CHAPTER 3

Lists, Stacks, and Queues

This chapter discusses three of the most simple and basic data structures. Virtually every significant program will use at least one of these structures explicitly, and a stack is always implicitly used in a program, whether or not you declare one. Among the highlights of this chapter, we will . . .

- Introduce the concept of Abstract Data Types (ADTs).
- Show how to efficiently perform operations on lists.
- Introduce the stack ADT and its use in implementing recursion.
- Introduce the queue ADT and its use in operating systems and algorithm design.

In this chapter, we provide code that implements a significant subset of two library classes: `vector` and `list`.

3.1 Abstract Data Types (ADTs)

An **abstract data type** (ADT) is a set of objects together with a set of operations. Abstract data types are mathematical abstractions; nowhere in an ADT's definition is there any mention of *how* the set of operations is implemented. Objects such as lists, sets, and graphs, along with their operations, can be viewed as ADTs, just as integers, reals, and booleans are data types. Integers, reals, and booleans have operations associated with them, and so do ADTs. For the set ADT, we might have such operations as *add, remove, size,* and *contains.* Alternatively, we might only want the two operations *union* and *find,* which would define a different ADT on the set.

The C++ class allows for the implementation of ADTs, with appropriate hiding of implementation details. Thus, any other part of the program that needs to perform an operation on the ADT can do so by calling the appropriate method. If for some reason implementation details need to be changed, it should be easy to do so by merely changing the routines that perform the ADT operations. This change, in a perfect world, would be completely transparent to the rest of the program.

There is no rule telling us which operations must be supported for each ADT; this is a design decision. Error handling and tie breaking (where appropriate) are also generally up to the program designer. The three data structures that we will study in this chapter are primary examples of ADTs. We will see how each can be implemented in several ways, but if they are done correctly, the programs that use them will not necessarily need to know which implementation was used.

3.2 The List ADT

We will deal with a general list of the form $A_0, A_1, A_2, \ldots, A_{N-1}$. We say that the size of this list is N. We will call the special list of size 0 an **empty list.**

For any list except the empty list, we say that A_i follows (or succeeds) A_{i-1} ($i < N$) and that A_{i-1} precedes A_i ($i > 0$). The first element of the list is A_0, and the last element is A_{N-1}. We will not define the predecessor of A_0 or the successor of A_{N-1}. The **position** of element A_i in a list is i. Throughout this discussion, we will assume, to simplify matters, that the elements in the list are integers, but in general, arbitrarily complex elements are allowed (and easily handled by a class template).

Associated with these "definitions" is a set of operations that we would like to perform on the List ADT. Some popular operations are `printList` and `makeEmpty`, which do the obvious things; `find`, which returns the position of the first occurrence of an item; `insert` and `remove`, which generally insert and remove some element from some position in the list; and `findKth`, which returns the element in some position (specified as an argument). If the list is 34, 12, 52, 16, 12, then `find(52)` might return 2; `insert(x,2)` might make the list into 34, 12, `x`, 52, 16, 12 (if we insert into the position given); and `remove(52)` might turn that list into 34, 12, `x`, 16, 12.

Of course, the interpretation of what is appropriate for a function is entirely up to the programmer, as is the handling of special cases (for example, what does `find(1)` return above?). We could also add operations such as `next` and `previous`, which would take a position as argument and return the position of the successor and predecessor, respectively.

3.2.1 Simple Array Implementation of Lists

All these instructions can be implemented just by using an array. Although arrays are created with a fixed capacity, the `vector` class, which internally stores an array, allows the array to grow by doubling its capacity when needed. This solves the most serious problem with using an array—namely, that historically, to use an array, an estimate of the maximum size of the list was required. This estimate is no longer needed.

An array implementation allows `printList` to be carried out in linear time, and the `findKth` operation takes constant time, which is as good as can be expected. However, insertion and deletion are potentially expensive, depending on where the insertions and deletions occur. In the worst case, inserting into position 0 (in other words, at the front of the list) requires pushing the entire array down one spot to make room, and deleting the first element requires shifting all the elements in the list up one spot, so the worst case for these operations is $O(N)$. On average, half of the list needs to be moved for either operation, so linear time is still required. On the other hand, if all the operations occur at the high end of the list, then no elements need to be shifted, and then adding and deleting take $O(1)$ time.

There are many situations where the list is built up by insertions at the high end, and then only array accesses (i.e., `findKth` operations) occur. In such a case, the array is a suitable implementation. However, if insertions and deletions occur throughout the list and, in particular, at the front of the list, then the array is not a good option. The next section deals with the alternative: the *linked list*.

3.2.2 Simple Linked Lists

In order to avoid the linear cost of insertion and deletion, we need to ensure that the list is not stored contiguously, since otherwise entire parts of the list will need to be moved. Figure 3.1 shows the general idea of a **linked list.**

The linked list consists of a series of nodes, which are not necessarily adjacent in memory. Each node contains the element and a link to a node containing its successor. We call this the `next` link. The last cell's `next` link points to `nullptr`.

To execute `printList()` or `find(x)`, we merely start at the first node in the list and then traverse the list by following the `next` links. This operation is clearly linear-time, as in the array implementation; although, the constant is likely to be larger than if an array implementation were used. The `findKth` operation is no longer quite as efficient as an array implementation; `findKth(i)` takes $O(i)$ time and works by traversing down the list in the obvious manner. In practice, this bound is pessimistic, because frequently the calls to `findKth` are in sorted order (by i). As an example, `findKth(2)`, `findKth(3)`, `findKth(4)`, and `findKth(6)` can all be executed in one scan down the list.

The `remove` method can be executed in one `next` pointer change. Figure 3.2 shows the result of deleting the third element in the original list.

The `insert` method requires obtaining a new node from the system by using a `new` call and then executing two `next` pointer maneuvers. The general idea is shown in Figure 3.3. The dashed line represents the old pointer.

As we can see, in principle, if we know where a change is to be made, inserting or removing an item from a linked list does not require moving lots of items, and instead involves only a constant number of changes to node links.

The special case of adding to the front or removing the first item is thus a constant-time operation, presuming of course that a link to the front of the linked list is maintained.

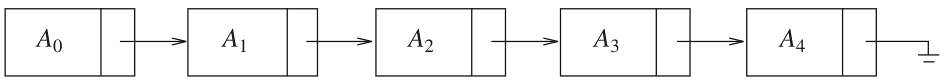


Figure 3.1 A linked list

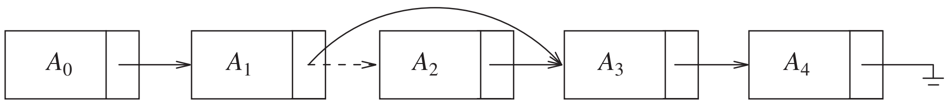


Figure 3.2 Deletion from a linked list

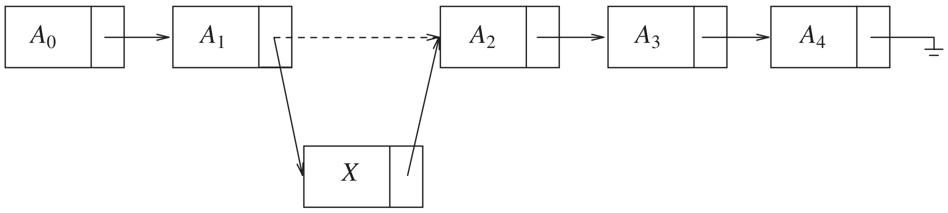


Figure 3.3 Insertion into a linked list

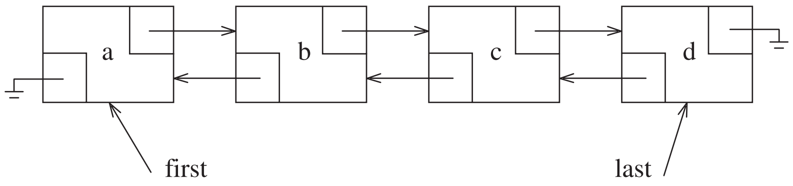


Figure 3.4 A doubly linked list

The special case of adding at the end (i.e., making the new item the last item) can be constant-time, as long as we maintain a link to the last node. Thus, a typical linked list keeps links to both ends of the list. Removing the last item is trickier, because we have to find the next-to-last item, change its *next* link to `nullptr`, and then update the link that maintains the last node. In the classic linked list, where each node stores a link to its next node, having a link to the last node provides no information about the next-to-last node.

The obvious idea of maintaining a third link to the next-to-last node doesn't work, because it too would need to be updated during a remove. Instead, we have every node maintain a link to its previous node in the list. This is shown in Figure 3.4 and is known as a **doubly linked list.**

3.3 `vector` and `list` in the STL

The C++ language includes, in its library, an implementation of common data structures. This part of the language is popularly known as the **Standard Template Library** (STL). The List ADT is one of the data structures implemented in the STL. We will see some others in Chapters 4 and 5. In general, these data structures are called **collections** or **containers**.

There are two popular implementations of the List ADT. The `vector` provides a growable array implementation of the List ADT. The advantage of using the `vector` is that it is indexable in constant time. The disadvantage is that insertion of new items and removal of existing items is expensive, unless the changes are made at the end of the `vector`. The `list` provides a doubly linked list implementation of the List ADT. The advantage of using the list is that insertion of new items and removal of existing items is cheap, provided that the position of the changes is known. The disadvantage is that the list is not easily indexable. Both vector and list are inefficient for searches. Throughout this discussion, list refers to the doubly linked list in the STL, whereas list (typeset without the monospace font) refers to the more general List ADT.

Both vector and list are class templates that are instantiated with the type of items that they store. Both have several methods in common. The first three methods shown are actually available for all the STL containers:

- `int size( ) const`: returns the number of elements in the container.
- `void clear( )`: removes all elements from the container.
- `bool empty( ) const`: returns true if the container contains no elements, and false otherwise.

Both vector and list support adding and removing from the end of the list in constant time. Both vector and list support accessing the front item in the list in constant time. The operations are:

- `void push_back( const Object & x )`: adds x to the end of the list.
- `void pop_back( )`: removes the object at the end of the list.
- `const Object & back( ) const`: returns the object at the end of the list (a mutator that returns a reference is also provided).
- `const Object & front( ) const`: returns the object at the front of the list (a mutator that returns a reference is also provided).

Because a doubly linked list allows efficient changes at the front, but a vector does not, the following two methods are available only for list:

- `void push_front( const Object & x )`: adds x to the front of the list.
- `void pop_front( )`: removes the object at the front of the list.

The vector has its own set of methods that are not part of list. Two methods allow efficient indexing. The other two methods allow the programmer to view and change the internal capacity. These methods are:

- `Object & operator[] ( int idx )`: returns the object at index idx in the vector, with no bounds-checking (an accessor that returns a constant reference is also provided).
- `Object & at( int idx )`: returns the object at index idx in the vector, with bounds-checking (an accessor that returns a constant reference is also provided).
- `int capacity( ) const`: returns the internal capacity of the vector. (See Section 3.4 for more details.)
- `void reserve( int newCapacity )`: sets the new capacity. If a good estimate is available, it can be used to avoid expansion of the vector. (See Section 3.4 for more details.)

3.3.1 Iterators

Some operations on lists, most critically those to insert and remove from the middle of the list, require the notion of a position. In the STL, a position is represented by a nested type, `iterator`. In particular, for a `list<string>`, the position is represented by the type `list<string>::iterator`; for a `vector<int>`, the position is represented by a class `vector<int>::iterator`, and so on. In describing some methods, we'll simply use `iterator` as a shorthand, but when writing code, we will use the actual nested class name.

Initially, there are three main issues to address: first, how one gets an iterator; second, what operations the iterators themselves can perform; third, which List ADT methods require iterators as parameters.

Getting an Iterator

For the first issue, the STL lists (and all other STL containers) define a pair of methods:

- `iterator begin( )`: returns an appropriate iterator representing the first item in the container.
- `iterator end( )`: returns an appropriate iterator representing the endmarker in the container (i.e., the position after the last item in the container).

The `end` method seems a little unusual, because it returns an iterator that is "out-of-bounds." To see the idea, consider the following code typically used to print the items in a `vector v` prior to the introduction of range-based `for` loops in C++11:

```
for( int i = 0; i != v.size( ); ++i )
    cout << v[ i ] << endl;
```

If we were to rewrite this code using iterators, we would see a natural correspondence with the `begin` and `end` methods:

```
for( vector<int>::iterator itr = v.begin( ); itr != v.end( ); itr.??? )
    cout << itr.??? << endl;
```

In the loop termination test, both `i!=v.size( )` and `itr!=v.end( )` are intended to test if the loop counter has become "out-of-bounds." The code fragment also brings us to the second issue, which is that the iterator must have methods associated with it (these unknown methods are represented by *???*).

Iterator Methods

Based on the code fragment above, it is obvious that iterators can be compared with `!=` and `==`, and likely have copy constructors and `operator=` defined. Thus, iterators have methods, and many of the methods use operator overloading. Besides copying, the most commonly used operations on iterators include the following:

- `itr++` and `++itr`: advances the iterator `itr` to the next location. Both the prefix and postfix forms are allowable.

- `*itr`: returns a reference to the object stored at iterator `itr`'s location. The reference returned may or may not be modifiable (we discuss these details shortly).
- `itr1==itr2`: returns true if iterators `itr1` and `itr2` refer to the same location and false otherwise.
- `itr1!=itr2`: returns true if iterators `itr1` and `itr2` refer to a different location and false otherwise.

With these operators, the code to print would be

```
for( vector<int>::iterator itr = v.begin( ); itr != v.end( ); ++itr )
    cout << *itr << endl;
```

The use of operator overloading allows one to access the current item, then advance to the next item using `*itr++`. Thus, an alternative to the fragment above is

```
vector<int>::iterator itr = v.begin( );
while( itr !=v.end( ) )
    cout << *itr++ << endl;
```

Container Operations That Require Iterators

For the last issue, the three most popular methods that require iterators are those that add or remove from the list (either a `vector` or `list`) at a specified position:

- `iterator insert( iterator pos, const Object & x )`: adds `x` into the list, prior to the position given by the iterator `pos`. This is a constant-time operation for `list`, but not for `vector`. The return value is an iterator representing the position of the inserted item.
- `iterator erase( iterator pos )`: removes the object at the position given by the iterator. This is a constant-time operation for `list`, but not for `vector`. The return value is the position of the element that followed `pos` prior to the call. This operation invalidates `pos`, which is now stale, since the container item it was viewing has been removed.
- `iterator erase( iterator start, iterator end )`: removes all items beginning at position `start`, up to, but not including `end`. Observe that the entire list can be erased by the call `c.erase( c.begin( ), c.end( ) )`.

3.3.2 Example: Using `erase` on a List

As an example, we provide a routine that removes every other item in a list, starting with the initial item. Thus if the list contains 6, 5, 1, 4, 2, then after the method is invoked it will contain 5, 4. We do this by stepping through the list and using the `erase` method on every second item. On a `list`, this will be a linear-time routine because each of the calls to `erase` takes constant time, but in a `vector` the entire routine will take quadratic time because each of the calls to `erase` is inefficient, using $O(N)$ time. As a result, we would normally write the code for a `list` only. However, for experimentation purposes, we write a general function template that will work with both a `list` or a `vector`, and then provide

```
 1  template <typename Container>
 2  void removeEveryOtherItem( Container & lst )
 3  {
 4      auto itr = lst.begin( );     // itr is a Container::iterator
 5
 6      while( itr != lst.end( ) )
 7      {
 8          itr = lst.erase( itr );
 9          if( itr != lst.end( ) )
10              ++itr;
11      }
12  }
```

Figure 3.5 Using iterators to remove every other item in a List (either a `vector` or `list`). Efficient for a `list`, but not for a `vector`.

timing information. The function template is shown in Figure 3.5. The use of `auto` at line 4 is a C++11 feature that allows us to avoid the longer type `Container::iterator`. If we run the code, passing a `list<int>`, it takes 0.039 sec for a 800,000-item `list`, and 0.073 sec for an 1,600,000-item `list`, and is clearly a linear-time routine, because the running time increases by the same factor as the input size. When we pass a `vector<int>`, the routine takes almost five minutes for an 800,000-item `vector` and about twenty minutes for an 1,600,000-item `vector`; the four fold increase in running time when the input increases by only a factor of two is consistent with quadratic behavior.

3.3.3 `const_iterators`

The result of `*itr` is not just the value of the item that the iterator is viewing but also the item itself. This distinction makes the iterators very powerful but also introduces some complications. To see the benefit, suppose we want to change all the items in a collection to a specified value. The following routine works for both `vector` and `list` and runs in linear time. It's a wonderful example of writing generic, type-independent code.

```
template <typename Container, typename Object>
void change( Container & c, const Object & newValue )
{
    typename Container::iterator itr = c.begin( );
    while( itr != c.end( ) )
        *itr++ = newValue;
}
```

To see the potential problem, suppose the `Container c` was passed to a routine using call-by-constant reference. This means we would expect that no changes would be allowed to `c`, and the compiler would ensure this by not allowing calls to any of `c`'s mutators. Consider the following code that prints a `list` of integers but also tries to sneak in a change to the `list`:

```
void print( const list<int> & lst, ostream & out = cout )
{
    typename Container::iterator itr = lst.begin( );
    while( itr != lst.end( ) )
    {
        out << *itr << endl;
        *itr = 0;   // This is fishy!!!
        ++itr;
    }
}
```

If this code were legal, then the const-ness of the `list` would be completely meaningless, because it would be so easily bypassed. The code is not legal and will not compile. The solution provided by the STL is that every collection contains not only an `iterator` nested type but also a `const_iterator` nested type. The main difference between an `iterator` and a `const_iterator` is that `operator*` for `const_iterator` returns a constant reference, and thus `*itr` for a `const_iterator` cannot appear on the left-hand side of an assignment statement.

Further, the compiler will force you to use a `const_iterator` to traverse a constant collection. It does so by providing two versions of `begin` and two versions of `end`, as follows:

- `iterator begin( )`
- `const_iterator begin( ) const`
- `iterator end( )`
- `const_iterator end( ) const`

The two versions of `begin` can be in the same class only because the const-ness of a method (i.e., whether it is an accessor or mutator) is considered to be part of the signature. We saw this trick in Section 1.7.2 and we will see it again in Section 3.4, both in the context of overloading `operator[]`.

If `begin` is invoked on a nonconstant container, the "mutator" version that returns an `iterator` is invoked. However, if `begin` is invoked on a constant container, what is returned is a `const_iterator`, and the return value may not be assigned to an `iterator`. If you try to do so, a compiler error is generated. Once `itr` is a `const_iterator`, `*itr=0` is easily detected as being illegal.

If you use `auto` to declare your iterators, the compiler will deduce for you whether an `iterator` or `const_iterator` is substituted; to a large extent, this relieves the programmer from having to keep track of the correct iterator type and is precisely one of the intended uses of `auto`. Additionally, library classes such as `vector` and `list` that provide iterators as described above are compatible with the range-based `for` loop, as are user-defined classes.

An additional feature in C++11 allows one to write code that works even if the `Container` type does not have `begin` and `end` member functions. Non-member free functions `begin` and `end` are defined that allow one to use `begin(c)` in any place where `c.begin()` is allowed. Writing generic code using `begin(c)` instead of `c.begin()` has the advantage that it allows the generic code to work on containers that have `begin/end` as members, as well as those that do not have `begin/end` but which can later be augmented with appropriate

```
template <typename Container>
void print( const Container & c, ostream & out = cout )
{
    if( c.empty( ) )
        out << "(empty)";
    else
    {
        auto itr = begin( c );   // itr is a Container::const_iterator

        out << "[ " << *itr++;   // Print first item

        while( itr != end( c ) )
            out << ", " << *itr++;
        out << " ]" << endl;
    }
}
```

Figure 3.6 Printing any container

non-member functions. The addition of `begin` and `end` as free functions in C++11 is made possible by the addition of language features `auto` and `decltype`, as shown in the code below.

```
template<typename Container>
auto begin( Container & c ) -> decltype( c.begin( ) )
{
    return c.begin( );
}

template<typename Container>
auto begin( const Container & c ) -> decltype( c.begin( ) )
{
    return c.begin( );
}
```

In this code, the return type of `begin` is deduced to be the type of `c.begin()` .

The code in Figure 3.6 makes use of `auto` to declare the iterator (as in Fig. 3.5) and uses non-member functions `begin` and `end`.

3.4 Implementation of `vector`

In this section, we provide the implementation of a usable `vector` class template. The `vector` will be a first-class type, meaning that unlike the primitive array in C++, the `vector` can be copied, and the memory it uses can be automatically reclaimed (via its destructor). In Section 1.5.7, we described some important features of C++ primitive arrays:

- The array is simply a pointer variable to a block of memory; the actual array size must be maintained separately by the programmer.
- The block of memory can be allocated via `new[]` but then must be freed via `delete[]`.
- The block of memory cannot be resized (but a new, presumably larger block can be obtained and initialized with the old block, and then the old block can be freed).

To avoid ambiguities with the library class, we will name our class template `Vector`. Before examining the `Vector` code, we outline the main details:

1. The `Vector` will maintain the primitive array (via a pointer variable to the block of allocated memory), the array capacity, and the current number of items stored in the `Vector`.
2. The `Vector` will implement the Big-Five to provide deep-copy semantics for the copy constructor and `operator=`, and will provide a destructor to reclaim the primitive array. It will also implement C++11 move semantics.
3. The `Vector` will provide a `resize` routine that will change (generally to a larger number) the size of the `Vector` and a `reserve` routine that will change (generally to a larger number) the capacity of the `Vector`. The capacity is changed by obtaining a new block of memory for the primitive array, copying the old block into the new block, and reclaiming the old block.
4. The `Vector` will provide an implementation of `operator[]` (as mentioned in Section 1.7.2, `operator[]` is typically implemented with both an accessor and mutator version).
5. The `Vector` will provide basic routines, such as `size`, `empty`, `clear` (which are typically one-liners), `back`, `pop_back`, and `push_back`. The `push_back` routine will call `reserve` if the size and capacity are same.
6. The `Vector` will provide support for the nested types `iterator` and `const_iterator`, and associated `begin` and `end` methods.

Figure 3.7 and Figure 3.8 show the `Vector` class. Like its STL counterpart, there is limited error checking. Later we will briefly discuss how error checking can be provided.

As shown on lines 118 to 120, the `Vector` stores the size, capacity, and primitive array as its data members. The constructor at lines 7 to 9 allows the user to specify an initial size, which defaults to zero. It then initializes the data members, with the capacity slightly larger than the size, so a few `push_back`s can be performed without changing the capacity.

The copy constructor, shown at lines 11 to 17, makes a new `Vector` and can then be used by a casual implementation of `operator=` that uses the standard idiom of swapping in a copy. This idiom works only if swapping is done by moving, which itself requires the implementation of the move constructor and move `operator=` shown at lines 29 to 44. Again, these use very standard idioms. Implementation of the copy assignment `operator=` using a copy constructor and swap, while simple, is certainly not the most efficient method, especially in the case where both `Vector`s have the same size. In that special case, which can be tested for, it can be more efficient to simply copy each element one by one using `Object`'s `operator=`.

```
#include <algorithm>

template <typename Object>
class Vector
{
  public:
    explicit Vector( int initSize = 0 ) : theSize{ initSize },
         theCapacity{ initSize + SPARE_CAPACITY }
      { objects = new Object[ theCapacity ]; }

    Vector( const Vector & rhs ) : theSize{ rhs.theSize },
        theCapacity{ rhs.theCapacity }, objects{ nullptr }
    {
        objects = new Object[ theCapacity  ];
        for( int k = 0; k < theSize; ++k )
            objects[ k ] = rhs.objects[ k ];
    }

    Vector & operator= ( const Vector & rhs )
    {
        Vector copy = rhs;
        std::swap( *this, copy );
        return *this;
    }

    ~Vector( )
      { delete [ ] objects; }

    Vector( Vector && rhs ) : theSize{ rhs.theSize },
        theCapacity{ rhs.theCapacity }, objects{ rhs.objects }
    {
        rhs.objects = nullptr;
        rhs.theSize = 0;
        rhs.theCapacity = 0;
    }

    Vector & operator= ( Vector && rhs )
    {
        std::swap( theSize, rhs.theSize );
        std::swap( theCapacity, rhs.theCapacity );
        std::swap( objects, rhs.objects );

        return *this;
    }

```

Figure 3.7 vector class (Part 1 of 2)

```
46      void resize( int newSize )
47      {
48          if( newSize > theCapacity )
49              reserve( newSize * 2 );
50          theSize = newSize;
51      }
52
53      void reserve( int newCapacity )
54      {
55          if( newCapacity < theSize )
56              return;
57
58          Object *newArray = new Object[ newCapacity ];
59          for( int k = 0; k < theSize; ++k )
60              newArray[ k ] = std::move( objects[ k ] );
61
62          theCapacity = newCapacity;
63          std::swap( objects, newArray );
64          delete [ ] newArray;
65      }
```

Figure 3.7 *(continued)*

The `resize` routine is shown at lines 46 to 51. The code simply sets the `theSize` data member, after possibly expanding the capacity. Expanding capacity is very expensive. So if the capacity is expanded, it is made twice as large as the size to avoid having to change the capacity again unless the size increases dramatically (the `+1` is used in case the size is 0). Expanding capacity is done by the `reserve` routine, shown at lines 53 to 65. It consists of allocation of a new array at line 58, moving the old contents at lines 59 and 60, and the reclaiming of the old array at line 64. As shown at lines 55 and 56, the `reserve` routine can also be used to shrink the underlying array, but only if the specified new capacity is at least as large as the size. If it isn't, the `reserve` request is ignored.

The two versions of `operator[]` are trivial (and in fact very similar to the implementations of `operator[]` in the `matrix` class in Section 1.7.2) and are shown in lines 67 to 70. Error checking is easily added by making sure that `index` is in the range `0` to `size()-1`, inclusive, and throwing an exception if it is not.

A host of short routines, namely, `empty`, `size`, `capacity`, `push_back`, `pop_back`, and `back`, are implemented in lines 72 to 101. At lines 83 and 90, we see the use of the postfix `++` operator, which uses `theSize` to index the array and then increases `theSize`. We saw the same idiom when discussing iterators: `*itr++` uses `itr` to decide which item to view and then advances `itr`. The positioning of the `++` matters: In the prefix `++` operator, `*++itr` advances `itr` and then uses the new `itr` to decide which item to view, and likewise, `objects[++theSize]` would increment `theSize` and use the new value to index the array (which is not what we would want). `pop_back` and `back` could both benefit from error checks in which an exception is thrown if the size is 0.

```
    Object & operator[]( int index )
      { return objects[ index ]; }
    const Object & operator[]( int index ) const
      { return objects[ index ]; }

    bool empty( ) const
      { return size( ) == 0; }
    int size( ) const
      { return theSize; }
    int capacity( ) const
      { return theCapacity; }

    void push_back( const Object & x )
    {
        if( theSize == theCapacity )
            reserve( 2 * theCapacity + 1 );
        objects[ theSize++ ] = x;
    }

    void push_back( Object && x )
    {
        if( theSize == theCapacity )
            reserve( 2 * theCapacity + 1 );
        objects[ theSize++ ] = std::move( x );
    }

    void pop_back( )
    {
        --theSize;
    }

    const Object & back ( ) const
    {
         return objects[ theSize - 1 ];
    }

    typedef Object * iterator;
    typedef const Object * const_iterator;

    iterator begin( )
      { return &objects[ 0 ]; }
    const_iterator begin( ) const
      { return &objects[ 0 ]; }
```

Figure 3.8 `vector` class (Part 2 of 2)

```
110       iterator end( )
111         { return &objects[ size( ) ]; }
112       const_iterator end( ) const
113         { return &objects[ size( ) ]; }
114
115       static const int SPARE_CAPACITY = 16;
116
117     private:
118       int theSize;
119       int theCapacity;
120       Object * objects;
121   };
```

Figure 3.8 *(continued)*

Finally, at lines 103 to 113 we see the declaration of the `iterator` and `const_iterator` nested types and the two `begin` and two `end` methods. This code makes use of the fact that in C++, a pointer variable has all the same operators that we expect for an `iterator`. Pointer variables can be copied and compared; the `*` operator yields the object being pointed at, and, most peculiarly, when `++` is applied to a pointer variable, the pointer variable then points at the object that would be stored next sequentially: If the pointer is pointing inside an array, incrementing the pointer positions it at the next array element. These semantics for pointers date back to the early 70s with the C programming language, upon which C++ is based. The STL iterator mechanism was designed in part to mimic pointer operations.

Consequently, at lines 103 and 104, we see `typedef` statements that state the `iterator` and `const_iterator` are simply other names for a pointer variable, and `begin` and `end` need to simply return the memory addresses representing the first array position and the first invalid array position, respectively.

The correspondence between iterators and pointers for the `vector` type means that using a `vector` instead of the C++ array is likely to carry little overhead. The disadvantage is that, as written, the code has no error checks. If the iterator `itr` goes crashing past the end marker, neither `++itr` nor `*itr` will necessarily signal an error. To fix this problem would require that the `iterator` and `const_iterator` be actual nested class types rather than simply pointer variables. Using nested class types is much more common and is what we will see in the `List` class in Section 3.5.

3.5 Implementation of `list`

In this section, we provide the implementation of a usable `list` class template. As in the case of the `vector` class, our list class will be named `List` to avoid ambiguities with the library class.

Recall that the `List` class will be implemented as a doubly linked list and that we will need to maintain pointers to both ends of the list. Doing so allows us to maintain constant time cost per operation, so long as the operation occurs at a known position. The known position can be at either end or at a position specified by an iterator.

In considering the design, we will need to provide four classes:

1. The `List` class itself, which contains links to both ends, the size of the list, and a host of methods.
2. The `Node` class, which is likely to be a private nested class. A node contains the data and pointers to the previous and next nodes, along with appropriate constructors.
3. The `const_iterator` class, which abstracts the notion of a position, and is a public nested class. The `const_iterator` stores a pointer to "current" node, and provides implementation of the basic iterator operations, all in the form of overloaded operators such as `=`, `==`, `!=`, and `++`.
4. The `iterator` class, which abstracts the notion of a position, and is a public nested class. The `iterator` has the same functionality as `const_iterator`, except that `operator*` returns a reference to the item being viewed, rather than a constant reference to the item. An important technical issue is that an `iterator` can be used in any routine that requires a `const_iterator`, but not vice versa. In other words, `iterator` *IS-A* `const_iterator`.

Because the iterator classes store a pointer to the "current node," and the end marker is a valid position, it makes sense to create an extra node at the end of the list to represent the endmarker. Further, we can create an extra node at the front of the list, logically representing the beginning marker. These extra nodes are sometimes known as **sentinel nodes**; specifically, the node at the front is sometimes known as a **header node**, and the node at the end is sometimes known as a **tail node**.

The advantage of using these extra nodes is that they greatly simplify the coding by removing a host of special cases. For instance, if we do not use a header node, then removing the first node becomes a special case, because we must reset the list's link to the first node during the remove and because the remove algorithm in general needs to access the node prior to the node being removed (and without a header node, the first node does not have a node prior to it). Figure 3.9 shows a doubly linked list with header and tail nodes. Figure 3.10 shows an empty list.

Figure 3.11 and Figure 3.12 show the outline and partial implementation of the `List` class.

We can see at line 5 the beginning of the declaration of the private nested `Node` class. Rather than using the `class` keyword, we use `struct`. In C++, the `struct` is a relic from the C programming language. A `struct` in C++ is essentially a `class` in which the members default to public. Recall that in a `class`, the members default to private. Clearly the `struct`

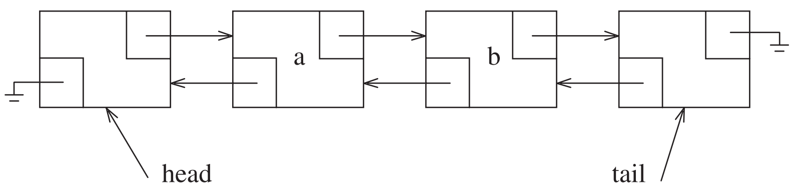


Figure 3.9 A doubly linked list with header and tail nodes

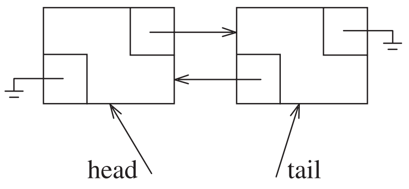


Figure 3.10 An empty doubly linked list with header and tail nodes

keyword is not needed, but you will often see it and it is commonly used by programmers to signify a type that contains mostly data that are accessed directly, rather than through methods. In our case, making the members public in the `Node` class will not be a problem, since the `Node` class is itself private and inaccessible outside of the `List` class.

At line 9 we see the beginning of the declaration of the public nested `const_iterator` class, and at line 12 we see the beginning of the declaration of the public nested `iterator` class. The unusual syntax is **inheritance**, which is a powerful construct not otherwise used in the book. The inheritance syntax states that `iterator` has exactly the same functionality as `const_iterator`, with possibly some additions, and that `iterator` is type-compatible with `const_iterator` and can be used wherever `const_iterator` is needed. We'll discuss those details when we see the actual implementations later.

Lines 80 to 82 contain the data members for `List`, namely, the pointers to the header and tail nodes. We also keep track of the size in a data member so that the `size` method can be implemented in constant time.

The rest of the `List` class consists of the constructor, the Big-Five, and a host of methods. Many of the methods are one-liners. `begin` and `end` return appropriate iterators; the call at line 30 is typical of the implementation, in which we return a constructed iterator (thus the `iterator` and `const_iterator` classes each have a constructor that takes a pointer to a `Node` as its parameter).

The `clear` method at lines 43 to 47 works by repeatedly removing items until the `List` is empty. Using this strategy allows `clear` to avoid getting its hands dirty reclaiming nodes because the node reclamation is now funneled to `pop_front`. The methods at lines 48 to 67 all work by cleverly obtaining and using an appropriate iterator. Recall that the `insert` method inserts prior to a position, so `push_back` inserts prior to the endmarker, as required. In `pop_back`, note that `erase(-end())` creates a temporary iterator corresponding to the endmarker, retreats the temporary iterator, and uses that iterator to `erase`. Similar behavior occurs in `back`. Note also that in the case of the `pop_front` and `pop_back` operations, we again avoid dealing with node reclamation.

Figure 3.13 shows the `Node` class, consisting of the stored item, pointers to the previous and next `Node`, and a constructor. All the data members are public.

Figure 3.14 shows the `const_iterator` class, and Figure 3.15 shows the `iterator` class. As we mentioned earlier, the syntax at line 39 in Figure 3.15 indicates an advanced feature known as *inheritance* and means that `iterator` *IS-A* `const_iterator`. When the `iterator` class is written this way, it inherits all the data and methods from `const_iterator`. It may then add new data, add new methods, and override (i.e., redefine) existing methods. In the most general scenario, there is significant syntactical baggage (often resulting in the keyword `virtual` appearing in the code).

```
template <typename Object>
class List
{
  private:
    struct Node
      { /* See Figure 3.13 */ };

  public:
    class const_iterator
      { /* See Figure 3.14 */ };

    class iterator : public const_iterator
      { /* See Figure 3.15 */ };

  public:
    List( )
      { /* See Figure 3.16 */ }
    List( const List & rhs )
      { /* See Figure 3.16 */ }
    ~List( )
      { /* See Figure 3.16 */ }
    List & operator= ( const List & rhs )
      { /* See Figure 3.16 */ }
    List( List && rhs )
      { /* See Figure 3.16 */ }
    List & operator= ( List && rhs )
      { /* See Figure 3.16 */ }

    iterator begin( )
      { return { head->next }; }
    const_iterator begin( ) const
      { return { head->next }; }
    iterator end( )
      { return { tail }; }
    const_iterator end( ) const
      { return { tail }; }

    int size( ) const
      { return theSize; }
    bool empty( ) const
      { return size( ) == 0; }

    void clear( )
    {
        while( !empty( ) )
            pop_front( );
    }
```
Figure 3.11 List class (Part 1 of 2)

```
    Object & front( )
      { return *begin( ); }
    const Object & front( ) const
      { return *begin( ); }
    Object & back( )
      { return *--end( ); }
    const Object & back( ) const
      { return *--end( ); }
    void push_front( const Object & x )
      { insert( begin( ), x ); }
    void push_front( Object && x )
      { insert( begin( ), std::move( x ) ); }
    void push_back( const Object & x )
      { insert( end( ), x ); }
    void push_back( Object && x )
      { insert( end( ), std::move( x ) ); }
    void pop_front( )
      { erase( begin( ) ); }
    void pop_back( )
      { erase( --end( ) ); }

    iterator insert( iterator itr, const Object & x )
      { /* See Figure 3.18 */ }
    iterator insert( iterator itr, Object && x )
      { /* See Figure 3.18 */ }

    iterator erase( iterator itr )
      { /* See Figure 3.20 */ }
    iterator erase( iterator from, iterator to )
      { /* See Figure 3.20 */ }

  private:
    int   theSize;
    Node *head;
    Node *tail;

    void init( )
      { /* See Figure 3.16 */ }
};
```

Figure 3.12 List class (Part 2 of 2)

```
    struct Node
    {
        Object  data;
        Node   *prev;
        Node   *next;

        Node( const Object & d = Object{ }, Node * p = nullptr,
                                            Node * n = nullptr )
          : data{ d }, prev{ p }, next{ n } { }

        Node( Object && d, Node * p = nullptr, Node * n = nullptr )
          : data{ std::move( d ) }, prev{ p }, next{ n } { }
    };
```

Figure 3.13 Nested `Node` class for `List` class

However, in our case, we can avoid much of the syntactical baggage because we are not adding new data, nor are we intending to change the behavior of an existing method. We are, however, adding some new methods in the `iterator` class (with very similar signatures to the existing methods in the `const_iterator` class). As a result, we can avoid using `virtual`. Even so, there are quite a few syntax tricks in `const_iterator`.

At lines 28 and 29, `const_iterator` stores as its single data member a pointer to the "current" node. Normally, this would be private, but if it were private, then `iterator` would not have access to it. Marking members of `const_iterator` as `protected` allows the classes that inherit from `const_iterator` to have access to these members, but does not allow other classes to have access.

At lines 34 and 35 we see the constructor for `const_iterator` that was used in the `List` class implementation of `begin` and `end`. We don't want all classes to see this constructor (iterators are not supposed to be visibly constructed from pointer variables), so it can't be public, but we also want the `iterator` class to be able to see it, so logically this constructor is made protected. However, this doesn't give `List` access to the constructor. The solution is the **friend declaration** at line 37, which grants the `List` class access to `const_iterator`'s nonpublic members.

The public methods in `const_iterator` all use operator overloading. `operator==`, `operator!=`, and `operator*` are straightforward. At lines 10 to 21 we see the implementation of `operator++`. Recall that the prefix and postfix versions of `operator++` are completely different in semantics (and precedence), so we need to write separate routines for each form. They have the same name, so they must have different signatures to be distinguished. C++ requires that we give them different signatures by specifying an empty parameter list for the prefix form and a single (anonymous) `int` parameter for the postfix form. Then `++itr` calls the zero-parameter `operator++`; and `itr++` calls the one-parameter `operator++`. The `int` parameter is never used; it is present only to give a different signature. The implementation suggests that, in many cases where there is a choice between using the prefix or postfix `operator++`, the prefix form will be faster than the postfix form.

In the `iterator` class, the protected constructor at line 64 uses an initialization list to initialize the inherited current node. We do not have to reimplement `operator==`

```
class const_iterator
{
  public:
    const_iterator( ) : current{ nullptr }
      { }

    const Object & operator* ( ) const
      { return retrieve( ); }

    const_iterator & operator++ ( )
    {
        current = current->next;
        return *this;
    }

    const_iterator operator++ ( int )
    {
        const_iterator old = *this;
        ++( *this );
        return old;
    }

    bool operator== ( const const_iterator & rhs ) const
      { return current == rhs.current; }
    bool operator!= ( const const_iterator & rhs ) const
      { return !( *this == rhs ); }

  protected:
    Node *current;

    Object & retrieve( ) const
      { return current->data; }

    const_iterator( Node *p ) : current{ p }
      { }

    friend class List<Object>;
};
```

Figure 3.14 Nested `const_iterator` class for `List` class

and `operator!=` because those are inherited unchanged. We do provide a new pair of `operator++` implementations (because of the changed return type) that hide the originals in the `const_iterator`, and we provide an accessor/mutator pair for `operator*`. The accessor `operator*`, shown at lines 47 and 48, simply uses the same implementation as in `const_iterator`. The accessor is explicitly implemented in `iterator` because otherwise the original implementation is hidden by the newly added mutator version.

```
    class iterator : public const_iterator
    {
      public:
        iterator( )
          { }

        Object & operator* ( )
          { return const_iterator::retrieve( ); }
        const Object & operator* ( ) const
          { return const_iterator::operator*( ); }

        iterator & operator++ ( )
        {
            this->current = this->current->next;
            return *this;
        }

        iterator operator++ ( int )
        {
            iterator old = *this;
            ++( *this );
            return old;
        }

      protected:
        iterator( Node *p ) : const_iterator{ p }
          { }

        friend class List<Object>;
    };
```

Figure 3.15 Nested `iterator` class for `List` class

Figure 3.16 shows the constructor and Big-Five. Because the zero-parameter constructor and copy constructor must both allocate the header and tail nodes, we provide a private `init` routine. `init` creates an empty `List`. The destructor reclaims the header and tail nodes; all the other nodes are reclaimed when the destructor invokes `clear`. Similarly, the copy constructor is implemented by invoking public methods rather than attempting low-level pointer manipulations.

Figure 3.17 illustrates how a new node containing `x` is spliced in between a node pointed at by `p` and `p.prev`. The assignment to the node pointers can be described as follows:

```
Node *newNode = new Node{ x, p->prev, p };  // Steps 1 and 2
p->prev->next = newNode;                    // Step 3
p->prev = newNode;                          // Step 4
```

```
    List( )
      { init( ); }

    ~List( )
    {
        clear( );
        delete head;
        delete tail;
    }

    List( const List & rhs )
    {
        init( );
        for( auto & x : rhs )
            push_back( x );
    }

    List & operator= ( const List & rhs )
    {
        List copy = rhs;
        std::swap( *this, copy );
        return *this;
    }


    List( List && rhs )
      : theSize{ rhs.theSize }, head{ rhs.head }, tail{ rhs.tail }
    {
        rhs.theSize = 0;
        rhs.head = nullptr;
        rhs.tail = nullptr;
    }

    List & operator= ( List && rhs )
    {
        std::swap( theSize, rhs.theSize );
        std::swap( head, rhs.head );
        std::swap( tail, rhs.tail );

        return *this;
    }

    void init( )
    {
        theSize = 0;
        head = new Node;
        tail = new Node;
        head->next = tail;
        tail->prev = head;
    }
```

Figure 3.16 Constructor, Big-Five, and private `init` routine for `List` class

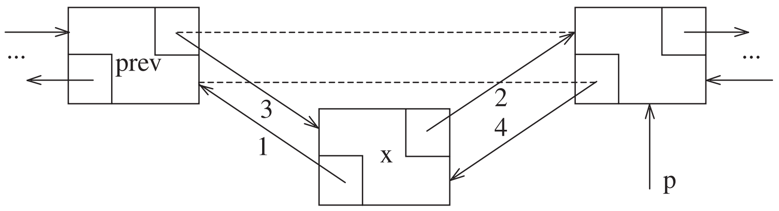


Figure 3.17 Insertion in a doubly linked list by getting a new node and then changing pointers in the order indicated

Steps 3 and 4 can be combined, yielding only two lines:

```
Node *newNode = new Node{ x, p->prev, p };  // Steps 1 and 2
p->prev = p->prev->next = newNode;          // Steps 3 and 4
```

But then these two lines can also be combined, yielding:

```
p->prev = p->prev->next = new Node{ x, p->prev, p };
```

This makes the `insert` routine in Figure 3.18 short.

Figure 3.19 shows the logic of removing a node. If `p` points to the node being removed, only two pointers change before the node can be reclaimed:

```
p->prev->next = p->next;
p->next->prev = p->prev;
delete p;
```

Figure 3.20 shows a pair of `erase` routines. The first version of `erase` contains the three lines of code shown above and the code to return an `iterator` representing the item after

```
    // Insert x before itr.
    iterator insert( iterator itr, const Object & x )
    {
        Node *p = itr.current;
        theSize++;
        return { p->prev = p->prev->next = new Node{ x, p->prev, p } };
    }

    // Insert x before itr.
    iterator insert( iterator itr, Object && x )
    {
        Node *p = itr.current;
        theSize++;
        return { p->prev = p->prev->next
                             = new Node{ std::move( x ), p->prev, p } };
    }
```

Figure 3.18 `insert` routine for `List` class

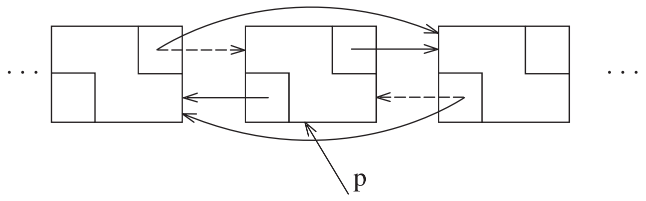


Figure 3.19 Removing node specified by `p` from a doubly linked list

```
 1      // Erase item at itr.
 2      iterator erase( iterator itr )
 3      {
 4          Node *p = itr.current;
 5          iterator retVal{ p->next };
 6          p->prev->next = p->next;
 7          p->next->prev = p->prev;
 8          delete p;
 9          theSize--;
10
11          return retVal;
12      }
13
14      iterator erase( iterator from, iterator to )
15      {
16          for( iterator itr = from; itr != to; )
17              itr = erase( itr );
18
19          return to;
20      }
```

Figure 3.20 `erase` routines for `List` class

the erased element. Like `insert`, `erase` must update `theSize`. The second version of `erase` simply uses an `iterator` to call the first version of `erase`. Note that we cannot simply use `itr++` in the `for` loop at line 16 and ignore the return value of `erase` at line 17. The value of `itr` is stale immediately after the call to `erase`, which is why `erase` returns an `iterator`.

In examining the code, we can see a host of errors that can occur and for which no checks are provided. For instance, iterators passed to `erase` and `insert` can be uninitialized or for the wrong list! Iterators can have `++` or `*` applied to them when they are already at the endmarker or are uninitialized.

An uninitialized iterator will have `current` pointing at `nullptr`, so that condition is easily tested. The endmarker's `next` pointer points at `nullptr`, so testing for `++` or `*` on an endmarker condition is also easy. However, in order to determine if an iterator passed to `erase` or `insert` is an iterator for the correct list, the iterator must store an additional data member representing a pointer to the `List` from which it was constructed.

```
  protected:
    const List<Object> *theList;
    Node *current;

    const_iterator( const List<Object> & lst, Node *p )
      : theList{ &lst }, current{ p }
    {
    }

    void assertIsValid( ) const
    {
        if( theList == nullptr || current == nullptr || current == theList->head )
            throw IteratorOutOfBoundsException{ };
    }
```

Figure 3.21 Revised protected section of `const_iterator` that incorporates ability to perform additional error checks

We will sketch the basic idea and leave the details as an exercise. In the `const_iterator` class, we add a pointer to the `List` and modify the protected constructor to take the `List` as a parameter. We can also add methods that throw an exception if certain assertions aren't met. The revised protected section looks something like the code in Figure 3.21. Then all calls to `iterator` and `const_iterator` constructors that formerly took one parameter now take two, as in the `begin` method for `List`:

```
const_iterator begin( ) const
{
    const_iterator itr{ *this, head };
    return ++itr;
}
```

Then `insert` can be revised to look something like the code in Figure 3.22. We leave the details of these modifications as an exercise.

```
    // Insert x before itr.
    iterator insert( iterator itr, const Object & x )
    {
        itr.assertIsValid( );
        if( itr.theList != this )
            throw IteratorMismatchException{ };

        Node *p = itr.current;
        theSize++;
        return { *this, p->prev = p->prev->next = new Node{ x, p->prev, p } };
    }
```

Figure 3.22 `List insert` with additional error checks

3.6 The Stack ADT

A **stack** is a list with the restriction that insertions and deletions can be performed in only one position, namely, the end of the list, called the **top.**

3.6.1 Stack Model

The fundamental operations on a stack are `push`, which is equivalent to an insert, and `pop`, which deletes the most recently inserted element. The most recently inserted element can be examined prior to performing a `pop` by use of the `top` routine. A `pop` or `top` on an empty stack is generally considered an error in the stack ADT. On the other hand, running out of space when performing a `push` is an implementation limit but not an ADT error.

Stacks are sometimes known as LIFO (last in, first out) lists. The model depicted in Figure 3.23 signifies only that `push`es are input operations and `pop`s and `top`s are output. The usual operations to make empty stacks and test for emptiness are part of the repertoire, but essentially all that you can do to a stack is `push` and `pop`.

Figure 3.24 shows an abstract stack after several operations. The general model is that there is some element that is at the top of the stack, and it is the only element that is visible.

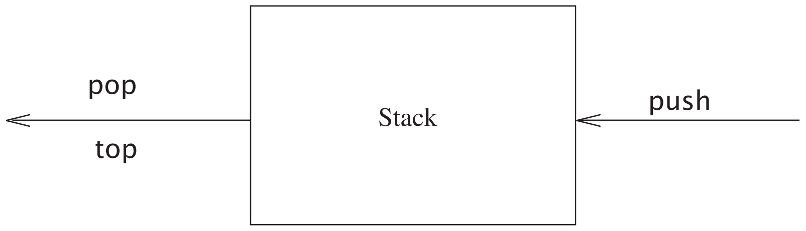


Figure 3.23 Stack model: Input to a stack is by `push`; output is by `pop` and `top`

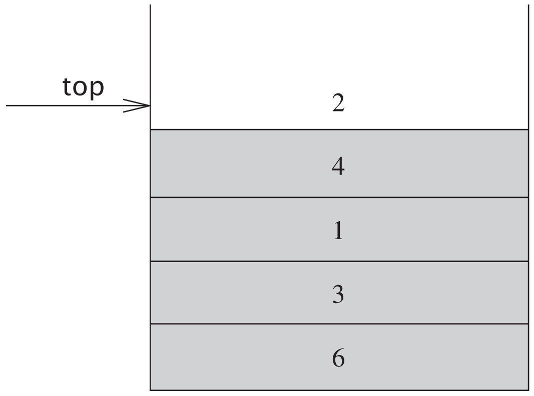


Figure 3.24 Stack model: Only the top element is accessible

3.6.2 Implementation of Stacks

Since a stack is a list, any list implementation will do. Clearly `list` and `vector` support stack operations; 99% of the time they are the most reasonable choice. Occasionally it can be faster to design a special-purpose implementation. Because stack operations are constant-time operations, this is unlikely to yield any discernable improvement except under very unique circumstances.

For these special times, we will give two popular stack implementations. One uses a linked structure, and the other uses an array, and both simplify the logic in `vector` and `list`, so we do not provide code.

Linked List Implementation of Stacks

The first implementation of a stack uses a singly linked list. We perform a `push` by inserting at the front of the list. We perform a `pop` by deleting the element at the front of the list. A `top` operation merely examines the element at the front of the list, returning its value. Sometimes the `pop` and `top` operations are combined into one.

Array Implementation of Stacks

An alternative implementation avoids links and is probably the more popular solution. It uses the `back`, `push_back`, and `pop_back` implementation from `vector`, so the implementation is trivial. Associated with each stack is `theArray` and `topOfStack`, which is -1 for an empty stack (this is how an empty stack is initialized). To push some element `x` onto the stack, we increment `topOfStack` and then set `theArray[topOfStack] = x`. To pop, we set the return value to `theArray[topOfStack]` and then decrement `topOfStack`.

Notice that these operations are performed in not only constant time but very fast constant time. On some machines, `push`es and `pop`s (of integers) can be written in one machine instruction, operating on a register with auto-increment and auto-decrement addressing. The fact that most modern machines have stack operations as part of the instruction set enforces the idea that the stack is probably the most fundamental data structure in computer science, after the array.

3.6.3 Applications

It should come as no surprise that if we restrict the operations allowed on a list, those operations can be performed very quickly. The big surprise, however, is that the small number of operations left are so powerful and important. We give three of the many applications of stacks. The third application gives a deep insight into how programs are organized.

Balancing Symbols

Compilers check your programs for syntax errors, but frequently a lack of one symbol (such as a missing brace or comment starter) can cause the compiler to spill out a hundred lines of diagnostics without identifying the real error.

A useful tool in this situation is a program that checks whether everything is balanced. Thus, every right brace, bracket, and parenthesis must correspond to its left counterpart.

The sequence `[()]` is legal, but `[(])` is wrong. Obviously, it is not worthwhile writing a huge program for this, but it turns out that it is easy to check these things. For simplicity, we will just check for balancing of parentheses, brackets, and braces and ignore any other character that appears.

The simple algorithm uses a stack and is as follows:

> Make an empty stack. Read characters until end of file. If the character is an opening symbol, push it onto the stack. If it is a closing symbol and the stack is empty, report an error. Otherwise, pop the stack. If the symbol popped is not the corresponding opening symbol, then report an error. At end of file, if the stack is not empty, report an error.

You should be able to convince yourself that this algorithm works. It is clearly linear and actually makes only one pass through the input. It is thus online and quite fast. Extra work can be done to attempt to decide what to do when an error is reported—such as identifying the likely cause.

Postfix Expressions

Suppose we have a pocket calculator and would like to compute the cost of a shopping trip. To do so, we add a list of numbers and multiply the result by 1.06; this computes the purchase price of some items with local sales tax added. If the items are 4.99, 5.99, and 6.99, then a natural way to enter this would be the sequence

$$4.99 + 5.99 + 6.99 * 1.06 =$$

Depending on the calculator, this produces either the intended answer, 19.05, or the scientific answer, 18.39. Most simple four-function calculators will give the first answer, but many advanced calculators know that multiplication has higher precedence than addition.

On the other hand, some items are taxable and some are not, so if only the first and last items were actually taxable, then the sequence

$$4.99 * 1.06 + 5.99 + 6.99 * 1.06 =$$

would give the correct answer (18.69) on a scientific calculator and the wrong answer (19.37) on a simple calculator. A scientific calculator generally comes with parentheses, so we can always get the right answer by parenthesizing, but with a simple calculator we need to remember intermediate results.

A typical evaluation sequence for this example might be to multiply 4.99 and 1.06, saving this answer as A_1. We then add 5.99 and A_1, saving the result in A_1. We multiply 6.99 and 1.06, saving the answer in A_2, and finish by adding A_1 and A_2, leaving the final answer in A_1. We can write this sequence of operations as follows:

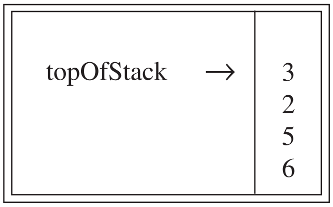
$$4.99\ 1.06 * 5.99 + 6.99\ 1.06\ \ * +$$

This notation is known as **postfix**, or **reverse Polish notation**, and is evaluated exactly as we have described above. The easiest way to do this is to use a stack. When a number is seen, it is pushed onto the stack; when an operator is seen, the operator is applied to the two numbers (symbols) that are popped from the stack, and the result is pushed onto the stack. For instance, the postfix expression

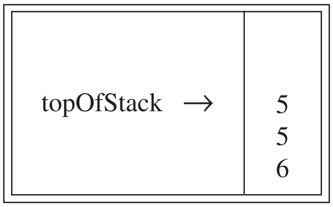
$$6\ 5\ 2\ 3 + 8 * +3 + *$$

is evaluated as follows:

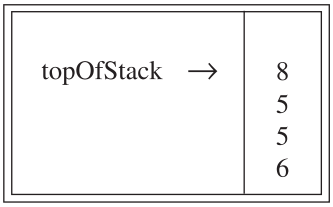
The first four symbols are placed on the stack. The resulting stack is



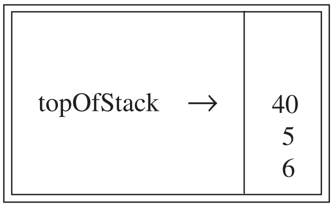
Next, a '+' is read, so 3 and 2 are popped from the stack, and their sum, 5, is pushed.



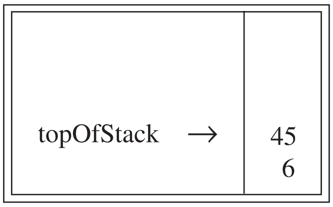
Next, 8 is pushed.



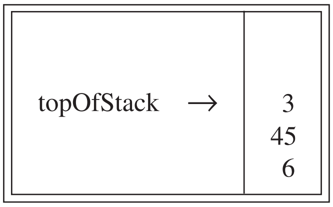
Now a '*' is seen, so 8 and 5 are popped, and $5 * 8 = 40$ is pushed.



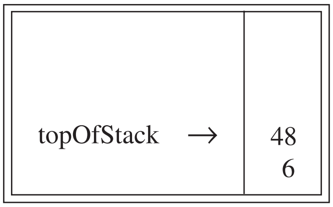
Next, a '+' is seen, so 40 and 5 are popped, and $5 + 40 = 45$ is pushed.



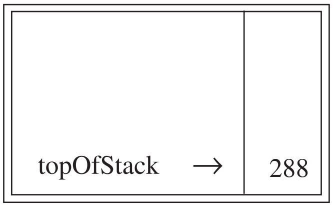
Now, 3 is pushed.



Next, '+' pops 3 and 45 and pushes $45 + 3 = 48$.



Finally, a '∗' is seen and 48 and 6 are popped; the result, $6 * 48 = 288$, is pushed.



The time to evaluate a postfix expression is $O(N)$, because processing each element in the input consists of stack operations and therefore takes constant time. The algorithm to do so is very simple. Notice that when an expression is given in postfix notation, there is no need to know any precedence rules; this is an obvious advantage.

Infix to Postfix Conversion

Not only can a stack be used to evaluate a postfix expression, but we can also use a stack to convert an expression in standard form (otherwise known as **infix**) into postfix. We will concentrate on a small version of the general problem by allowing only the operators `+`, `*`, `(`, `)`, and insisting on the usual precedence rules. We will further assume that the expression is legal. Suppose we want to convert the infix expression

```
a + b * c + ( d * e + f ) * g
```

into postfix. A correct answer is `a b c * + d e * f + g * +`.

When an operand is read, it is immediately placed onto the output. Operators are not immediately output, so they must be saved somewhere. The correct thing to do is to place operators that have been seen, but not placed on the output, onto the stack. We will also stack left parentheses when they are encountered. We start with an initially empty stack.

If we see a right parenthesis, then we pop the stack, writing symbols until we encounter a (corresponding) left parenthesis, which is popped but not output.

If we see any other symbol (`+`, `*`, `(`), then we pop entries from the stack until we find an entry of lower priority. One exception is that we never remove a `(` from the stack except when processing a `)`. For the purposes of this operation, `+` has lowest priority and `(` highest. When the popping is done, we push the operator onto the stack.

Finally, if we read the end of input, we pop the stack until it is empty, writing symbols onto the output.

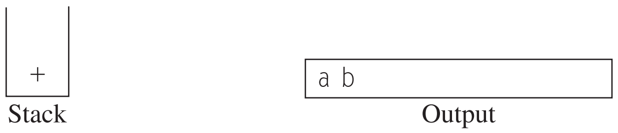
The idea of this algorithm is that when an operator is seen, it is placed on the stack. The stack represents pending operators. However, some of the operators on the stack that have high precedence are now known to be completed and should be popped, as they will no longer be pending. Thus prior to placing the operator on the stack, operators that are on the stack, and which are to be completed prior to the current operator, are popped. This is illustrated in the following table:

Expression	Stack When Third Operator Is Processed	Action
`a*b-c+d`	`-`	`-` is completed; `+` is pushed
`a/b+c*d`	`+`	Nothing is completed; `*` is pushed
`a-b*c/d`	`- *`	`*` is completed; `/` is pushed
`a-b*c+d`	`- *`	`*` and `-` are completed; `+` is pushed

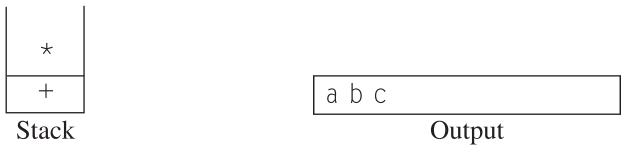
Parentheses simply add an additional complication. We can view a left parenthesis as a high-precedence operator when it is an input symbol (so that pending operators remain pending) and a low-precedence operator when it is on the stack (so that it is not accidentally removed by an operator). Right parentheses are treated as the special case.

To see how this algorithm performs, we will convert the long infix expression above into its postfix form. First, the symbol `a` is read, so it is passed through to the output.

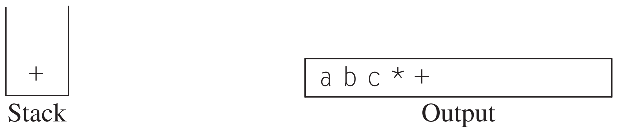
Then + is read and pushed onto the stack. Next b is read and passed through to the output. The state of affairs at this juncture is as follows:



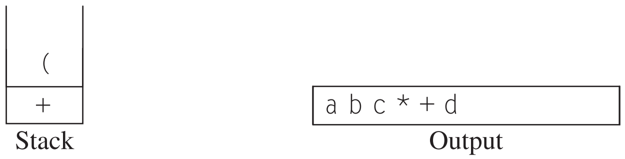
Next, a * is read. The top entry on the operator stack has lower precedence than *, so nothing is output and * is put on the stack. Next, c is read and output. Thus far, we have



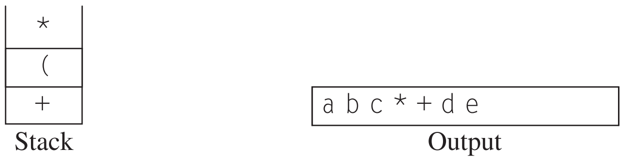
The next symbol is a +. Checking the stack, we find that we will pop a * and place it on the output; pop the other +, which is not of *lower* but equal priority, on the stack; and then push the +.



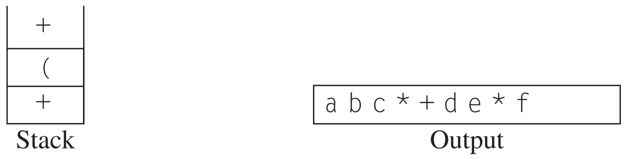
The next symbol read is a (. Being of highest precedence, this is placed on the stack. Then d is read and output.



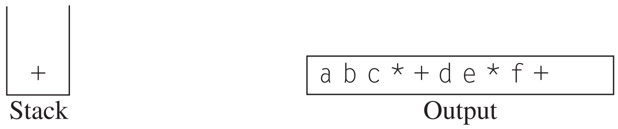
We continue by reading a *. Since open parentheses do not get removed except when a closed parenthesis is being processed, there is no output. Next, e is read and output.



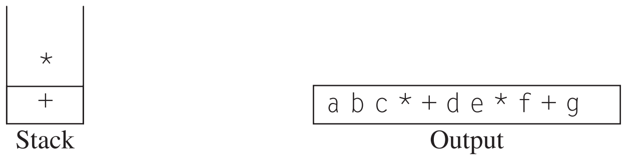
The next symbol read is a `+`. We pop and output `*` and then push `+`. Then we read and output `f`.



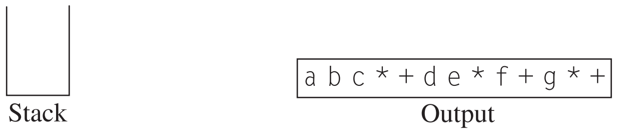
Now we read a `)`, so the stack is emptied back to the `(`. We output a `+`.



We read a `*` next; it is pushed onto the stack. Then `g` is read and output.



The input is now empty, so we pop and output symbols from the stack until it is empty.



As before, this conversion requires only $O(N)$ time and works in one pass through the input. We can add subtraction and division to this repertoire by assigning subtraction and addition equal priority and multiplication and division equal priority. A subtle point is that the expression `a - b - c` will be converted to `a b - c -` and not `a b c - -`. Our algorithm does the right thing, because these operators associate from left to right. This is not necessarily the case in general, since exponentiation associates right to left: $2^{2^3} = 2^8 = 256$, not $4^3 = 64$. We leave as an exercise the problem of adding exponentiation to the repertoire of operators.

Function Calls

The algorithm to check balanced symbols suggests a way to implement function calls in compiled procedural and object-oriented languages. The problem here is that when a call is made to a new function, all the variables local to the calling routine need to be saved by the system, since otherwise the new function will overwrite the memory used by the calling routine's variables. Furthermore, the current location in the routine must be saved so that the new function knows where to go after it is done. The variables have generally been assigned by the compiler to machine registers, and there are certain to be conflicts (usually all functions get some variables assigned to register #1), especially if recursion is involved. The reason that this problem is similar to balancing symbols is that a function call and function return are essentially the same as an open parenthesis and closed parenthesis, so the same ideas should work.

When there is a function call, all the important information that needs to be saved, such as register values (corresponding to variable names) and the return address (which can be obtained from the program counter, which is typically in a register), is saved "on a piece of paper" in an abstract way and put at the top of a pile. Then the control is transferred to the new function, which is free to replace the registers with its values. If it makes other function calls, it follows the same procedure. When the function wants to return, it looks at the "paper" at the top of the pile and restores all the registers. It then makes the return jump.

Clearly, all of this work can be done using a stack, and that is exactly what happens in virtually every programming language that implements recursion. The information saved is called either an **activation record** or **stack frame**. Typically, a slight adjustment is made: The current environment is represented at the top of the stack. Thus, a return gives the previous environment (without copying). The stack in a real computer frequently grows from the high end of your memory partition downward, and on many systems there is no checking for overflow. There is always the possibility that you will run out of stack space by having too many simultaneously active functions. Needless to say, running out of stack space is always a fatal error.

In languages and systems that do not check for stack overflow, programs crash without an explicit explanation. In normal events, you should not run out of stack space; doing so is usually an indication of runaway recursion (forgetting a base case). On the other hand, some perfectly legal and seemingly innocuous programs can cause you to run out of stack space. The routine in Figure 3.25, which prints out a container, is perfectly legal and actually correct. It properly handles the base case of an empty container, and the recursion is fine. This program can be *proven* correct. Unfortunately, if the container

```
/**
 * Print container from start up to but not including end.
 */
template <typename Iterator>
void print( Iterator start, Iterator end, ostream & out = cout )
{
    if( start == end )
        return;

    out << *start++ << endl;   // Print and advance start
    print( start, end, out );
}
```

Figure 3.25 A bad use of recursion: printing a container

```
/**
 * Print container from start up to but not including end.
 */
template <typename Iterator>
void print( Iterator start, Iterator end, ostream & out = cout )
{
    while( true )
    {
        if( start == end )
            return;

        out << *start++ << endl;   // Print and advance start
    }
}
```

Figure 3.26 Printing a container without recursion; a compiler might do this (you should not)

contains 200,000 elements to print, there will be a stack of 200,000 activation records representing the nested calls of line 11. Activation records are typically large because of all the information they contain, so this program is likely to run out of stack space. (If 200,000 elements are not enough to make the program crash, replace the number with a larger one.)

This program is an example of an extremely bad use of recursion known as **tail recursion.** Tail recursion refers to a recursive call at the last line. Tail recursion can be mechanically eliminated by enclosing the body in a `while` loop and replacing the recursive call with one assignment per function argument. This simulates the recursive call because nothing needs to be saved; after the recursive call finishes, there is really no need to know the saved values. Because of this, we can just go to the top of the function with the values that would have been used in a recursive call. The function in Figure 3.26 shows the mechanically improved version generated by this algorithm. Removal of tail recursion is so simple that some compilers do it automatically. Even so, it is best not to find out that yours does not.

Recursion can always be completely removed (compilers do so in converting to assembly language), but doing so can be quite tedious. The general strategy requires using a stack and is worthwhile only if you can manage to put the bare minimum on the stack. We will not dwell on this further, except to point out that although nonrecursive programs are certainly generally faster than equivalent recursive programs, the speed advantage rarely justifies the lack of clarity that results from removing the recursion.

3.7 The Queue ADT

Like stacks, **queues** are lists. With a queue, however, insertion is done at one end whereas deletion is performed at the other end.

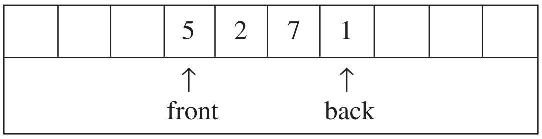
3.7.1 Queue Model

The basic operations on a queue are `enqueue`, which inserts an element at the end of the list (called the rear), and `dequeue`, which deletes (and returns) the element at the start of the list (known as the front). Figure 3.27 shows the abstract model of a queue.

3.7.2 Array Implementation of Queues

As with stacks, any list implementation is legal for queues. Like stacks, both the linked list and array implementations give fast $O(1)$ running times for every operation. The linked list implementation is straightforward and left as an exercise. We will now discuss an array implementation of queues.

For each queue data structure, we keep an array, `theArray`, and the positions `front` and `back`, which represent the ends of the queue. We also keep track of the number of elements that are actually in the queue, `currentSize`. The following table shows a queue in some intermediate state.



The operations should be clear. To `enqueue` an element `x`, we increment `currentSize` and `back`, then set `theArray[back] = x`. To `dequeue` an element, we set the return value to `theArray[front]`, decrement `currentSize`, and then increment `front`. Other strategies are possible (this is discussed later). We will comment on checking for errors presently.

There is one potential problem with this implementation. After 10 `enqueue`s, the queue appears to be full, since `back` is now at the last array index, and the next `enqueue` would be in a nonexistent position. However, there might only be a few elements in the queue, because several elements may have already been dequeued. Queues, like stacks, frequently stay small even in the presence of a lot of operations.

The simple solution is that whenever `front` or `back` gets to the end of the array, it is wrapped around to the beginning. The following tables show the queue during some operations. This is known as a **circular array** implementation.

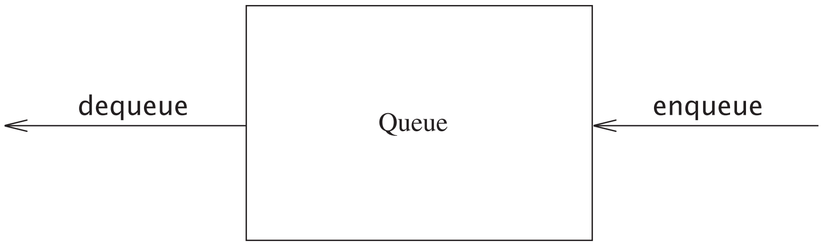
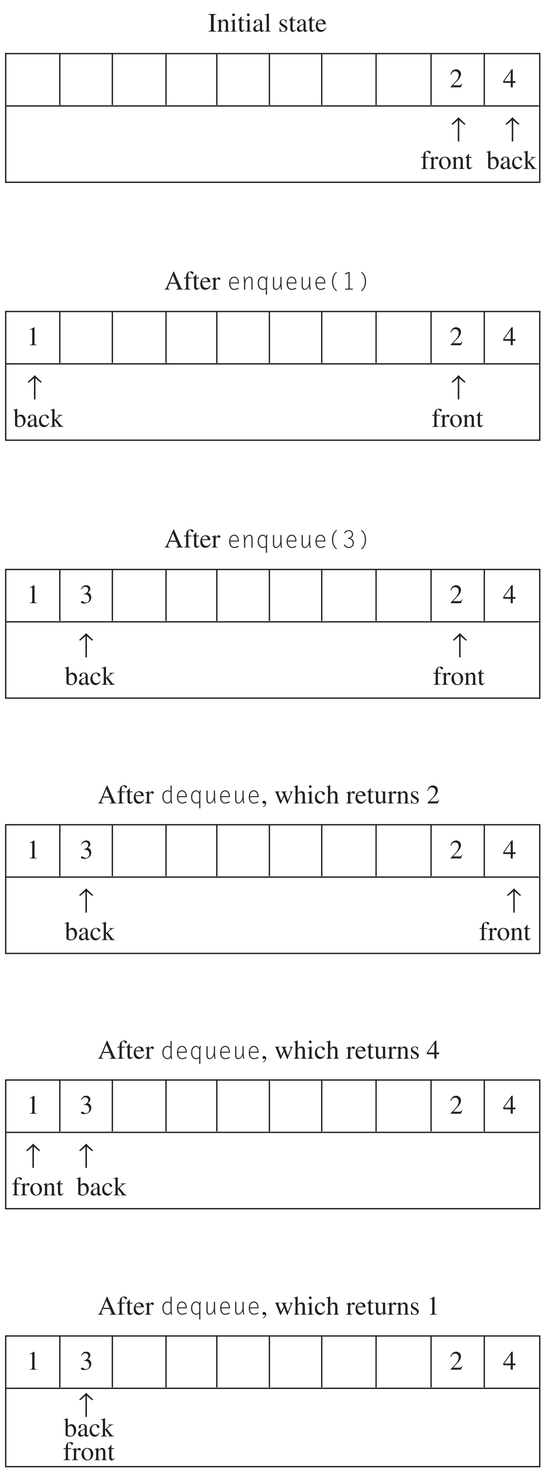
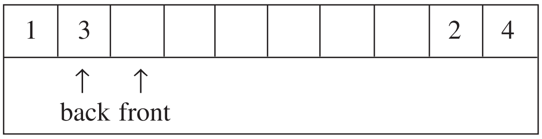


Figure 3.27 Model of a queue


Initial state
2 4
front back
After enqueue(1)
1 2 4
back front
After enqueue(3)
1 3 2 4
back front
After dequeue, which returns 2
1 3 2 4
back front
After dequeue, which returns 4
1 3 2 4
front back
After dequeue, which returns 1
1 3 2 4
back
front

After `dequeue`, which returns 3
and makes the queue empty



The extra code required to implement the wraparound is minimal (although it probably doubles the running time). If incrementing either `back` or `front` causes it to go past the array, the value is reset to the first position in the array.

Some programmers use different ways of representing the front and back of a queue. For instance, some do not use an entry to keep track of the size, because they rely on the base case that when the queue is empty, `back = front-1`. The size is computed implicitly by comparing `back` and `front`. This is a very tricky way to go, because there are some special cases, so be very careful if you need to modify code written this way. If the `currentSize` is not maintained as an explicit data member, then the queue is full when there are `theArray.capacity()-1` elements, since only `theArray.capacity()` different sizes can be differentiated and one of these is 0. Pick any style you like and make sure that all your routines are consistent. Since there are a few options for implementation, it is probably worth a comment or two in the code if you don't use the `currentSize` data member.

In applications where you are sure that the number of `enqueue`s is not larger than the capacity of the queue, the wraparound is not necessary. As with stacks, `dequeue`s are rarely performed unless the calling routines are certain that the queue is not empty. Thus error checks are frequently skipped for this operation, except in critical code. This is generally not justifiable, because the time savings that you are likely to achieve are minimal.

3.7.3 Applications of Queues

There are many algorithms that use queues to give efficient running times. Several of these are found in graph theory, and we will discuss them in Chapter 9. For now, we will give some simple examples of queue usage.

When jobs are submitted to a printer, they are arranged in order of arrival. Thus, essentially, jobs sent to a printer are placed on a queue.[1]

Virtually every real-life line is (supposed to be) a queue. For instance, lines at ticket counters are queues, because service is first-come first-served.

Another example concerns computer networks. There are many network setups of personal computers in which the disk is attached to one machine, known as the **file server.** Users on other machines are given access to files on a first-come first-served basis, so the data structure is a queue.

[1] We say *essentially* because jobs can be killed. This amounts to a deletion from the middle of the queue, which is a violation of the strict definition.

Further examples include the following:

- Calls to large companies are generally placed on a queue when all operators are busy.
- In large universities, where resources are limited, students must sign a waiting list if all computers are occupied. The student who has been at a computer the longest is forced off first, and the student who has been waiting the longest is the next user to be allowed on.

A whole branch of mathematics known as **queuing theory** deals with computing, probabilistically, how long users expect to wait on a line, how long the line gets, and other such questions. The answer depends on how frequently users arrive to the line and how long it takes to process a user once the user is served. Both of these parameters are given as probability distribution functions. In simple cases, an answer can be computed analytically. An example of an easy case would be a phone line with one operator. If the operator is busy, callers are placed on a waiting line (up to some maximum limit). This problem is important for businesses, because studies have shown that people are quick to hang up the phone.

If there are k operators, then this problem is much more difficult to solve. Problems that are difficult to solve analytically are often solved by a simulation. In our case, we would need to use a queue to perform the simulation. If k is large, we also need other data structures to do this efficiently. We shall see how to do this simulation in Chapter 6. We could then run the simulation for several values of k and choose the minimum k that gives a reasonable waiting time.

Additional uses for queues abound, and as with stacks, it is staggering that such a simple data structure can be so important.

Summary

This chapter describes the concept of ADTs and illustrates the concept with three of the most common abstract data types. The primary objective is to separate the implementation of the ADTs from their function. The program must know what the operations do, but it is actually better off not knowing how it is done.

Lists, stacks, and queues are perhaps the three fundamental data structures in all of computer science, and their use is documented through a host of examples. In particular, we saw how stacks are used to keep track of function calls and how recursion is actually implemented. This is important to understand, not just because it makes procedural languages possible, but because knowing how recursion is implemented removes a good deal of the mystery that surrounds its use. Although recursion is very powerful, it is not an entirely free operation; misuse and abuse of recursion can result in programs crashing.

Exercises

3.1 You are given a list, `L`, and another list, `P`, containing integers sorted in ascending order. The operation `printLots(L,P)` will print the elements in `L` that are in positions specified by `P`. For instance, if `P` $= 1, 3, 4, 6$, the elements in positions 1, 3, 4, and 6 in `L` are printed. Write the procedure `printLots(L,P)`. You may use only the public STL container operations. What is the running time of your procedure?

3.2 Swap two adjacent elements by adjusting only the links (and not the data) using
a. singly linked lists
b. doubly linked lists

3.3 Implement the STL `find` routine that returns the `iterator` containing the first occurrence of `x` in the range that begins at `start` and extends up to but not including `end`. If `x` is not found, `end` is returned. This is a nonclass (global function) with signature

```
template <typename Iterator, typename Object>
iterator find( Iterator start, Iterator end, const Object & x );
```

3.4 Given two sorted lists, L_1 and L_2, write a procedure to compute $L_1 \cap L_2$ using only the basic list operations.

3.5 Given two sorted lists, L_1 and L_2, write a procedure to compute $L_1 \cup L_2$ using only the basic list operations.

3.6 The *Josephus problem* is the following game: N people, numbered 1 to N, are sitting in a circle. Starting at person 1, a hot potato is passed. After M passes, the person holding the hot potato is eliminated, the circle closes ranks, and the game continues with the person who was sitting after the eliminated person picking up the hot potato. The last remaining person wins. Thus, if $M = 0$ and $N = 5$, players are eliminated in order, and player 5 wins. If $M = 1$ and $N = 5$, the order of elimination is 2, 4, 1, 5.
a. Write a program to solve the Josephus problem for general values of M and N. Try to make your program as efficient as possible. Make sure you dispose of cells.
b. What is the running time of your program?
c. If $M = 1$, what is the running time of your program? How is the actual speed affected by the `delete` routine for large values of N ($N > 100{,}000$)?

3.7 Modify the `Vector` class to add bounds checks for indexing.

3.8 Add `insert` and `erase` to the `Vector` class.

3.9 According to the C++ standard, for the `vector`, a call to `push_back`, `pop_back`, `insert`, or `erase` invalidates (potentially makes stale) all iterators viewing the `vector`. Why?

3.10 Modify the `Vector` class to provide stringent iterator checking by making iterators class types rather than pointer variables. The hardest part is dealing with stale iterators, as described in Exercise 3.9.

3.11 Assume that a singly linked list is implemented with a header node, but no tail node, and that it maintains only a pointer to the header node. Write a class that includes methods to
a. return the size of the linked list
b. print the linked list
c. test if a value `x` is contained in the linked list
d. add a value `x` if it is not already contained in the linked list
e. remove a value `x` if it is contained in the linked list

3.12 Repeat Exercise 3.11, maintaining the singly linked list in sorted order.

3.13 Add support for `operator-` to the `List` iterator classes.

3.14 Looking ahead in an STL iterator requires an application of `operator++`, which in turn advances the iterator. In some cases looking at the next item in the list, without advancing to it, may be preferable. Write the member function with the declaration

```
const_iterator operator+( int k ) const;
```

to facilitate this in a general case. The binary `operator+` returns an iterator that corresponds to `k` positions ahead of current.

3.15 Add the `splice` operation to the `List` class. The method declaration

```
void splice( iterator position, List<T> & lst );
```

removes all the items from `lst`, placing them prior to `position` in `List *this. lst` and `*this` must be different lists. Your routine must run in constant time.

3.16 Add reverse iterators to the STL `List` class implementation. Define `reverse_iterator` and `const_reverse_iterator`. Add the methods `rbegin` and `rend` to return appropriate reverse iterators representing the position prior to the endmarker and the position that is the header node. Reverse iterators internally reverse the meaning of the `++` and `--` operators. You should be able to print a list `L` in reverse by using the code

```
List<Object>::reverse_iterator itr = L.rbegin( );
while( itr != L.rend( ) )
    cout << *itr++ << endl;
```

3.17 Modify the `List` class to provide stringent iterator checking by using the ideas suggested at the end of Section 3.5.

3.18 When an `erase` method is applied to a `list`, it invalidates any `iterator` that is referencing the removed node. Such an iterator is called **stale**. Describe an efficient algorithm that guarantees that any operation on a stale iterator acts as though the iterator's `current` is `nullptr`. Note that there may be many stale iterators. You must explain which classes need to be rewritten in order to implement your algorithm.

3.19 Rewrite the `List` class without using header and tail nodes and describe the differences between the class and the class provided in Section 3.5.

3.20 An alternative to the deletion strategy we have given is to use **lazy deletion.** To delete an element, we merely mark it deleted (using an extra bit field). The number of deleted and nondeleted elements in the list is kept as part of the data structure. If there are as many deleted elements as nondeleted elements, we traverse the entire list, performing the standard deletion algorithm on all marked nodes.

a. List the advantages and disadvantages of lazy deletion.

b. Write routines to implement the standard linked list operations using lazy deletion.

3.21 Write a program to check for balancing symbols in the following languages:

a. Pascal (`begin/end`, `()`, `[]`, `{}`).

b. C++ (`/* */`, `()`, `[]`, `{}`).

* c. Explain how to print out an error message that is likely to reflect the probable cause.

3.22 Write a program to evaluate a postfix expression.

3.23 a. Write a program to convert an infix expression that includes `(`, `)`, `+`, `-`, `*`, and `/` to postfix.
b. Add the exponentiation operator to your repertoire.
c. Write a program to convert a postfix expression to infix.

3.24 Write routines to implement two stacks using only one array. Your stack routines should not declare an overflow unless every slot in the array is used.

3.25 * a. Propose a data structure that supports the stack `push` and `pop` operations and a third operation `findMin`, which returns the smallest element in the data structure, all in $O(1)$ worst-case time.
* b. Prove that if we add the fourth operation `deleteMin` which finds and removes the smallest element, then at least one of the operations must take $\Omega(\log N)$ time. (This requires reading Chapter 7.)

* **3.26** Show how to implement three stacks in one array.

3.27 If the recursive routine in Section 2.4 used to compute Fibonacci numbers is run for $N = 50$, is stack space likely to run out? Why or why not?

3.28 A *deque* is a data structure consisting of a list of items on which the following operations are possible:

`push(x)`: Insert item `x` on the front end of the deque.

`pop()`: Remove the front item from the deque and return it.

`inject(x)`: Insert item `x` on the rear end of the deque.

`eject()`: Remove the rear item from the deque and return it.

Write routines to support the deque that take $O(1)$ time per operation.

3.29 Write an algorithm for printing a singly linked list in reverse, using only constant extra space. This instruction implies that you cannot use recursion but you may assume that your algorithm is a list member function. Can such an algorithm be written if the routine is a constant member function?

3.30 a. Write an array implementation of self-adjusting lists. In a **self-adjusting list**, all insertions are performed at the front. A self-adjusting list adds a `find` operation, and when an element is accessed by a `find`, it is moved to the front of the list without changing the relative order of the other items.
b. Write a linked list implementation of self-adjusting lists.
* c. Suppose each element has a fixed probability, p_i, of being accessed. Show that the elements with highest access probability are expected to be close to the front.

3.31 Efficiently implement a stack class using a singly linked list, with no header or tail nodes.

3.32 Efficiently implement a queue class using a singly linked list, with no header or tail nodes.

3.33 Efficiently implement a queue class using a circular array. You may use a `vector` (rather than a primitive array) as the underlying array structure.

3.34 A linked list contains a cycle if, starting from some node p, following a sufficient number of `next` links brings us back to node p. p does not have to be the first node in the list. Assume that you are given a linked list that contains N nodes; however, the value of N is unknown.

a. Design an $O(N)$ algorithm to determine if the list contains a cycle. You may use $O(N)$ extra space.

⋆ b. Repeat part (a), but use only $O(1)$ extra space. (*Hint:* Use two iterators that are initially at the start of the list but advance at different speeds.)

3.35 One way to implement a queue is to use a circular linked list. In a **circular linked list**, the last node's `next` pointer points at the first node. Assume the list does not contain a header and that we can maintain, at most, one iterator corresponding to a node in the list. For which of the following representations can all basic queue operations be performed in constant worst-case time? Justify your answers.

a. Maintain an iterator that corresponds to the first item in the list.

b. Maintain an iterator that corresponds to the last item in the list.

3.36 Suppose we have a pointer to a node in a singly linked list that is guaranteed *not to be the last node* in the list. We do not have pointers to any other nodes (except by following links). Describe an $O(1)$ algorithm that logically removes the value stored in such a node from the linked list, maintaining the integrity of the linked list. (*Hint:* Involve the next node.)

3.37 Suppose that a singly linked list is implemented with both a header and a tail node. Describe constant-time algorithms to

a. insert item x before position p (given by an iterator)

b. remove the item stored at position p (given by an iterator)