4

Lists, Stacks, and Queues

If your program needs to store a few things — numbers, payroll records, or job descriptions for example — the simplest and most effective approach might be to put them in a list. Only when you have to organize and search through a large number of things do more sophisticated data structures usually become necessary. (We will study how to organize and search through medium amounts of data in Chapters 5, 7, and 9, and discuss how to deal with large amounts of data in Chapters 8–10.) Many applications don't require any form of search, and they do not require that any ordering be placed on the objects being stored. Some applications require processing in a strict chronological order, processing objects in the order that they arrived, or perhaps processing objects in the reverse of the order that they arrived. For all these situations, a simple list structure is appropriate.

This chapter describes representations for lists in general, as well as two important list-like structures called the stack and the queue. Along with presenting these fundamental data structures, the other goals of the chapter are to: (1) Give examples of separating a logical representation in the form of an ADT from a physical implementation for a data structure. (2) Illustrate the use of asymptotic analysis in the context of some simple operations that you might already be familiar with. In this way you can begin to see how asymptotic analysis works, without the complications that arise when analyzing more sophisticated algorithms and data structures. (3) Introduce the concept and use of dictionaries.

We begin by defining an ADT for lists in Section 4.1. Two implementations for the list ADT — the array-based list and the linked list — are covered in detail and their relative merits discussed. Sections 4.2 and 4.3 cover stacks and queues, respectively. Sample implementations for each of these data structures are presented. Section 4.4 presents the Dictionary ADT for storing and retrieving data, which sets a context for implementing search structures such as the Binary Search Tree of Section 5.4.

4.1 Lists

We all have an intuitive understanding of what we mean by a "list." Our first step is to define precisely what is meant so that this intuitive understanding can eventually be converted into a concrete data structure and its operations. The most important concept related to lists is that of **position**. In other words, we perceive that there is a first element in the list, a second element, and so on. We should view a list as embodying the mathematical concepts of a sequence, as defined in Section 2.1.

We define a **list** to be a finite, ordered sequence of data items known as **elements**. "Ordered" in this definition means that each element has a position in the list. (We will not use "ordered" in this context to mean that the list elements are sorted by value.) Each list element has a data type. In the simple list implementations discussed in this chapter, all elements of the list have the same data type, although there is no conceptual objection to lists whose elements have differing data types if the application requires it (see Section 12.1). The operations defined as part of the list ADT do not depend on the elemental data type. For example, the list ADT can be used for lists of integers, lists of characters, lists of payroll records, even lists of lists.

A list is said to be **empty** when it contains no elements. The number of elements currently stored is called the **length** of the list. The beginning of the list is called the **head**, the end of the list is called the **tail**. There might or might not be some relationship between the value of an element and its position in the list. For example, **sorted lists** have their elements positioned in ascending order of value, while **unsorted lists** have no particular relationship between element values and positions. This section will consider only unsorted lists. Chapters 7 and 9 treat the problems of how to create and search sorted lists efficiently.

When presenting the contents of a list, we use the same notation as was introduced for sequences in Section 2.1. To be consistent with **C++** array indexing, the first position on the list is denoted as 0. Thus, if there are n elements in the list, they are given positions 0 through $n-1$ as $\langle a_0,\ a_1,\ ...,\ a_{n-1}\rangle$. The subscript indicates an element's position within the list. Using this notation, the empty list would appear as $\langle\rangle$.

Before selecting a list implementation, a program designer should first consider what basic operations the implementation must support. Our common intuition about lists tells us that a list should be able to grow and shrink in size as we insert and remove elements. We should be able to insert and remove elements from anywhere in the list. We should be able to gain access to any element's value, either to read it or to change it. We must be able to create and clear (or reinitialize) lists. It is also convenient to access the next or previous element from the "current" one.

The next step is to define the ADT for a list object in terms of a set of operations on that object. We will use the **C++** notation of anabstract class to formally define

the list ADT. An abstract class is one whose member functions are all declared to be "pure virtual" as indicated by the "`=0`" notation at the end of the member function declarations. Class **`List`** defines the member functions that any list implementation inheriting from it must support, along with their parameters and return types. We increase the flexibility of the list ADT by writing it as a **C**++ template.

True to the notion of an ADT, anabstract class does not specify how operations are implemented. Two complete implementations are presented later in this section, both of which use the same list ADT to define their operations, but they are considerably different in approaches and in their space/time tradeoffs.

Figure 4.1 presents our list ADT. Class **`List`** is a template of one parameter, named **`E`** for "element". **`E`** serves as a placeholder for whatever element type the user would like to store in a list. The comments given in Figure 4.1 describe precisely what each member function is intended to do. However, some explanation of the basic design is in order. Given that we wish to support the concept of a sequence, with access to any position in the list, the need for many of the member functions such as **`insert`** and **`moveToPos`** is clear. The key design decision embodied in this ADT is support for the concept of a **current position**. For example, member **`moveToStart`** sets the current position to be the first element on the list, while methods **`next`** and **`prev`** move the current position to the next and previous elements, respectively. The intention is that any implementation for this ADT support the concept of a current position. The current position is where any action such as insertion or deletion will take place.

Since insertions take place at the current position, and since we want to be able to insert to the front or the back of the list as well as anywhere in between, there are actually $n + 1$ possible "current positions" when there are n elements in the list.

It is helpful to modify our list display notation to show the position of the current element. I will use a vertical bar, such as $\langle 20,\ 23 \mid 12,\ 15\rangle$ to indicate the list of four elements, with the current position being to the right of the bar at element 12. Given this configuration, calling **`insert`** with value 10 will change the list to be $\langle 20,\ 23 \mid 10,\ 12,\ 15\rangle$.

If you examine Figure 4.1, you should find that the list member functions provided allow you to build a list with elements in any desired order, and to access any desired position in the list. You might notice that the **`clear`** method is not necessary, in that it could be implemented by means of the other member functions in the same asymptotic time. It is included merely for convenience.

Method **`getValue`** returns a pointer to the current element. It is considered a violation of **`getValue`**'s preconditions to ask for the value of a non-existent element (i.e., there must be something to the right of the vertical bar). In our concrete list implementations, assertions are used to enforce such preconditions. In a commercial implementation, such violations would be best implemented by the **C**++ exception mechanism.

```
template <typename E> class List { // List ADT
private:
  void operator =(const List&) {}      // Protect assignment
  List(const List&) {}           // Protect copy constructor
public:
  List() {}          // Default constructor
  virtual ~List() {} // Base destructor

  // Clear contents from the list, to make it empty.
  virtual void clear() = 0;

  // Insert an element at the current location.
  // item: The element to be inserted
  virtual void insert(const E& item) = 0;

  // Append an element at the end of the list.
  // item: The element to be appended.
  virtual void append(const E& item) = 0;

  // Remove and return the current element.
  // Return: the element that was removed.
  virtual E remove() = 0;

  // Set the current position to the start of the list
  virtual void moveToStart() = 0;

  // Set the current position to the end of the list
  virtual void moveToEnd() = 0;

  // Move the current position one step left. No change
  // if already at beginning.
  virtual void prev() = 0;

  // Move the current position one step right. No change
  // if already at end.
  virtual void next() = 0;

  // Return: The number of elements in the list.
  virtual int length() const = 0;

  // Return: The position of the current element.
  virtual int currPos() const = 0;

  // Set current position.
  // pos: The position to make current.
  virtual void moveToPos(int pos) = 0;

  // Return: The current element.
  virtual const E& getValue() const = 0;
};
```

Figure 4.1 The ADT for a list.

A list can be iterated through as shown in the following code fragment.

```
for (L.moveToStart(); L.currPos()<L.length(); L.next()) {
  it = L.getValue();
  doSomething(it);
}
```

In this example, each element of the list in turn is stored in **it**, and passed to the **doSomething** function. The loop terminates when the current position reaches the end of the list.

The declaration for abstract class **List** also makes private the class copy constructor and an overloading for the assignment operator. This protects the class from accidentally being copied. This is done in part to simplify the example code used in this book. A full-featured list implementation would likely support copying and assigning list objects.

The list class declaration presented here is just one of many possible interpretations for lists. Figure 4.1 provides most of the operations that one naturally expects to perform on lists and serves to illustrate the issues relevant to implementing the list data structure. As an example of using the list ADT, we can create a function to return **true** if there is an occurrence of a given integer in the list, and **false** otherwise. The **find** method needs no knowledge about the specific list implementation, just the list ADT.

```
// Return true if "K" is in list "L", false otherwise
bool find(List<int>& L, int K) {
  int it;
  for (L.moveToStart(); L.currPos()<L.length(); L.next()) {
    it = L.getValue();
    if (K == it) return true;  // Found K
  }
  return false;                // K not found
}
```

While this implementation for **find** could be written as a template with respect to the element type, it would still be limited in its ability to handle different data types stored on the list. In particular, it only works when the description for the object being searched for (**k** in the function) is of the same type as the objects themselves, and that can meaningfully be compared when using the **==** comparison operator. A more typical situation is that we are searching for a record that contains a key field who's value matches **k**. Similar functions to find and return a composite element based on a key value can be created using the list implementation, but to do so requires some agreement between the list ADT and the **find** function on the concept of a key, and on how keys may be compared. This topic will be discussed in Section 4.4.

4.1.1 Array-Based List Implementation

There are two standard approaches to implementing lists, the **array-based** list, and the **linked** list. This section discusses the array-based approach. The linked list is presented in Section 4.1.2. Time and space efficiency comparisons for the two are discussed in Section 4.1.3.

Figure 4.2 shows the array-based list implementation, named **`AList`**. **`AList`** inherits from abstract class **`List`** and so must implement all of the member functions of **`List`**.

Class **`AList`**'s private portion contains the data members for the array-based list. These include **`listArray`**, the array which holds the list elements. Because **`listArray`** must be allocated at some fixed size, the size of the array must be known when the list object is created. Note that an optional parameter is declared for the **`AList`** constructor. With this parameter, the user can indicate the maximum number of elements permitted in the list. The phrase "**`=defaultSize`**" indicates that the parameter is optional. If no parameter is given, then it takes the value **`defaultSize`**, which is assumed to be a suitably defined constant value.

Because each list can have a differently sized array, each list must remember its maximum permitted size. Data member **`maxSize`** serves this purpose. At any given time the list actually holds some number of elements that can be less than the maximum allowed by the array. This value is stored in **`listSize`**. Data member **`curr`** stores the current position. Because **`listArray`**, **`maxSize`**, **`listSize`**, and **`curr`** are all declared to be **`private`**, they may only be accessed by methods of Class **`AList`**.

Class **`AList`** stores the list elements in the first **`listSize`** contiguous array positions. Array positions correspond to list positions. In other words, the element at position i in the list is stored at array cell i. The head of the list is always at position 0. This makes random access to any element in the list quite easy. Given some position in the list, the value of the element in that position can be accessed directly. Thus, access to any element using the **`moveToPos`** method followed by the **`getValue`** method takes $\Theta(1)$ time.

Because the array-based list implementation is defined to store list elements in contiguous cells of the array, the **`insert`**, **`append`**, and **`remove`** methods must maintain this property. Inserting or removing elements at the tail of the list is easy, so the **`append`** operation takes $\Theta(1)$ time. But if we wish to insert an element at the head of the list, all elements currently in the list must shift one position toward the tail to make room, as illustrated by Figure 4.3. This process takes $\Theta(n)$ time if there are n elements already in the list. If we wish to insert at position i within a list of n elements, then $n - i$ elements must shift toward the tail. Removing an element from the head of the list is similar in that all remaining elements must shift toward the head by one position to fill in the gap. To remove the element at position

```
template <typename E> // Array-based list implementation
class AList : public List<E> {
private:
  int maxSize;        // Maximum size of list
  int listSize;       // Number of list items now
  int curr;           // Position of current element
  E* listArray;    // Array holding list elements

public:
  AList(int size=defaultSize) { // Constructor
    maxSize = size;
    listSize = curr = 0;
    listArray = new E[maxSize];
  }

  ~AList() { delete [] listArray; } // Destructor

  void clear() {                    // Reinitialize the list
    delete [] listArray;            // Remove the array
    listSize = curr = 0;            // Reset the size
    listArray = new E[maxSize];  // Recreate array
  }

  // Insert "it" at current position
  void insert(const E& it) {
    Assert(listSize < maxSize, "List capacity exceeded");
    for(int i=listSize; i>curr; i--)  // Shift elements up
      listArray[i] = listArray[i-1];  //   to make room
    listArray[curr] = it;
    listSize++;                       // Increment list size
  }

  void append(const E& it) {       // Append "it"
    Assert(listSize < maxSize, "List capacity exceeded");
    listArray[listSize++] = it;
  }

  // Remove and return the current element.
  E remove() {
    Assert((curr>=0) && (curr < listSize), "No element");
    E it = listArray[curr];           // Copy the element
    for(int i=curr; i<listSize-1; i++)  // Shift them down
      listArray[i] = listArray[i+1];
    listSize--;                           // Decrement size
    return it;
  }
```

Figure 4.2 An array-based list implementation.

```
  void moveToStart() { curr = 0; }        // Reset position
  void moveToEnd() { curr = listSize; }     // Set at end
  void prev() { if (curr != 0) curr--; }       // Back up
  void next() { if (curr < listSize) curr++; } // Next

  // Return list size
  int length() const  { return listSize; }

  // Return current position
  int currPos() const { return curr; }

  // Set current list position to "pos"
  void moveToPos(int pos) {
    Assert ((pos>=0)&&(pos<=listSize), "Pos out of range");
    curr = pos;
  }

  const E& getValue() const { // Return current element
    Assert((curr>=0)&&(curr<listSize),"No current element");
    return listArray[curr];
  }
};
```

Figure 4.2 (continued)

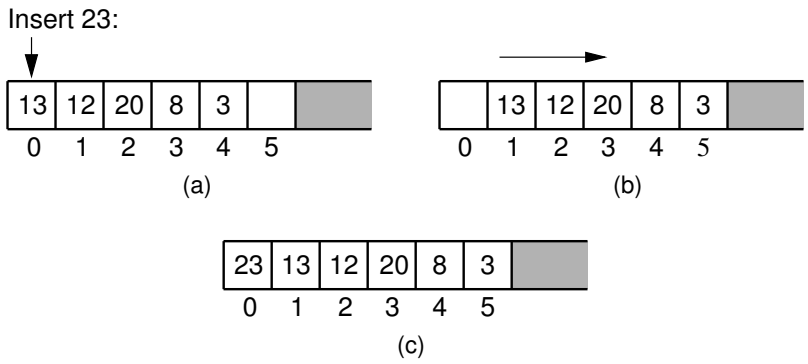

Figure 4.3 Inserting an element at the head of an array-based list requires shifting all existing elements in the array by one position toward the tail. (a) A list containing five elements before inserting an element with value 23. (b) The list after shifting all existing elements one position to the right. (c) The list after 23 has been inserted in array position 0. Shading indicates the unused part of the array.

i, $n - i - 1$ elements must shift toward the head. In the average case, insertion or removal requires moving half of the elements, which is $\Theta(n)$.

Most of the other member functions for Class **AList** simply access the current list element or move the current position. Such operations all require $\Theta(1)$ time. Aside from **insert** and **remove**, the only other operations that might require

```
// Singly linked list node
template <typename E> class Link {
public:
  E element;       // Value for this node
  Link *next;         // Pointer to next node in list
  // Constructors
  Link(const E& elemval, Link* nextval =NULL)
    { element = elemval;  next = nextval; }
  Link(Link* nextval =NULL) { next = nextval; }
};
```

Figure 4.4 A simple singly linked list node implementation.

more than constant time are the constructor, the destructor, and **clear**. These three member functions each make use of the system free-storeoperators **new** and **delete**. As discussed further in Section 4.1.2, system free-store operations can be expensive. In particular, the cost to delete **listArray** depends in part on the type of elements it stores, and whether the **delete** operator must call a destructor on each one.

4.1.2 Linked Lists

The second traditional approach to implementing lists makes use of pointers and is usually called a **linked list**. The linked list uses **dynamic memory allocation**, that is, it allocates memory for new list elements as needed.

A linked list is made up of a series of objects, called the **nodes** of the list. Because a list node is a distinct object (as opposed to simply a cell in an array), it is good practice to make a separate list node class. An additional benefit to creating a list node class is that it can be reused by the linked implementations for the stack and queue data structures presented later in this chapter. Figure 4.4 shows the implementation for list nodes, called the **Link** class. Objects in the **Link** class contain an **element** field to store the element value, and a **next** field to store a pointer to the next node on the list. The list built from such nodes is called a **singly linked list**, or a **one-way list**, because each list node has a single pointer to the next node on the list.

The **Link** class is quite simple. There are two forms for its constructor, one with an initial element value and one without. Because the **Link** class is also used by the stack and queue implementations presented later, its data members are made public. While technically this is breaking encapsulation, in practice the **Link** class should be implemented as a private class of the linked list (or stack or queue) implementation, and thus not visible to the rest of the program.

Figure 4.5(a) shows a graphical depiction for a linked list storing four integers. The value stored in a pointer variable is indicated by an arrow "pointing" to something. C++ uses the special symbol **NULL** for a pointer value that points nowhere, such as for the last list node's **next** field. A **NULL** pointer is indicated graphically

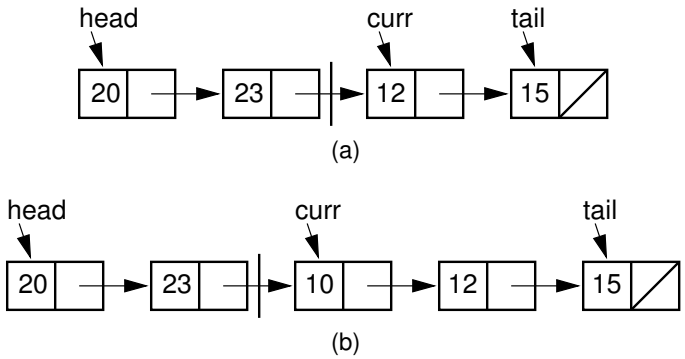

Figure 4.5 Illustration of a faulty linked-list implementation where **curr** points directly to the current node. (a) Linked list prior to inserting element with value 10. (b) Desired effect of inserting element with value 10.

by a diagonal slash through a pointer variable's box. The vertical line between the nodes labeled 23 and 12 in Figure 4.5(a) indicates the current position (immediately to the right of this line).

The list's first node is accessed from a pointer named **head**. To speed access to the end of the list, and to allow the **append** method to be performed in constant time, a pointer named **tail** is also kept to the last link of the list. The position of the current element is indicated by another pointer, named **curr**. Finally, because there is no simple way to compute the length of the list simply from these three pointers, the list length must be stored explicitly, and updated by every operation that modifies the list size. The value **cnt** stores the length of the list.

Class **LList** also includes private helper methods **init** and **removeall**. They are used by **LList**'s constructor, destructor, and **clear** methods.

Note that **LList**'s constructor maintains the optional parameter for minimum list size introduced for Class **AList**. This is done simply to keep the calls to the constructor the same for both variants. Because the linked list class does not need to declare a fixed-size array when the list is created, this parameter is unnecessary for linked lists. It is ignored by the implementation.

A key design decision for the linked list implementation is how to represent the current position. The most reasonable choices appear to be a pointer to the current element. But there is a big advantage to making **curr** point to the element preceding the current element.

Figure 4.5(a) shows the list's **curr** pointer pointing to the current element. The vertical line between the nodes containing 23 and 12 indicates the logical position of the current element. Consider what happens if we wish to insert a new node with value 10 into the list. The result should be as shown in Figure 4.5(b). However, there is a problem. To "splice" the list node containing the new element into the list, the list node storing 23 must have its **next** pointer changed to point to the new

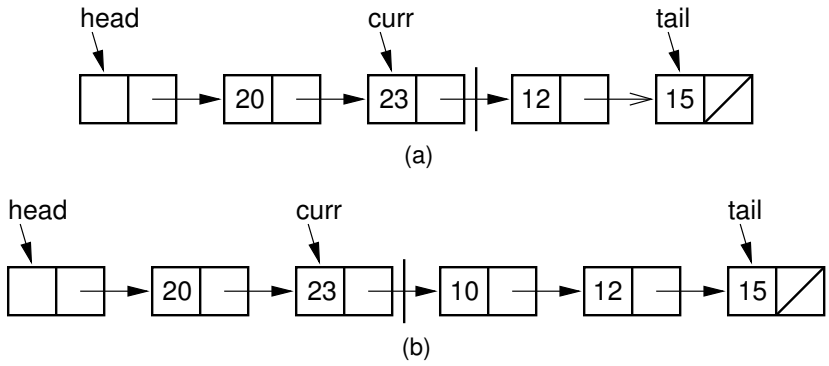

Figure 4.6 Insertion using a header node, with **curr** pointing one node head of the current element. (a) Linked list before insertion. The current node contains 12. (b) Linked list after inserting the node containing 10.

node. Unfortunately, there is no convenient access to the node preceding the one pointed to by **curr**.

There is an easy solution to this problem. If we set **curr** to point directly to the preceding element, there is no difficulty in adding a new element after **curr**. Figure 4.6 shows how the list looks when pointer variable **curr** is set to point to the node preceding the physical current node. See Exercise 4.5 for further discussion of why making **curr** point directly to the current element fails.

We encounter a number of potential special cases when the list is empty, or when the current position is at an end of the list. In particular, when the list is empty we have no element for **head**, **tail**, and **curr** to point to. Implementing special cases for **insert** and **remove** increases code complexity, making it harder to understand, and thus increases the chance of introducing a programming bug.

These special cases can be eliminated by implementing linked lists with an additional **header node** as the first node of the list. This header node is a link node like any other, but its value is ignored and it is not considered to be an actual element of the list. The header node saves coding effort because we no longer need to consider special cases for empty lists or when the current position is at one end of the list. The cost of this simplification is the space for the header node. However, there are space savings due to smaller code size, because statements to handle the special cases are omitted. In practice, this reduction in code size typically saves more space than that required for the header node, depending on the number of lists created. Figure 4.7 shows the state of an initialized or empty list when using a header node.

Figure 4.8 shows the definition for the linked list class, named **LList**. Class **LList** inherits from the abstract list class and thus must implement all of Class **List**'s member functions.

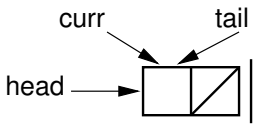

Figure 4.7 Initial state of a linked list when using a header node.

Implementations for most member functions of the **list** class are straightforward. However, **insert** and **remove** should be studied carefully.

Inserting a new element is a three-step process. First, the new list node is created and the new element is stored into it. Second, the **next** field of the new list node is assigned to point to the current node (the one *after* the node that **curr** points to). Third, the **next** field of node pointed to by **curr** is assigned to point to the newly inserted node. The following line in the **insert** method of Figure 4.8 does all three of these steps.

```
curr->next = new Link<E>(it, curr->next);
```

Operator **new** creates the new link node and calls the **Link** class constructor, which takes two parameters. The first is the element. The second is the value to be placed in the list node's **next** field, in this case"**curr->next**." Figure 4.9 illustrates this three-step process. Once the new node is added, **tail** is pushed forward if the new element was added to the end of the list. Insertion requires $\Theta(1)$ time.

Removing a node from the linked list requires only that the appropriate pointer be redirected around the node to be deleted. The following lines from the **remove** method of Figure 4.8 do precisely this.

```
Link<E>* ltemp = curr->next;   // Remember link node
curr->next = curr->next->next; // Remove from list
```

We must be careful not to "lose" the memory for the deleted link node. So, temporary pointer **ltemp** is first assigned to point to the node being removed. A call to **delete** is later used to return the old node to free storage. Figure 4.10 illustrates the **remove** method.Assuming that the free-store **delete** operator requires constant time, removing an element requires $\Theta(1)$ time.

Method **next** simply moves **curr** one position toward the tail of the list, which takes $\Theta(1)$ time. Method **prev** moves **curr** one position toward the head of the list, but its implementation is more difficult. In a singly linked list, there is no pointer to the previous node. Thus, the only alternative is to march down the list from the beginning until we reach the current node (being sure always to remember the node before it, because that is what we really want). This takes $\Theta(n)$ time in the average and worst cases. Implementation of method **moveToPos** is similar in that finding the ith position requires marching down i positions from the head of the list, taking $\Theta(i)$ time.

Implementations for the remaining operations each require $\Theta(1)$ time.

```
// Linked list implementation
template <typename E> class LList: public List<E> {
private:
  Link<E>* head;       // Pointer to list header
  Link<E>* tail;       // Pointer to last element
  Link<E>* curr;       // Access to current element
  int cnt;             // Size of list

  void init() {        // Intialization helper method
    curr = tail = head = new Link<E>;
    cnt = 0;
  }

  void removeall() {   // Return link nodes to free store
    while(head != NULL) {
      curr = head;
      head = head->next;
      delete curr;
    }
  }

public:
  LList(int size=defaultSize) { init(); }    // Constructor
  ~LList() { removeall(); }                   // Destructor
  void print() const;                 // Print list contents
  void clear() { removeall(); init(); }       // Clear list

  // Insert "it" at current position
  void insert(const E& it) {
    curr->next = new Link<E>(it, curr->next);
    if (tail == curr) tail = curr->next;  // New tail
    cnt++;
  }

  void append(const E& it) { // Append "it" to list
    tail = tail->next = new Link<E>(it, NULL);
    cnt++;
  }

  // Remove and return current element
  E remove() {
    Assert(curr->next != NULL, "No element");
    E it = curr->next->element;       // Remember value
    Link<E>* ltemp = curr->next;      // Remember link node
    if (tail == curr->next) tail = curr; // Reset tail
    curr->next = curr->next->next;    // Remove from list
    delete ltemp;                     // Reclaim space
    cnt--;                            // Decrement the count
    return it;
  }
```

Figure 4.8 A linked list implementation.

```
  void moveToStart() // Place curr at list start
    { curr = head; }

  void moveToEnd()   // Place curr at list end
    { curr = tail; }

  // Move curr one step left; no change if already at front
  void prev() {
    if (curr == head) return;       // No previous element
    Link<E>* temp = head;
    // March down list until we find the previous element
    while (temp->next!=curr) temp=temp->next;
    curr = temp;
  }

  // Move curr one step right; no change if already at end
  void next()
    { if (curr != tail) curr = curr->next; }

  int length() const  { return cnt; } // Return length

  // Return the position of the current element
  int currPos() const {
    Link<E>* temp = head;
    int i;
    for (i=0; curr != temp; i++)
      temp = temp->next;
    return i;
  }

  // Move down list to "pos" position
  void moveToPos(int pos) {
    Assert ((pos>=0)&&(pos<=cnt), "Position out of range");
    curr = head;
    for(int i=0; i<pos; i++) curr = curr->next;
  }

  const E& getValue() const { // Return current element
    Assert(curr->next != NULL, "No value");
    return curr->next->element;
  }
};
```

Figure 4.8 (continued)

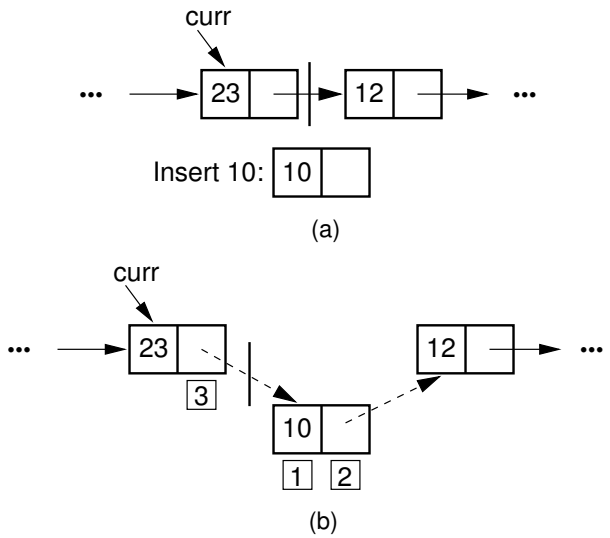

Figure 4.9 The linked list insertion process. (a) The linked list before insertion. (b) The linked list after insertion. [1] marks the **`element`** field of the new link node. [2] marks the **`next`** field of the new link node, which is set to point to what used to be the current node (the node with value 12). [3] marks the **`next`** field of the node preceding the current position. It used to point to the node containing 12; now it points to the new node containing 10.

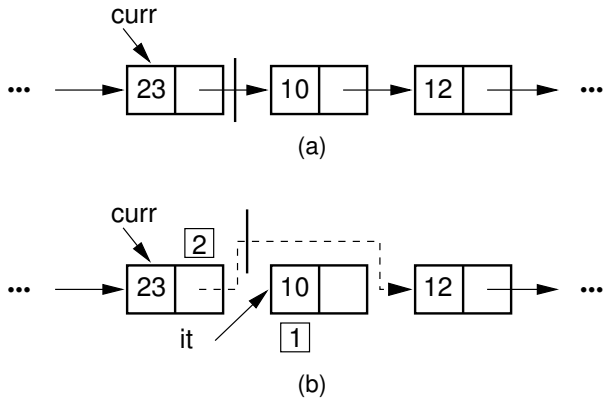

Figure 4.10 The linked list removal process. (a) The linked list before removing the node with value 10. (b) The linked list after removal. [1] marks the list node being removed. **`it`** is set to point to the element. [2] marks the **`next`** field of the preceding list node, which is set to point to the node following the one being deleted.

Freelists

The C++ free-store management operators **new** and **delete** are relatively expensive to use. Section 12.3 discusses how general-purpose memory managers are implemented. The expense comes from the fact that free-store routines must be capable of handling requests to and from free store with no particular pattern, as well as memory requests of vastly different sizes. Thismakes them inefficient compared to what might be implemented for more controlled patterns of memory access.

List nodes are created and deleted in a linked list implementation in a way that allows the **Link** class programmer to provide simple but efficient memory management routines. Instead of making repeated calls to **new** and **delete**, the **Link** class can handle its own **freelist**. A freelist holds those list nodes that are not currently being used. When a node is deleted from a linked list, it is placed at the head of the freelist. When a new element is to be added to a linked list, the freelist is checked to see if a list node is available. If so, the node is taken from the freelist. If the freelist is empty, the standard **new** operator must then be called.

Freelists are particularly useful for linked lists that periodically grow and then shrink. The freelist will never grow larger than the largest size yet reached by the linked list. Requests for new nodes (after the list has shrunk) can be handled by the freelist. Another good opportunity to use a freelist occurs when a program uses multiple lists. So long as they do not all grow and shrink together, the free list can let link nodes move between the lists.

One approach to implementing freelists would be to create two new operators to use instead of the standard free-store routines **new** and **delete**. This requires that the user's code, such as the linked list class implementation of Figure 4.8, be modified to call these freelist operators. A second approach is to use **C++ operator overloading** to replace the meaning of **new** and **delete** when operating on **Link** class objects. In this way, programs that use the **LList** class need not be modified at all to take advantage of a freelist. Whether the **Link** class is implemented with freelists, or relies on the regular free-store mechanism, is entirely hidden from the list class user. Figure 4.11 shows the reimplementation for the **Link** classwith freelist methods overloading the standard free-store operators. Note how simple they are, because they need only remove and add an element to the front of the freelist, respectively. The freelist versions of **new** and **delete** both run in $\Theta(1)$ time, except in the case where the freelist is exhausted and the **new** operation must be called. On my computer, a call to the overloaded **new** and **delete** operators requires about one tenth of the time required by the system free-store operators.

There is an additional efficiency gain to be had from a freelist implementation. The implementation of Figure 4.11 makes a separate call to the system **new** operator for each link node requested whenever the freelist is empty. These link nodes tend to be small — only a few bytes more than the size of the **element** field. If at some point in time the program requires thousands of active link nodes, these will

```
// Singly linked list node with freelist support
template <typename E> class Link {
private:
  static Link<E>* freelist; // Reference to freelist head
public:
  E element;                // Value for this node
  Link* next;                  // Point to next node in list

  // Constructors
  Link(const E& elemval, Link* nextval =NULL)
    { element = elemval;  next = nextval; }
  Link(Link* nextval =NULL) { next = nextval; }

  void* operator new(size_t) {  // Overloaded new operator
    if (freelist == NULL) return ::new Link; // Create space
    Link<E>* temp = freelist; // Can take from freelist
    freelist = freelist->next;
    return temp;                 // Return the link
  }

  // Overloaded delete operator
  void operator delete(void* ptr) {
    ((Link<E>*)ptr)->next = freelist; // Put on freelist
    freelist = (Link<E>*)ptr;
  }
};

// The freelist head pointer is actually created here
template <typename E>
Link<E>* Link<E>::freelist = NULL;
```

Figure 4.11 Implementation for the **Link** class with a freelist. Note that the redefinition for **new** refers to **::new** on the third line. This indicates that the standard C++ **new** operator is used, rather than the redefined **new** operator. If the colons had not been used, then the **Link** class **new** operator would be called, setting up an infinite recursion. The **static** declaration for member **freelist** means that all **Link** class objects share the same freelist pointer variable instead of each object storing its own copy.

have been created by many calls to the system version of **new**. An alternative is to allocate many link nodes in a single call to the system version of **new**, anticipating that if the freelist is exhausted now, more nodes will be needed soon. It is faster to make one call to **new** to get space for 100 **link** nodes, and then load all 100 onto the freelist at once, rather than to make 100 separate calls to **new**. The following statement will assign **ptr** to point to an array of 100 link nodes.

```
ptr = ::new Link[100];
```

The implementation for the **new** operator in the **link** class could then place each of these 100 nodes onto the freelist.

The **freelist** variable declaration uses the keyword **static**. This creates a single variable shared among all instances of the **Link** nodes. We want only a single freelist for all **Link** nodes of a given type. A program might create multiple lists. If they are all of the same type (that is, their element types are the same), then they can and should share the same freelist. This will happen with the implementation of Figure 4.11. If lists are created that have different element types, because this code is implemented with a template, the need for different list implementations will be discovered by the compiler at compile time. Separate versions of the list class will be generated for each element type. Thus, each element type will also get its own separate copy of the **Link** class. And each distinct **Link** class implementation will get a separate freelist.

4.1.3 Comparison of List Implementations

Now that you have seen two substantially different implementations for lists, it is natural to ask which is better. In particular, if you must implement a list for some task, which implementation should you choose?

Array-based lists have the disadvantage that their size must be predetermined before the array can be allocated. Array-based lists cannot grow beyond their predetermined size. Whenever the list contains only a few elements, a substantial amount of space might be tied up in a largely empty array. Linked lists have the advantage that they only need space for the objects actually on the list. There is no limit to the number of elements on a linked list, as long as there is free-store memory available. The amount of space required by a linked list is $\Theta(n)$, while the space required by the array-based list implementation is $\Omega(n)$, but can be greater.

Array-based lists have the advantage that there is no wasted space for an individual element. Linked lists require that an extra pointer be added to every list node. If the element size is small, then the overhead for links can be a significant fraction of the total storage. When the array for the array-based list is completely filled, there is no storage overhead. The array-based list will then be more space efficient, by a constant factor, than the linked implementation.

A simple formula can be used to determine whether the array-based list or linked list implementation will be more space efficient in a particular situation. Call n the number of elements currently in the list, P the size of a pointer in storage units (typically four bytes), E the size of a data element in storage units (this could be anything, from one bit for a Boolean variable on up to thousands of bytes or more for complex records), and D the maximum number of list elements that can be stored in the array. The amount of space required for the array-based list is DE, regardless of the number of elements actually stored in the list at any given time. The amount of space required for the linked list is $n(P + E)$. The smaller of these expressions for a given value n determines the more space-efficient implementation for n elements. In general, the linked implementation requires less space

than the array-based implementation when relatively few elements are in the list. Conversely, the array-based implementation becomes more space efficient when the array is close to full. Using the equation, we can solve for n to determine the break-even point beyond which the array-based implementation is more space efficient in any particular situation. This occurs when

$$n > DE/(P + E).$$

If $P = E$, then the break-even point is at $D/2$. This would happen if the element field is either a four-byte **int** value or a pointer, and the next field is a typical four-byte pointer. That is, the array-based implementation would be more efficient (if the link field and the element field are the same size) whenever the array is more than half full.

As a rule of thumb, linked lists are more space efficient when implementing lists whose number of elements varies widely or is unknown. Array-based lists are generally more space efficient when the user knows in advance approximately how large the list will become.

Array-based lists are faster for random access by position. Positions can easily be adjusted forwards or backwards by the **next** and **prev** methods. These operations always take $\Theta(1)$ time. In contrast, singly linked lists have no explicit access to the previous element, and access by position requires that we march down the list from the front (or the current position) to the specified position. Both of these operations require $\Theta(n)$ time in the average and worst cases, if we assume that each position on the list is equally likely to be accessed on any call to **prev** or **moveToPos**.

Given a pointer to a suitable location in the list, the **insert** and **remove** methods for linked lists require only $\Theta(1)$ time. Array-based lists must shift the remainder of the list up or down within the array. This requires $\Theta(n)$ time in the average and worst cases. For many applications, the time to insert and delete elements dominates all other operations. For this reason, linked lists are often preferred to array-based lists.

When implementing the array-based list, an implementor could allow the size of the array to grow and shrink depending on the number of elements that are actually stored. This data structure is known as a **dynamic array**. Both the Java and C++/STL **Vector** classes implement a dynamic array. Dynamic arrays allow the programmer to get around the limitation on the standard array that its size cannot be changed once the array has been created. This also means that space need not be allocated to the dynamic array until it is to be used. The disadvantage of this approach is that it takes time to deal with space adjustments on the array. Each time the array grows in size, its contents must be copied. A good implementation of the dynamic array will grow and shrink the array in such a way as to keep the overall cost for a series of insert/delete operations relatively inexpensive, even though an

occasional insert/delete operation might be expensive. A simple rule of thumb is to double the size of the array when it becomes full, and to cut the array size in half when it becomes one quarter full. To analyze the overall cost of dynamic array operations over time, we need to use a technique known as **amortized analysis**, which is discussed in Section 14.3.

4.1.4 Element Implementations

List users must decide whether they wish to store a copy of any given element on each list that contains it. For small elements such as an integer, this makes sense. If the elements are payroll records, it might be desirable for the list node to store a pointer to the record rather than store a copy of the record itself. This change would allow multiple list nodes (or other data structures) to point to the same record, rather than make repeated copies of the record. Not only might this save space, but it also means that a modification to an element's value is automatically reflected at all locations where it is referenced. The disadvantage of storing a pointer to each element is that the pointer requires space of its own. If elements are never duplicated, then this additional space adds unnecessary overhead.

The C++ implementations for lists presented in this section give the user of the list the choice of whether to store copies of elements or pointers to elements. The user can declare `E` to be, for example, a pointer to a payroll record. In this case, multiple lists can point to the same copy of the record. On the other hand, if the user declares `E` to be the record itself, then a new copy of the record will be made when it is inserted into the list.

Whether it is more advantageous to use pointers to shared elements or separate copies depends on the intended application. In general, the larger the elements and the more they are duplicated, the more likely that pointers to shared elements is the better approach.

A second issue faced by implementors of a list class (or any other data structure that stores a collection of user-defined data elements) is whether the elements stored are all required to be of the same type. This is known as **homogeneity** in a data structure. In some applications, the user would like to define the class of the data element that is stored on a given list, and then never permit objects of a different class to be stored on that same list. In other applications, the user would like to permit the objects stored on a single list to be of differing types.

For the list implementations presented in this section, the compiler requires that all objects stored on the list be of the same type. In fact, because the lists are implemented using templates, a new class is created by the compiler for each data type. For implementors who wish to minimize the number of classes created by the compiler, the lists can all store a `void*` pointer, with the user performing the necessary casting to and from the actual object type for each element. However, this

approach requires that the user do his or her own type checking, either to enforce homogeneity or to differentiate between the various object types.

Besides **C**++ templates, there are other techniques that implementors of a list class can use to ensure that the element type for a given list remains fixed, while still permitting different lists to store different element types. One approach is to store an object of the appropriate type in the header node of the list (perhaps an object of the appropriate type is supplied as a parameter to the list constructor), and then check that all insert operations on that list use the same element type.

The third issue that users of the list implementations must face is primarily of concern when programming in languages that do not support automatic garbage collection. That is how to deal with the memory of the objects stored on the list when the list is deleted or the **`clear`** method is called. The list destructor and the **`clear`** method are problematic in that there is a potential that they will bemisused, thus causing a memory leak. The type of the element stored determines whether there is a potential for trouble here. If the elements are of a simple type such as an **`int`**, then there is no need to delete the elements explicitly. If the elements are of a user-defined class, then their own destructor will be called. However, what if the list elements are pointers to objects? Then deleting **`listArray`** in the array-based implementation, or deleting a link node in the linked list implementation, might remove the only reference to an object, leaving its memory space inaccessible. Unfortunately, there is no way for the list implementation to know whether a given object is pointed to in another part of the program or not. Thus, the user of the list must be responsible for deleting these objects when that is appropriate.

4.1.5 Doubly Linked Lists

The singly linked list presented in Section 4.1.2 allows for direct access from a list node only to the next node in the list. A **doubly linked list** allows convenient access from a list node to the next node and also to the preceding node on the list. The doubly linked list node accomplishes this in the obvious way by storing two pointers: one to the node following it (as in the singly linked list), and a second pointer to the node preceding it. The most common reason to use a doubly linked list is because it is easier to implement than a singly linked list. While the code for the doubly linked implementation is a little longer than for the singly linked version, it tends to be a bit more "obvious" in its intention, and so easier to implement and debug. Figure 4.12 illustrates the doubly linked list concept. Whether a list implementation is doubly or singly linked should be hidden from the **`List`** class user.

Like our singly linked list implementation, the doubly linked list implementation makes use of a header node. We also add a tailer node to the end of the list. The tailer is similar to the header, in that it is a node that contains no value, and it always exists. When the doubly linked list is initialized, the header and tailer nodes

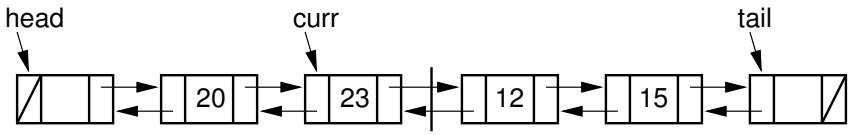

Figure 4.12 A doubly linked list.

are created. Data member **head** points to the header node, and **tail** points to the tailer node. The purpose of these nodes is to simplify the **insert**, **append**, and **remove** methods by eliminating all need for special-case code when the list is empty, or when we insert at the head or tail of the list.

For singly linked lists we set **curr** to point to the node preceding the node that contained the actual current element, due to lack of access to the previous node during insertion and deletion. Since we do have access to the previous node in a doubly linked list, this is no longer necessary. We could set **curr** to point directly to the node containing the current element. However, I have chosen to keep the same convention for the **curr** pointer as we set up for singly linked lists, purely for the sake of consistency.

Figure 4.13 shows the complete implementation for a **Link** class to be used with doubly linked lists. This code is a little longer than that for the singly linked list node implementation since the doubly linked list nodes have an extra data member.

Figure 4.14 shows the implementation for the **insert**, **append**, **remove**, and **prev** doubly linked list methods. The class declaration and the remaining member functions for the doubly linked list class are nearly identical to the singly linked list version.

The **insert** method is especially simple for our doubly linked list implementation, because most of the work is done by the node's constructor. Figure 4.15 shows the list before and after insertion of a node with value 10.

The three parameters to the **new** operator allow the list node class constructor to set the **element**, **prev**, and **next** fields, respectively, for the new link node. The **new** operator returns a pointer to the newly created node. The nodes to either side have their pointers updated to point to the newly created node. The existence of the header and tailer nodes mean that there are no special cases to worry about when inserting into an empty list.

The **append** method is also simple. Again, the **Link** class constructor sets the **element**, **prev**, and **next** fields of the node when the **new** operator is executed.

Method **remove** (illustrated by Figure 4.16) is straightforward, though the code is somewhat longer. First, the variable **it** is assigned the value being removed. Note that we must separate the element, which is returned to the caller, from the link object. The following lines then adjust the list.

```
// Doubly linked list link node with freelist support
template <typename E> class Link {
private:
  static Link<E>* freelist; // Reference to freelist head

public:
  E element;       // Value for this node
  Link* next;         // Pointer to next node in list
  Link* prev;         // Pointer to previous node

  // Constructors
  Link(const E& it, Link* prevp, Link* nextp) {
    element = it;
    prev = prevp;
    next = nextp;
  }
  Link(Link* prevp =NULL, Link* nextp =NULL) {
    prev = prevp;
    next = nextp;
  }

  void* operator new(size_t) {  // Overloaded new operator
    if (freelist == NULL) return ::new Link; // Create space
    Link<E>* temp = freelist; // Can take from freelist
    freelist = freelist->next;
    return temp;                 // Return the link
  }

  // Overloaded delete operator
  void operator delete(void* ptr) {
    ((Link<E>*)ptr)->next = freelist; // Put on freelist
    freelist = (Link<E>*)ptr;
  }
};

// The freelist head pointer is actually created here
template <typename E>
Link<E>* Link<E>::freelist = NULL;
```

Figure 4.13 Doubly linked list node implementation with a freelist.

```
// Insert "it" at current position
void insert(const E& it) {
  curr->next = curr->next->prev =
    new Link<E>(it, curr, curr->next);
  cnt++;
}

// Append "it" to the end of the list.
void append(const E& it) {
  tail->prev = tail->prev->next =
    new Link<E>(it, tail->prev, tail);
  cnt++;
}

// Remove and return current element
E remove() {
  if (curr->next == tail)        // Nothing to remove
    return NULL;
  E it = curr->next->element;    // Remember value
  Link<E>* ltemp = curr->next;   // Remember link node
  curr->next->next->prev = curr;
  curr->next = curr->next->next; // Remove from list
  delete ltemp;                  // Reclaim space
  cnt--;                         // Decrement cnt
  return it;
}

// Move fence one step left; no change if left is empty
void prev() {
  if (curr != head)  // Can't back up from list head
    curr = curr->prev;
}
```

Figure 4.14 Implementations for doubly linked list **insert**, **append**, **remove**, and **prev** methods.

```
Link<E>* ltemp = curr->next;   // Remember link node
curr->next->next->prev = curr;
curr->next = curr->next->next; // Remove from list
delete ltemp;                  // Reclaim space
```

The first line sets a temporary pointer to the node being removed. The second line makes the next node's **prev** pointer point to the left of the node being removed. Finally, the **next** field of the node preceding the one being deleted is adjusted. The final steps of method **remove** are to update the listlength, return the deleted node to free store, and return the value of the deleted element.

The only disadvantage of the doubly linked list as compared to the singly linked list is the additional space used. The doubly linked list requires two pointers per node, and so in the implementation presented it requires twice as much overhead as the singly linked list.

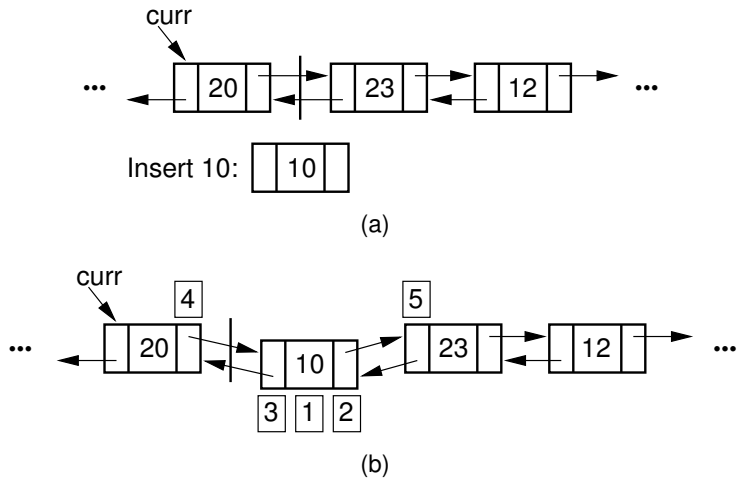

Figure 4.15 Insertion for doubly linked lists. The labels [1], [2], and [3] correspond to assignments done by the linked list node constructor. [4] marks the assignment to **`curr->next`**. [5] marks the assignment to the **`prev`** pointer of the node following the newly inserted node.

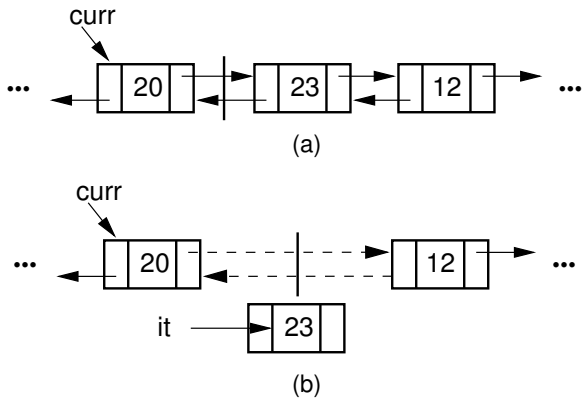

Figure 4.16 Doubly linked list removal. Element **`it`** stores the element of the node being removed. Then the nodes to either side have their pointers adjusted.

Example 4.1 There is a space-saving technique that can be employed to eliminate the additional space requirement, though it will complicate the implementation and be somewhat slower. Thus, this is an example of a space/time tradeoff. It is based on observing that, if we store the sum of two values, then we can get either value back by subtracting the other. That is, if we store $a + b$ in variable c, then $b = c - a$ and $a = c - b$. Of course, to recover one of the values out of the stored summation, the other value must be supplied. A pointer to the first node in the list, along with the value of one of its two link fields, will allow access to all of the remaining nodes of the list in order. This is because the pointer to the node must be the same as the value of the following node's **`prev`** pointer, as well as the previous node's **`next`** pointer. It is possible to move down the list breaking apart the summed link fields as though you were opening a zipper. Details for implementing this variation are left as an exercise.

The principle behind this technique is worth remembering, as it has many applications. The following code fragment will swap the contents of two variables without using a temporary variable (at the cost of three arithmetic operations).

```
a = a + b;
b = a - b; // Now b contains original value of a
a = a - b; // Now a contains original value of b
```

A similar effect can be had by using the exclusive-or operator. This fact is widely used in computer graphics. A region of the computer screen can be highlighted by XORing the outline of a box around it. XORing the box outline a second time restores the original contents of the screen.

4.2 Stacks

The **stack** is a list-like structure in which elements may be inserted or removed from only one end. While this restriction makes stacks less flexible than lists, it also makes stacks both efficient (for those operations they can do) and easy to implement. Many applications require only the limited form of insert and remove operations that stacks provide. In such cases, it is more efficient to use the simpler stack data structure rather than the generic list. For example, the freelist of Section 4.1.2 is really a stack.

Despite their restrictions, stacks have many uses. Thus, a special vocabulary for stacks has developed. Accountants used stacks long before the invention of the computer. They called the stack a "LIFO" list, which stands for "Last-In, First-

```
// Stack abtract class
template <typename E> class Stack {
private:
  void operator =(const Stack&) {}     // Protect assignment
  Stack(const Stack&) {}          // Protect copy constructor

public:
  Stack() {}                          // Default constructor
  virtual ~Stack() {}                 // Base destructor

  // Reinitialize the stack.  The user is responsible for
  // reclaiming the storage used by the stack elements.
  virtual void clear() = 0;

  // Push an element onto the top of the stack.
  // it: The element being pushed onto the stack.
  virtual void push(const E& it) = 0;

  // Remove the element at the top of the stack.
  // Return: The element at the top of the stack.
  virtual E pop() = 0;

  // Return: A copy of the top element.
  virtual const E& topValue() const = 0;

  // Return: The number of elements in the stack.
  virtual int length() const = 0;
};
```

Figure 4.17 The stack ADT.

Out." Note that one implication of the LIFO policy is that stacks remove elements in reverse order of their arrival.

The accessible element of the stack is called the **top** element. Elements are not said to be inserted, they are **pushed** onto the stack. When removed, an element is said to be **popped** from the stack. Figure 4.17 shows a sample stack ADT.

As with lists, there are many variations on stack implementation. The two approaches presented here are **array-based** and **linked stacks**, which are analogous to array-based and linked lists, respectively.

4.2.1 Array-Based Stacks

Figure 4.18 shows a complete implementation for the array-based stack class. As with the array-based list implementation, **`listArray`** must be declared of fixed size when the stack is created. In the stack constructor, **`size`** serves to indicate this size. Method **`top`** acts somewhat like a current position value (because the "current" position is always at the top of the stack), as well as indicating the number of elements currently in the stack.

```
// Array-based stack implementation
template <typename E> class AStack: public Stack<E> {
private:
  int maxSize;              // Maximum size of stack
  int top;                  // Index for top element
  E *listArray;          // Array holding stack elements

public:
  AStack(int size =defaultSize)   // Constructor
    { maxSize = size; top = 0; listArray = new E[size]; }

  ~AStack() { delete [] listArray; }  // Destructor

  void clear() { top = 0; }           // Reinitialize

  void push(const E& it) {         // Put "it" on stack
    Assert(top != maxSize, "Stack is full");
    listArray[top++] = it;
  }

  E pop() {                 // Pop top element
    Assert(top != 0, "Stack is empty");
    return listArray[--top];
  }

  const E& topValue() const {     // Return top element
    Assert(top != 0, "Stack is empty");
    return listArray[top-1];
  }

  int length() const { return top; }  // Return length
};
```

Figure 4.18 Array-based stack class implementation.

The array-based stack implementation is essentially a simplified version of the array-based list. The only important design decision to be made is which end of the array should represent the top of the stack. One choice is to make the top be at position 0 in the array. In terms of list functions, all **insert** and **remove** operations would then be on the element in position 0. This implementation is inefficient, because now every **push** or **pop** operation will require that all elements currently in the stack be shifted one position in the array, for a cost of $\Theta(n)$ if there are n elements. The other choice is have the top element be at position $n-1$ when there are n elements in the stack. In other words, as elements are pushed onto the stack, they are appended to the tail of the list. Method **pop** removes the tail element. In this case, the cost for each **push** or **pop** operation is only $\Theta(1)$.

For the implementation of Figure 4.18, **top** is defined to be the array index of the first free position in the stack. Thus, an empty stack has **top** set to 0, the first available free position in the array. (Alternatively, **top** could have been defined to

be the index for the top element in the stack, rather than the first free position. If this had been done, the empty list would initialize **top** as -1.) Methods **push** and **pop** simply place an element into, or remove an element from, the array position indicated by **top**. Because **top** is assumed to be at the first free position, **push** first inserts its value into the top position and then increments **top**, while **pop** first decrements **top** and then removes the top element.

4.2.2 Linked Stacks

The linked stack implementation is quite simple. The freelist of Section 4.1.2 is an example of a linked stack. Elements are inserted and removed only from the head of the list. A header node is not used because no special-case code is required for lists of zero or one elements. Figure 4.19 shows the complete linked stack implementation. The only data member is **top**, a pointer to the first (top) link node of the stack. Method **push** first modifies the **next** field of the newly created link node to point to the top of the stack and then sets **top** to point to the new link node. Method **pop** is also quite simple. Variable **temp** stores the top nodes' value, while **ltemp** links to the top node as it is removed from the stack. The stack is updated by setting **top** to point to the next link in the stack. The old top node is then returned to free store (or the freelist), and the element value is returned.

4.2.3 Comparison of Array-Based and Linked Stacks

All operations for the array-based and linked stack implementations take constant time, so from a time efficiency perspective, neither has a significant advantage. Another basis for comparison is the total space required. The analysis is similar to that done for list implementations. The array-based stack must declare a fixed-size array initially, and some of that space is wasted whenever the stack is not full. The linked stack can shrink and grow but requires the overhead of a link field for every element.

When multiple stacks are to be implemented, it is possible to take advantage of the one-way growth of the array-based stack. This can be done by using a single array to store two stacks. One stack grows inward from each end as illustrated by Figure 4.20, hopefully leading to less wasted space. However, this only works well when the space requirements of the two stacks are inversely correlated. In other words, ideally when one stack grows, the other will shrink. This is particularly effective when elements are taken from one stack and given to the other. If instead both stacks grow at the same time, then the free space in the middle of the array will be exhausted quickly.

```
// Linked stack implementation
template <typename E> class LStack: public Stack<E> {
private:
  Link<E>* top;             // Pointer to first element
  int size;                    // Number of elements

public:
  LStack(int sz =defaultSize) // Constructor
    { top = NULL; size = 0; }

  ~LStack() { clear(); }           // Destructor

  void clear() {                   // Reinitialize
    while (top != NULL) {          // Delete link nodes
      Link<E>* temp = top;
      top = top->next;
      delete temp;
    }
    size = 0;
  }

  void push(const E& it) { // Put "it" on stack
    top = new Link<E>(it, top);
    size++;
  }

  E pop() {                 // Remove "it" from stack
    Assert(top != NULL, "Stack is empty");
    E it = top->element;
    Link<E>* ltemp = top->next;
    delete top;
    top = ltemp;
    size--;
    return it;
  }

  const E& topValue() const { // Return top value
    Assert(top != 0, "Stack is empty");
    return top->element;
  }

  int length() const { return size; } // Return length
};
```

Figure 4.19 Linked stack class implementation.

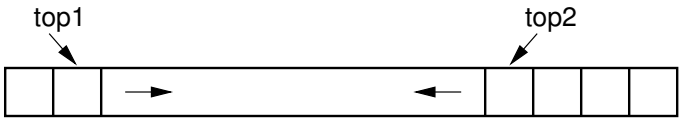

Figure 4.20 Two stacks implemented within in a single array, both growing toward the middle.

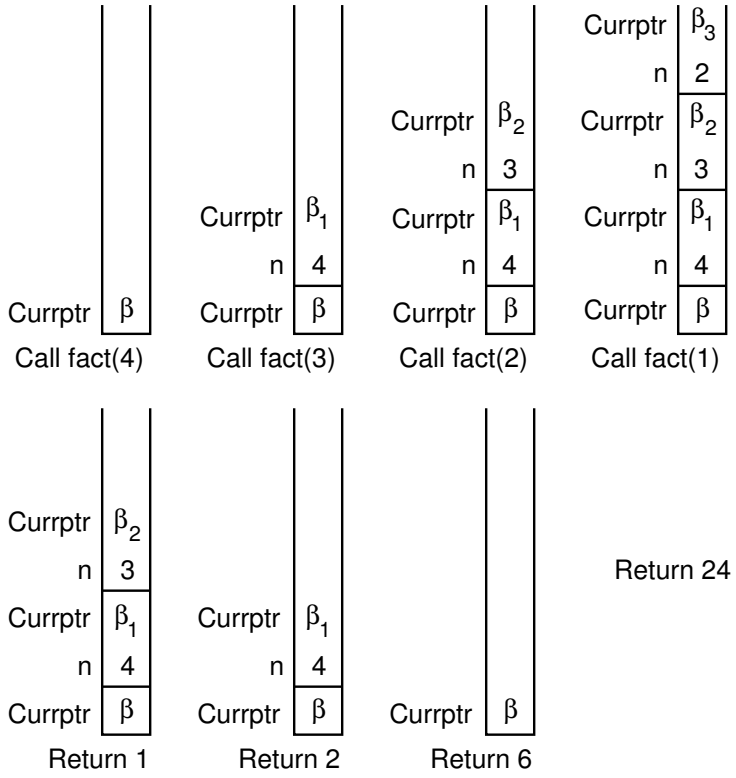

Figure 4.21 Implementing recursion with a stack. β values indicate the address of the program instruction to return to after completing the current function call. On each recursive function call to **`fact`** (as implemented in Section 2.5), both the return address and the current value of **`n`** must be saved. Each return from **`fact`** pops the top activation record off the stack.

4.2.4 Implementing Recursion

Perhaps the most common computer application that uses stacks is not even visible to its users. This is the implementation of subroutine calls in most programming language runtime environments. A subroutine call is normally implemented by placing necessary information about the subroutine (including the return address, parameters, and local variables) onto a stack. This information is called an **activation record**. Further subroutine calls add to the stack. Each return from a subroutine pops the top activation record off the stack. Figure 4.21 illustrates the implementation of the recursive factorial function of Section 2.5 from the runtime environment's point of view.

Consider what happens when we call **`fact`** with the value 4. We use β to indicate the address of the program instruction where the call to **`fact`** is made. Thus, the stack must first store the address β, and the value 4 is passed to **`fact`**.

Next, a recursive call to **fact** is made, this time with value 3. We will name the program address from which the call is made β_1. The address β_1, along with the current value for n (which is 4), is saved on the stack. Function **fact** is invoked with input parameter 3.

In similar manner, another recursive call is made with input parameter 2, requiring that the address from which the call is made (say β_2) and the current value for n (which is 3) are stored on the stack. A final recursive call with input parameter 1 is made, requiring that the stack store the calling address (say β_3) and current value (which is 2).

At this point, we have reached the base case for **fact**, and so the recursion begins to unwind. Each return from **fact** involves popping the stored value for n from the stack, along with the return address from the function call. The return value for **fact** is multiplied by the restored value for n, and the result is returned.

Because an activation record must be created and placed onto the stack for each subroutine call, making subroutine calls is a relatively expensive operation. While recursion is often used to make implementation easy and clear, sometimes you might want to eliminate the overhead imposed by the recursive function calls. In some cases, such as the factorial function of Section 2.5, recursion can easily be replaced by iteration.

Example 4.2 As a simple example of replacing recursion with a stack, consider the following non-recursive version of the factorial function.

```
long fact(int n, Stack<int>& S) { // Compute n!
  // To fit n! in a long variable, require n <= 12
  Assert((n >= 0) && (n <= 12), "Input out of range");
  while (n > 1) S.push(n--);  // Load up the stack
  long result = 1;            // Holds final result
  while(S.length() > 0)
     result = result * S.pop();   // Compute
  return result;
}
```

Here, we simply push successively smaller values of n onto the stack until the base case is reached, then repeatedly pop off the stored values and multiply them into the result.

An iterative form of the factorial function is both simpler and faster than the version shown in Example 4.2. But it is not always possible to replace recursion with iteration. Recursion, or some imitation of it, is necessary when implementing algorithms that require multiple branching such as in the Towers of Hanoi algorithm, or when traversing a binary tree. The Mergesort and Quicksort algorithms of Chapter 7 are also examples in which recursion is required. Fortunately, it is always possible to imitate recursion with a stack. Let us now turn to a non-recursive version of the Towers of Hanoi function, which cannot be done iteratively.

Example 4.3 The **TOH** function shown in Figure 2.2 makes two recursive calls: one to move $n-1$ rings off the bottom ring, and another to move these $n-1$ rings back to the goal pole. We can eliminate the recursion by using a stack to store a representation of the three operations that **TOH** must perform: two recursive calls and a move operation. To do so, we must first come up with a representation of the various operations, implemented as a class whose objects will be stored on the stack.

Figure 4.22 shows such a class. We first define an enumerated type called **TOHop**, with two values MOVE and TOH, to indicate calls to the **move** function and recursive calls to **TOH**, respectively. Class **TOHobj** stores five values: an operation field (indicating either a move or a new TOH operation), the number of rings, and the three poles. Note that the move operation actually needs only to store information about two poles. Thus, there are two constructors: one to store the state when imitating a recursive call, and one to store the state for a move operation.

An array-based stack is used because we know that the stack will need to store exactly $2n+1$ elements. The new version of **TOH** begins by placing on the stack a description of the initial problem for n rings. The rest of the function is simply a **while** loop that pops the stack and executes the appropriate operation. In the case of a **TOH** operation (for $n > 0$), we store on the stack representations for the three operations executed by the recursive version. However, these operations must be placed on the stack in reverse order, so that they will be popped off in the correct order.

Recursive algorithms lend themselves to efficient implementation with a stack when the amount of information needed to describe a sub-problem is small. For example, Section 7.5 discusses a stack-based implementation for Quicksort.

4.3 Queues

Like the stack, the **queue** is a list-like structure that provides restricted access to its elements. Queue elements may only be inserted at the back (called an **enqueue** operation) and removed from the front (called a **dequeue** operation). Queues operate like standing in line at a movie theater ticket counter.[1] If nobody cheats, then newcomers go to the back of the line. The person at the front of the line is the next to be served. Thus, queues release their elements in order of arrival. Accountants have used queues since long before the existence of computers. They call a queue a "FIFO" list, which stands for "First-In, First-Out." Figure 4.23 shows a sample

[1] In Britain, a line of people is called a "queue," and getting into line to wait for service is called "queuing up."

```
// Operation choices: DOMOVE will move a disk
// DOTOH corresponds to a recursive call
enum TOHop { DOMOVE, DOTOH };
class TOHobj { // An operation object
public:
  TOHop op;              // This operation type
  int num;               // How many disks
  Pole start, goal, tmp; // Define pole order

  // DOTOH operation constructor
  TOHobj(int n, Pole s, Pole g, Pole t) {
    op = DOTOH; num = n;
    start = s; goal = g; tmp = t;
  }

  // DOMOVE operation constructor
  TOHobj(Pole s, Pole g)
    { op = DOMOVE; start = s; goal = g; }
};

void TOH(int n, Pole start, Pole goal, Pole tmp,
         Stack<TOHobj*>& S) {
  S.push(new TOHobj(n, start, goal, tmp)); // Initial
  TOHobj* t;
  while (S.length() > 0) {       // Grab next task
    t = S.pop();
    if (t->op == DOMOVE)   // Do a move
      move(t->start, t->goal);
    else if (t->num > 0) {
      // Store (in reverse) 3 recursive statements
      int num = t->num;
      Pole tmp = t->tmp;  Pole goal = t->goal;
      Pole start = t->start;
      S.push(new TOHobj(num-1, tmp, goal, start));
      S.push(new TOHobj(start, goal));
      S.push(new TOHobj(num-1, start, tmp, goal));
    }
    delete t; // Must delete the TOHobj we made
  }
}
```

Figure 4.22 Stack-based implementation for Towers of Hanoi.

queue ADT. This section presents two implementations for queues: the array-based queue and the linked queue.

4.3.1 Array-Based Queues

The array-based queue is somewhat tricky to implement effectively. A simple conversion of the array-based list implementation is not efficient.

Assume that there are n elements in the queue. By analogy to the array-based list implementation, we could require that all elements of the queue be stored in the first n positions of the array. If we choose the rear element of the queue to be in

```
// Abstract queue class
template <typename E> class Queue {
private:
  void operator =(const Queue&) {}     // Protect assignment
  Queue(const Queue&) {}          // Protect copy constructor

public:
  Queue() {}           // Default
  virtual ~Queue() {} // Base destructor

  // Reinitialize the queue.  The user is responsible for
  // reclaiming the storage used by the queue elements.
  virtual void clear() = 0;

  // Place an element at the rear of the queue.
  // it: The element being enqueued.
  virtual void enqueue(const E&) = 0;

  // Remove and return element at the front of the queue.
  // Return: The element at the front of the queue.
  virtual E dequeue() = 0;

  // Return: A copy of the front element.
  virtual const E& frontValue() const = 0;

  // Return: The number of elements in the queue.
  virtual int length() const = 0;
};
```

Figure 4.23 The C++ ADT for a queue.

position 0, then **dequeue** operations require only $\Theta(1)$ time because the front element of the queue (the one being removed) is the last element in the array. However, **enqueue** operations will require $\Theta(n)$ time, because the n elements currently in the queue must each be shifted one position in the array. If instead we chose the rear element of the queue to be in position $n-1$, then an **enqueue** operation is equivalent to an **append** operation on a list. This requires only $\Theta(1)$ time. But now, a **dequeue** operation requires $\Theta(n)$ time, because all of the elements must be shifted down by one position to retain the property that the remaining $n-1$ queue elements reside in the first $n-1$ positions of the array.

A far more efficient implementation can be obtained by relaxing the requirement that all elements of the queue must be in the first n positions of the array. We will still require that the queue be stored be in contiguous array positions, but the contents of the queue will be permitted to drift within the array, as illustrated by Figure 4.24. Now, both the **enqueue** and the **dequeue** operations can be performed in $\Theta(1)$ time because no other elements in the queue need be moved.

This implementation raises a new problem. Assume that the front element of the queue is initially at position 0, and that elements are added to successively

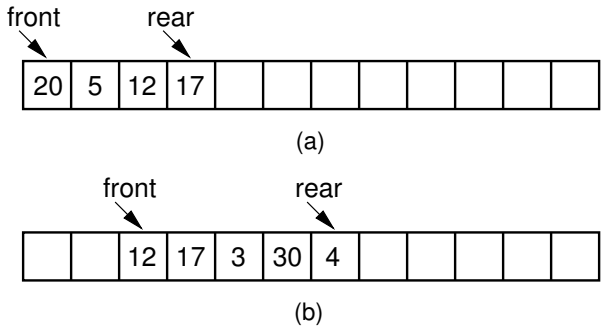

Figure 4.24 After repeated use, elements in the array-based queue will drift to the back of the array. (a) The queue after the initial four numbers 20, 5, 12, and 17 have been inserted. (b) The queue after elements 20 and 5 are deleted, following which 3, 30, and 4 are inserted.

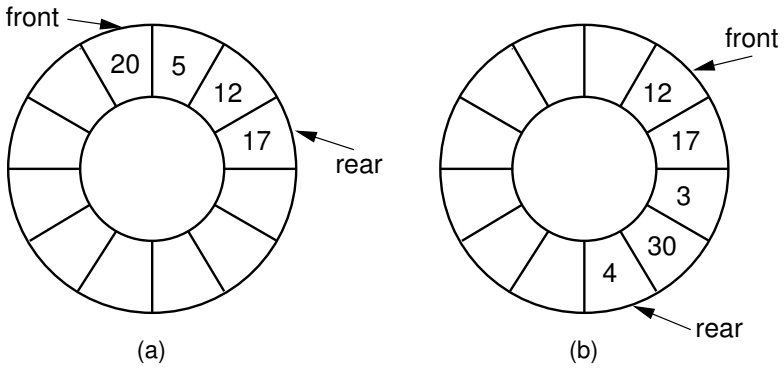

Figure 4.25 The circular queue with array positions increasing in the clockwise direction. (a) The queue after the initial four numbers 20, 5, 12, and 17 have been inserted. (b) The queue after elements 20 and 5 are deleted, following which 3, 30, and 4 are inserted.

higher-numbered positions in the array. When elements are removed from the queue, the front index increases. Over time, the entire queue will drift toward the higher-numbered positions in the array. Once an element is inserted into the highest-numbered position in the array, the queue has run out of space. This happens despite the fact that there might be free positions at the low end of the array where elements have previously been removed from the queue.

The "drifting queue" problem can be solved by pretending that the array is circular and so allow the queue to continue directly from the highest-numbered position in the array to the lowest-numbered position. This is easily implemented through use of the modulus operator (denoted by `%` in C++). In this way, positions in the array are numbered from 0 through **`size`**-1, and position **`size`**-1 is defined to immediately precede position 0 (which is equivalent to position **`size % size`**). Figure 4.25 illustrates this solution.

There remains one more serious, though subtle, problem to the array-based queue implementation. How can we recognize when the queue is empty or full? Assume that **front** stores the array index for the front element in the queue, and **rear** stores the array index for the rear element. If both **front** and **rear** have the same position, then with this scheme there must be one element in the queue. Thus, an empty queue would be recognized by having **rear** be *one less* than **front** (taking into account the fact that the queue is circular, so position **size**-1 is actually considered to be one less than position 0). But what if the queue is completely full? In other words, what is the situation when a queue with n array positions available contains n elements? In this case, if the front element is in position 0, then the rear element is in position **size**-1. But this means that the value for **rear** is one less than the value for **front** when the circular nature of the queue is taken into account. In other words, the full queue is indistinguishable from the empty queue!

You might think that the problem is in the assumption about **front** and **rear** being defined to store the array indices of the front and rear elements, respectively, and that some modification in this definition will allow a solution. Unfortunately, the problem cannot be remedied by a simple change to the definition for **front** and **rear**, because of the number of conditions or **states** that the queue can be in. Ignoring the actual position of the first element, and ignoring the actual values of the elements stored in the queue, how many different states are there? There can be no elements in the queue, one element, two, and so on. At most there can be n elements in the queue if there are n array positions. This means that there are $n + 1$ different states for the queue (0 through n elements are possible).

If the value of **front** is fixed, then $n+1$ different values for **rear** are needed to distinguish among the $n+1$ states. However, there are only n possible values for **rear** unless we invent a special case for, say, empty queues. This is an example of the Pigeonhole Principle defined in Exercise 2.30. The Pigeonhole Principle states that, given n pigeonholes and $n + 1$ pigeons, when all of the pigeons go into the holes we can be sure that at least one hole contains more than one pigeon. In similar manner, we can be sure that two of the $n + 1$ states are indistinguishable by the n relative values of **front** and **rear**. We must seek some other way to distinguish full from empty queues.

One obvious solution is to keep an explicit count of the number of elements in the queue, or at least a Boolean variable that indicates whether the queue is empty or not. Another solution is to make the array be of size $n + 1$, and only allow n elements to be stored. Which of these solutions to adopt is purely a matter of the implementor's taste in such affairs. My choice is to use an array of size $n + 1$.

Figure 4.26 shows an array-based queue implementation. **listArray** holds the queue elements, and as usual, the queue constructor allows an optional parameter to set the maximum size of the queue. The array as created is actually large enough to hold one element more than the queue will allow, so that empty queues

```
// Array-based queue implementation
template <typename E> class AQueue: public Queue<E> {
private:
  int maxSize;               // Maximum size of queue
  int front;                 // Index of front element
  int rear;                  // Index of rear element
  E *listArray;           // Array holding queue elements

public:
  AQueue(int size =defaultSize) {  // Constructor
    // Make list array one position larger for empty slot
    maxSize = size+1;
    rear = 0;  front = 1;
    listArray = new E[maxSize];
  }

  ~AQueue() { delete [] listArray; } // Destructor

  void clear() { rear = 0; front = 1; } // Reinitialize

  void enqueue(const E& it) {     // Put "it" in queue
    Assert(((rear+2) % maxSize) != front, "Queue is full");
    rear = (rear+1) % maxSize;       // Circular increment
    listArray[rear] = it;
  }

  E dequeue() {           // Take element out
    Assert(length() != 0, "Queue is empty");
    E it = listArray[front];
    front = (front+1) % maxSize;    // Circular increment
    return it;
  }

  const E& frontValue() const {  // Get front value
    Assert(length() != 0, "Queue is empty");
    return listArray[front];
  }

  virtual int length() const         // Return length
   { return ((rear+maxSize) - front + 1) % maxSize; }
};
```

Figure 4.26 An array-based queue implementation.

can be distinguished from full queues. Member **maxSize** is used to control the circular motion of the queue (it is the base for the modulus operator). Member **rear** is set to the position of the current rear element, while **front** is the position of the current front element.

In this implementation, the front of the queue is defined to be toward the lower numbered positions in the array (in the counter-clockwise direction in Figure 4.25), and the rear is defined to be toward the higher-numbered positions. Thus, **enqueue** increments the rear pointer (modulus **size**), and **dequeue** increments the front pointer. Implementation of all member functions is straightforward.

4.3.2 Linked Queues

The linked queue implementation is a straightforward adaptation of the linked list. Figure 4.27 shows the linked queue class declaration. Methods **front** and **rear** are pointers to the front and rear queue elements, respectively. We will use a header link node, which allows for a simpler implementation of the enqueue operation by avoiding any special cases when the queue is empty. On initialization, the **front** and **rear** pointers will point to the header node, and front will always point to the header node while rear points to the true last link node in the queue. Method **enqueue** places the new element in a link node at the end of the linked list (i.e., the node that **rear** points to) and then advances **rear** to point to the new link node. Method **dequeue** removes and returns the first element of the list.

4.3.3 Comparison of Array-Based and Linked Queues

All member functions for both the array-based and linked queue implementations require constant time. The space comparison issues are the same as for the equivalent stack implementations. Unlike the array-based stack implementation, there is no convenient way to store two queues in the same array, unless items are always transferred directly from one queue to the other.

4.4 Dictionaries

The most common objective of computer programs is to store and retrieve data. Much of this book is about efficient ways to organize collections of data records so that they can be stored and retrieved quickly. In this section we describe a simple interface for such a collection, called a **dictionary**. The dictionary ADT provides operations for storing records, finding records, and removing records from the collection. This ADT gives us a standard basis for comparing various data structures.

Before we can discuss the interface for a dictionary, we must first define the concepts of a **key** and **comparable** objects. If we want to search for a given record

```
// Linked queue implementation
template <typename E> class LQueue: public Queue<E> {
private:
  Link<E>* front;       // Pointer to front queue node
  Link<E>* rear;        // Pointer to rear queue node
  int size;                // Number of elements in queue

public:
  LQueue(int sz =defaultSize) // Constructor
    { front = rear = new Link<E>(); size = 0; }

  ~LQueue() { clear(); delete front; }      // Destructor

  void clear() {           // Clear queue
    while(front->next != NULL) { // Delete each link node
      rear = front;
      delete rear;
    }
    rear = front;
    size = 0;
  }

  void enqueue(const E& it) { // Put element on rear
    rear->next = new Link<E>(it, NULL);
    rear = rear->next;
    size++;
  }

  E dequeue() {              // Remove element from front
    Assert(size != 0, "Queue is empty");
    E it = front->next->element;  // Store dequeued value
    Link<E>* ltemp = front->next; // Hold dequeued link
    front->next = ltemp->next;       // Advance front
    if (rear == ltemp) rear = front; // Dequeue last element
    delete ltemp;                    // Delete link
    size --;
    return it;                       // Return element value
  }

  const E& frontValue() const { // Get front element
    Assert(size != 0, "Queue is empty");
    return front->next->element;
  }

  virtual int length() const { return size; }
};
```

Figure 4.27 Linked queue class implementation.

in a database, how should we describe what we are looking for? A database record could simply be a number, or it could be quite complicated, such as a payroll record with many fields of varying types. We do not want to describe what we are looking for by detailing and matching the entire contents of the record. If we knew everything about the record already, we probably would not need to look for it. Instead, we typically define what record we want in terms of a key value. For example, if searching for payroll records, we might wish to search for the record that matches a particular ID number. In this example the ID number is the **search key**.

To implement the search function, we require that keys be comparable. At a minimum, we must be able to take two keys and reliably determine whether they are equal or not. That is enough to enable a sequential search through a database of records and find one that matches a given key. However, we typically would like for the keys to define a total order (see Section 2.1), which means that we can tell which of two keys is greater than the other. Using key types with total orderings gives the database implementor the opportunity to organize a collection of records in a way that makes searching more efficient. An example is storing the records in sorted order in an array, which permits a binary search. Fortunately, in practice most fields of most records consist of simple data types with natural total orders. For example, integers, floats, doubles, and character strings all are totally ordered. Ordering fields that are naturally multi-dimensional, such as a point in two or three dimensions, present special opportunities if we wish to take advantage of their multidimensional nature. This problem is addressed in Section 13.3.

Figure 4.28 shows the definition for a simple abstract dictionary class. The methods **`insert`** and **`find`** are the heart of the class. Method **`insert`** takes a record and inserts it into the dictionary. Method **`find`** takes a key value and returns some record from the dictionary whose key matches the one provided. If there are multiple records in the dictionary with that key value, there is no requirement as to which one is returned.

Method **`clear`** simply re-initializes the dictionary. The **`remove`** method is similar to **`find`**, except that it also deletes the record returned from the dictionary. Once again, if there are multiple records in the dictionary that match the desired key, there is no requirement as to which one actually is removed and returned. Method **`size`** returns the number of elements in the dictionary.

The remaining Method is **`removeAny`**. This is similar to **`remove`**, except that it does not take a key value. Instead, it removes an arbitrary record from the dictionary, if one exists. The purpose of this method is to allow a user the ability to iterate over all elements in the dictionary (of course, the dictionary will become empty in the process). Without the **`removeAny`** method, a dictionary user could not get at a record of the dictionary that he didn't already know the key value for. With the **`removeAny`** method, the user can process all records in the dictionary as shown in the following code fragment.

```
// The Dictionary abstract class.
template <typename Key, typename E>
class  Dictionary {
private:
  void operator =(const Dictionary&) {}
  Dictionary(const Dictionary&) {}

public:
  Dictionary() {}           // Default constructor
  virtual ~Dictionary() {} // Base destructor

  // Reinitialize dictionary
  virtual void clear() = 0;

  // Insert a record
  // k: The key for the record being inserted.
  // e: The record being inserted.
  virtual void insert(const Key& k, const E& e) = 0;

  // Remove and return a record.
  // k: The key of the record to be removed.
  // Return: A maching record. If multiple records match
  // "k", remove an arbitrary one. Return NULL if no record
  // with key "k" exists.
  virtual E remove(const Key& k) = 0;

  // Remove and return an arbitrary record from dictionary.
  // Return: The record removed, or NULL if none exists.
  virtual E removeAny() = 0;

  // Return: A record matching "k" (NULL if none exists).
  // If multiple records match, return an arbitrary one.
  // k: The key of the record to find
  virtual E find(const Key& k) const = 0;

  // Return the number of records in the dictionary.
  virtual int size() = 0;
};
```

Figure 4.28 The ADT for a simple dictionary.

```
// A simple payroll entry with ID, name, address fields
class Payroll {
private:
  int ID;
  string name;
  string address;

public:
  // Constructor
  Payroll(int inID, string inname, string inaddr) {
    ID = inID;
    name = inname;
    address = inaddr;
  }

  ~Payroll() {}  // Destructor

  // Local data member access functions
  int getID() { return ID; }
  string getname() { return name; }
  string getaddr() { return address; }
};
```

Figure 4.29 A payroll record implementation.

```
while (dict.size() > 0) {
  it = dict.removeAny();
  doSomething(it);
}
```

There are other approaches that might seem more natural for iterating though a dictionary, such as using a "first" and a "next" function. But not all data structures that we want to use to implement a dictionary are able to do "first" efficiently. For example, a hash table implementation cannot efficiently locate the record in the table with the smallest key value. By using **`RemoveAny`**, we have a mechanism that provides generic access.

Given a database storing records of a particular type, we might want to search for records in multiple ways. For example, we might want to store payroll records in one dictionary that allows us to search by ID, and also store those same records in a second dictionary that allows us to search by name.

Figure 4.29 shows an implementation for a payroll record. Class **`Payroll`** has multiple fields, each of which might be used as a search key. Simply by varying the type for the key, and using the appropriate field in each record as the key value, we can define a dictionary whose search key is the ID field, another whose search key is the name field, and a third whose search key is the address field. Figure 4.30 shows an example where **`Payroll`** objects are stored in two separate dictionaries, one using the ID field as the key and the other using the name field as the key.

```
int main() {
  // IDdict organizes Payroll records by ID
  UALdict<int, Payroll*> IDdict;
  // namedict organizes Payroll records by name
  UALdict<string, Payroll*> namedict;
  Payroll *foo1, *foo2, *findfoo1, *findfoo2;

  foo1 = new Payroll(5, "Joe", "Anytown");
  foo2 = new Payroll(10, "John", "Mytown");

  IDdict.insert(foo1->getID(), foo1);
  IDdict.insert(foo2->getID(), foo2);
  namedict.insert(foo1->getname(), foo1);
  namedict.insert(foo2->getname(), foo2);

  findfoo1 = IDdict.find(5);
  if (findfoo1 != NULL) cout << findfoo1;
  else cout << "NULL ";
  findfoo2 = namedict.find("John");
  if (findfoo2 != NULL) cout << findfoo2;
  else cout << "NULL ";
}
```

Figure 4.30 A dictionary search example. Here, payroll records are stored in two dictionaries, one organized by ID and the other organized by name. Both dictionaries are implemented with an unsorted array-based list.

The fundamental operation for a dictionary is finding a record that matches a given key. This raises the issue of how to extract the key from a record. We would like any given dictionary implementation to support arbitrary record types, so we need some mechanism for extracting keys that is sufficiently general. One approach is to require all record types to support some particular method that returns the key value. For example, in Java the **`Comparable`** interface can be used to provide this effect. Unfortunately, this approach does not work when the same record type is meant to be stored in multiple dictionaries, each keyed by a different field of the record. This is typical in database applications. Another, more general approach is to supply a class whose job is to extract the key from the record. Unfortunately, this solution also does not work in all situations, because there are record types for which it is not possible to write a key extraction method.[2]

[2]One example of such a situation occurs when we have a collection of records that describe books in a library. One of the fields for such a record might be a list of subject keywords, where the typical record stores a few keywords. Our dictionary might be implemented as a list of records sorted by keyword. If a book contains three keywords, it would appear three times on the list, once for each associated keyword. However, given the record, there is no simple way to determine which keyword on the keyword list triggered this appearance of the record. Thus, we cannot write a function that extracts the key from such a record.

```
// Container for a key-value pair
template <typename Key, typename E>
class KVpair {
private:
  Key k;
  E e;
public:
  // Constructors
  KVpair() {}
  KVpair(Key kval, E eval)
    { k = kval; e = eval; }
  KVpair(const KVpair& o)  // Copy constructor
    { k = o.k; e = o.e; }

  void operator =(const KVpair& o) // Assignment operator
    { k = o.k; e = o.e; }

  // Data member access functions
  Key key() { return k; }
  void setKey(Key ink) { k = ink; }
  E value() { return e; }
};
```

Figure 4.31 Implementation for a class representing a key-value pair.

The fundamental issue is that the key value for a record is not an intrinsic property of the record's class, or of any field within the class. The key for a record is actually a property of the context in which the record is used.

A truly general alternative is to explicitly store the key associated with a given record, as a separate field in the dictionary. That is, each entry in the dictionary will contain both a record and its associated key. Such entries are known as key-value pairs. It is typical that storing the key explicitly duplicates some field in the record. However, keys tend to be much smaller than records, so this additional space overhead will not be great. A simple class for representing key-value pairs is shown in Figure 4.31. The **`insert`** method of the dictionary class supports the key-value pair implementation because it takes two parameters, a record and its associated key for that dictionary.

Now that we have defined the dictionary ADT and settled on the design approach of storing key-value pairs for our dictionary entries, we are ready to consider ways to implement it. Two possibilities would be to use an array-based or linked list. Figure 4.32 shows an implementation for the dictionary using an (unsorted) array-based list.

Examining class **`UALdict`** (UAL stands for "unsorted array-based list), we can easily see that **`insert`** is a constant-time operation, because it simply inserts the new record at the end of the list. However, **`find`**, and **`remove`** both require $\Theta(n)$ time in the average and worst cases, because we need to do a sequential search. Method **`remove`** in particular must touch every record in the list, because once the

```
// Dictionary implemented with an unsorted array-based list
template <typename Key, typename E>
class UALdict : public Dictionary<Key, E> {
private:
  AList<KVpair<Key,E> >* list;
public:
  UALdict(int size=defaultSize)   // Constructor
    { list = new AList<KVpair<Key,E> >(size); }
  ~UALdict() { delete list; }          // Destructor
  void clear() { list->clear(); }      // Reinitialize

  // Insert an element: append to list
  void insert(const Key&k, const E& e) {
    KVpair<Key,E> temp(k, e);
    list->append(temp);
  }

  // Use sequential search to find the element to remove
  E remove(const Key& k) {
    E temp = find(k); // "find" will set list position
    if(temp != NULL) list->remove();
    return temp;
  }

  E removeAny() { // Remove the last element
    Assert(size() != 0, "Dictionary is empty");
    list->moveToEnd();
    list->prev();
    KVpair<Key,E> e = list->remove();
    return e.value();
  }

  // Find "k" using sequential search
  E find(const Key& k) const {
    for(list->moveToStart();
        list->currPos() < list->length(); list->next()) {
      KVpair<Key,E> temp = list->getValue();
      if (k == temp.key())
        return temp.value();
    }
    return NULL; // "k" does not appear in dictionary
  }
```

Figure 4.32 A dictionary implemented with an unsorted array-based list.

```
  int size() // Return list size
    { return list->length(); }
};
```

Figure 4.32 (continued)

```
// Sorted array-based list
// Inherit from AList as a protected base class
template <typename Key, typename E>
class SAList: protected AList<KVpair<Key,E> > {
public:
  SAList(int size=defaultSize) :
    AList<KVpair<Key,E> >(size) {}

  ~SAList() {}                    // Destructor

  // Redefine insert function to keep values sorted
  void insert(KVpair<Key,E>& it) { // Insert at right
    KVpair<Key,E> curr;
    for (moveToStart(); currPos() < length(); next()) {
      curr = getValue();
      if(curr.key() > it.key())
        break;
    }
    AList<KVpair<Key,E> >::insert(it); // Do AList insert
  }

  // With the exception of append, all remaining methods are
  // exposed from AList. Append is not available to SAlist
  // class users since it has not been explicitly exposed.
  AList<KVpair<Key,E> >::clear;
  AList<KVpair<Key,E> >::remove;
  AList<KVpair<Key,E> >::moveToStart;
  AList<KVpair<Key,E> >::moveToEnd;
  AList<KVpair<Key,E> >::prev;
  AList<KVpair<Key,E> >::next;
  AList<KVpair<Key,E> >::length;
  AList<KVpair<Key,E> >::currPos;
  AList<KVpair<Key,E> >::moveToPos;
  AList<KVpair<Key,E> >::getValue;
};
```

Figure 4.33 An implementation for a sorted array-based list.

desired record is found, the remaining records must be shifted down in the list to fill the gap. Method **removeAny** removes the last record from the list, so this is a constant-time operation.

As an alternative, we could implement the dictionary using a linked list. The implementation would be quite similar to that shown in Figure 4.32, and the cost of the functions should be the same asymptotically.

Another alternative would be to implement the dictionary with a sorted list. The advantage of this approach would be that we might be able to speed up the **find** operation by using a binary search. To do so, first we must define a variation on the **List** ADT to support sorted lists.An implementation for the array-based sorted list is shown in Figure 4.33. A sorted list is somewhat different from an unsorted list in that it cannot permit the user to control where elements get inserted. Thus,

the **insert** method must be quite different in a sorted list than in an unsorted list. Likewise, the user cannot be permitted to append elements onto the list. For these reasons, a sorted list cannot be implemented with straightforward inheritance from the **List** ADT.

Class **SAList** (SAL stands for "sorted array-based list") does inherit from class **AList**; however it does so using class **AList** as a protected base class. This means that **SAList** has available for its use any member functions of **AList**, but those member functions are not necessarily available to the user of **SAList**. However, many of the **AList** member functions are useful to the **SALlist** user. Thus, most of the **AList** member functions are passed along directly to the **SAList** user without change. For example, the line

```
AList<KVpair<Key,E> >::remove;
```

provides **SAList**'s clients with access to the **remove** method of **AList**. However, the original **insert** method from class **AList** is replaced, and the **append** method of **AList** is kept hidden.

The dictionary ADT can easily be implemented from class **SAList**, as shown in Figure 4.34. Method **insert** for the dictionary simply calls the **insert** method of the sorted list. Method **find** uses a generalization of the binary search function originally shown in Section 3.5. The cost for **find** in a sorted list is $\Theta(\log n)$ for a list of length n. This is a great improvement over the cost of **find** in an unsorted list. Unfortunately, the cost of **insert** changes from constant time in the unsorted list to $\Theta(n)$ time in the sorted list. Whether the sorted list implementation for the dictionary ADT is more or less efficient than the unsorted list implementation depends on the relative number of **insert** and **find** operations to be performed. If many more **find** operations than **insert** operations are used, then it might be worth using a sorted list to implement the dictionary. In both cases, **remove** requires $\Theta(n)$ time in the worst and average cases. Even if we used binary search to cut down on the time to find the record prior to removal, we would still need to shift down the remaining records in the list to fill the gap left by the **remove** operation.

Given two keys, we have not properly addressed the issue of how to compare them. One possibility would be to simply use the basic **==**, **<=**, and **>=** operators built into **C**++. This is the approach taken by our implementations for dictionaries shown inFigures 4.32 and 4.34. If the key type is **int**, for example, this will work fine. However, if the key is a pointer to a string or any other type of object, then this will not give the desired result. When we compare two strings we probably want to know which comes first in alphabetical order, but what we will get from the standard comparison operators is simply which object appears first in memory. Unfortunately, the code will compile fine, but the answers probably will not be fine.

In a language like **C**++ that supports operator overloading, we could require that the user of the dictionary overload the **==**, **<=**, and **>=** operators for the given

```
// Dictionary implemented with a sorted array-based list
template <typename Key, typename E>
class SALdict : public Dictionary<Key, E> {
private:
  SAList<Key,E>* list;
public:
  SALdict(int size=defaultSize)   // Constructor
    { list = new SAList<Key,E>(size); }
  ~SALdict() { delete list; }         // Destructor
  void clear() { list->clear(); }     // Reinitialize

  // Insert an element: Keep elements sorted
  void insert(const Key&k, const E& e) {
    KVpair<Key,E> temp(k, e);
    list->insert(temp);
  }

  // Use sequential search to find the element to remove
  E remove(const Key& k) {
    E temp = find(k);
    if (temp != NULL) list->remove();
    return temp;
  }

  E removeAny() { // Remove the last element
    Assert(size() != 0, "Dictionary is empty");
    list->moveToEnd();
    list->prev();
    KVpair<Key,E> e = list->remove();
    return e.value();
  }

  // Find "K" using binary search
  E find(const Key& k) const {
    int l = -1;
    int r = list->length();
    while (l+1 != r) { // Stop when l and r meet
      int i = (l+r)/2; // Check middle of remaining subarray
      list->moveToPos(i);
      KVpair<Key,E> temp = list->getValue();
      if (k < temp.key()) r = i;        // In left
      if (k == temp.key()) return temp.value(); // Found it
      if (k > temp.key()) l = i;        // In right
    }
    return NULL; // "k" does not appear in dictionary
  }
```

Figure 4.34 Dictionary implementation using a sorted array-based list.

```
  int size() // Return list size
    { return list->length(); }
};
```

Figure 4.34 (continued)

key type. This requirement then becomes an obligation on the user of the dictionary class. Unfortunately, this obligation is hidden within the code of the dictionary (and possibly in the user's manual) rather than exposed in the dictionary's interface. As a result, some users of the dictionary might neglect to implement the overloading, with unexpected results. Again, the compiler will not catch this problem.

The most general solution is to have users supply their own definition for comparing keys. The concept of a class that does comparison (called a **comparator**) is quite important. By making these operations be template parameters, the requirement to supply the comparator class becomes part of the interface. This design is an example of the Strategy design pattern, because the "strategies" for comparing and getting keys from records are provided by the client. In some cases, it makes sense for the comparator class to extract the key from the record type, as an alternative to storing key-value pairs.

Here is an example of the required class for comparing two integers.

```
class intintCompare { // Comparator class for integer keys
public:
  static bool lt(int x, int y) { return x < y; }
  static bool eq(int x, int y) { return x == y; }
  static bool gt(int x, int y) { return x > y; }
};
```

Class **intintCompare** provides methods for determining if two **int** variables are equal (**eq**), or if the first is less than the second (**lt**), or greater than the second (**gt**).

Here is a class for comparing two C-style character strings. It makes use of the standard library function **strcmp** to do the actual comparison.

```
class CCCompare { // Compare two character strings
public:
  static bool lt(char* x, char* y)
    { return strcmp(x, y) < 0; }
  static bool eq(char* x, char* y)
    { return strcmp(x, y) == 0; }
  static bool gt(char* x, char* y)
    { return strcmp(x, y) > 0; }
};
```

We will usea comparator in Section 5.5 to implement comparison in heaps, and in Chapter 7 to implement comparison in sorting algorithms.

4.5 Further Reading

For more discussion on choice of functions used to define the **`List`** ADT, see the work of the Reusable Software Research Group from Ohio State. Their definition for the **`List`** ADT can be found in [SWH93]. More information about designing such classes can be found in [SW94].

4.6 Exercises

4.1 Assume a list has the following configuration:

$$\langle \mid 2,\ 23,\ 15,\ 5,\ 9 \rangle.$$

Write a series of C++ statements using the **`List`** ADT of Figure 4.1 to delete the element with value 15.

4.2 Show the list configuration resulting from each series of list operations using the **`List`** ADT of Figure 4.1. Assume that lists **`L1`** and **`L2`** are empty at the beginning of each series. Show where the current position is in the list.

(a)
```
L1.append(10);
L1.append(20);
L1.append(15);
```

(b)
```
L2.append(10);
L2.append(20);
L2.append(15);
L2.moveToStart();
L2.insert(39);
L2.next();
L2.insert(12);
```

4.3 Write a series of C++ statements that uses the **`List`** ADT of Figure 4.1 to create a list capable of holding twenty elements and which actually stores the list with the following configuration:

$$\langle\ 2,\ 23 \mid 15,\ 5,\ 9\ \rangle.$$

4.4 Using the list ADT of Figure 4.1, write a function to interchange the current element and the one following it.

4.5 In the linked list implementation presented in Section 4.1.2, the current position is implemented using a pointer to the element ahead of the logical current node. The more "natural" approach might seem to be to have **`curr`** point directly to the node containing the current element. However, if this was done, then the pointer of the node preceding the current one cannot be

updated properly because there is no access to this node from **curr**. An alternative is to add a new node *after* the current element, copy the value of the current element to this new node, and then insert the new value into the old current node.

(a) What happens if **curr** is at the end of the list already? Is there still a way to make this work? Is the resulting code simpler or more complex than the implementation of Section 4.1.2?

(b) Will deletion always work in constant time if **curr** points directly to the current node? In particular, can you make several deletions in a row?

4.6 Add to the **LList** class implementation a member function to reverse the order of the elements on the list. Your algorithm should run in $\Theta(n)$ time for a list of n elements.

4.7 Write a function to merge two linked lists. The input lists have their elements in sorted order, from lowest to highest. The output list should also be sorted from lowest to highest. Your algorithm should run in linear time on the length of the output list.

4.8 A **circular linked list** is one in which the **next** field for the last link node of the list points to the first link node of the list. This can be useful when you wish to have a relative positioning for elements, but no concept of an absolute first or last position.

(a) Modify the code of Figure 4.8 to implement circular singly linked lists.

(b) Modify the code of Figure 4.14 to implement circular doubly linked lists.

4.9 Section 4.1.3 states "the space required by the array-based list implementation is $\Omega(n)$, but can be greater." Explain why this is so.

4.10 Section 4.1.3 presents an equation for determining the break-even point for the space requirements of two implementations of lists. The variables are D, E, P, and n. What are the dimensional units for each variable? Show that both sides of the equation balance in terms of their dimensional units.

4.11 Use the space equation of Section 4.1.3 to determine the break-even point for an array-based list and linked list implementation for lists when the sizes for the data field, a pointer, and the array-based list's array are as specified. State when the linked list needs less space than the array.

(a) The data field is eight bytes, a pointer is four bytes, and the array holds twenty elements.

(b) The data field is two bytes, a pointer is four bytes, and the array holds thirty elements.

(c) The data field is one byte, a pointer is four bytes, and the array holds thirty elements.

(d) The data field is 32 bytes, a pointer is four bytes, and the array holds forty elements.

4.12 Determine the size of an **int** variable, a **double** variable, and a pointer on your computer. (The **C**++ operator **sizeof** might be useful here if you do not already know the answer.)

(a) Calculate the break-even point, as a function of n, beyond which the array-based list is more space efficient than the linked list for lists whose elements are of type **int**.

(b) Calculate the break-even point, as a function of n, beyond which the array-based list is more space efficient than the linked list for lists whose elements are of type **double**.

4.13 Modify the code of Figure 4.18 to implement two stacks sharing the same array, as shown in Figure 4.20.

4.14 Modify the array-based queue definition of Figure 4.26 to use a separate Boolean member to keep track of whether the queue is empty, rather than require that one array position remain empty.

4.15 A **palindrome** is a string that reads the same forwards as backwards. Using only a fixed number of stacks and queues, the stack and queue ADT functions, and a fixed number of **int** and **char** variables, write an algorithm to determine if a string is a palindrome. Assume that the string is read from standard input one character at a time. The algorithm should output **true** or **false** as appropriate.

4.16 Re-implement function **fibr** from Exercise 2.11, using a stack to replace the recursive call as described in Section 4.2.4.

4.17 Write a recursive algorithm to compute the value of the recurrence relation

$$\mathbf{T}(n) = \mathbf{T}(\lceil n/2 \rceil) + \mathbf{T}(\lfloor n/2 \rfloor) + n; \quad \mathbf{T}(1) = 1.$$

Then, rewrite your algorithm to simulate the recursive calls with a stack.

4.18 Let Q be a non-empty queue, and let S be an empty stack. Using only the stack and queue ADT functions and a single element variable X, write an algorithm to reverse the order of the elements in Q.

4.19 A common problem for compilers and text editors is to determine if the parentheses (or other brackets) in a string are balanced and properly nested. For example, the string "((())())()" contains properly nested pairs of parentheses, but the string ")()(" does not, and the string "())" does not contain properly matching parentheses.

(a) Give an algorithm that returns **true** if a string contains properly nested and balanced parentheses, and **false** otherwise. Use a stack to keep track of the number of left parentheses seen so far. *Hint*: At no time while scanning a legal string from left to right will you have encountered more right parentheses than left parentheses.

(b) Give an algorithm that returns the position in the string of the first offending parenthesis if the string is not properly nested and balanced. That is, if an excess right parenthesis is found, return its position; if there are too many left parentheses, return the position of the first excess left parenthesis. Return -1 if the string is properly balanced and nested. Use a stack to keep track of the number and positions of left parentheses seen so far.

4.20 Imagine that you are designing an application where you need to perform the operations **`Insert`**, **`Delete_Maximum`**, and **`Delete_Minimum`**. For this application, the cost of inserting is not important, because it can be done off-line prior to startup of the time-critical section, but the performance of the two deletion operations are critical. Repeated deletions of either kind must work as fast as possible. Suggest a data structure that can support this application, and justify your suggestion. What is the time complexity for each of the three key operations?

4.21 Write a function that reverses the order of an array of n items.

4.7 Projects

4.1 A **deque** (pronounced "deck") is like a queue, except that items may be added and removed from both the front and the rear. Write either an array-based or linked implementation for the deque.

4.2 One solution to the problem of running out of space for an array-based list implementation is to replace the array with a larger array whenever the original array overflows. A good rule that leads to an implementation that is both space and time efficient is to double the current size of the array when there is an overflow. Re-implement the array-based **`List`** class of Figure 4.2 to support this array-doubling rule.

4.3 Use singly linked lists to implement integers of unlimited size. Each node of the list should store one digit of the integer. You should implement addition, subtraction, multiplication, and exponentiation operations. Limit exponents to be positive integers. What is the asymptotic running time for each of your operations, expressed in terms of the number of digits for the two operands of each function?

4.4 Implement doubly linked lists by storing the sum of the **`next`** and **`prev`** pointers in a single pointer variable as described in Example 4.1.

4.5 Implement a city database using unordered lists. Each database record contains the name of the city (a string of arbitrary length) and the coordinates of the city expressed as integer x and y coordinates. Your database should allow records to be inserted, deleted by name or coordinate, and searched by name or coordinate. Another operation that should be supported is to

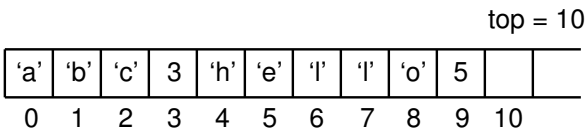

Figure 4.35 An array-based stack storing variable-length strings. Each position stores either one character or the length of the string immediately to the left of it in the stack.

print all records within a given distance of a specified point. Implement the database using an array-based list implementation, and then a linked list implementation. Collect running time statistics for each operation in both implementations. What are your conclusions about the relative advantages and disadvantages of the two implementations? Would storing records on the list in alphabetical order by city name speed any of the operations? Would keeping the list in alphabetical order slow any of the operations?

4.6 Modify the code of Figure 4.18 to support storing variable-length strings of at most 255 characters. The stack array should have type **`char`**. A string is represented by a series of characters (one character per stack element), with the length of the string stored in the stack element immediately above the string itself, as illustrated by Figure 4.35. The **`push`** operation would store an element requiring i storage units in the i positions beginning with the current value of **`top`** and store the size in the position i storage units above **`top`**. The value of **`top`** would then be reset above the newly inserted element. The **`pop`** operation need only look at the size value stored in position `top` -1 and then pop off the appropriate number of units. You may store the string on the stack in reverse order if you prefer, provided that when it is popped from the stack, it is returned in its proper order.

4.7 Define an ADT for a bag (see Section 2.1) and create an array-based implementation for bags. Be sure that your bag ADT does not rely in any way on knowing or controlling the position of an element. Then, implement the dictionary ADT of Figure 4.28 using your bag implementation.

4.8 Implement the dictionary ADT of Figure 4.28 using an unsorted linked list as defined by class **`LList`** in Figure 4.8. Make the implementation as efficient as you can, given the restriction that your implementation must use the unsorted linked list and its access operations to implement the dictionary. State the asymptotic time requirements for each function member of the dictionary ADT under your implementation.

4.9 Implement the dictionary ADT of Figure 4.28 based on stacks. Your implementation should declare and use two stacks.

4.10 Implement the dictionary ADT of Figure 4.28 based on queues. Your implementation should declare and use two queues.