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

If you wish to have a printed version of this document, print copies are published by Dover Publications
(see http://store.doverpublications.com/048648582x.html).
Further information about this text is available at
http://people.cs.vt.edu/~shaffer/Book/.

```
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)

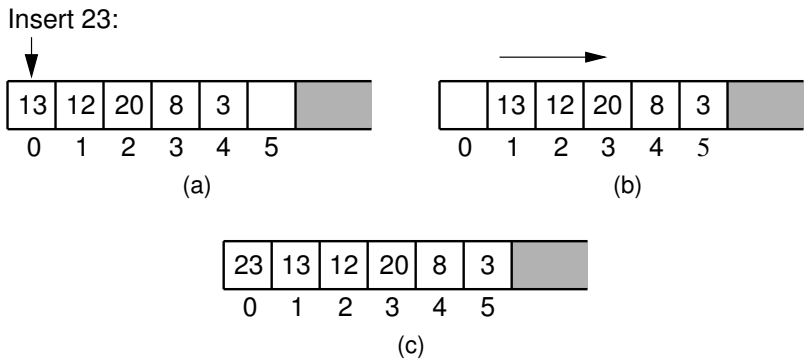

Figure 4.3 Inserting an element at the head of an array-based list requires shifting all existing elements in the array by one position toward the tail. (a) A list containing five elements before inserting an element with value 23. (b) The list after shifting all existing elements one position to the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