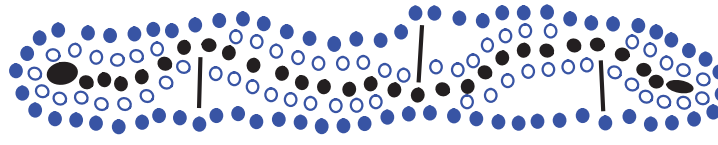
Chapter 6 List and Iterator ADTs



Contents

6.1 Vectors

Suppose we have a collection S of n elements stored in a certain linear order, so that we can refer to the elements in S as first, second, third, and so on. Such a collection is generically referred to as a ***list*** or ***sequence***. We can uniquely refer to each element e in S using an integer in the range $[0, n-1]$ that is equal to the number of elements of S that precede e in S. The ***index*** of an element e in S is the number of elements that are before e in S. Hence, the first element in S has index 0 and the last element has index $n-1$. Also, if an element of S has index i, its previous element (if it exists) has index $i-1$, and its next element (if it exists) has index $i+1$. This concept of index is related to that of the ***rank*** of an element in a list, which is usually defined to be one more than its index; so the first element is at rank 1, the second is at rank 2, and so on.

A sequence that supports access to its elements by their indices is called a ***vector***. Since our index definition is more consistent with the way arrays are indexed in C++ and other common programming languages, we refer to the place where an element is stored in a vector as its "index," rather than its "rank."

This concept of index is a simple yet powerful notion, since it can be used to specify where to insert a new element into a list or where to remove an old element.

6.1.1 The Vector Abstract Data Type

A ***vector***, also called an ***array list***, is an ADT that supports the following fundamental functions (in addition to the standard size() and empty() functions). In all cases, the index parameter i is assumed to be in the range $0 \leq i \leq \text{size}() - 1$.

at(i): Return the element of V with index i; an error condition occurs if i is out of range.

set(i, e): Replace the element at index i with e; an error condition occurs if i is out of range.

insert(i, e): Insert a new element e into V to have index i; an error condition occurs if i is out of range.

erase(i): Remove from V the element at index i; an error condition occurs if i is out of range.

We do ***not*** insist that an array be used to implement a vector, so that the element at index 0 is stored at index 0 in the array, although that is one (very natural) possibility. The index definition offers us a way to refer to the "place" where an element is stored in a sequence without having to worry about the exact implementation of that sequence. The index of an element may change when the sequence is updated, however, as we illustrate in the following example.

Example 6.1: *We show below some operations on an initially empty vector V.*

Operation	*Output*	V
insert$(0,7)$	–	(7)
insert$(0,4)$	–	$(4,7)$
at(1)	7	$(4,7)$
insert$(2,2)$	–	$(4,7,2)$
at(3)	*"error"*	$(4,7,2)$
erase(1)	–	$(4,2)$
insert$(1,5)$	–	$(4,5,2)$
insert$(1,3)$	–	$(4,3,5,2)$
insert$(4,9)$	–	$(4,3,5,2,9)$
at(2)	5	$(4,3,5,2,9)$
set$(3,8)$	–	$(4,3,5,8,9)$

6.1.2 A Simple Array-Based Implementation

An obvious choice for implementing the vector ADT is to use a fixed size array A, where $A[i]$ stores the element at index i. We choose the size N of array A to be sufficiently large, and we maintain the number $n < N$ of elements in the vector in a member variable.

The details of the implementation of the functions of the vector ADT are reasonably simple. To implement the at(i) operation, for example, we just return $A[i]$. The implementations of the functions insert(i,e) and erase(i) are given in Code Fragment 6.1.

Algorithm insert(i,e):
  for $j = n-1, n-2, \ldots, i$ **do**
    $A[j+1] \leftarrow A[j]$ {make room for the new element}
  $A[i] \leftarrow e$
  $n \leftarrow n+1$

Algorithm erase(i):
  for $j = i+1, i+2, \ldots, n-1$ **do**
    $A[j-1] \leftarrow A[j]$ {fill in for the removed element}
  $n \leftarrow n-1$

Code Fragment 6.1: Methods insert(i,e) and erase(i) in the array implementation of the vector ADT. The member variable n stores the number of elements.

An important (and time-consuming) part of this implementation involves the shifting of elements up or down to keep the occupied cells in the array contiguous. These shifting operations are required to maintain our rule of always storing an

element whose list index i at index i in the array A. (See Figure 6.1.)

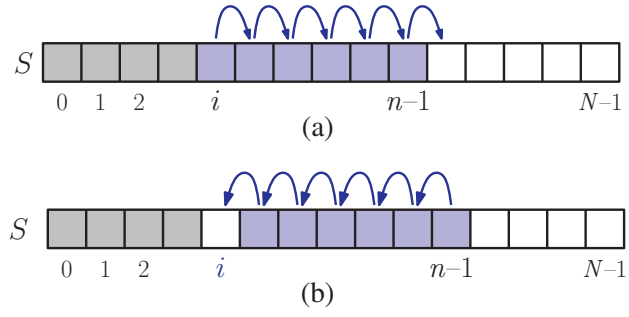


Figure 6.1: Array-based implementation of a vector V that is storing n elements: (a) shifting up for an insertion at index i; (b) shifting down for a removal at index i.

The Performance of a Simple Array-Based Implementation

Table 6.1 shows the worst-case running times of the functions of a vector with n elements realized by means of an array. Methods empty, size, at, and set clearly run in $O(1)$ time, but the insertion and removal functions can take much longer than this. In particular, insert(i,e) runs in time $O(n)$. Indeed, the worst case for this operation occurs when $i = 0$, since all the existing n elements have to be shifted forward. A similar argument applies to function erase(i), which runs in $O(n)$ time, because we have to shift backward $n-1$ elements in the worst case ($i = 0$). In fact, assuming that each possible index is equally likely to be passed as an argument to these operations, their average running time is $O(n)$ because we have to shift $n/2$ elements on average.

Operation	*Time*
size()	$O(1)$
empty()	$O(1)$
at(i)	$O(1)$
set(i,e)	$O(1)$
insert(i,e)	$O(n)$
erase(i)	$O(n)$

Table 6.1: Performance of a vector with n elements realized by an array. The space usage is $O(N)$, where N is the size of the array.

Looking more closely at insert(i,e) and erase(i), we note that they each run in time $O(n-i+1)$, for only those elements at index i and higher have to be shifted

up or down. Thus, inserting or removing an item at the end of a vector, using the functions insert(n,e) and erase$(n-1)$, take $O(1)$ time each respectively. Moreover, this observation has an interesting consequence for the adaptation of the vector ADT to the deque ADT given in Section 5.3.1. If the vector ADT in this case is implemented by means of an array as described above, then functions insertBack and eraseBack of the deque each run in $O(1)$ time. However, functions insertFront and eraseFront of the deque each run in $O(n)$ time.

Actually, with a little effort, we can produce an array-based implementation of the vector ADT that achieves $O(1)$ time for insertions and removals at index 0, as well as insertions and removals at the end of the vector. Achieving this requires that we give up on our rule that an element at index i is stored in the array at index i, however, as we would have to use a circular array approach like the one we used in Section 5.2 to implement a queue. We leave the details of this implementation for an exercise (R-6.17).

6.1.3 An Extendable Array Implementation

A major weakness of the simple array implementation for the vector ADT given in Section 6.1.2 is that it requires advance specification of a fixed capacity, N, for the total number of elements that may be stored in the vector. If the actual number of elements, n, of the vector is much smaller than N, then this implementation will waste space. Worse, if n increases past N, then this implementation will crash. Fortunately, there is a simple way to fix this major drawback.

Let us provide a means to grow the array A that stores the elements of a vector V. Of course, in C++ (and most other programming languages) we cannot actually grow the array A; its capacity is fixed at some number N, as we have already observed. Instead, when an ***overflow*** occurs, that is, when $n = N$ and function insert is called, we perform the following steps:

1. Allocate a new array B of capacity N
2. Copy $A[i]$ to $B[i]$, for $i = 0, \ldots, N-1$
3. Deallocate A and reassign A to point to the new array B

This array replacement strategy is known as an ***extendable array***, for it can be viewed as extending the end of the underlying array to make room for more elements. (See Figure 6.2.) Intuitively, this strategy is much like that of the hermit crab, which moves into a larger shell when it outgrows its previous one.

We give an implementation of the vector ADT using an extendable array in Code Fragment 6.2. To avoid the complexities of templated classes, we have adopted our earlier practice of using a type definition to specify the base element type, which is an **int** in this case. The class is called ArrayVector. We leave the details of producing a fully generic templated class as an exercise (R-6.7).

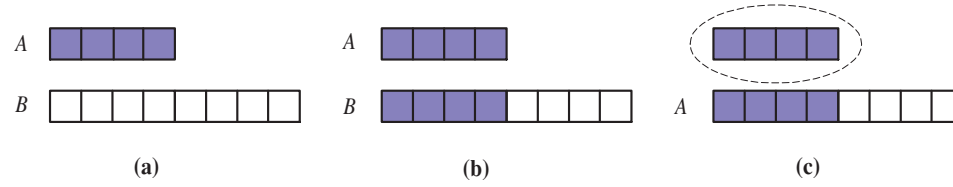


Figure 6.2: The three steps for "growing" an extendable array: (a) create new array B; (b) copy elements from A to B; (c) reassign A to refer to the new array and delete the old array.

Our class definition differs slightly from the operations given in our ADT. For example, we provide two means for accessing individual elements of the vector. The first involves overriding the C++ array index operator ("[]"), and the second is the at function. The two functions behave the same, except that the at function performs a range test before each access. (Note the similarity with the STL vector class given in Section 6.1.4.) If the index i is not in bounds, this function throws an exception. Because both of these access operations return a reference, there is no need to explicitly define a set function. Instead, we can simply use the assignment operator. For example, the ADT function v.set$(i,5)$ could be implemented either as $v[i] = 5$ or, more safely, as v.at$(i) = 5$.

```
typedef int Elem;                          // base element type
class ArrayVector {
public:
  ArrayVector();                           // constructor
  int size() const;                        // number of elements
  bool empty() const;                      // is vector empty?
  Elem& operator[](int i);                 // element at index
  Elem& at(int i) throw(IndexOutOfBounds); // element at index
  void erase(int i);                       // remove element at index
  void insert(int i, const Elem& e);       // insert element at index
  void reserve(int N);                     // reserve at least N spots
  // ... (housekeeping functions omitted)
private:
  int capacity;                            // current array size
  int n;                                   // number of elements in vector
  Elem* A;                                 // array storing the elements
};
```

Code Fragment 6.2: A vector implementation using an extendable array.

The member data for class ArrayVector consists of the array storage A, the current number n of elements in the vector, and the current storage capacity. The class ArrayVector also provides the ADT functions insert and remove. We discuss their implementations below. We have added a new function, called reserve, that

is not part of the ADT. This function allows the user to explicitly request that the array be expanded to a capacity of a size at least n. If the capacity is already larger than this, then the function does nothing.

Even though we have not bothered to show them, the class also provides some of the standard housekeeping functions. These consist of a copy constructor, an assignment operator, and a destructor. Because this class allocates memory, their inclusion is essential for a complete and robust class implementation. We leave them as an exercise (R-6.6). We should also add versions of the indexing operators that return constant references.

In Code Fragment 6.3, we present the class constructor and a number of simple member functions. When the vector is constructed, we do not allocate any storage and simply set A to NULL. Note that the first attempt to add an element results in array storage being allocated.

```
ArrayVector::ArrayVector()                    // constructor
  : capacity(0), n(0), A(NULL) { }

int ArrayVector::size() const                 // number of elements
  { return n; }

bool ArrayVector::empty() const               // is vector empty?
  { return size() == 0; }

Elem& ArrayVector::operator[](int i)          // element at index
  { return A[i]; }
                                              // element at index (safe)
Elem& ArrayVector::at(int i) throw(IndexOutOfBounds) {
  if (i < 0 || i >= n)
    throw IndexOutOfBounds("illegal index in function at()");
  return A[i];
}
```

Code Fragment 6.3: The simple member functions for class ArrayVector.

In Code Fragment 6.4, we present the member function erase. As mentioned above, it removes an element at index i by shifting all subsequent elements from index $i+1$ to the last element of the array down by one position.

```
void ArrayVector::erase(int i) {              // remove element at index
  for (int j = i+1; j < n; j++)               // shift elements down
    A[j - 1] = A[j];
  n--;                                        // one fewer element
}
```

Code Fragment 6.4: The member function remove for class ArrayVector.

Finally, in Code Fragment 6.5, we present the reserve and insert functions. The reserve function first checks whether the capacity already exceeds n, in which case nothing needs to be done. Otherwise, it allocates a new array B of the desired sizes, copies the contents of A to B, deletes A, and makes B the current array. The insert function first checks whether there is sufficient capacity for one more element. If not, it sets the capacity to the maximum of 1 and twice the current capacity. Then starting at the insertion point, it shifts elements up by one position, and stores the new element in the desired position.

```
void ArrayVector::reserve(int N) {              // reserve at least N spots
  if (capacity >= N) return;                    // already big enough
  Elem* B = new Elem[N];                        // allocate bigger array
  for (int j = 0; j < n; j++)                   // copy contents to new array
    B[j] = A[j];
  if (A != NULL) delete [] A;                   // discard old array
  A = B;                                        // make B the new array
  capacity = N;                                 // set new capacity
}
void ArrayVector::insert(int i, const Elem& e) {
  if (n >= capacity)                            // overflow?
    reserve(max(1, 2 * capacity));              // double array size
  for (int j = n - 1; j >= i; j--)              // shift elements up
    A[j+1] = A[j];
  A[i] = e;                                     // put in empty slot
  n++;                                          // one more element
}
```

Code Fragment 6.5: The member functions reserve and insert for class ArrayVector.

In terms of efficiency, this array replacement strategy might, at first, seem rather slow. After all, performing just one array replacement required by an element insertion takes $O(n)$ time, which is not very good. Notice, however, that, after we perform an array replacement, our new array allows us to add n new elements to the vector before the array must be replaced again.

This simple observation allows us to show that the running time of a series of operations performed on an initially empty vector is proportional to the total number of elements added. As a shorthand notation, let us refer to the insertion of an element meant to be the last element in a vector as a "push" operation. Using a design pattern called ***amortization***, we show below that performing a sequence of push operations on a vector implemented with an extendable array is quite efficient.

Proposition 6.2: *Let V be a vector implemented by means of an extendable array A, as described above. The total time to perform a series of n push operations in V, starting from V being empty and A having size $N = 1$, is $O(n)$.*

Justification: To perform this analysis, we view the computer as a coin-operated

appliance, which requires the payment of one ***cyber-dollar*** for a constant amount of computing time. When an operation is executed, we should have enough cyber-dollars available in our current "bank account" to pay for that operation's running time. Thus, the total amount of cyber-dollars spent for any computation is proportional to the total time spent on that computation. The beauty of using this analysis method is that we can overcharge some operations in order to save up cyber-dollars to pay for others.

Let us assume that one cyber-dollar is enough to pay for the execution of each push operation in V, excluding the time spent for growing the array. Also, let us assume that growing the array from size k to size $2k$ requires k cyber-dollars for the time spent copying the elements. We shall charge each push operation three cyber-dollars. Thus, we overcharge each push operation that does not cause an overflow by two cyber-dollars. Think of the two cyber-dollars profited in an insertion that does not grow the array as being "stored" at the element inserted.

An overflow occurs when the vector V has 2^i elements, for some $i \geq 0$, and the size of the array used by V is 2^i. Thus, doubling the size of the array requires 2^i cyber-dollars. Fortunately, these cyber-dollars can be found at the elements stored in cells 2^{i-1} through $2^i - 1$. (See Figure 6.3.) Note that the previous overflow occurred when the number of elements became larger than 2^{i-1} for the first time, and thus the cyber-dollars stored in cells 2^{i-1} through $2^i - 1$ were not previously spent. Therefore, we have a valid amortization scheme in which each operation is charged three cyber-dollars and all the computing time is paid for. That is, we can pay for the execution of n push operations using $3n$ cyber-dollars. ■

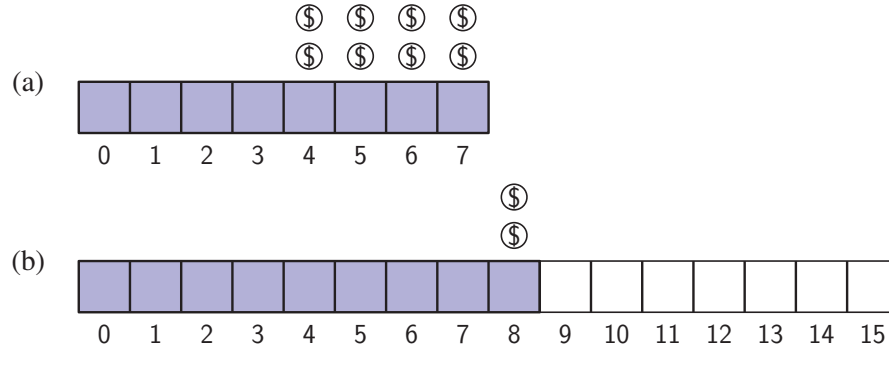


Figure 6.3: A series of push operations on a vector: (a) an 8-cell array is full, with two cyber-dollars "stored" at cells 4 through 7; (b) a push operation causes an overflow and a doubling of capacity. Copying the eight old elements to the new array is paid for by the cyber-dollars already stored in the table; inserting the new element is paid for by one of the cyber-dollars charged to the push operation; and two cyber-dollars profited are stored at cell 8.

6.1.4 STL Vectors

In Section 1.5.5, we introduced the vector class of the C++ Standard Template Library (STL). We mentioned that STL vectors behave much like standard arrays in C++, but they are superior to standard arrays in many respects. In this section we explore this class in greater detail.

The Standard Template Library provides C++ programmers a number of useful built-in classes and algorithms. The classes provided by the STL are organized in various groups. Among the most important of these groups is the set of classes called containers. A ***container*** is a data structure that stores a collection of objects. Many of the data structures that we study later in this book, such as stacks, queues, and lists, are examples of STL containers. The class vector is perhaps the most basic example of an STL container class. We discuss containers further in Section 6.2.1.

The definition of class vector is given in the system include file named "vector." The vector class is part of the std namespace, so it is necessary either to use "std::vector" or to provide an appropriate **using** statement. The vector class is templated with the class of the individual elements. For example, the code fragment below declares a vector containing 100 integers.

```
#include <vector>                    // provides definition of vector
using std::vector;                   // make vector accessible

vector<int> myVector(100);           // a vector with 100 integers
```

We refer to the type of individual elements as the vector's ***base type***. Each element is initialized to the base type's default value, which for integers is zero.

STL vector objects behave in many respects like standard C++ arrays, but they provide many additional features.

- As with arrays, individual elements of a vector object can be indexed using the usual index operator ("[]"). Elements can also be accessed by a member function called at. The advantage of this member function over the index operator is that it performs range checking and generates an error exception if the index is out of bounds.
- Unlike C++ arrays, STL vectors can be dynamically resized, and new elements may be efficiently appended or removed from the end of an array.
- When an STL vector of class objects is destroyed, it automatically invokes the destructor for each of its elements. (With C++ arrays, it is the obligation of the programmer to do this explicitly.)
- STL vectors provide a number of useful functions that operate on entire vectors, not just on individual elements. This includes, for example, the ability to copy all or part of one vector to another, the ability to compare the contents of two arrays, and the ability to insert and erase multiple elements.

Here are the principal member functions of the vector class. Let V be declared to be an STL vector of some base type, and let e denote a single object of this same base type. (For example, V is a vector of integers, and e is an integer.)

vector(n): Construct a vector with space for n elements; if no argument is given, create an empty vector.

size(): Return the number of elements in V.

empty(): Return true if V is empty and false otherwise.

resize(n): Resize V, so that it has space for n elements.

reserve(n): Request that the allocated storage space be large enough to hold n elements.

operator[i]: Return a reference to the ith element of V.

at(i): Same as $V[i]$, but throw an out_of_range exception if i is out of bounds, that is, if $i < 0$ or $i \geq V$.size().

front(): Return a reference to the first element of V.

back(): Return a reference to the last element of V.

push_back(e): Append a copy of the element e to the end of V, thus increasing its size by one.

pop_back(): Remove the last element of V, thus reducing its size by one.

When the base type of an STL vector is class, all copying of elements (for example, in push_back) is performed by invoking the class's copy constructor. Also, when elements are destroyed (for example, by invoking the destroyer or the pop_back member function) the class's destructor is invoked on each deleted element. STL vectors are expandable—when the current array space is exhausted, its storage size is increased.

Although we have not discussed it here, the STL vector also supports functions for inserting elements at arbitrary positions within the vector, and for removing arbitrary elements of the vector. These are discussed in Section 6.1.4.

There are both similarities and differences between our ArrayVector class of Section 6.1.3 and the STL vector class. One difference is that the STL constructor allows for an arbitrary number of initial elements, whereas our ArrayVect constructor always starts with an empty vector. The STL vector functions V.front() and V.back() are equivalent to our functions $V[0]$ and $V[n-1]$, respectively, where n is equal to V.size(). The STL vector functions V.push_back(e) and V.pop_back() are equivalent to our ArrayVect functions V.insert(n,e) and V.remove($n-1$), respectively.

6.2 Lists

Using an index is not the only means of referring to the place where an element appears in a list. If we have a list L implemented with a (singly or doubly) linked list, then it could possibly be more natural and efficient to use a ***node*** instead of an index as a means of identifying where to access and update a list. In this section, define the list ADT, which abstracts the concrete linked list data structure (presented in Sections 3.2 and 3.3) using a related position ADT that abstracts the notion of "place" in a list.

6.2.1 Node-Based Operations and Iterators

Let L be a (singly or doubly) linked list. We would like to define functions for L that take nodes of the list as parameters and provide nodes as return types. Such functions could provide significant speedups over index-based functions, because finding the index of an element in a linked list requires searching through the list incrementally from its beginning or end, counting elements as we go.

For instance, we might want to define a hypothetical function $\mathsf{remove}(v)$ that removes the element of L stored at node v of the list. Using a node as a parameter allows us to remove an element in $O(1)$ time by simply going directly to the place where that node is stored and then "linking out" this node through an update of the *next* and *prev* links of its neighbors. Similarly, in $O(1)$ time, we could insert a new element e into L with an operation such as $\mathsf{insert}(v,e)$, which specifies the node v before which the node of the new element should be inserted. In this case, we simply "link in" the new node.

Defining functions of a list ADT by adding such node-based operations raises the issue of how much information we should be exposing about the implementation of our list. Certainly, it is desirable for us to be able to use either a singly or doubly linked list without revealing this detail to a user. Likewise, we do not wish to allow a user to modify the internal structure of a list without our knowledge. Such modifications would be possible, however, if we provided a pointer to a node in our list in a form that allows the user to access internal data in that node (such as the *next* or *prev* field).

To abstract and unify the different ways of storing elements in the various implementations of a list, we introduce a data type that abstracts the notion of the relative position or place of an element within a list. Such an object might naturally be called a ***position***. Because we want this object not only to access individual elements of a list, but also to move around in order to enumerate all the elements of a list, we adopt the convention used in the C++ Standard Template Library, and call it an ***iterator***.

Containers and Positions

In order to safely expand the set of operations for lists, we abstract a notion of "position" that allows us to enjoy the efficiency of doubly or singly linked list implementations without violating object-oriented design principles. In this framework, we think of a list as an instance of a more general class of objects, called a container. A ***container*** is a data structure that stores any collection of elements. We assume that the elements of a container can be arranged in a linear order. A ***position*** is defined to be an abstract data type that is associated with a particular container and which supports the following function.

element(): Return a reference to the element stored at this position.

C++'s ability to overload operators provides us with an elegant alternative manner in which to express the element operation. In particular, we overload the dereferencing operator ("*"), so that, given a position variable p, the associated element can be accessed by $*p$, rather than p.element(). This can be used both for accessing and modifying the element's value.

A position is always defined in a ***relative*** manner, that is, in terms of its neighbors. Unless it is the first or last of the container, a position q is always "after" some position p and "before" some position r (see Figure 6.4). A position q, which is associated with some element e in a container, does not change, even if the index of e changes in the container, unless we explicitly remove e. If the associated node is removed, we say that q is ***invalidated***. Moreover, the position q does not change even if we replace or swap the element e stored at q with another element.

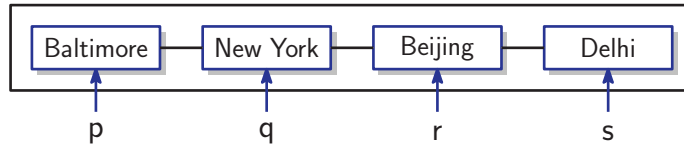


Figure 6.4: A list container. The positions in the current order are p, q, r, and s.

Iterators

Although a position is a useful object, it would be more useful still to be able to navigate through the container, for example, by advancing to the next position in the container. Such an object is called an ***iterator***. An iterator is an extension of a position. It supports the ability to access a node's element, but it also provides the ability to navigate forwards (and possibly backwards) through the container.

There are a number of ways in which to define an ADT for an iterator object. For example, given an iterator object p, we could define an operation p.next(), which returns an iterator that refers to the node just after p in the container. Because of C++'s ability to overload operators, there is a more elegant way to do

this by overloading the increment operator ("++"). In particular, the operation $++p$ advances p to the next position of the container. By repeatedly applying this operation, we can step through all the elements of the container. For some implementations of containers, such as a doubly linked list, navigation may be possible both forwards and backwards. If so, we can also overload the decrement operator ("−−") to move the iterator to the previous position in the container.

In addition to navigating through the container, we need some way of initializing an iterator to the first node of a container and determining whether it has gone beyond the end of the container. To do this, we assume that each container provides two special iterator values, ***begin*** and ***end***. The beginning iterator refers to the first position of the container. We think of the ending iterator as referring to an imaginary position that lies ***just after*** the last node of the container. Given a container object L, the operation L.begin() returns an instance of the beginning iterator for L, and the operation L.end() returns an instance of the ending iterator. (See Figure 6.5.)

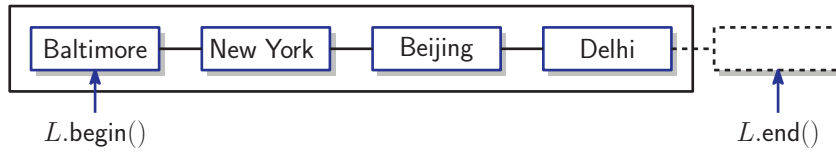


Figure 6.5: The special iterators L.begin() and L.end() for a list L.

In order to enumerate all the elements of a given container L, we define an iterator p whose value is initialized to L.begin(). The associated element is accessed using $*p$. We can enumerate all of the elements of the container by advancing p to the next node using the operation $++p$. We repeat this until p is equal to L.end(), which means that we have fallen off the end of the list.

6.2.2 The List Abstract Data Type

Using the concept of an iterator to encapsulate the idea of "node" in a list, we can define another type of sequence ADT, called simply the ***list*** ADT. In addition to the above functions, we include the generic functions size and empty with the usual meanings. This ADT supports the following functions for a list L and an iterator p for this list.

begin(): Return an iterator referring to the first element of L; same as end() if L is empty.

end(): Return an iterator referring to an imaginary element just after the last element of L.

insertFront(e): Insert a new element e into L as the first element.

insertBack(e): Insert a new element e into L as the last element.

insert(p,e): Insert a new element e into L before position p in L.

eraseFront(): Remove the first element of L.

eraseBack(): Remove the last element of L.

erase(p): Remove from L the element at position p; invalidates p as a position.

The functions insertFront(e) and insertBack(e) are provided as a convenience, since they are equivalent to insert(L.begin(),e) and insert(L.end(),e), respectively. Similarly, eraseFront and eraseBack can be performed by the more general function erase.

An error condition occurs if an invalid position is passed as an argument to one of the list operations. Reasons for a position p to be invalid include:

- p was never initialized or was set to a position in a different list
- p was previously removed from the list
- p results from an illegal operation, such as attempting to perform $++p$, where $p = L$.end(), that is, attempting to access a position beyond the end position

We do not check for these errors in our implementation. Instead, it is the responsibility of the programmer to be sure that only legal positions are used.

Example 6.3: *We show a series of operations for an initially empty list L below. We use variables p and q to denote different positions, and we show the object currently stored at such a position in parentheses in the Output column.*

Operation	***Output***	L
insertFront(8)	–	(8)
$p =$ begin()	p : (8)	(8)
insertBack(5)	–	(8,5)
$q = p;\ ++q$	q : (5)	(8,5)
$p ==$ begin()	*true*	(8,5)
insert(q,3)	–	(8,3,5)
$*q = 7$	–	(8,3,7)
insertFront(9)	–	(9,8,3,7)
eraseBack()	–	(9,8,3)
erase(p)	–	(9,3)
eraseFront()	–	(3)

The list ADT, with its built-in notion of position, is useful in a number of settings. For example, a program that models several people playing a game of cards could model each person's hand as a list. Since most people like to keep cards of the same suit together, inserting and removing cards from a person's hand could

be implemented using the functions of the list ADT, with the positions being determined by a natural ordering of the suits. Likewise, a simple text editor embeds the notion of positional insertion and removal, since such editors typically perform all updates relative to a ***cursor***, which represents the current position in the list of characters of text being edited.

6.2.3 Doubly Linked List Implementation

There are a number of different ways to implement our list ADT in C++. Probably the most natural and efficient way is to use a doubly linked list, similar to the one we introduced in Section 3.3. Recall that our doubly linked list structure is based on two ***sentinel nodes***, called the ***header*** and ***trailer***. These are created when the list is first constructed. The other elements of the list are inserted between these sentinels.

Following our usual practice, in order to keep the code simple, we sacrifice generality by forgoing the use of class templates. Instead, we provide a type definition Elem, which is the base element type of the list. We leave the details of producing a fully generic templated class as an exercise (R-6.11).

Before defining the class, which we call NodeList, we define two important structures. The first represents a node of the list and the other represents an iterator for the list. Both of these objects are defined as nested classes within NodeList. Since users of the class access nodes exclusively through iterators, the node is declared a private member of NodeList, and the iterator is a public member.

The node object is called Node and is presented in Code Fragment 6.6. This is a simple C++ structure, which has only (public) data members, consisting of the node's element, a link to the previous node of the list, and a link to the next node of the list. Since it is declared to be private to NodeList, its members are accessible only within NodeList.

```
struct Node {                          // a node of the list
  Elem elem;                           // element value
  Node* prev;                          // previous in list
  Node* next;                          // next in list
};
```

Code Fragment 6.6: The declaration of a node of a doubly linked list.

Our iterator object is called Iterator. To users of class NodeList, it can be accessed by the qualified type name NodeList::Iterator. Its definition, which is presented in Code Fragment 6.7, is placed in the public part of NodeList. An element associated with an iterator can be accessed by overloading the dereferencing operator ("*"). In order to make it possible to compare iterator objects, we overload the

equality and inequality operators ("==" and "!="). We provide the ability to move forward or backward in the list by providing the increment and decrement operators ("++" and "−−"). We declare NodeList to be a friend, so that it may access the private members of Iterator. The private data member consists of a pointer v to the associated node of the list. We also provide a private constructor, which initializes the node pointer. (The constructor is private so that only NodeList is allowed to create new iterators.)

```
class Iterator {                              // an iterator for the list
public:
  Elem& operator*();                          // reference to the element
  bool operator==(const Iterator& p) const; // compare positions
  bool operator!=(const Iterator& p) const;
  Iterator& operator++();                     // move to next position
  Iterator& operator--();                     // move to previous position
  friend class NodeList;                      // give NodeList access
private:
  Node* v;                                    // pointer to the node
  Iterator(Node* u);                          // create from node
};
```

Code Fragment 6.7: Class Iterator, realizing an iterator for a doubly linked list.

In Code Fragment 6.8 we present the implementations of the member functions for the Iterator class. These all follow directly from the definitions given earlier.

```
NodeList::Iterator::Iterator(Node* u)          // constructor from Node*
  { v = u; }

Elem& NodeList::Iterator::operator*()          // reference to the element
  { return v->elem; }
                                               // compare positions
bool NodeList::Iterator::operator==(const Iterator& p) const
  { return v == p.v; }

bool NodeList::Iterator::operator!=(const Iterator& p) const
  { return v != p.v; }
                                               // move to next position
NodeList::Iterator& NodeList::Iterator::operator++()
  { v = v->next; return *this; }
                                               // move to previous position
NodeList::Iterator& NodeList::Iterator::operator--()
  { v = v->prev; return *this; }
```

Code Fragment 6.8: Implementations of the Iterator member functions.

To keep the code simple, we have not implemented any error checking. We assume that the functions of Code Fragment 6.8 are defined outside the class body. Because of this, when referring to the nested class Iterator, we need to apply the scope resolution operator, as in NodeList::Iterator, so the compiler knows that we are referring to the iterator type associated with NodeList. Observe that the increment and decrement operators not only update the position, but they also return a reference to the updated position. This makes it possible to use the result of the increment operation, as in "$q = ++p$."

Having defined the supporting structures Node and Iterator, let us now present the declaration of class NodeList, which is given in Code Fragment 6.9. The class declaration begins by inserting the Node and Iterator definitions from Code Fragments 6.6 and 6.7. This is followed by the public members, that consist of a simple default constructor and the members of the list ADT. We have omitted the standard housekeeping functions from our class definition. These include the class destructor, a copy constructor, and an assignment operator. The definition of the destructor is important, since this class allocates memory, so it is necessary to delete this memory when an object of this type is destroyed. We leave the implementation of these housekeeping functions as an exercise (R-6.12).

```
typedef int Elem;                                  // list base element type
class NodeList {                                   // node-based list
private:
  // insert Node declaration here...
public:
  // insert Iterator declaration here...
public:
  NodeList();                                      // default constructor
  int size() const;                                // list size
  bool empty() const;                              // is the list empty?
  Iterator begin() const;                          // beginning position
  Iterator end() const;                            // (just beyond) last position
  void insertFront(const Elem& e);                 // insert at front
  void insertBack(const Elem& e);                  // insert at rear
  void insert(const Iterator& p, const Elem& e); // insert e before p
  void eraseFront();                               // remove first
  void eraseBack();                                // remove last
  void erase(const Iterator& p);                   // remove p
  // housekeeping functions omitted...
private:                                           // data members
  int     n;                                       // number of items
  Node*  header;                                   // head-of-list sentinel
  Node*  trailer;                                  // tail-of-list sentinel
};
```

Code Fragment 6.9: Class NodeList realizing the C++-based list ADT.

The private data members include pointers to the header and trailer sentinel nodes. In order to implement the function size efficiently, we also provide a variable n, which stores the number of elements in the list.

Next, let us see how the various functions of class NodeList are implemented. In Code Fragment 6.10, we begin by presenting a number of simple functions, including the constructor, the size and empty functions, and the begin and end functions. The constructor creates an initially empty list by setting n to zero, then allocating the header and trailer nodes and linking them together. The function begin returns the position of the first node of the list, which is the node just following the header sentinel, and the function end returns the position of the trailer. As desired, this is the position following the last element of the list. In both cases, we are invoking the private constructor declared within class Iterator. We are allowed to do so because NodeList is a friend of Iterator.

```
NodeList::NodeList() {                          // constructor
  n = 0;                                        // initially empty
  header = new Node;                            // create sentinels
  trailer = new Node;
  header->next = trailer;                       // have them point to each other
  trailer->prev = header;
}

int NodeList::size() const                      // list size
  { return n; }

bool NodeList::empty() const                    // is the list empty?
  { return (n == 0); }

NodeList::Iterator NodeList::begin() const      // begin position is first item
  { return Iterator(header->next); }

NodeList::Iterator NodeList::end() const        // end position is just beyond last
  { return Iterator(trailer); }
```

Code Fragment 6.10: Implementations of a number of simple member functions of class NodeList.

Let us next see how to implement insertion. There are three different public insertion functions, insert, insertFront, and insertBack, which are shown in Code Fragment 6.11. The function insert(p,e) performs insertion into the doubly linked list using the same approach that we was explained in Section 3.3. In particular, let w be a pointer to p's node, let u be a pointer to p's predecessor. We create a new node v, and link it before w and after u. Finally, we increment n to indicate that there is one additional element in the list.

```
                                                    // insert e before p
void NodeList::insert(const NodeList::Iterator& p, const Elem& e) {
  Node* w = p.v;                                    // p's node
  Node* u = w->prev;                                // p's predecessor
  Node* v = new Node;                               // new node to insert
  v->elem = e;
  v->next = w; w->prev = v;                         // link in v before w
  v->prev = u; u->next = v;                         // link in v after u
  n++;
}

void NodeList::insertFront(const Elem& e)  // insert at front
  { insert(begin(), e); }

void NodeList::insertBack(const Elem& e)   // insert at rear
  { insert(end(), e); }
```

Code Fragment 6.11: Implementations of the insertion functions of class NodeList.

The function insertFront invokes insert on the beginning of the list, and the function insertBack invokes insert on the list's trailer.

Finally, in Code Fragment 6.12 we present the implementation of the erase function, which removes a node from the list. Again, our approach follows directly from the method described in Section 3.3 for removal of a node from a doubly linked list. Let v be a pointer to the node to be deleted, and let w be its successor and u be its predecessor. We unlink v by linking u and w to each other. Once v has been unlinked from the list, we need to return its allocated storage to the system in order to avoid any memory leaks. Finally, we decrement the number of elements in the list.

```
void NodeList::erase(const Iterator& p) {   // remove p
  Node* v = p.v;                            // node to remove
  Node* w = v->next;                        // successor
  Node* u = v->prev;                        // predecessor
  u->next = w; w->prev = u;                 // unlink p
  delete v;                                 // delete this node
  n--;                                      // one fewer element
}

void NodeList::eraseFront()                 // remove first
  { erase(begin()); }

void NodeList::eraseBack()                  // remove last
  { erase(--end()); }
```

Code Fragment 6.12: Implementations of the function erase of class NodeList.

There are number of other enhancements that could be added to our implementation of NodeList, such as better error checking, a more sophisticated suite of constructors, and a post-increment operator for the iterator class.

You might wonder why we chose to define the iterator function end to return an imaginary position that lies just beyond the end of the list, rather than the last node of the list. This choice offers a number of advantages. First, it is well defined, even if the list is empty. Second, the function insert(p,e) can be used to insert a new element at any position of the list. In particular, it is possible to insert an element at the end of the list by invoking insert(end(),e). If we had instead defined end to return the last position of the list, this would not be possible, and the only way to insert an element at the end of the list would be through the insertBack function.

Observe that our implementation is quite efficient with respect to both time and space. All of the operations of the list ADT run in time $O(1)$. (The only exceptions to this are the omitted housekeeping functions, the destructor, copy constructor, and assignment operator. They require $O(n)$ time, where n is the number of elements in the list.) The space used by the data structure is proportional to the number of elements in the list.

6.2.4 STL Lists

The C++ Standard Template Library provides an implementation of a list, which is called list. Like the STL vector, the STL list is an example of an STL container. As in our implementation of class NodeList, the STL list is implemented as a doubly linked list.

In order to define an object to be of type list, it is necessary to first include the appropriate system definition file, which is simply called "list." As with the STL vector, the list class is a member of the std namespace, it is necessary either to preface references to it with the namespace resolution operator, as in "std::list", or to provide an appropriate **using** statement. The list class is templated with the ***base type*** of the individual elements. For example, the code fragment below declares a list of floats. By default, the initial list is empty.

```
#include <list>
using std::list;                    // make list accessible
list<float> myList;                 // an empty list of floats
```

Below is a list of the principal member functions of the list class. Let L be declared to be an STL list of some base type, and let x denote a single object of this same base type. (For example, L is a list of integers, and e is an integer.)

list(n): Construct a list with n elements; if no argument list is given, an empty list is created.

size(): Return the number of elements in L.

empty(): Return true if L is empty and false otherwise.
front(): Return a reference to the first element of L.
back(): Return a reference to the last element of L.
push_front(e): Insert a copy of e at the beginning of L.
push_back(e): Insert a copy of e at the end of L.
pop_front(): Remove the fist element of L.
pop_back(): Remove the last element of L.

The functions push_front and push_back are the STL equivalents of our functions insertFront and insertBack, respectively, of our list ADT. Similarly, the functions pop_front and pop_back are equivalent to the respective functions eraseFront and eraseBack.

Note that, when the base type of an STL vector is class object, all copying of elements (for example, in push_back) is performed by invoking the base class's copy constructor. Whenever elements are destroyed (for example, by invoking the destroyer or the pop_back member function) the class's destructor is invoked on each deleted element.

6.2.5 STL Containers and Iterators

In order to develop a fuller understanding of STL vectors and lists, it is necessary to understand the concepts of STL ***containers*** and ***iterators***. Recall that a container is a data structure that stores a collection of elements. The STL provides a variety of different container classes, many of which are discussed later.

STL Container	*Description*
vector	Vector
deque	Double ended queue
list	List
stack	Last-in, first-out stack
queue	First-in, first-out queue
priority_queue	Priority queue
set (and multiset)	Set (and multiset)
map (and multimap)	Map (and multi-key map)

Different containers organize their elements in different ways, and hence support different methods for accessing individual elements. STL iterators provide a relatively uniform method for accessing and enumerating the elements stored in containers.

Before introducing how iterators work for STL containers, let us begin with a simple function that sums the elements of an STL vector, denoted by V, shown in

Code Fragment 6.13. The elements are accessed in the standard manner through the indexing operator.

```
int vectorSum1(const vector<int>& V) {
  int sum = 0;
  for (int i = 0; i < V.size(); i++)
    sum += V[i];
  return sum;
}
```

Code Fragment 6.13: A simple C++ function that sums the entries of an STL vector.

This particular method of iterating through the elements of a vector should be quite familiar by now. Unfortunately, this method would not be applicable to other types of containers, because it relies on the fact that the elements of a vector can be accessed efficiently through indexing. This is not true for all containers, such as lists. What we desire is a uniform mechanism for accessing elements.

STL Iterators

Every STL container class defines a special associated class called an ***iterator***. As mentioned earlier, an iterator is an object that specifies a position within a container and which is endowed with the ability to navigate to other positions. If p is an iterator that refers to some position within a container, then $*p$ yields a reference to the associated element.

Advancing to the next element of the container is done by incrementing the iterator. For example, either $++p$ or $p++$ advances p to point to the next element of the container. The former returns the updated value of the iterator, and the latter returns its original value. (In our implementation of an iterator for class NodeList in Code Fragment 6.7, we defined only the preincrement operator, but the postincrement operator would be an easy extension. See Exercise R-6.13.)

Each STL container class provides two member functions, begin and end, each of which returns an iterator for this container. The first returns an iterator that points to the first element of the container, and the second returns an iterator that can be thought of as pointing to an imaginary element ***just beyond*** the last element of the container. An example for the case of lists was shown in Figure 6.5, and an example of how this works for the STL vector is shown in Figure 6.6.

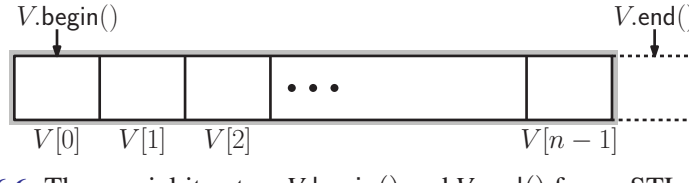


Figure 6.6: The special iterators V.begin() and V.end() for an STL vector V.

Using Iterators

Let us see how we can use iterators to enumerate the elements of an STL container C. Suppose, for example, that C is of type vector<int>, that is, it is an STL list of integers. The associated iterator type is denoted "vector<int>::iterator." In general, if C is an STL container of some type cont and the base type is of type base, then the iterator type would be denoted "cont<base>::iterator."

For example, Code Fragment 6.14 demonstrates how to sum the elements of an STL vector V using an iterator. We begin by providing a type definition to the iterator type, called Iterator. We then create a loop, which is controlled by an iterator p. We start with V.begin(), and we terminate when p reaches V.end(). Although this approach is less direct than the approach based on indexing individual elements, it has the advantage that it can be applied to ***any*** STL container class, not just vectors.

```
int vectorSum2(vector<int> V) {
  typedef vector<int>::iterator Iterator;          // iterator type
  int sum = 0;
  for (Iterator p = V.begin(); p != V.end(); ++p)
    sum += *p;
  return sum;
}
```

Code Fragment 6.14: Using an iterator to sum the elements of an STL vector.

Different containers provide iterators with different capabilities. Most STL containers (including lists, sets, and maps) provide the ability to move not only forwards, but backwards as well. For such containers the decrement operators $--p$ and $p--$ are also defined for their iterators. This is called a ***bidirectional iterator***.

A few STL containers (including vectors and deques) support the additional feature of allowing the addition and subtraction of an integer. For example, for such an iterator, p, the value $p+3$ references the element three positions after p in the container. This is called a ***random-access iterator***.

As with pointers, care is needed in the use of iterators. For example, it is up to the programmer to be sure that an iterator points to a valid element of the container before attempting to dereference it. Attempting to dereference an invalid iterator can result in your program aborting. As mentioned earlier, iterators can be ***invalid*** for various reasons. For example, an iterator becomes invalid if the position that it refers to is deleted.

Const Iterators

Observe that in Code Fragment 6.14, we passed the vector V into the function ***by value*** (recall Section 1.4). This can be quite inefficient, because the system constructs a complete copy of the actual argument. Since our function does not

modify V, the best approach would be to declare the argument to be a constant reference instead, that is, "**const** vector<**int**>&."

A problem arises, however, if we declare an iterator for such a vector. Many STL implementations generate an error message if we attempt to use a regular iterator with a constant vector reference, since such an iterator may lead to an attempt to modify the vector's contents.

The solution is a special read-only iterator, called a ***const iterator***. When using a const iterator, it is possible to read the values of the container by dereferencing the iterator, but it is not possible to modify the container's values. For example, if p is a const iterator, it is possible to read the value of $*p$, but you cannot assign it a value. The const iterator type for our vector type is denoted "vector<**int**>::const_iterator." We make use of the **typedef** command to rename this lengthy definition to the more concise ConstIterator. The final code fragment is presented in Code Fragment 6.15.

```
int vectorSum3(const vector<int>& V) {
  typedef vector<int>::const_iterator ConstIterator;  // iterator type
  int sum = 0;
  for (ConstIterator p = V.begin(); p != V.end(); ++p)
    sum += *p;
  return sum;
}
```

Code Fragment 6.15: Using a constant iterator to sum the elements of a vector.

STL Iterator-Based Container Functions

STL iterators offer a flexible and uniform way to access the elements of STL containers. Many of the member functions and predefined algorithms that work with STL containers use iterators as their arguments. Here are a few of the member functions of the STL vector class that use iterators as arguments. Let V be an STL vector of some given base type, and let e be an object of this base type. Let p and q be iterators over this base type, both drawn from the same container.

vector(p,q): Construct a vector by iterating between p and q, copying each of these elements into the new vector.

assign(p,q): Delete the contents of V, and assigns its new contents by iterating between p and q and copying each of these elements into V.

insert(p,e): Insert a copy of e just prior to the position given by iterator p and shifts the subsequent elements one position to the right.

erase(p): Remove and destroy the element of V at the position given by p and shifts the subsequent elements one po-

sition to the left.

erase(p,q): Iterate between p and q, removing and destroying all these elements and shifting subsequent elements to the left to fill the gap.

clear(): Delete all these elements of V.

When presenting a range of iterator values (as we have done above with the constructor $V(p,q)$, assign(p,q), and erase(p,q)), the iterator range is understood to start with p and end ***just prior*** to q. Borrowing from interval notation in mathematics, this iterator range is often expressed as $[p,q)$, implying that p is included in the range, but q is not. This convention holds whenever dealing with iterator ranges.

Note that the vector member functions insert and erase move elements around in the vector. They should be used with care, because they can be quite slow. For example, inserting or erasing an element at the beginning of a vector causes all the later elements of the vector to be shifted one position.

The above functions are also defined for the STL list and the STL deque (but, of course, the constructors are named differently). Since the list is defined as a doubly linked list, there is no need to shift elements when performing insert or erase. These three STL containers (vector, list, and deque) are called ***sequence containers***, because they explicitly store elements in sequential order. The STL containers set, multiset, map, and multimap support all of the above functions except assign. They are called ***associated containers***, because elements are typically accessed by providing an associated key value. We discuss them in Chapter 9.

It is worthwhile noting that, in the constructor and assignment functions, the iterators p and q do not need to be drawn from the same type of container as V, as long as the container they are drawn from has the same base type. For example, suppose that L is an STL list container of integers. We can create a copy of L in the form of an STL vector V as follows:

```
list<int> L;                          // an STL list of integers
// ...
vector<int> V(L.begin(), L.end());    // initialize V to be a copy of L
```

The iterator-based form of the constructor is quite handy, since it provides an easy way to initialize the contents of an STL container from a standard C++ array. Here, we make use of a low-level feature of C++, which is inherited from its predecessor, the C programming language. Recall from Section 1.1.3 that a C++ array A is represented as a pointer to its first element $A[0]$. In addition, $A+1$ points to $A[1]$, $A+2$ points to $A[2]$, and generally $A+i$ points to $A[i]$.

Addressing the elements of an array in this manner is called ***pointer arithmetic***. It is generally frowned upon as poor programming practice, but in this instance it

provides an elegant way to initialize a vector from an C++ array, as follows.

```
int A[] = {2, 5, -3, 8, 6};          // a C++ array of 5 integers
vector<int> V(A, A+5);               // V = (2, 5, -3, 8, 6)
```

Even though the pointers A and $A+5$ are not STL iterators, through the magic of pointer arithmetic, they essentially behave as though they were. This same trick can be used to initialize any of the STL sequence containers.

STL Vectors and Algorithms

In addition to the above member functions for STL vectors, the STL also provides a number of algorithms that operate on containers in general, and vectors in particular. To access these functions, use the include statement "#include <algorithm>." Let p and q be iterators over some base type, and let e denote an object of this base type. As above, these operations apply to the iterator range $[p,q)$, which starts at p and ends just prior to q.

- sort(p,q): Sort the elements in the range from p to q in ascending order. It is assumed that less-than operator ("<") is defined for the base type.
- random_shuffle(p,q): Rearrange the elements in the range from p to q in random order.
- reverse(p,q): Reverse the elements in the range from p to q.
- find(p,q,e): Return an iterator to the first element in the range from p to q that is equal to e; if e is not found, q is returned.
- min_element(p,q): Return an iterator to the minimum element in the range from p to q.
- max_element(p,q): Return an iterator to the maximum element in the range from p to q.
- for_each(p,q,f): Apply the function f the elements in the range from p to q.

For example, to sort an entire vector V, we would use sort$(V$.begin$(),V$.end$())$. To sort just the first 10 elements, we could use sort$(V$.begin$(),V$.begin$()+10)$.

All of the above functions are supported for the STL deque. All of the above functions, except sort and random_shuffle are supported for the STL list.

An Illustrative Example

In Code Fragment 6.16, we provide a short example program of the functionality of the STL vector class. The program begins with the necessary "#include"

statements. We include <vector>, for the definitions of vectors and iterators, and <algorithm>, for the definitions of sort and random_shuffle.

We first initialize a standard C++ array containing six integers, and we then use the iterator-based constructor to create a six-element vector containing these values. We show how to use the member functions size, pop_back, push_back, front, and back. Observe that popping decreases the array size and pushing increases the array size. We then show how to use iterator arithmetic to sort a portion of the vector, in this case, the first four elements. The call to the erase member function removes two of the last four elements of the vector. After the removal, the remaining two elements at the end have been shifted forward to fill the empty positions.

Next, we demonstrate how to generate a vector of characters. We apply the function random_shuffle to permute the elements of the vector randomly. Finally, we show how to use the member function insert, to insert a character at the beginning of the vector. Observe how the other elements are shifted to the right.

```
#include <cstdlib>                                  // provides EXIT_SUCCESS
#include <iostream>                                 // I/O definitions
#include <vector>                                   // provides vector
#include <algorithm>                                // for sort, random_shuffle

using namespace std;                                // make std:: accessible

int main () {
  int a[] = {17, 12, 33, 15, 62, 45};
  vector<int> v(a, a + 6);                          // v: 17 12 33 15 62 45
  cout << v.size() << endl;                         // outputs: 6
  v.pop_back();                                     // v: 17 12 33 15 62
  cout << v.size() << endl;                         // outputs: 5
  v.push_back(19);                                  // v: 17 12 33 15 62 19
  cout << v.front() << " " << v.back() << endl; // outputs: 17 19
  sort(v.begin(), v.begin() + 4);                   // v: (12 15 17 33) 62 19
  v.erase(v.end() - 4, v.end() - 2);                // v: 12 15 62 19
  cout << v.size() << endl;                         // outputs: 4

  char b[] = {'b', 'r', 'a', 'v', 'o'};
  vector<char> w(b, b + 5);                         // w: b r a v o
  random_shuffle(w.begin(), w.end());               // w: o v r a b
  w.insert(w.begin(), 's');                         // w: s o v r a b

  for (vector<char>::iterator p = w.begin(); p != w.end(); ++p)
    cout << *p << " ";                              // outputs: s o v r a b
  cout << endl;
  return EXIT_SUCCESS;
}
```

Code Fragment 6.16: An example of the use of the STL vector and iterators.

6.3 Sequences

In this section, we define an abstract data type that generalizes the vector and list ADTs. This ADT therefore provides access to its elements using both indices and positions, and is a versatile data structure for a wide variety of applications.

6.3.1 The Sequence Abstract Data Type

A ***sequence*** is an ADT that supports all the functions of the list ADT (discussed in Section 6.2), but it also provides functions for accessing elements by their index, as we did in the vector ADT (discussed in Section 6.1). The interface consists of the operations of the list ADT, plus the following two "bridging" functions, which provide connections between indices and positions.

atIndex(i): Return the position of the element at index i.

indexOf(p): Return the index of the element at position p.

6.3.2 Implementing a Sequence with a Doubly Linked List

One possible implementation of a sequence, of course, is with a doubly linked list. By doing so, all of the functions of the list ADT can be easily implemented to run in $O(1)$ time each. The functions atIndex and indexOf from the vector ADT can also be implemented with a doubly linked list, though in a less efficient manner.

In particular, if we want the functions from the list ADT to run efficiently (using position objects to indicate where accesses and updates should occur), then we can no longer explicitly store the indices of elements in the sequence. Hence, to perform the operation atIndex(i), we must perform link "hopping" from one of the ends of the list until we locate the node storing the element at index i. As a slight optimization, we could start hopping from the closest end of the sequence, which would achieve a running time of

$$O(\min(i+1, n-i)).$$

This is still $O(n)$ in the worst case (when i is near the middle index), but it would be more efficient in applications where many calls to atIndex(i) are expected to involve index values that are significantly closer to either end of the list. In our implementation, we just use the simple approach of walking from the front of the list, and we leave the two-sided solution as an exercise (R-6.14).

In Code Fragment 6.17, we present a definition of a class NodeSequence, which implements the sequence ADT. Observe that, because a sequence extends the definition of a list, we have inherited our class by extending the NodeList class

that was introduced in Section 6.2. We simply add definitions of our bridging functions. Through the magic of inheritance, users of our class NodeSequence have access to all the members of the NodeList class, including its nested class, NodeList::Iterator.

```
class NodeSequence : public NodeList {
public:
  Iterator atIndex(int i) const;          // get position from index
  int indexOf(const Iterator& p) const;   // get index from position
};
```

Code Fragment 6.17: The definition of class NodeSequence, which implements the sequence ADT using a doubly linked list.

Next, in Code Fragment 6.18, we show the implementations of the atIndex and indexOf member functions. The function atIndex(i) hops i positions to the right, starting at the beginning, and returns the resulting position. The function indexOf hops through the list until finding the position that matches the given position p. Observe that the conditional "q != p" uses the overloaded comparison operator for positions defined in Code Fragment 6.8.

```
                                          // get position from index
NodeSequence::Iterator NodeSequence::atIndex(int i) const {
  Iterator p = begin();
  for (int j = 0; j < i; j++) ++p;
  return p;
}
                                          // get index from position
int NodeSequence::indexOf(const Iterator& p) const {
  Iterator q = begin();
  int j = 0;
  while (q != p) {                        // until finding p
    ++q; ++j;                             // advance and count hops
  }
  return j;
}
```

Code Fragment 6.18: Definition of the functions atIndex and indexOf of class NodeSequence.

Both of these functions are quite fragile, and are likely to abort if their arguments are not in bounds. A more careful implementation of atIndex would first check that the argument i lies in the range from 0 to $n-1$, where n is the size of the sequence. The function indexOf should check that it does not run past the end of the sequence. In either case, an appropriate exception should be thrown.

The worst-case running times of both of the functions atIndex and indexOf are $O(n)$, where n is the size of the list. Although this is not very efficient, we may take consolation in the fact that all the other operations of the list ADT run in time $O(1)$. A natural alternative approach would be to implement the sequence ADT using an array. Although we could now provide very efficient implementations of atIndex and indexOf, the insertion and removal operations of the list ADT would now require $O(n)$ time. Thus, neither solution is perfect under all circumstances. We explore this alternative in the next section.

6.3.3 Implementing a Sequence with an Array

Suppose we want to implement a sequence S by storing each element e of S in a cell $A[i]$ of an array A. We can define a position object p to hold an index i and a reference to array A, as member variables. We can then implement function element(p) by simply returning a reference to $A[i]$. A major drawback with this approach, however, is that the cells in A have no way to reference their corresponding positions. For example, after performing an insertFront operation, the elements have been shifted to new positions, but we have no way of informing the existing positions for S that the associated positions of their elements have changed. (Remember that positions in a sequence are defined relative to their neighboring positions, not their ranks.) Hence, if we are going to implement a general sequence with an array, we need to find a different approach.

Consider an alternate solution in which, instead of storing the elements of S in array A, we store a pointer to a new kind of position object in each cell of A. Each new position object p stores a pair consisting of the index i and the element e associated with p. We can easily scan through the array to update the i value associated with each position whose rank changes as the result of an insertion or deletion. An example is shown in Figure 6.7, where a new element containing BWI is inserted at index 1 of an existing sequence. After the insertion, the elements PVD and SFO are shifted to the right, so we increment the index value associated with their index-element pairs.

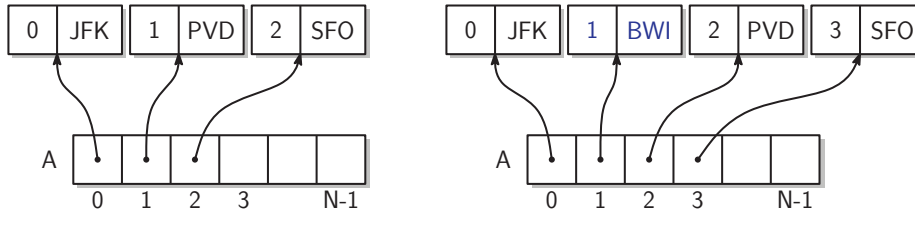


Figure 6.7: An array-based implementation of the sequence ADT.

In this array-based implementation of a sequence, the functions insertFront, insert, and erase take $O(n)$ time because we have to shift position objects to make room for the new position or to fill in the hole created by the removal of the old position (just as in the insert and remove functions based on rank). All the other position-based functions take $O(1)$ time.

Note that we can use an array in a circular fashion, as we did for implementing a queue (see Section 5.2.4). With a little work, we can then perform functions insertFront in $O(1)$ time. Note that functions insert and erase still take $O(n)$ time. Now, the worst case occurs when the element to be inserted or removed has rank $\lfloor n/2 \rfloor$.

Table 6.2 compares the running times of the implementations of the general sequence ADT, by means of a circular array and a doubly linked list.

Operations	*Circular Array*	*List*
size, empty	$O(1)$	$O(1)$
atIndex, indexOf	$O(1)$	$O(n)$
begin, end	$O(1)$	$O(1)$
$*p$, $++p$, $--p$	$O(1)$	$O(1)$
insertFront, insertBack	$O(1)$	$O(1)$
insert, erase	$O(n)$	$O(1)$

Table 6.2: Comparison of the running times of the functions of a sequence implemented with either an array (used in a circular fashion) or a doubly linked list. We denote with n the number of elements in the sequence at the time the operation is performed. The space usage is $O(n)$ for the doubly linked list implementation, and $O(N)$ for the array implementation, where N is the size of the array.

Summarizing this table, we see that the array-based implementation is superior to the linked-list implementation on the rank-based access operations, atIndex and indexOf. It is equal in performance to the linked-list implementation on all the other access operations. Regarding update operations, the linked-list implementation beats the array-based implementation in the position-based update operations, insert and erase. For update operations insertFront and insertBack, the two implementations have comparable performance.

Considering space usage, note that an array requires $O(N)$ space, where N is the size of the array (unless we utilize an extendable array), while a doubly linked list uses $O(n)$ space, where n is the number of elements in the sequence. Since n is less than or equal to N, this implies that the asymptotic space usage of a linked-list implementation is superior to that of a fixed-size array, although there is a small constant factor overhead that is larger for linked lists, since arrays do not need links to maintain the ordering of their cells.

6.4 Case Study: Bubble-Sort on a Sequence

In this section, we illustrate the use of the sequence ADT and its implementation trade-offs with sample C++ functions using the well-known ***bubble-sort*** algorithm.

6.4.1 The Bubble-Sort Algorithm

Consider a sequence of n elements such that any two elements in the sequence can be compared according to an order relation (for example, companies compared by revenue, states compared by population, or words compared lexicographically). The ***sorting*** problem is to reorder the sequence so that the elements are in non-decreasing order. The ***bubble-sort*** algorithm (see Figure 6.8) solves this problem by performing a series of ***passes*** over the sequence. In each pass, the elements are scanned by increasing rank, from rank 0 to the end of the sequence. At each position in a pass, an element is compared with its neighbor, and if these two consecutive elements are found to be in the wrong relative order (that is, the preceding element is larger than the succeeding one), then the two elements are swapped. The sequence is sorted by completing n such passes.

pass	swaps	sequence
		$(5,7,2,6,9,3)$
1st	$7 \leftrightarrow 2$ $7 \leftrightarrow 6$ $9 \leftrightarrow 3$	$(5,2,6,7,3,9)$
2nd	$5 \leftrightarrow 2$ $7 \leftrightarrow 3$	$(2,5,6,3,7,9)$
3rd	$6 \leftrightarrow 3$	$(2,5,3,6,7,9)$
4th	$5 \leftrightarrow 3$	$(2,3,5,6,7,9)$

Figure 6.8: The bubble-sort algorithm on a sequence of integers. For each pass, the swaps performed and the sequence after the pass are shown.

The bubble-sort algorithm has the following properties:

- In the first pass, once the largest element is reached, it keeps on being swapped until it gets to the last position of the sequence.
- In the second pass, once the second largest element is reached, it keeps on being swapped until it gets to the second-to-last position of the sequence.
- In general, at the end of the ith pass, the right-most i elements of the sequence (that is, those at indices from $n-1$ down to $n-i$) are in final position.

The last property implies that it is correct to limit the number of passes made by a bubble-sort on an n-element sequence to n. Moreover, it allows the ith pass to be limited to the first $n-i+1$ elements of the sequence.

6.4.2 A Sequence-Based Analysis of Bubble-Sort

Assume that the implementation of the sequence is such that the accesses to elements and the swaps of elements performed by bubble-sort take $O(1)$ time each. That is, the running time of the ith pass is $O(n-i+1)$. We have that the overall running time of bubble-sort is

$$O\left(\sum_{i=1}^{n}(n-i+1)\right).$$

We can rewrite the sum inside the big-Oh notation as

$$n+(n-1)+\cdots+2+1=\sum_{i=1}^{n} i.$$

By Proposition 4.3, we have

$$\sum_{i=1}^{n} i=\frac{n(n+1)}{2}.$$

Thus, bubble-sort runs in $O(n^2)$ time, provided that accesses and swaps can each be implemented in $O(1)$ time. As we see in future chapters, this performance for sorting is quite inefficient. We discuss the bubble-sort algorithm here. Our aim is to demonstrate, not as an example of a good sorting algorithm.

Code Fragments 6.19 and 6.20 present two implementations of bubble-sort on a sequence of integers. The parameter S is of type Sequence, but we do not specify whether it is a node-based or array-based implementation. The two bubble-sort implementations differ in the preferred choice of functions to access and modify the sequence. The first is based on accessing elements by their index. We use the function atIndex to access the two elements of interest.

Since function bubbleSort1 accesses elements only through the index-based interface functions atIndex, this implementation is suitable only for the array-based implementation of the sequence, for which atIndex takes $O(1)$ time. Given such an array-based sequence, this bubble-sort implementation runs in $O(n^2)$ time.

On the other hand, if we had used our node-based implementation of the sequence, each atIndex call would take $O(n)$ time in the worst case. Since this function is called with each iteration of the inner loop, the entire function would run in $O(n^3)$ worst-case time, which is quite slow if n is large.

In contrast to bubbleSort1, our second function, bubbleSort2, accesses the elements entirely through the use of iterators. The iterators *prec* and *succ* play the roles that indices $j-1$ and j play, respectively, in bubbleSort1. Observe that when we first enter the inner loop of bubbleSort1, the value of $j-1$ is 0, that is, it refers to the first element of the sequence. This is why we initialize *prec* to the beginning

```
void bubbleSort1(Sequence& S) {                    // bubble-sort by indices
  int n = S.size();
  for (int i = 0; i < n; i++) {                    // i-th pass
    for (int j = 1; j < n−i; j++) {
      Sequence::Iterator prec = S.atIndex(j−1);    // predecessor
      Sequence::Iterator succ = S.atIndex(j);      // successor
      if (*prec > *succ) {                         // swap if out of order
        int tmp = *prec; *prec = *succ; *succ = tmp;
      }
    }
  }
}
```

Code Fragment 6.19: A C++ implementations of bubble-sort based on indices.

of the sequence before entering the inner loop. Whenever we reenter the inner loop, we initialize *succ* to *prec* and then immediately increment it. Thus, *succ* refers to the position immediately after *prec*. Before resuming the loop, we increment *prec*.

```
void bubbleSort2(Sequence& S) {                    // bubble-sort by positions
  int n = S.size();
  for (int i = 0; i < n; i++) {                    // i-th pass
    Sequence::Iterator prec = S.begin();           // predecessor
    for (int j = 1; j < n−i; j++) {
      Sequence::Iterator succ = prec;
      ++succ;                                      // successor
      if (*prec > *succ) {                         // swap if out of order
        int tmp = *prec; *prec = *succ; *succ = tmp;
      }
      ++prec;                                      // advance predecessor
    }
  }
}
```

Code Fragment 6.20: Two C++ implementations of bubble-sort.

Since the iterator increment operator takes $O(1)$ time in either the array-based or node-based implementation of a sequence, this second implementation of bubble-sort would run in $O(n^2)$ worst-case time, irrespective of the manner in which the sequence was implemented.

The two bubble-sort implementations given above show the importance of providing efficient implementations of ADTs. Nevertheless, in spite of its implementation simplicity, computing researchers generally feel that the bubble-sort algorithm is not a good sorting method, because, even if implemented in the best possible way, it still takes quadratic time. Indeed, there are much more efficient sorting algorithms that run in $O(n \log n)$ time. We explore these in Chapters 8 and 11.

6.5 Exercises

For help with exercises, please visit the web site, www.wiley.com/college/goodrich.

Reinforcement

R-6.1 Give a C++ code fragment for reversing an array.

R-6.2 Give a C++ code fragment for randomly permuting an array.

R-6.3 Give a C++ code fragment for circularly rotating an array by distance d.

R-6.4 Draw a representation of an initially empty vector A after performing the following sequence of operations: insert$(0,4)$, insert$(0,3)$, insert$(0,2)$, insert$(2,1)$, insert$(1,5)$, insert$(1,6)$, insert$(3,7)$, insert$(0,8)$.

R-6.5 Give an adapter class to support the Stack interface using the functions of the vector ADT.

R-6.6 Provide the missing housekeeping functions (copy constructor, assignment operator, and destructor) for the class ArrayVector of Code Fragment 6.2.

R-6.7 Provide a fully generic version of the class ArrayVector of Code Fragment 6.2 using a templated class.

R-6.8 Give a templated C++ function sum(v) that returns the sum of elements in an STL vector v. Use an STL iterator to enumerate the elements of v. Assume that the element type of v is any numeric type that supports the $+$ operator.

R-6.9 Rewrite the justification of Proposition 6.2 under the assumption that the cost of growing the array from size k to size $2k$ is $3k$ cyber-dollars. How much should each push operation be charged to make the amortization work?

R-6.10 Draw pictures illustrating each of the major steps in the algorithms for functions insert(p,e), insertFront(e), and insertBack(e) of Code Fragment 6.5.

R-6.11 Provide a fully generic version of the class NodeList of Code Fragment 6.9 using a templated class.

R-6.12 Provide the missing housekeeping functions (copy constructor, assignment operator, and destructor) for the class NodeList, which was presented in Code Fragment 6.9.

R-6.13 In our implementation of an iterator for class NodeList in Code Fragment 6.7, we defined only the preincrement operator. Provide a definition for a postincrement operator.

R-6.14 In our implementation of the atRank(i) function in Code Fragment 6.18 for class NodeSequence, we walked from the front of the list. Present a more efficient implementation, which walks from whichever end of the list is closer to index i.

R-6.15 Provide the details of an array implementation of the list ADT.

R-6.16 Suppose that we have made kn total accesses to the elements in a list L of n elements, for some integer $k \geq 1$. What are the minimum and maximum number of elements that have been accessed fewer than k times?

R-6.17 Give pseudo-code describing how to implement all the operations in the sequence ADT using an array used in a circular fashion. What is the running time for each of these functions?

R-6.18 Using the Sequence interface functions, describe a recursive function for determining if a sequence S of n integer objects contains a given integer k. Your function should not contain any loops. How much space does your function use in addition to the space used for S?

R-6.19 Briefly describe how to perform a new sequence function makeFirst(p) that moves an element of a sequence S at position p to be the first element in S while keeping the relative ordering of the remaining elements in S unchanged. Your function should run in $O(1)$ time if S is implemented with a doubly linked list.

R-6.20 Describe how to implement an iterator for the class ArrayVector of Code Fragment 6.2, based on an integer index. Include pseudo-code fragments describing the dereferencing operator ("$*$"), equality test ("$==$"), and increment and decrement ("$++$" and "$--$").

Creativity

C-6.1 Describe what changes need to be made to the extendable array implementation given in Code Fragment 6.2 in order to avoid unexpected termination due to an error. Specify the new types of exceptions you would add, and when and where they should be thrown.

C-6.2 Give complete C++ code for a new class, ShrinkingVector, that extends the ArrayVector class shown in Code Fragment 6.2 and adds a function, shrinkToFit, which replaces the underlying array with an array whose capacity is exactly equal to the number of elements currently in the vector.

C-6.3 Describe what changes need to be made to the extendable array implementation given in Code Fragment 6.2 in order to shrink the size N of the array by half any time the number of elements in the vector goes below $N/4$.

C-6.4 Show that, using an extendable array that grows and shrinks as in the previous exercise, the following series of $2n$ operations takes $O(n)$ time: (i) n push operations on a vector with initial capacity $N = 1$; (ii) n pop (removal of the last element) operations.

C-6.5 Describe a function for performing a ***card shuffle*** of an array of $2n$ elements, by converting it into two lists. A card shuffle is a permutation where a list L is cut into two lists, L_1 and L_2, where L_1 is the first half of L and L_2 is the second half of L, and then these two lists are merged into one by taking the first element in L_1, then the first element in L_2, followed by the second element in L_1, the second element in L_2, and so on.

C-6.6 Show how to improve the implementation of function insert(i,e) in Code Fragment 6.5 so that, in case of an overflow, the elements are copied into their final place in the new array.

C-6.7 Consider an implementation of the vector ADT using an extendable array, but instead of copying the elements into an array of double the size (that is, from N to $2N$) when its capacity is reached, we copy the elements into an array with $\lceil N/4 \rceil$ additional cells, going from capacity N to $N + \lceil N/4 \rceil$. Show that performing a sequence of n push operations (that is, insertions at the end) still runs in $O(n)$ time in this case.

C-6.8 The NodeList implementation given in Code Fragments 6.9 through 6.12 does not do any checking to determine whether a given position p is actually a member of this particular list. For example, if p is a position in list S and we call T.insert(p,e) on a different list T, then we actually will add the element to S just before p. Describe how to change the NodeList implementation in an efficient manner to disallow such misuses.

C-6.9 Describe an implementation of the functions insertBack and insertFront realized by using combinations of only the functions empty and insert.

C-6.10 Consider the following fragment of C++ code, assuming that the constructor Sequence creates an empty sequence of integer objects. Recall that division between integers performs truncation (for example, $7/2 = 3$).

```
Sequence<int> seq;
for (int i = 0; i < n; i++)
    seq.insertAtRank(i/2, i);
```

a. Assume that the **for** loop is executed 10 times, that is, $n = 10$, and show the sequence after each iteration of the loop.

b. Draw a schematic illustration of the sequence at the end of the **for** loop, for a generic number n of iterations.

C-6.11 Suppose we want to extend the Sequence abstract data type with functions indexOfElement(e) and positionOfElement(e), which respectively return the index and the position of the (first occurrence of) element e in the

sequence. Show how to implement these functions by expressing them in terms of other functions of the Sequence interface.

C-6.12 Describe the structure and pseudo-code for an array-based implementation of the vector ADT that achieves $O(1)$ time for insertions and removals at index 0, as well as insertions and removals at the end of the vector. Your implementation should also provide for a constant time elemAtRank function.

C-6.13 Describe an efficient way of putting a vector representing a deck of n cards into random order. You may use a function, randomInteger(n), which returns a random number between 0 and $n-1$, inclusive. Your method should guarantee that every possible ordering is equally likely. What is the running time of your function?

C-6.14 Design a circular node list ADT that abstracts a circularly linked list in the same way that the node list ADT abstracts a doubly linked list.

C-6.15 An array is ***sparse*** if most of its entries are NULL. A list L can be used to implement such an array, A, efficiently. In particular, for each nonnull cell $A[i]$, we can store an entry (i,e) in L, where e is the element stored at $A[i]$. This approach allows us to represent A using $O(m)$ storage, where m is the number of nonnull entries in A. Describe and analyze efficient ways of performing the functions of the vector ADT on such a representation.

C-6.16 Show that only $n-1$ passes are needed in the execution of bubble-sort on a sequence with n elements.

C-6.17 Give a pseudo-code description of an implementation of the bubble-sort algorithm that uses only two stacks and, at most, five additional variables to sort a collection of objects stored initially in one of the stacks. You may operate on the stacks using only functions of the stack ADT. The final output should be one of the stacks containing all the elements so that a sequence of pop operations would list the elements in order.

C-6.18 A useful operation in databases is the ***natural join***. If we view a database as a list of ordered pairs of objects, then the natural join of databases A and B is the list of all ordered triples (x,y,z) such that the pair (x,y) is in A and the pair (y,z) is in B. Describe and analyze an efficient algorithm for computing the natural join of a list A of n pairs and a list B of m pairs.

C-6.19 When Bob wants to send Alice a message M on the Internet, he breaks M into n ***data packets***, numbers the packets consecutively, and injects them into the network. When the packets arrive at Alice's computer, they may be out of order, so Alice must assemble the sequence of n packets in order before she can be sure she has the entire message. Describe an efficient scheme for Alice to do this. What is the running time of this algorithm?

C-6.20 Given a list L of n positive integers, each represented with $k = \lceil \log n \rceil + 1$ bits, describe an $O(n)$-time function for finding a k-bit integer not in L.

C-6.21 Argue why any solution to the previous problem must run in $\Omega(n)$ time.

C-6.22 Given a list L of n arbitrary integers, design an $O(n)$-time function for finding an integer that cannot be formed as the sum of two integers in L.

Projects

P-6.1 Implement the vector ADT by means of an extendable array used in a circular fashion, so that insertions and deletions at the beginning and end of the vector run in constant time.

P-6.2 Implement the vector ADT using a doubly linked list. Show experimentally that this implementation is worse than the array-based approach.

P-6.3 Write a simple text editor, which stores a string of characters using the list ADT, together with a cursor object that highlights the position of some character in the string (or possibly the position before the first character). Your editor should support the following operations and redisplay the current text (that is, the list) after performing any one of them.

- left: Move cursor left one character (or nothing if at the beginning)
- right: Move cursor right one character (or do nothing if at the end)
- delete: Delete the character to the right of the cursor (or do nothing if at the end)
- insert c: Insert the character c just after the cursor

P-6.4 Implement the sequence ADT by means of an extendable array used in a circular fashion, so that insertions and deletions at the beginning and end of the sequence run in constant time.

P-6.5 Implement the sequence ADT by means of a singly linked list.

P-6.6 Write a complete adapter class that implements the sequence ADT using an STL vector object.

Chapter Notes

Sequences and iterators are pervasive concepts in the C++ Standard Template Library (STL) [81], and they play fundamental roles in JDSL, the data structures library in Java. For further information on STL vector and list classes, see books by Stroustrup [91], Lippmann and Lajoie [67], and Musser and Saini [81]. The list ADT was proposed by several authors, including Aho, Hopcroft, and Ullman [5], who introduce the "position" abstraction, and Wood [104], who defines a list ADT similar to ours. Implementations of sequences via arrays and linked lists are discussed in Knuth's seminal book, *Fundamental Algorithms* [56]. Knuth's companion volume, *Sorting and Searching* [57], describes the bubble-sort function and the history of this and other sorting algorithms.