called a *sequential* data structure. By this we mean that, as we have already noted, to access any item we must move sequentially down the links from the start. There is no easy way to randomly access an element from the middle of the collection, as we can do with an array or a Vector.

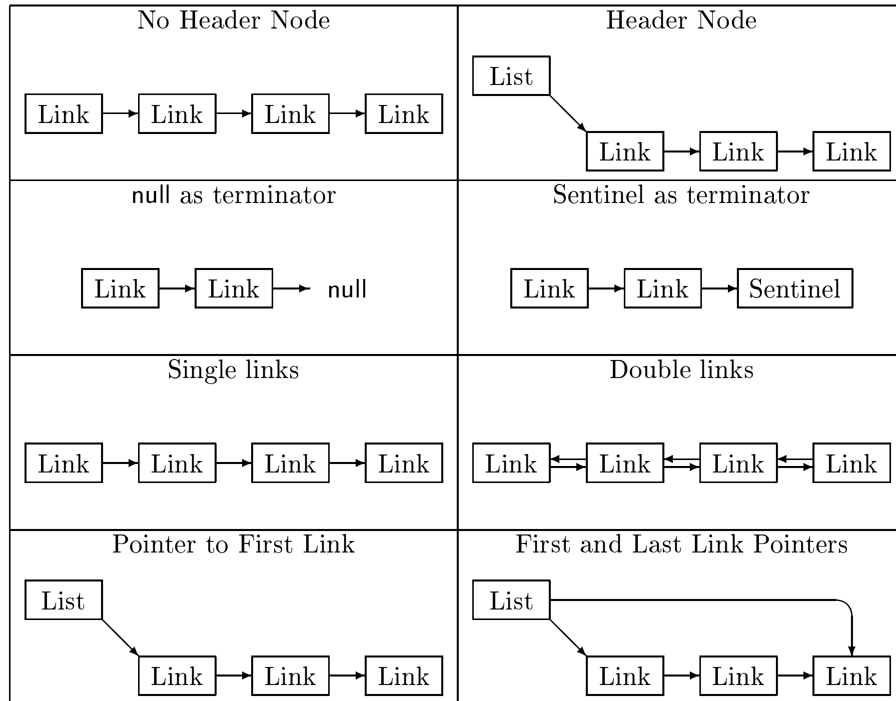


Figure 12.1: Design Issues in the Linked List abstraction

12.1 Varieties of Linked Lists

While the concept of linking is common to all linked lists, there are a number of other issues that can effect a design in different ways. Figure 12.1 lists four of the most important design decisions. For each issue we identify two different solutions. These issues are independent of each other, yielding at least sixteen different designs that can all be called linked list abstractions. Most of these sixteen variations are, in the right circumstances, an appropriate solution to a given problem. The four issues noted in Figure 12.1 can be described as follows:

- **Header object**. Should the list have a special object that maintains the reference to the first link in the list. The advantage of using a header is that there is always an object that "represents" the list. Operations on the list are performed by sending messages to the list header. When a list header is not used the list is simply represented by the first link. An empty list is represented by a null. The disadvantage of having a header node is that there may be two types of objects: the list header and the link

Embedding Links in Data

In older computer programs you will sometimes find a simple but not particularly elegant type of linked list. The idea was to simply place the link field inside a data object. For example, suppose one is manipulating a list of Card objects. One could define these as follows:

```
public class Card {
    Card (int is, int ir) { // initialize a new Card
        suit = is; // suit
        rank = ir; // rank
    }

    public Card link; // next Card
    ...
}
```

A card is then automatically a link in a linked list. The list consists of a reference to the first card, from which you can get to the next card, and so on. The final card will hold a null value in the link field.

While this approach is easy to implement, it should be avoided, for several reasons. It makes your code very rigid, for example you cannot move the Card abstraction to another program in which Cards are not on lists, or need to be on more than one list. It confuses issues by combining the Card specific operations with list operations. And finally, you end up duplicating code that you can more easily write once and reuse by using a standard linked list container.

field. (This is not strictly a requirement, as in some of our abstractions we will use an ordinary link as a header).

- **List Termination**. How should the chain of links be terminated? One approach, the one we have been using in our discussion up to this point, is to simply use the special value null as the link in the last element. Another approach, one that we will see has certain advantages, is to use a designated element as a marker for the end of the list. An element used in this fashion is called a *sentinel*. (A sentinel can be viewed, in some sense, as the opposite of a header).
- **Single or Double Links**. Single links permit the list to be traversed in one direction only. It is sometimes useful to be able to move both forward and backward through a list. By maintaining links in both a forward and a backward fashion we can more

easily move through the list in either direction.

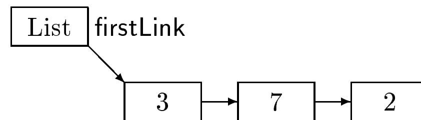
- **First and Last Link References**. A characteristic of the list is that efficient (constant time) access to the elements is only possible at the ends of the list. To access an element in the middle requires traversing all the links in between, and thus will vary in execution time depending upon the number of elements in the list. If we only keep a reference to the first link, then accessing the last element in the list is slowest of all. By keeping references to both the first and last link, we can have fast (constant time) access to both the first and last elements in the list. (A programming project at the end of the chapter explores a third possibility, maintaining a reference to the last element in the list, and not the first).

We will illustrate the range of possibilities by presenting a number of different abstractions that make use of the linked list concept. We should not give the mistaken impression that the sixteen possibilities given by Figure 12.1 exhaust all the variations on the theme of lists. We will be presenting other alternatives in the next few chapters. Simiarly the programming exercises at the end of each of the chapters discuss more varieties of lists.

12.2 A Linked List Stack

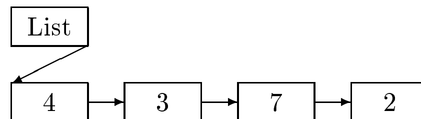
The simplest list style abstraction is the list stack, shown in Figure 12.2. Elements in this class are stored in instances of an internal class called StackLink. Each StackLink object maintains two data fields; an object representing one element in the collection, and a reference to the next element. The value null is used as a terminator on the final link.

The list itself has one data field, named firstLink. This value references the first element in the list, or the value null if the collection is empty.



Note that the value of firstLink is initially null, and this value is tested by the method isEmpty.

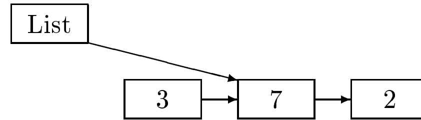
To insert a new value, a new StackLink is created, and the value of firstLink is updated. The fields in the newly created link are set by the constructor, while the change to firstLink occurs as a consequence of the assignment inside the method push.



```
public class ListStack implements Stack {

    private StackLink firstLink = null;

    public boolean isEmpty () {
        return firstLink == null;
    }

    public void push (Object v) {
        firstLink = new StackLink(v, firstLink);
    }

    public synchronized void pop () {
        if (isEmpty())
            throw new NoSuchElementException();
        firstLink = firstLink.link;
    }

    public synchronized Object top () {
        if (isEmpty())
            throw new NoSuchElementException();
        return firstLink.value;
    }

    private class StackLink {
        public StackLink (Object obj, StackLink lnk)
            { value = obj; link = lnk; }

        public StackLink link;
        public Object value;
    }
}
```

Figure 12.2: Single Link Lists can implement a Stack

To remove a value, value of firstLink is set to the next element in sequence. The garbage collection system will eventually recover the deleted link, which is no longer being referenced by any value.



There are no loops in any of the methods associated with this abstraction, and therefore all operations are constant, $O(1)$. Compare this with our earlier stack abstraction, the VectorStack presented in Chapter 7.

	VectorStack	ListStack
push	$O(1)$ expected, $O(n)$ worst case	$O(1)$ always
pop	$O(1)$	$O(1)$
top	$O(1)$	$O(1)$

It might appear at first that the list stack has the advantage, as it is uniformly constant time. However, the vector stack only rarely exhibits its $O(n)$ time behavior, and it is only at this point that significant memory allocation occurs. The linked list version, on the other hand, must allocate new memory (using the new StackLink operator) each time a value is inserted. It is difficult to predict in advance how these two factors will balance each other. Empirical measurements can help resolve the question.

There are two different experiments that one could imagine. The first experiment would simply push a value onto the stack n times, then pop the stack until it became empty. In order to obtain reliable and measurable figures, it is useful to repeat this task a number of times, as in the following:

```
for (int i = 0; i < 100; i++) {
    for (int j = 0; j < n; j++)
        stk.push(value);
    while (! stk.isEmpty())
        stk.pop();
}
```

This experiment would measure the cost of the push and pop operations for ever increasing stack sizes. This would tend to favor the list, as the vector stack is being forced to constantly reallocate new space.

A second experiment would invert the two loops, performing a constant number of push and pop operations, but for ever increasing number of steps:

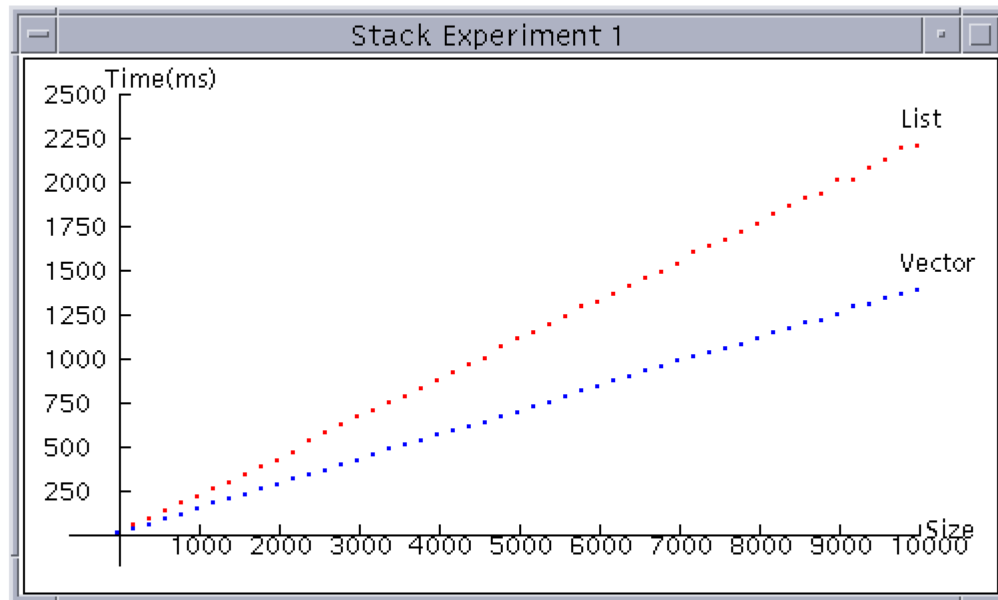


Figure 12.3: Timings of Stack Operations

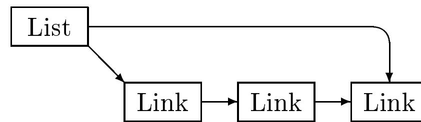
```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < 100; j++)
        stk.push(value);
    while (! stk.isEmpty())
        stk.pop();
}
```

In this case the stack size remains small, and so the effect of vector reallocation is reduced. As a consequence we would hope to get a more reliable measure of the relative costs of the stack operations.

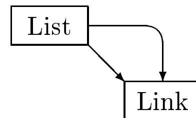
The results of the first experiment are shown in Figure 12.3. From this we see that the stack operations provided by the ListStack appear to be slightly faster than those provided by the VectorStack, although the difference is small, and in practice either implementation should work. Deciding which data structure to use should then be based on other factors. The results of the second experiment are similar, and are not shown here.

12.3 A Linked List Queue

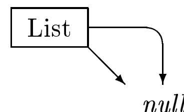
Just as it was difficult to implement a Queue using only a Vector, it is difficult to implement a Queue using a singly linked list with only a reference to the first element. The reason is also similar. The queue requires manipulation of both ends of the collection, and the first link only provides access to the front. This problem is easily solved by maintaining links to both the first and the *last* element:



The implementation of a queue based on these observations is shown in Figure 12.4. The need to manage both the first and last links complicates the code slightly. For example, when a value is inserted into an empty list it is important that both the values firstLink and lastLink are set to the new object.



Similarly, when the last element is removed, it is important that both firstLink *and* lastLink be set to null.



As was true with the stack abstraction, no operations here involve loops, and thus all are constant, or $O(1)$ time. This compares favorably with the array based queue, which had constant expected time, although could occasionally require $O(n)$ time for an insertion should reallocation of the buffer be necessary. Since the asympototic execution times are similar, other factors such as memory allocation will become important. As with the stack comparison, in most cases the vector class will tend to have a slight advantage in execution time, since it performs fewer memory allocations.

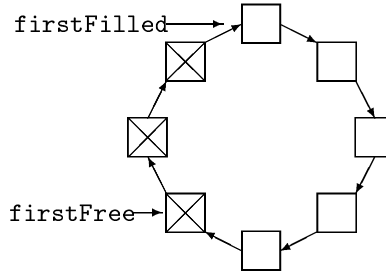
Programming Project 1 at the end of the chapter explores an alternative design of a linked list queue, one that maintains only a single pointer to the end of the queue.

```
public class ListQueue implements Queue {

    private QueueLink firstLink = null;
    private QueueLink lastLink = null;

    public boolean isEmpty() {
        return firstLink == null;
    }

    public synchronized void enqueue (Object newElement) {
        lastLink = new QueueLink(newElement, lastLink);
        if (firstLink == null)
            firstLink = lastLink;
    }

    public synchronized Object frontElement () {
        if (isEmpty())
            throw new NoSuchElementException();
        return firstLink.value;
    }

    public synchronized void dequeue () {
        if (isEmpty())
            throw new NoSuchElementException();
        firstLink = firstLink.link;
        if (firstLink == null)
            lastLink = null;
    }

    private class QueueLink {
        public QueueLink (Object obj, QueueLink lnk)
            { value = obj; link = lnk; }

        public QueueLink link;
        public Object value;
    }
};
```

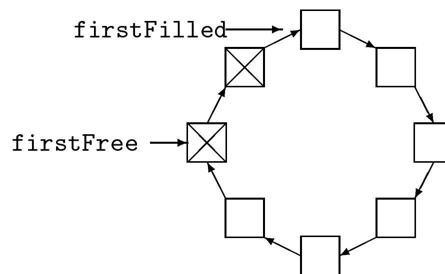
Figure 12.4: A Linked List Based Queue Abstraction

12.3.1 The Circular Queue *

Another implementation technique for the Queue abstraction is frequently encountered, particularly in the design of operating systems (where a queue might be used, for example, to hold a sequence of jobs waiting to be serviced by a printer). This technique is termed a *circular queue*, or sometimes a *ring buffer queue*. This basic idea is similar to the linked list implementation of a queue, only the list is circular, and so has no beginning and no end. As the list is traversed in only one direction, only single links are necessary. And, unlike the ListQueue abstraction, nodes in the list are not recovered when elements are removed. Instead, they remain in the list and are reused as new values are inserted. This reduces the cost of memory management involved in link allocation and deallocation, but may tie up memory in unused links.

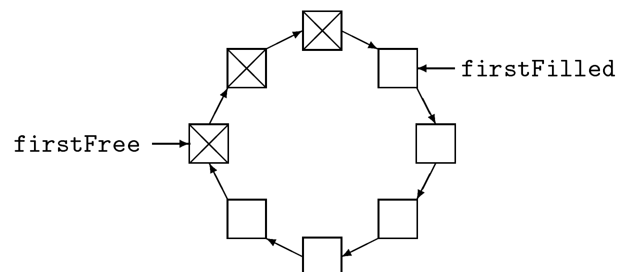


Two references point into the list of elements. One reference denotes the first available empty location, and one the first filled location. When an element is inserted, the first available empty position is advanced:



Similarly, when a value is removed the reference to the first filled location is advanced:

*Section headings followed by an asterisk indicate optional material.



As with all queue abstractions, an exception will be thrown if an attempt is made to remove an element from an empty queue (that is, when the two references point to the same element). The other extreme occurs if an attempt is made to insert an element into a completely full queue. For reasons we will explore in the following sequence of questions, there must always be at least one unused node in the ring buffer elements.

Figure 12.5 gives an implementation of a queue that illustrates the circular buffer technique. The following series of questions is intended to help you better understand the cicrcular buffer implementation, and provide an informal proof of correctness.

1. Draw a representation of the ring buffer as it is first created by the constructor. Verify that the method isEmpty will return true.

2. Trace the execution of the method enqueue, and verify that if we ignore the execution time cost of allocating the new node, the cost of all other operations is bounded by a constant, therefore addition of a new value is $O(1)$. Then, draw a representation of the ring buffer after one element has been inserted. Verify that the method isEmpty will now return false.

3. Add a second element, and once more draw a representation of the resulting structure.

4. To understand why it is necessary to always maintain an unused node in the ring buffer, imagine changing the test in the method addList to the following:
   ```
   if (firstFree == firstFilled)
   ```
   this would have the effect of forcing the addition of a new node when the structure was completely filled, rather than when it was almost filled. Simulate the addition of another new element. Now simulate the execution of the method isEmpty. What do you observe? For an alternative solution to this problem, see Programming Project 3.

5. The preceding questions have examined the validity of the method enqueue when the queue was entirely empty, and when it was almost full. One more test is necessary. Imagine that several elements have been added and removed, so that the buffer contains six nodes, but only three are filled. Simulate the addition of a fourth value, and draw the representation of the state of the queue before and after the insertion.

```
public class CircularQueue implements Queue {

    CircularQueue () {
        firstFilled = firstFree = new CircularLink(null);
        firstFree.next = firstFree;
    }

    private CircularLink firstFree, firstFilled;

    public boolean isEmpty () { return firstFilled == firstFree; }

    public synchronized void enqueue (Object val) {
        if (firstFree.next == firstFilled)
            firstFree.next = new CircularLink(firstFree.next);
        firstFree.value = val;
        firstFree = firstFree.next;
    }

    public synchronized Object frontElement () {
        if (firstFilled == firstFree) throw new NoSuchElementException();
        return firstFilled.value;
    }

    public synchronized void dequeue () {
        if (firstFilled == firstFree) throw new NoSuchElementException();
        firstFilled = firstFilled.next;
    }

    private class CircularLink {
        public CircularLink (CircularLink n) { next = n; }

        public Object value;
        public CircularLink next;
    }
}
```

Figure 12.5: Implementation of the class CircularQueue

6. Simulate the method size on the structure left at the end of the last question, and verify that it returns the correct value.

7. Simulate the execution of the methods frontElement and dequeue on the queue left after the previous question. What value is yielded? What is the resulting structure of the ring buffer queue? Verify that in all situations the execution time of the method will be bounded, and thus removal is an $O(1)$ time operation.

8. What will happen if an attempt is made to access the first value in an empty queue?

9. Consider a ring buffer with five nodes, only one of which is filled. Simulate the execution of the method dequeue. Draw a representation of the structure both before and after the removal, and verify that the method isEmpty will return true.

10. To understand why the enqueue must be synchronized, simulate the following sequence of events that could potentially occur were two threads permitted to execute the method at the same time. Draw a picture of the state of the buffer after each step. Explain why the ring buffer queue is left in an invalid state.

    (a) Two calls on enqueue are made on the same almost full queue. The first invocation of the method gets as far as allocating a new node before it is suspended.

    (b) The second invocation completes the if statement, including allocating a new node and assigning the value to firstFree.next, before it is suspended.

    (c) The first method then runs to completion.

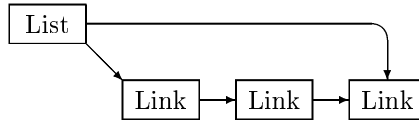
    (d) the second method runs to completion.

Experiment: Evaluate Circular Queue Execution Times

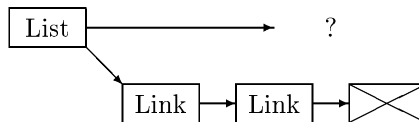
Under what conditions is the ring buffer implementation more efficient than the other two implementation techniques that have been described? Test the ring buffer implementation using the two different distributions of operations described in the previous section, and compare the running time to that of the ArrayQueue and the ListQueue.

12.4 A Linked List Deque

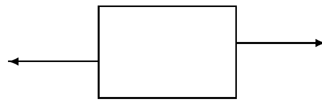
Maintaining a link to the last element is all that is necessary in order to allow elements to be *added* to the end of a list, but if we want to allow elements to be *removed* from the end of the list it is more complicated:



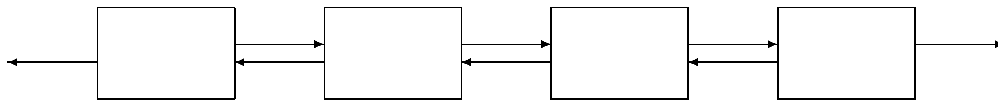
Having removed the last value, how do we discover the new value to be assigned to lastLink without looping over the entire collection of values?



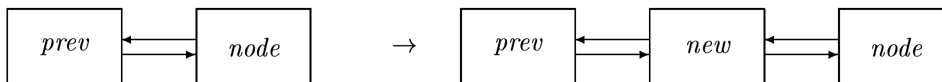
To overcome this problem and allow a linked list implementation of the Deque abstraction it is necesary to maintain links that point both to the *next* element and to the *previous*:



When we combine a series of DequeLink objects together, the result is a sequence of links chained in both directions:



Inserting a value into a doubly linked list is more complicated than with a singly linked list. This is because the pointers in *both* neighbors must be changed so that they reference the new value:



Similarly, when a link is removed the references in both neighbors must be updated:

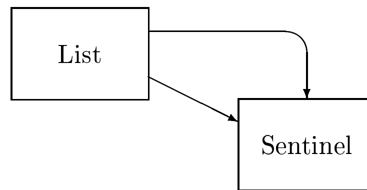
Like the ListQueue, instances of the class ListDeque will maintain a header node, pointing to the first link, and a reference to the final node. Instead of terminating our linked lists with a null value, we will use this opportunity to illustrate the utility of a programming

```
public class ListDeque implements Deque {

    public ListDeque () { firstLink = lastLink; }

    private DequeLink firstLink;
    private final DequeLink lastLink = new DequeLink(null, null, null);

    public boolean isEmpty ()
        { return firstLink == lastLink; }

    public synchronized void addFront (Object newElement)
        { firstLink.insert(newElement); }

    public synchronized void addBack (Object newElement)
        { lastLink.insert(newElement); }

    public synchronized void removeFront () {
        if (isEmpty())
            throw new NoSuchElementException();
        firstLink.remove();
    }

    public synchronized void removeback () {
        if (isEmpty())
            throw new NoSuchElementException();
        lastLink.prev.remove();
    }

    public synchronized Object frontElement () {
        if (isEmpty())
            throw new NoSuchElementException();
        return firstLink.value;
    }

    public synchronized Object backElement () {
        if (isEmpty())
            throw new NoSuchElementException();
        return lastLink.prev.value;
    }

    private class DequeLink { ... }
}
```

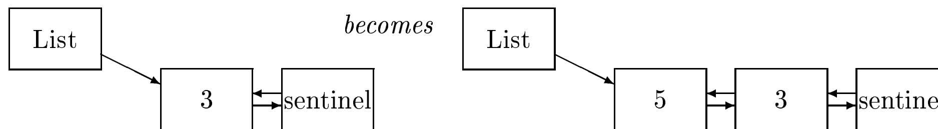
Figure 12.6: Class Description for ListDeque

technique known as a *sentinel*. A sentinel value is a marker that indicates the end of the list. It is created as part of the initialization of the list, and is never deleted. The ListDeque abstraction is shown in Figure 12.6.

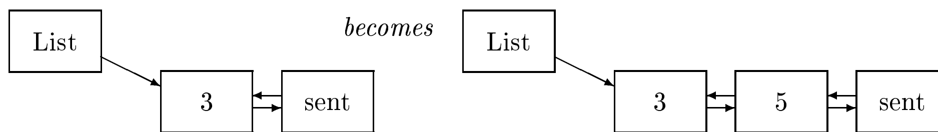
Notice that the sentinel field lastLink can be declared using the final modifier, which indicates that once assigned it will never be reassigned. The firstLink field, on the other hand, does not carry this modifier since it will change frequently. Both links will be instances of the class inner class DequeLink. We will examine this class in a moment. An instance of this class, one with a null value, will be used as a sentinel to mark the end of the list. The use of the sentinel means that the fields firstLink and the sentinel lastLink will never be null, as in an empty list they will both refer to the sentinel:



Because we are using a sentinel value, our list will always have at least one position. This makes insertions particularly easy. To insert at the front, we merely use the DequeLink.insert method on the first link:



On the other end, to insert at the end, we use the DequeLink.insert method on the sentinel:



Removals are similarly handled by passing a message to either the first link, or to the link immediately preceding the sentinel.

All of this is possible because of the way in which the internal class DequeLink implements the methods for insertion and removal:

```
private class DequeLink {
    public DequeLink (Object v, DequeLink n, DequeLink p)
        { value = v; next = n; prev = p; }
```

The Sentinel Design Pattern

name Sentinel. (From the idea of "one that keeps guard")

forces A sentinel can be used whenever there is a sequence of indefinite length, and it is necessary to know when the end has been reached.

synopsis A sentinel is used to mark the end of a sequence. A simple example is reading values from an input file. A special value, for example -1, might be used to mark the end of the input. In Chapter 6 we used a blank word as sentinel to mark the end of the vector in the word frequency example. Sentinels can also be used in recursive data structures, for example in a linked list. Here it is necessary to know when we have reached the last element. A special value, for example `null`, marks the last entry. The sentinel is in this situation acting as the base case for the recursive definition.

A list that uses a sentinel is never completely empty. This eliminates special code that would otherwise need to handle an empty list situation. As we saw in the word frequency program in Chapter 6, sentinels can also make it easier to detect the end of a sequence, and to ensure that all elements are processed.

counterforces Any traversal of the data structure must remember to check for the sentinel value. These explicit checks can sometimes be avoided by converting a passive sentinel into an active sentinel.

related patterns The term sentinel usually refers to a passive data value. In object-oriented languages an alternative is the Active Sentinel (see page 451).

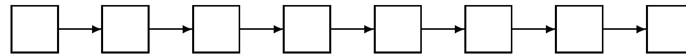
```
public Object value;
public DequeLink next, prev;

public void insert (Object newValue) {
    DequeLink newLink = new DequeLink(newValue, this, prev);
    newLink.next = this;
    newLink.prev = prev;
    if (prev == null)
        firstLink = newLink;
    else
        prev.next = newLink;
}
```

```
        public void remove () {
            if (prev == null)
                firstLink = next;
            else
                prev.next = next;
            next.prev = prev;
        }
    }
```

The link classes are careful to maintain the previous and next pointers correctly, and to update the value of firstLink should the first list element be modified.

12.5 Chapter Summary

A linked list is a dynamic data structure that will grow and shrink in size as elements are added and removed. Values in a linked list are stored in links; small objects that each maintain one element and a reference to the next link.



A linked list is also a sequential data structure. This means to access any element it is necessary to move down a chain of links starting from the first. The use of link fields allows a very flexibile data structure, but this must be weighed against the cost of memory allocation and recovery as link fields are added and deleted.

In this chapter we have shown how link lists can be used to implement the stack, queue, and deque interfaces. They can also be used to implement most of the other classic data structure abstractions, something we will investigate in the programming assignments and in subsequent chapters.

Key Concepts

- linked list
- dynamic data type
- single link
- double link
- header
- sentinel
- circular queue

Further Information

Variations on linked list structures are discussed by Knuth in volume 1 of his three-volume set [Knuth 97]. The few examples in this chapter do not come close to exhausting the range of possible variations on the theme of linked lists. Some other abstractions will be examined later in this book, many others are discussed by other authors. (should cite some)

Study Questions

1. What is the key characteristic common to all types of linked list?
2. What is the difference between a single link and a double link? What are the advantages of double links? What are the disadvantages?
3. Although sometimes used, why in general is it not a good idea to create a linked list by storing a link field directly in a data object?
4. What does it mean to say a list uses a header object?
5. What is the advantage of maintaining a reference to the last link in a linked list, as well as to the first link?
6. What is a circular queue?
7. What is a sentinel?
8. How does the use of a sentinel simplify the implementation of the ListDeque data structure?

Exercises

1. Using invariants, provide a proof of correctness for each of the methods implemented by the ListStack.
2. Using invariants, provide a proof of correctness for each of the methods implemented by the ListQueue.
3. To see why why the pop method in class ListStack must be synchronized, imagine the modifier is removed and described what would happen in the following situation. A stack contains a single element. One thread starts a pop operation, but is halted following the if statement. A second thread then performs a pop on the same stack, and runs to completion. The first thread then resumes.
4. Describe a similar scenario that illustrates why the top method in ListStack must be synchronized.

5. Describe scenarios that illustrate why the enqueue and dequeue methods in ListQueue must be synchronized.

6. Rewrite the ListDeque using a null as the terminator for links. What special cases must now be handled that are not treated as special cases in the code given in the text?

7. Fill in the asymptotic execution times for the following operations provided by the class ListDeque. For each, indicate whether the execution time will always be required, is an expected (or average) time, or is worst case.

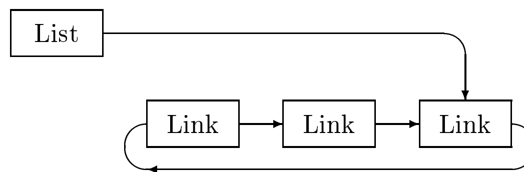
isEmpty	$O(\quad)$
enqueue	$O(\quad)$
frontElement	$O(\quad)$
dequeue	$O(\quad)$

8. Fill in the asymptotic execution times for the following operations provided by the class CircularQueue. For each, indicate whether the execution time will always be required, is an expected (or average) time, or is worst case.

isEmpty	$O(\quad)$
enqueue	$O(\quad)$
frontElement	$O(\quad)$
dequeue	$O(\quad)$

Programming Projects

1. As we noted in Chapter 7, an *output-restricted* queue, or *scroll*, allows insertions from either the front or back, but allows removals only from the front. A simple way to implement a scroll is as a singly-linked list that maintains a reference to the *last* link, instead of the first, and to use the link on the final element to point back to the first, in the fashion of a circular queue:



Insertions to the front of the queue can be made by adding a new node following the last link. Insertions to the end of the queue are the same, except the reference to the

last link is changed. Removals to the front are simply change the link field of the final link. Implement the Scroll abstraction using these ideas. Why is it difficult to extend this technique so that it implements the complete Deque interface?

2. Add the stack and queue interfaces to the data structure you developed in the previous project, and empirically compare the running time of these operations to that of the ListStack and the ListQueue.

3. Rewrite the CircularQueue class so that it maintains a data member count that stores the number of elements in the queue. Show that the requirement that the ring buffer queue always maintain an empty position can now be removed.

   By performing a large number of insertions and removals, experimentally evaluate the effect on execution time of the this modification. Does the cost of keeping track of the queue size substantially alter the time to perform these operations?

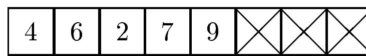
4. Yet another alternative solution to the problem explored in the previous project is to maintain a boolean data member named full. This value will be false except when the queue is entirely full. Implement a version of the CircularQueue using this approach. Show how this approach permits the queue to be completely full without encountering the problem described in Section 12.3.1. What is the new implementation of the member function isEmpty?

   Experimentally compare the execution time of the queue modification described in the previous question to the original circular queue described in Section 12.3.1.

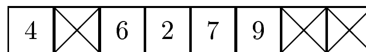
Chapter 13

Sequential Containers

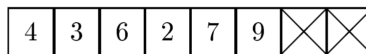
In Chapter 12 we saw how simple linked lists could be used to implement stacks, queues, deques, and other ordered containers. However, the concept of linking is much more general, and can be employed in the implementation of all types of containers. The most important difference between linked lists and the earlier vector based abstractions does not even arize when lists are used for ordered containers. This difference becomes clear when we consider the task of inserting a value into the *middle* of a collection, a situation that does not occur in the stack. Recall that for a vector, this process required several steps. To insert a value into the second position of the following vector:



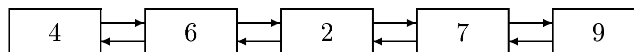
A "hole" must be first opened up to make space for the new element:



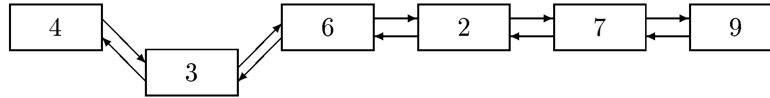
Before the element could be inserted:



The creation of this hole required moving each of the succeeding elements in the vector, a process that could require, in the worst case, $O(n)$ steps. To insert a value into the middle of the corresponding linked list is much easier. The list:



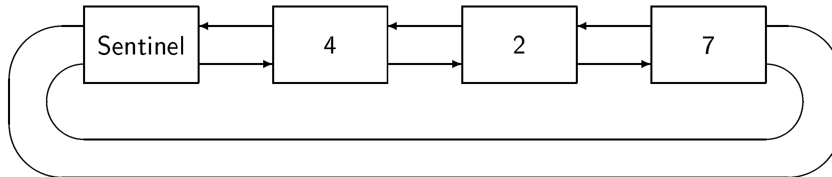
becomes



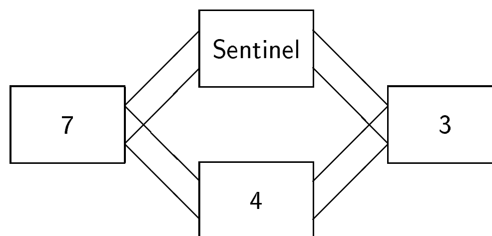
Notice how the insertion process required making changes to only the two adjacent elements. This is true regardless of the size of the list. Thus, we can say that the task of insertion is a constant time operation. (We need to be careful here and note that it is only the insertion itself that is constant time. Finding the location in the list where an element should be inserted may, depending upon the nature of the list, require a linear traversal, and hence may still be $O(n)$.)

13.1 A Linked List Bag

As we saw in the previous chapter, the most general type of list structure requires both forward and backward links. The ListBag, shown in Figure 13.1, is yet another variation on the theme of linking and sentinels. In this case we have constructed a list using doubly-linked node and a sentinel value that wraps back on itself in both the forward and backward directions:



Another way to view this is as a circularly linked list:



By linking the sentinel in this fashion we never have a completely empty list (the list may be empty of values, but there is still a sentinel), and furthermore all links have both a forward and backward reference. The latter property means we eliminate the special cases that occurred in some of the earlier abstractions when we reached the end of the list.

```
public class ListBag implements Bag {

    public ListBag ()
        { sentinel.next = sentinel; sentinel.prev = sentinel; }

    private final DoubleLink sentinel = new DoubleLink(null, null, null);
    private int elementCount = 0;

    public synchronized void addElement (Object newElement)
        { sentinel.insert(newElement); }

    public synchronized boolean containsElement (Object testElement) {
        DoubleLink lnk = sentinel.next;
        while (lnk != sentinel) {
            if (testElement.equals(lnk.value))
                return true;
            lnk = lnk.next;
        }
        return false;
    }

    public Enumeration elements () { return new ListBagEnumeration(); }

    public synchronized boolean removeElement (Object oldElement) {
        DoubleLink lnk = sentinel.next;
        while (lnk != sentinel) {
            if (oldElement.equals(lnk.value)) {
                lnk.remove();
                return true;
                }
            lnk = lnk.next;
        }
        return false;
    }

    public int size () { return elementCount; }

    private class DoubleLink { ... }

    private class ListBagEnumeration implements ListEnumeration { ... }
}
```

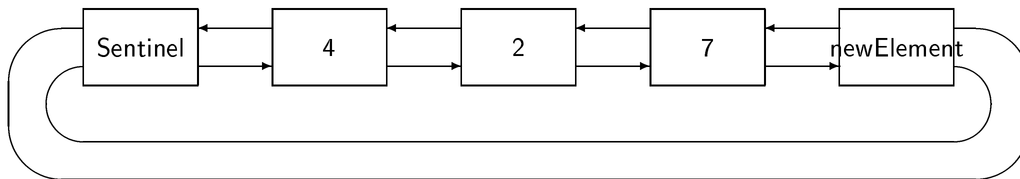
Figure 13.1: The class ListBag

```
protected class DoubleLink {
    public DoubleLink (Object v, DoubleLink n, DoubleLink p)
        { value = v; next = n; prev = p; }

    public Object value;
    public DoubleLink next, prev;

    public void insert (Object newValue) {
        DoubleLink newLink = new DoubleLink(newValue, this, prev);
        prev.next = newLink;
        prev = newLink;
        elementCount++;
    }

    public void remove () {
        prev.next = next;
        next.prev = prev;
        elementCount--;
    }
}
```

Figure 13.2: The Inner Class DoubleLink

Because of the way we are using the sentinel, it marks both the front *and* the back of the list.

The method addElement will invoke the method insert provided by the inner class DoubleLink. This method will ensure that links are properly maintained when the new element in inserted. It will place the value at the end of the list:



The methods containsElement and removeElement illustrate a loop that will cycle through the elements of a list. In the case of removeElement, it uses a method provided by the inner class DoubleLink to actually perform the removal.

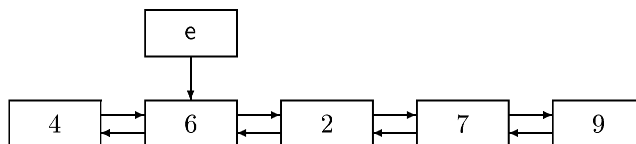
The inner class DoubleLink is shown in Figure 13.2. This class is a variation on the link classes we developed in the previous chapter. As we noted earlier, the use of the sentinel means that every link contains both a valid forward and backward pointer, and we can

therefore eliminate the special code that was formerly necessary to handle the situation where the first node in a list did not have a forward link. The methods insert and remove are careful to ensure the elementCount field is properly updated.

13.1.1 List Enumerators

At we noted at the beginning of this chapter, a distinguishing characteristic of the linked list abstraction is the ability to easily add and remove elements from the middle of a collection. But how do we describe an arbitrary location in the middle of a list? Most of the time, we access the elements of a list by means of an enumerator:

```
Enumeration e = lst.elements();
while (e.hasMoreElements()) {
    Object obj = e.nextElement();
    ...
}
```



The enumerator is acting as our link into the middle of the list. In order to make it easy to perform insertions and removals, we extend the Enumeration interface so as to allow insertions and deletions using methods associated with an enumerator, rather than with the list itself. This extended interface is named ListEnumeration:[1]

```
interface ListEnumeration extends Enumeration  {
    public void add (Object newElement);
    public void remove ();
    public void set (Object newValue);
}
```

The ListEnumeration class extends Enumeration by adding three new methods. The first adds a new value to the collection immediately prior to the current element. The second method removes the current element, while the third method changes the value associated

[1]While the Enumeration iterface is taken directly from the Sun library, the ListEnumeration extension is particular to the data structure containers for this book. The Sun Java library does provide a class ListIterator with similar functionality, however the ListIterator class is much more complex than suits our purpose here.

with the current link. Notice that the method elements in the ListBag enumeration actually returns a value of type ListEnumeration, although it is declared as only returning an Enumeration. This type of situation is sometimes known as a *software factory* (see sidebar).

A private inner class named ListBagEnumeration implements the ListEnumeration interface within the the ListBag container. Because of the way that we use a sentinel and the functionality provided by the class DoubleLink, no method takes more than one or two statements.

```
private class ListBagEnumeration implements ListEnumeration {
    private DoubleLink lnk = sentinel;

    public boolean hasMoreElements ()
        { lnk = lnk.next; return lnk != sentinel; }

    public Object nextElement ()
        { return lnk.value; }

    public void add (Object newElement)
        { lnk.insert(newElement); }

    public void remove ()
        { lnk.remove(); }

    public void set (Object newValue)
        { lnk.value = newValue; }
}
```

Because the sentinel marks both the front and the back of the list, an addition to an exhausted enumerator (that is, one that has finished iterating over all the elements in the collection) will place the new element at the end of a list.

We explore the working of some of these methods in more detail in exercises at the end of the chapter.

13.2 List Insertion Sort and List Quick Sort

We can illustrate the use of the ListBag data type by rewriting some of the sorting algorithms we introduced in earlier chapters, altering them to work with lists. The first example we will consider is insertion sort, which we explored back in Chapter 4. Recall that an insertion sort examined each element in turn, performing a linear traversal of a collection of already sorted values in order to place the new element into its correct location. Linear traversals

and insertions into the middle of a collection seem ideally suited for linked lists, and so it is not surprizing that the sorting algorithm is not difficult to recast in this new form:

```
public class ListInsertionSort {
    public ListInsertionSort (Comparator t) { test = t; }
    private Comparator test;

    public Bag sort (Bag s) {
        Bag newList = new ListBag();
        Enumeration e = s.elements();
        while (e.hasMoreElements()) {
            Object a = e.nextElement();
            ListEnumeration f = (ListEnumeration) newList.elements();
            while (f.hasMoreElements() &&
                (test.compare(a, f.nextElement()) > 0)) ; // do nothing
            f.add(a); // insert at new location
        }
        return newList;
    }
}
```

A new ListBag is constructed to hold the sorted collection. The outer while loop is examining each element in turn. The inner while loop finds the location to place the new value in the sorted collection. Note that all the important actions for this loop can be written in the test, and the body of the loop does nothing. The inner loop exits either when the end of the collection is reached (indicating the new element should be placed at the end) or when a value that is larger than the new element is found. In either case, the ListEnumeration value in f is then the correct location to place the new value.

The QuickSort algorithm introduced in Chapter 10 is another sorting algorithm that can be easily adapted for linked lists. Recall that a quick sort operated by selecting a value as pivot, then dividing a list into two partitions, the first consisting of those values less than or equal to the pivot, and the second, those elements larger than the pivot. These two collections were then recursively sorted, and the resulting values placed back together. Recast to use a ListBag, this algorithm looks as follows:

```
public class ListQuicksort {
    public ListQuicksort (Comparator t) { test = t; }
    private Comparator test;

    public Bag sort (Bag s) {
        Enumeration e = s.elements();
        Object pivot;
```

```
        if (e.hasMoreElements())
            pivot = e.nextElement();
        else
            return s;
            // first partition
        Bag one = new ListBag();
        Bag two = new ListBag();
        while (e.hasMoreElements()) {
            Object a = e.nextElement();
            if (test.compare(a, pivot) < 0)
                one.addElement(a);
            else
                two.addElement(a);
        }
            // then recursively sort
        one = sort(one);
        two = sort(two);
            // then put back together
        one.addElement(pivot);
        e = two.elements();
        while (e.hasMoreElements())
            one.addElement(e.nextElement());
        return one;
    }
}
```

Following the recursive calls the pivot is appended to the first list (remember that elements get added to the end of a collection), and all the values in the second list are similarly appended. The final sorted list is then contained in the first list.

13.3 Application–Infinite Precision Integers

Some numerical applications, such as dealing with the national debt, or factoring 200-digit numbers as part of a cryptographic package, require manipulating integer quantities that are larger than will fit in a normal Java integer. For these we need a library that will represent infinite precision integers. In this example application we will present the beginning of such a library, and outline how it can be extended.

The basic idea will be to represent a large integer as a list of smaller numbers, numbers which *can* be maintained in a simple `int` form. In our example we will divide numbers by units of 1000. Each link in our list will be a positive number that is less than 1000. Because both addition and subtraction operate from the least significant (smallest) digit to

the largest, it will be convenient to store the numbers in our list in a backwards fashion. A number such as 72345876, for example, will be held in a list of three nodes, as follows:

We will begin with a class that can hold large positive integers. The constructor for the class and the procedure to convert our value back into a string can be written as follows:

```
public class LargePositiveInteger implements Comparable {
    public LargePositiveInteger (int start) {
        while (start > 0) {
            digits.addElement(new Integer(start%1000));
            start = start / 1000;
        }
    }

    private LargePositiveInteger (ListBag d) { digits = d; }

    private ListBag digits = new ListBag();

    public String toString () {
        if (digits.size() == 0) return "0";
        String result = "";
        Enumeration e = digits.elements();
        while (e.hasMoreElements()) {
                // make each block exactly three digits long
            String block = "000" + e.nextElement();
            result = block.substring(block.length()-3) + result;
            }
        return result;
    }

    public LargePositiveInteger add (LargePositiveInteger right) { ... }

    public int compareTo (Object right) {  ... }

    public LargePositiveInteger sub (LargePositiveInteger right) { ... }
}
```

One interesting feature of this class is the inclusion of a private constructor. By declaring the constructor that takes a ListBag as argument as private we ensure that it can be used

only from within the class, specifically from within the add method we will describe shortly. The other constructor, one that takes a normal integer as argument, is declared public.

We begin with the algorithm for adding two integers. The technique basically follows the approach you learned in grade school, except using units of 1000 rather than units of 10. At each stage we are summing two groups of numbers that are each less than 1000, and adding in a *carry*. The result is a number that may or may not be larger than 1000. We divide the result by 1000 to get the new carry that will be used in the next block, and the remainder after dividing by 1000 is the result for this position.

0	1	0	*carry*
242	476	342	*left*
31	332	792	*right*
273	809	134	*result*
0	0	1	*new carry*

One complication comes from the problem that one list may be shorter than the second. This problem is easily addressed by appending zeros to one or the other until they are the same length.

```
private void makeSameSize (LargePositiveInteger right) {
    while (digits.size() < right.digits.size())
        digits.addElement(new Integer(0));
    while (right.digits.size() < digits.size())
        right.digits.addElement(new Integer(0));
}
```

The implementation of the addition algorithm is then straightforward:

```
public LargePositiveInteger add (LargePositiveInteger right) {
    makeSameSize(right);
    Enumeration le = digits.elements();
    Enumeration re = right.digits.elements();
    ListBag result = new ListBag();
    int sum = 0;
    int carry = 0;
    while (le.hasMoreElements() && re.hasMoreElements()) {
        Integer leftDigit = (Integer) le.nextElement();
        Integer rightDigit = (Integer) re.nextElement();
        sum = leftDigit.intValue() + rightDigit.intValue() + carry;
        carry = sum / 1000;
        result.addElement(new Integer(sum % 1000));
    }
```

```
    if (carry != 0)
        result.addElement(new Integer(carry));
    return new LargePositiveInteger(result);
}
```

To decide the relative order of two numbers is similar. We run down the two lists in parallel, comparing elements item by item. A flag named test will initially mark that the two values are equal, and will only be modified when we find corresponding values that are not equal. When we are finished, the *last* value in which the comparison was changed will determine the result:

```
public int compareTo (Object r) {
    LargePositiveInteger right = (LargePositiveInteger) r;
    makeSameSize(right);
    int test = 0;
    Enumeration le = digits.elements();
    Enumeration re = right.digits.elements();
    while (le.hasMoreElements() && re.hasMoreElements()) {
        Integer leftDigit = (Integer) le.nextElement();
        Integer rightDigit = (Integer) re.nextElement();
        int ld = leftDigit.intValue();
        int rd = rightDigit.intValue();
        if (ld < rd) test = -1;
        if (ld > rd) test = 1;
    }
    return test;
}
```

The subtraction code only handles the case where the right argument is less than the left, and hence the result is positive. The algorithm is similar to the addition algorithm:

```
public LargePositiveInteger sub (LargePositiveInteger right) {
        // pre: assumes that the argument is smaller than our value
    makeSameSize(right);
    Enumeration le = digits.elements();
    Enumeration re = right.digits.elements();
    ListBag result = new ListBag();
    int borrow = 0;
    while (le.hasMoreElements() && re.hasMoreElements()) {
        Integer leftDigit = (Integer) le.nextElement();
        Integer rightDigit = (Integer) re.nextElement();
        int ld = leftDigit.intValue();
```

```
        int rd = rightDigit.intValue();
        ld = ld - borrow;
        borrow = 0;
        if (ld < rd) {
            ld = ld + 1000;
            borrow = 1;
            }
        result.addElement(new Integer(ld - rd));
    }
    return new LargePositiveInteger(result);
}
```

What is surprising is that these three procedures are all that is necessary to implement addition and subtraction for all infinite precision numbers. To create a number package, numbers will be represented by an instance of class LargeInteger in sign/magnitude form. An internal boolean will keep track of the sign, while a LargePositiveInteger will hold the magnitude. Adding two positive integers we have seen, as well as subtracting a smaller number from a larger. Subtracting a larger positive number from a smaller is the same as inverting the subtraction and changing the sign on the result. Adding a positive number to a negative one is the same as subtracting the second from the first, and so on. In this fashion all the operations can be provided.

13.4 List-Based Maps

A pair of parallel ListBags can be used to implement a list-based Map abstraction. This is shown in Figure 13.3. The method containsKey is here used for two purposes; while performing the test to determine if the argument is a valid key in the collection, it also sets the values of a list enumeration for the key collection and another list enumeration for the value collection. This side effect is then used by the methods get, set, and removeKey.

Of course, an association-style list-based map could be created by surrounding a ListBag with a MapAdapter:

```
Map newMap = new MapAdapter(new ListBag());
```

Comparing the relative speeds of these two techniques is left as an exercise.

13.4.1 The MultiMap adapter

As preparation for our next example application, we first describe yet another form of map adapter, termed a MultiMap. Like the MapAdapter, the MultiMap, shown in Figure 13.4,

```
public class ListMap implements Map {

    private ListBag keyData = new ListBag();
    private ListBag valueData = new ListBag();
    private ListEnumeration ke, ve;
    private Object vo;

    public synchronized boolean containsKey (Object key) {
        ke = (ListEnumeration) keyData.elements();
        ve = (ListEnumeration) valueData.elements();
        while (ke.hasMoreElements() && ve.hasMoreElements()) {
            vo = ve.nextElement();
            if (key.equals(ke.nextElement()))
                return true;
        }
        return false;
    }

    public synchronized Object get (Object key) {
        if (containsKey(key))
            return vo;
        throw new NoSuchElementException();
    }

    public Enumeration keys () { return keyData.elements(); }

    public synchronized void set (Object key, Object newValue) {
        if (containsKey(key))
            ve.set(newValue);
        else {
            keyData.addElement(key);
            valueData.addElement(newValue);
        }
    }

    public synchronized boolean removeKey (Object key) {
        if (containsKey(key)) {
            ke.remove();
            ve.remove();
            return true;
        }
        return false;
    }
}
```

Figure 13.3: A List-based Map Abstraction

Side Effects

Whenever a method has an effect that is not reflected in the description or the return value it is said to have a *side effect*. The method containsKey in the ListMap is a good example. Setting the values of the two list enumerations is not necessary for the functioning of this method by itself, and is not needed for the result of this method. Side effects should always be treated with great care. They have a tendency to make programs difficult to read, since they create hidden dependences between methods. However, there are times when they can greatly simplify tasks, such as in the example shown here.

wraps around another another container that has been passed as argument to the constructor. In this case the container is a Map. The MultiMap also implements the Map interface.[2]

A MultiMap allows several values to be stored using the same key. To accomplish this, the values stored by the underlying map are actually containers. In this case, we use a ListBag. When a value is placed into the map, the MultiMap first checks to see if a value is already entered under the given key. If so, then the new value is simply added to the list of values stored under the given key. If the key is new, then a new ListBag is contructed, and the element is inserted as the first value in the list.

When the values associated with a given key are requested, an Enumeration for the container holding the elements is constructed. The get method for a MultiMap will always return an Enumeration. The other methods for the adapter are easily built using the functionality provided by the underlying Map.

13.4.2 Application – A Concordance

We can illustrate the use of the MultiMap data type with a program to build a concordance. A concordance is an alphabetical listing of words in a text, one that indicates the line numbers on which each word occurs. A MultiMap is an appropriate data structure for this problem because the same word will often appear on multiple different lines; indeed, discovering such connections is one of the primary purposes of a concordance.

Our implementation of a concordance is as follows:

```
class Concordance {
    private Map dict = new MultiMap(new ListMap());

    public void readLines (BufferedReader input) throws IOException {
```

[2] A component such as this, one that both wraps around an object of a given type, and itself implements the interface for that type, is sometimes termed a *decorator*.

```
public class MultiMap implements Map {

    public MultiMap (Map m) { data = m; }
    private Map data;

    public boolean containsKey (Object key)
        { return data.containsKey(key); }

    public Object get (Object key) {
        Bag b = (Bag) data.get(key);
        return b.elements();
    }

    public Enumeration keys () { return data.keys(); }

    public synchronized void set (Object key, Object newValue) {
        Bag bag;
        if (data.containsKey(key))
            bag = (Bag) data.get(key);
        else {
            bag = new ListBag();
            data.set(key, bag);
        }
        bag.addElement(newValue);
    }

    public synchronized boolean removeKey (Object key)
        { return data.removeKey(key); }
}
```

Figure 13.4: The MultiMap adapter

```
        String delims = " \t\n.,!?;:";
        for (int line = 1; true; line++ ) {
            String text = input.readLine();
            Integer iline = new Integer(line);
            if (text == null) return;
            text = text.toLowerCase();
            Enumeration e = new StringTokenizer(text, delims);
            while (e.hasMoreElements()) {
                 String word = (String) e.nextElement();
                 if (! dict.contains(word, iline))
                    dict.set(word, iline);
            }
        }
    }

    public void generateOutput (PrintStream output) {
        Enumeration e = dict.elements();
        while (e.hasMoreElements() ) {
            String word = (String) e.nextElement();
            Enumeration f = (Enumeration) dict.get(word);
            output.print(word + ": ");
            while (f.hasMoreElements())
                output.print(f.nextElement() + " ");
            output.println(" ");
        }
    }
}
```

The concordance has two methods, the first to read lines from an input and create the concordance, and the second to print the concordance entries.

As with the silly sentence program, the method readLines uses a BufferedReader as the input source. An integer variable counts the line numbers as they are read. Once a line is read, it is converted to lower case by the String method toLowerCase. Then it is split into individual words using a StringTokenizer. Each word is then inserted into the dictionary.

Output is produced using the method generateOutput. An outer loop iterates over the keys in the dictionary, which are the individual words. An inner loop iterates over entries associated with the word, which are the line numbers on which the word was found.

To test our concordance we create a buffered reader from the standard input stream, as follows:

```
public static void main (String [ ] args) {
    Concordance cd = new Concordance();
```

```
    try {
        cd.readLines(new  BufferedReader(new InputStreamReader(System.in)));
    } catch(IOException e) { System.err.println("received IO error " + e);}
    cd.generateOutput(System.out);
}
```

If, for example, the input was the text:

It was the best of times,
it was the worst of times.

The output, from best to worst, would be:

best: 1
it: 1 2
of: 1 2
the: 1 2
times: 1 2
was: 1 2
worst: 2

13.5 Self Organizing Lists *

In some situations that involve searching a list, the distribution of requests is not uniform, and some values will be sought much more frequently than others. Telephone companies use this principle, for example, when they place frequently referenced numbers, such as governmental offices, in a special section near the front of a telephone directory. In a similar fashion a linked list can improve upon the sequential nature of a search for an element by placing it near the front of a list.

But what if you know that searches will have this non-uniform property but you cannot predict which elements will be the most frequently requested? One common approach is to assume that when an element is requested there will be a high probability that it will be subsequently requested again. We can exploit this property by always moving an element to the front of a list on a successful search. A data structure that tries to improve *future* performance based on current usage is said to be *self-organizing*.

To implement the self organizing principle we need only create a subclass of ListBag and override the containsElement method:

```
public class SelfOrgList extends ListBag {

    public boolean containsElement (Object testElement) {
```

*Section headings followed by an asterisk indicate optional material.

Use Inheritance or Not?

It is useful to consider why inheritance is the appropriate mechanism to use here, whereas we used composition for many of our earlier abstractions. Both mechanisms create a new data abstraction by leveraging from an earlier software component. But the key insight is that the self organizing list does not reduce the number of operations provided by the underlying container, indeed in this case it does not change it at all. By using inheritance, we need describe only the difference between the two abstractions, and all other operations are automatically provided for free.

```
        ListEnumeration e = (ListEnumeration) elements();
        while (e.hasMoreElements()) {
            Object a = e.nextElement();
            if (testElement.equals(a)) {
                e.remove(); // remove it
                    // move to front
                e = (ListEnumeration) elements();
                e.add(a);
                return true;
            }
        }
        return false;
    }
}
```

To measure the effectiveness of the self-organizing concept, we need to artificially create a series of requests in which certain elements will appear repeatedly. One approach is to create a list of n random elements, and then perform n tests for inclusion, but on every 7th test ask for the number 42. (These numbers were selected arbitrarily, and have no intrinsic meaning). If we compare the running time of the search process under these assumptions to the regular ListBag, we get timing figures such as are shown in Figure 13.5.

Of course, the improvements obtained through the use of the self organizing list are only possible when the series of requests fit a very particular pattern. In the next section we will examine a different technique that will speed up the search for values in much more general situations.

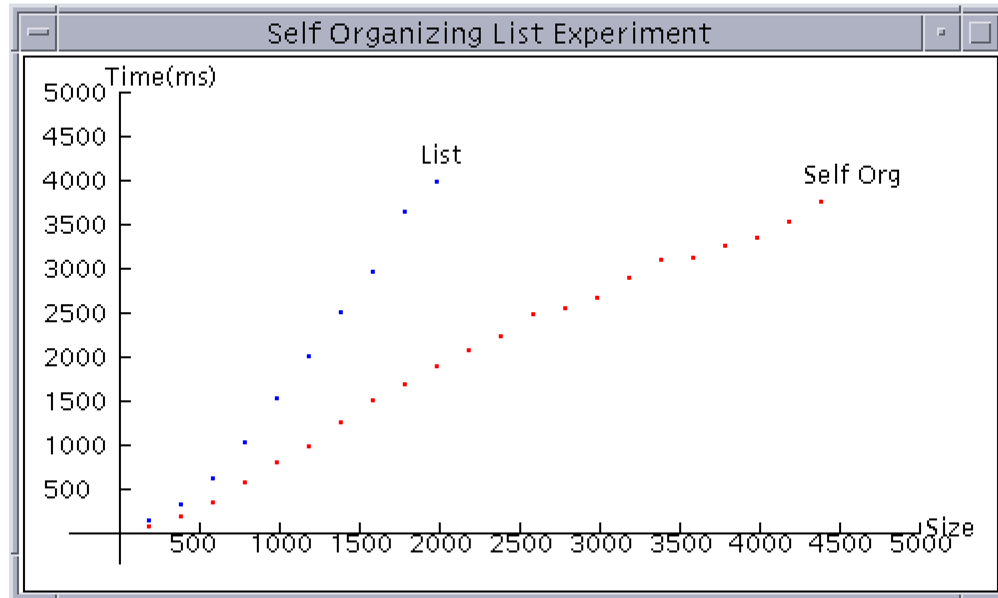


Figure 13.5: Search Timings for Self Organizing Lists

Experiment: Finding the Limits of Self Organization

These experimental results seem to show that the self organization principle is useful when every 7th request repeats an earlier request. What happens if this ratio is changed? What would happen if instead of every 7th request, we artificially made every 30th request match an earlier one. At what point does the overhead of moving the elements outweigh the benefits?

13.6 Chapter Summary

In this chapter we have continued our exploration of the concept of linked lists, concentrating on the ability of linked lists to efficiently add new values into the middle of a container. The ListBag data type showed how linked lists could be used to implement the Bag abstraction. List enumerators extend the ability of enumerations, making it easy to insert and remove values from an enumeration as it is looping. Many sorting algorithms, such as insertion sort or quick sort, can be made to work with linked list structures given these new abilities.

A linked list can be used to implement a container that supports the Map interface. We have described a variation on the Map, called a MultiMap. The MultiMap allows more than one value to be stored under the same key. The MultiMap accomplishes this by maintaining a list for each key stored in the collection.

Key Concepts

- List insertions
- List Enumerators
- Map and MultiMap
- Self Organizing Lists

Further Information

A number of different approaches to self-organizing data structures are discussed by Gonnet and Baeza-Yates [Gonnet 91]. Sleator and Tarjan show that the move-to-front rule for self organizing lists is within a constant factor of the best possible self-organizing rule [Sleator 85].

Study Questions

1. Why is it more efficient to insert a value into a middle of a linked list than into the middle of a vector?
2. What new abilities does the ListEnumeration class add to the class Enumeration?
3. When a value is added to a list using a ListEnumeration, where is the new value placed in relation to the current element being referenced by the enumeration?
4. Compare the List Insertion sort to the array Insertion sort described in Chapter 4. In what ways are they similar? In what ways are they different?
5. Compare the List quick sort to the vector quick sort described in Chapter 13. In what ways are they similar? In what ways are they different?
6. What is a side effect? Why should programmers try to avoid side effects whenever possible?
7. In what ways is a MultiMap different from a Map? In what ways are they similar?
8. What do we mean when we say that a data abstraction is self-organizing?
9. Under what circumstances will a self-organizing list perform better than an ordinary linked list?

Exercises

1. Fill in the asymptotic execution times for the following operations provided by the class ListBag. For each, indicate whether the execution time will always be required, is an expected (or average) time, or is worst case.

addElement	$O(\quad)$
containsElement	$O(\quad)$
removeElement	$O(\quad)$
size	$O(\quad)$

2. Imagine you have a linked list with five elements. A ListEnumerator is constructed and has moved to the third element. Draw the state of the list including all links prior to the addition of a new element, then after the addition, then after the next call on the methods hasMoreElements and nextElement.

3. Do the same, but this time assume the operation is a remove.

4. Using invariants, prove that the method containsElement in the class ListBag is correct.

5. Using invariants, prove that the method removeElement in the class ListBag is correct.

6. Show why the method containsElement in class ListBag must be synchronized, by presenting a scenario where an unexpected result will be produced if the modifier were omitted.

7. Is the list insertion sort algorithm stable? Give an argument showing why, or an example to show that it is not.

8. Is the list quick sort algorithm stable? Give an argument showing why, or an example to show that it is not.

9. Assume that we would like to say two lists are equal if they have the same elements, regardless of their ordering. What would be the asymptotic complexity of determining if two lists of n elements each are equal?

10. Would it be possible to add the Vector index methods elementAt and setElementAt to a ListBag? What would be the complexity of accessing an element?

11. Fill in the asymptotic execution times for the following operations provided by the class ListMap. For each, indicate whether the execution time will always be required, is an expected (or average) time, or is worst case.

containsKey	$O(\quad)$
get	$O(\quad)$
set	$O(\quad)$
removeKey	$O(\quad)$

12. Fill in the asymptotic execution times for the following operations provided by the class MapAdapter when used in conjunction with a ListBag. For each, indicate whether the execution time will always be required, is an expected (or average) time, or is worst case.

containsKey	$O(\quad)$
get	$O(\quad)$
set	$O(\quad)$
removeKey	$O(\quad)$

13. Using invariants, prove that the method containsKey in the class ListMap is correct.

14. Using invariants, prove that the methods get, set and removekey in the class ListMap are correct. What assertions do you need to make concerning the execution of the method containsKey?

15. To see why the methods set and removeKey must be synchronized in the ListMap class, imagine the modifiers have been removed, and the following sequence of events occur. One thread performs a removeKey, but is suspended after the return from calling containsKey. Next, a second thread invokes set on the same map, but using a different key. Finally, the first thread is resumed. What is the resulting state of the collection?

16. Fill in the asymptotic execution times for the following operations provided by the class MultiMap when used in conjunction with a ListMap. For each, indicate whether the execution time will always be required, is an expected (or average) time, or is worst case.

containsKey	$O(\quad)$
get	$O(\quad)$
set	$O(\quad)$
removeKey	$O(\quad)$

17. Imagine we construct a MultiMap around a MapAdapter and a ListBag, and perform the following set of operations:

```
Map m = new MultiMap(new MapAdapter(new ListBag()));
m.set("one", new Integer(1));
m.set("two", new Integer(2));
m.set("one", new Integer(10));
m.set("three", new Integer(3));
```

Draw a pictorial representation of the data fields maintained by the value m and all their associated values.

Programming Projects

1. A number of useful operations could be added to our ListBag abstraction:

    (a) add a constructor that will take as argument another Bag, and initialize the list with the elements from the collection.

    (b) Do the same for an argument of type Enumeration.

    (c) Add a method named unique that will replace repeated sequences of an element by a single occurrence. For example, the list (2 3 3 1 1 1 5 5 2 2) would become (2 3 1 5 2).

    (d) Add a method that will reverse the elements in a linked list without creating any new link nodes.

2. Empirically compare the execution time performance of the ListMap and the MapAdapter used with a ListBag. Are there any operations that are clearly faster in one abstraction or the other? Can you explain why?

3. Concordances often filter out common words, such as "the" or "and". Suppose you have a file of common words, which is then read into a Set. Modify the concordance program so that it checks words from this set, and if found does not enter them into the database.

4. Modify the concordance class by adding a method that will print the entries in the concordance in order of their frequency of use, with most frequently used words first, and least frequently used words last. Show how you can use a specially built Comparison object in conjunction with a different Map data abstraction to make this task easier.

5. Another approach to silly sentence generation is the following. An existing text is analyzed, and a MultiMap is formed by using each word as key, and storing with each word the set of all words that *followed* the key in the original text. To generate new sentences, an arbitrary first word is selected as the starting point. A random word in the successor list is then selected and printed. This is then used as the key and the next word is randomly selected. The process is continued as long as desired.

6. An alternative approach to the self-organizing list concept is termed the *transpose method*. When a successful search is found, the selected element is transposed with its neighbor, moving one location closer to the front of the list. In this fashion elements move much more slowly to the front of the list; however, the results are more stable as a series of requests for rare values cannot get in the way of a commonly accessed element moving to the front. Implement a self organizing list using the transpose method, and experimentally compare the running time of this technique to the mechanism studied in this chapter.

7. There is a famous fable about a servant who does a favor for the king. When the king asks what reward he would desire, the servent produces a chessboard, and asks for one grain of rice to be placed on the first square. The 2nd square should have twice as many as the first. The 3rd square should have twice as many as the 2nd, and so on.

   If we try to compute the number of grains of rice that the last (or 64th) square will contain using an int value we quickly overflow the number size. However, using the fact that a number can be doubled by adding it to itself (that is, $2 \times x$ is the same as $x + x$) we can compute this value using nothing more than the addition operator provided by LargePositiveInteger. Use this idea to determine how many grains of rice the servent will receive in the last square, and how many grains we will receive in total.

8. Finish the infinite precision number package described in Section 13.3 by writing the class LargeInteger as described. Your class should be able to handle arbitrary addition and subtraction of large integers. Multiplication of large integers can be handled in two steps. First write a procedure to multiply a large positive integer by a primitive integer value (that is, less than 1000). Multiplication of two infinite precision integers is then performed as a series of multiplies by primitive ints and additions. The product 10242×3673, for example, can be computed as $(10242 \times 3) \times 1000 + 10242 \times 673$. Note that to multiply an intermediate result by 1000 it is only necessary to add a zero entry to the end of the list.

9. In Chapter 4 we described some of the objectives a good set of test cases should possess. How would you go about testing the infinite precision number package? Write a harness program, and execute the code using your test values.