Data Structures and Algorithm Analysis

Edition 3.2 (C++ Version)

Clifford A. Shaffer
Department of Computer Science
Virginia Tech
Blacksburg, VA 24061

March 28, 2013
Update 3.2.0.10
For a list of changes, see
`http://people.cs.vt.edu/~shaffer/Book/errata.html`

```
// 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

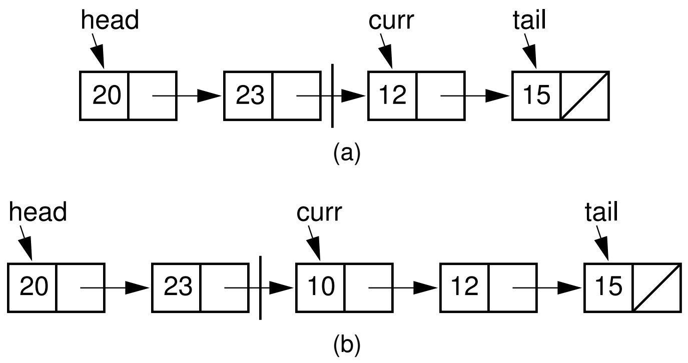

Figure 4.5 Illustration of a faulty linked-list implementation where **`curr`** points directly to the current node. (a) Linked list prior to inserting element with value 10. (b) Desired effect of inserting element with value 10.

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

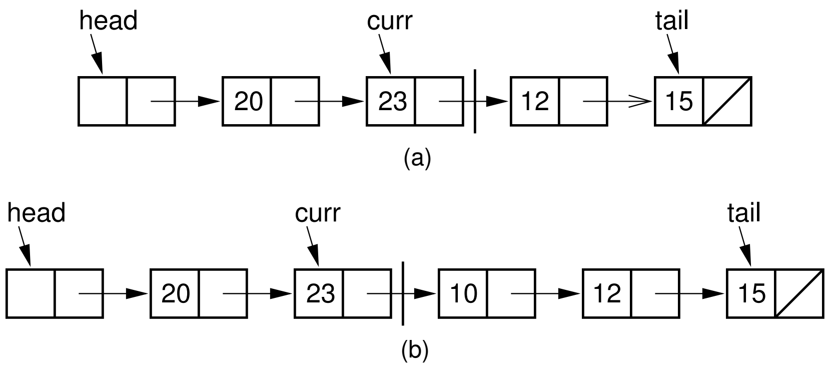

Figure 4.6 Insertion using a header node, with **curr** pointing one node head of the current element. (a) Linked list before insertion. The current node contains 12. (b) Linked list after inserting the node containing 10.

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.

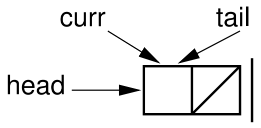

Figure 4.7 Initial state of a linked list when using a header node.

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)

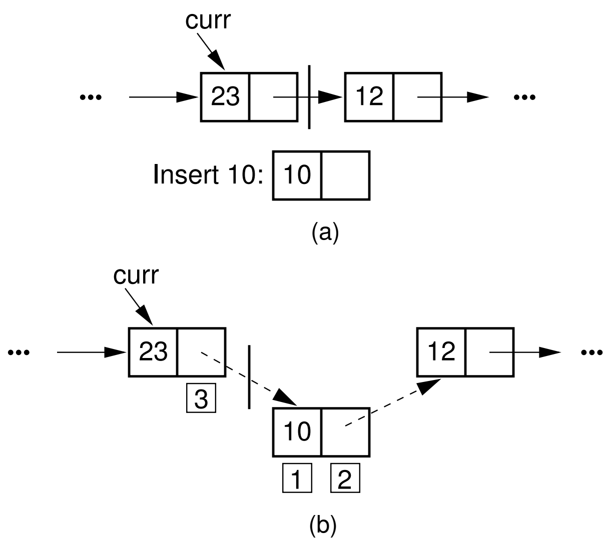

Figure 4.9 The linked list insertion process. (a) The linked list before insertion. (b) The linked list after insertion. $\boxed{1}$ marks the **`element`** field of the new link node. $\boxed{2}$ marks the **`next`** field of the new link node, which is set to point to what used to be the current node (the node with value 12). $\boxed{3}$ 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.

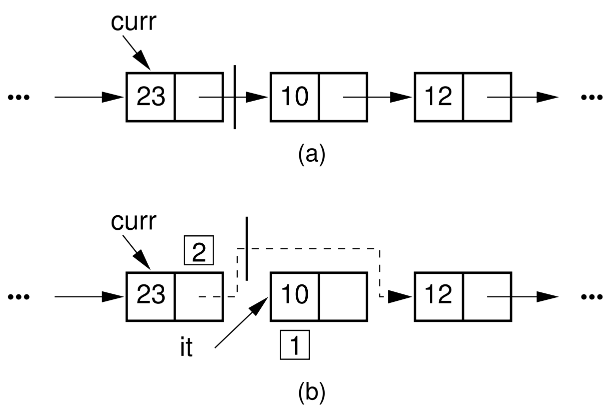

Figure 4.10 The linked list removal process. (a) The linked list before removing the node with value 10. (b) The linked list after removal. $\boxed{1}$ marks the list node being removed. **`it`** is set to point to the element. $\boxed{2}$ marks the **`next`** field of the preceding list node, which 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

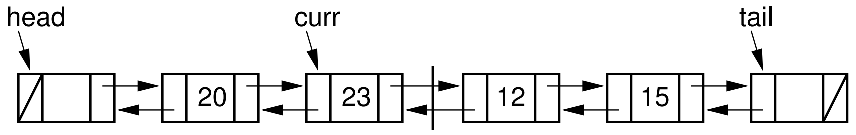

Figure 4.12 A doubly linked list.

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.

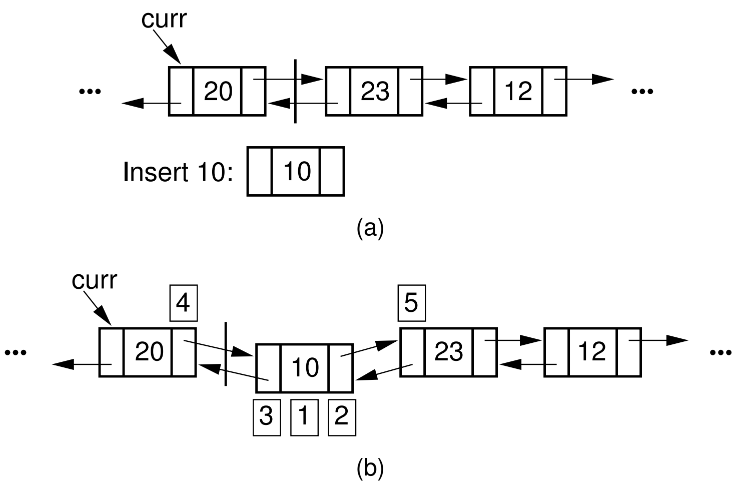

Figure 4.15 Insertion for doubly linked lists. The labels $\boxed{1}$, $\boxed{2}$, and $\boxed{3}$ correspond to assignments done by the linked list node constructor. $\boxed{4}$ marks the assignment to **`curr->next`**. $\boxed{5}$ marks the assignment to the **`prev`** pointer of the node following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

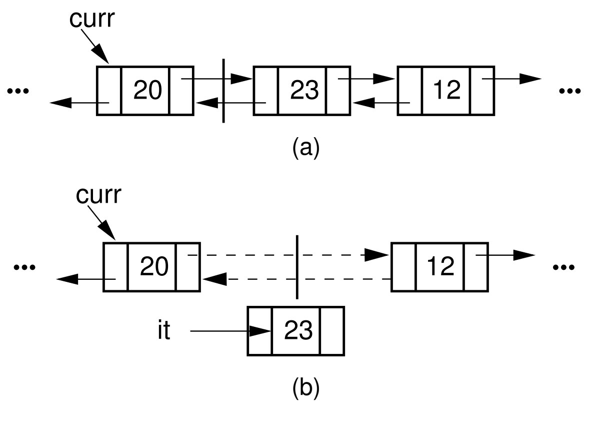

Figure 4.16 Doubly linked list removal. Element **`it`** stores the element of the node being removed. Then the nodes to either side have their pointers adjusted.

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-