After studying this chapter, you should be able to:

- Describe the Abstract Data Type Unsorted List from three perspectives
- Use the Unsorted List operations to implement utility routines to do the following application-level tasks:
  - Print the list of elements
  - Create a list of elements from a file
- Implement the following Unsorted List operations using an array-based implementation:
  - Create and destroy a list
  - Determine whether the list is full
  - Insert an element
  - Retrieve an element
  - Delete an element
- Write and execute a test plan for an abstract data type
- Declare variables of pointer types
- Access the variables to which pointers point
- Implement the list operations outlined above using a linked implementation
- Compare the two implementations of the ADT Unsorted List in terms of Big-O approximations
Chapter 3: ADT Unsorted List

In Chapter 2, we defined an abstract data type and showed how all data can be viewed from three perspectives: from the logical perspective, the implementation perspective, and the application perspective. The logical perspective is the abstract view of what the ADT does. The implementation perspective offers a picture of how the logical operations are carried out. The application perspective shows why the ADT behaves as it does—that is, how the behavior can be useful in a real-world problem.

In this chapter, we look at an ADT that should be familiar to all of us: the list. We all know intuitively what a “list” is; in our everyday lives we use lists constantly—grocery lists, lists of things to do, lists of addresses, lists of party guests. Lists are places where we write down things that we want to remember.

3.1 Lists

In computer programs, lists are very useful abstract data types. They are members of a general category of abstract data types called containers, whose purpose is to hold other objects. In some languages, the list is a built-in structure. In Lisp, for example, the list is the main data structure provided in the language. In C++, while lists are provided in the Standard Template Library, the techniques for building lists and other abstract data types are so important that we show you how to design and write your own.

From a theoretical point of view, a list is a homogeneous collection of elements, with a linear relationship between elements. Linear means that, at the logical level, each element in the list except the first one has a unique predecessor, and each element except the last one has a unique successor. (At the implementation level, a relationship also exists between the elements, but the physical relationship may not be the same as the logical one.) The number of items in the list, which we call the length of the list, is a property of a list. That is, every list has a length.

Lists can be unsorted—their elements may be placed into the list in no particular order—or they can be sorted in a variety of ways. For instance, a list of numbers can be sorted by value, a list of strings can be sorted alphabetically, and a list of grades can be sorted numerically. When the elements in a sorted list are of composite types, their logical (and often physical) order is determined by one of the members of the structure, called the key member. For example, a list of students on the honor roll can be sorted alphabetically by name or numerically by student identification number. In the first case, the name is the key; in the second case, the identification number is the key. Such sorted lists are also called key-sorted lists.

If a list cannot contain items with duplicate keys, it is said to have unique keys. This chapter deals with unsorted lists. In Chapter 4, we examine lists of elements with unique keys, sorted from smallest to largest key value.
3.2 Abstract Data Type Unsorted List

Logical Level

Programmers can provide many different operations for lists. For different applications we can imagine all kinds of things users might need to do to a list of elements. In this chapter we formally define a list and develop a set of general-purpose operations for creating and manipulating lists. By doing so, we build an abstract data type.

In the next section we design the specifications for a List ADT where the items in the list are unsorted; that is, no semantic relationship exists between an item and its predecessor or successor. Items simply appear next to one another in the list.

Abstract Data Type Operations

The first step in designing any abstract data type is to stand back and consider what a user of the data type would want it to provide. Recall that there are four kinds of operations: constructors, transformers, observers, and iterators. We begin by reviewing each type and consider each kind of operation with respect to the List ADT. We use a handwriting font for operation names at the logical level and change to a monospaced font when we refer to specific implementation.

Constructors  A constructor creates an instance of the data type. It is usually implemented with a language-level declaration.

Transformers (also called mutators)  Transformers are operations that change the structure in some way: They may make the structure empty, put an item into the structure, or remove a specific item from the structure. For our Unsorted List ADT, let’s call these transformers MakeEmpty, InsertItem, and DeleteItem.

MakeEmpty  needs only the list, no other parameters. As we implement our operations as member functions, the list is the object to which the function is applied. InsertItem and DeleteItem need an additional parameter: the item to be inserted or removed. For this Unsorted List ADT, let’s assume that the item to be inserted is not currently in the list and the item to be deleted is in the list.

A transformer that takes two lists and creates a third list by appending one list to the other would be a binary transformer.

Observers  Observers come in several forms. They ask true/false questions about the data type (Is the structure empty?), select or access a particular item (Give me a copy of the last item.), or return a property of the structure (How many items are in the structure?). The Unsorted List ADT needs at least two observers: IsFull and GetLength.

IsFull  returns true if the list is full; GetLength  tells us how many items appear in the list. Another useful observer searches the list for an item with a particular key and returns a copy of the associated information if it is found; let’s call it RetrieveItem.
Chapter 3: ADT Unsorted List

If an abstract data type places limits on the component type, we could define other observers. For example, if we know that our abstract data type is a list of numerical values, we could define statistical observers such as Minimum, Maximum, and Average. Here, we are interested in generality; we know nothing about the type of the items on the list, so we use only general observers in our ADT.

In most of our discussions of error checking to date, we have put the responsibility of checking for error conditions on the user through the use of preconditions that prohibit the operation’s call if these error conditions exist. In making the client responsible for checking for error conditions, however, we must make sure that the ADT gives the user the tools with which to check for the conditions. In another approach, we could keep an error variable in our list, have each operation record whether an error occurs, and provide operations that test this variable. The operations that check whether an error has occurred would be observers. However, in the Unsorted List ADT we are specifying, let’s have the user prevent error conditions by obeying the preconditions of the ADT operations.

Iterators

Iterators are used with composite types to allow the user to process an entire structure, component by component. To give the user access to each item in sequence, we provide two operations: one to initialize the iteration process (analogous to Reset or Open with a file) and one to return a copy of the “next component” each time it is called. The user can then set up a loop that processes each component. Let’s call these operations ResetList and GetNextItem. Note that ResetList is not an iterator itself, but rather an auxiliary operation that supports the iteration. Another type of iterator takes an operation and applies it to every element in the list.

C++ Declarations and Definitions

In general programming terminology, a declaration associates an identifier with a data object, an action (such as a function), or a data type. C++ terminology distinguishes between a declaration and a definition. A declaration becomes a definition when it binds storage to the identifier. Hence, all definitions are declarations, but not all declarations are definitions. For example, a function prototype is a declaration, but a function heading with a body is a function definition. On the other hand, declarations such as typedef can never be definitions, because they are not bound to storage. Because of the way that C++ treats classes, their specification is also a definition. Because the ISO/ANSI C++ standard uses the term “definition” rather than “declaration” when referring to a class, we do the same here.

Generic Data Types

A generic data type is one for which the operations are defined but the types of the items being manipulated are not. Some programming languages have a built-in mechanism for defining generic data types; others lack this feature. Although C++ does have such a mechanism (called a template), we postpone its description until Chapter 6. Here we present a simple, general-purpose way of simulating
generics that works in any programming language. We let the user define the type of the items on the list in a class named `ItemType` and have our Unsorted List ADT include the class definition.

Two of the list operations (DeleteItem and RetrieveItem) will involve the comparison of the keys of two list components (as does InsertItem if the list is sorted by key value). We could require the user to name the key data member “key” and compare the key data members using the C++ relational operators. However, this approach isn’t a very satisfactory solution for two reasons: “key” is not always a meaningful identifier in an application program, and the keys would be limited to values of simple types. C++ does have a way to change the meaning of the relational operators (called overloading them), but for now we present a general solution rather than a language-dependent one.

We let the user define a member function `ComparedTo` in the class `ItemType`. This function compares two items and returns LESS, GREATER, or EQUAL depending on whether the key of one item comes before the key of the other item, the first key comes after it, or the keys of the two items are equal, respectively. If the keys are of a simple type such as an identification number, `ComparedTo` would be implemented using the relational operators. If the keys are strings, function `ComparedTo` would use the string-comparison operators supplied in `<string>`. If the keys are people’s names, both the last name and the first name would be compared. Therefore, our specification assumes that `ComparedTo` is a member of `ItemType`.

Our ADT needs one more piece of information from the client: the maximum number of items on the list. As this information varies from application to application, it is logical for the client to provide it.

Let’s summarize our observations in two CRC cards: one for `ItemType` and the other for `UnsortedType`. Note that `UnsortedType` collaborates with `ItemType`.

<table>
<thead>
<tr>
<th>Class Name:</th>
<th>Superclass:</th>
<th>Subclasses:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ItemType</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Responsibilities**

- Provide
- `MAX_ITEMS`
- `enum RelationType (LESS, GREATER, EQUAL)`
- `ComparedTo (item) returns RelationType`
Now we can formalize the specification for the Unsorted List ADT.

**Unsorted List ADT Specification**

**Structure:**

The list elements are of ItemType. The list has a special property called the current position—the position of the last element accessed by GetNextItem during an iteration through the list. Only ResetList and GetNextItem affect the current position.

**Definitions and Operation (provided by user in class ItemType):**

- **MAX_ITEMS:** A constant specifying the maximum number of items on the list.
- **RelationType:** An enumeration type that consists of LESS, GREATER, EQUAL.

**Member function of ItemType that must be included:**

- **RelationType ComparedTo(ItemType item)**
  - **Function:** Determines the ordering of two ItemType objects based on their keys.
  - **Precondition:** Self and item have their key members initialized.
Postcondition:
Function value = LESS if the key of self is less than the key of item.
  = GREATER if the key of self is greater than the key of item.
  = EQUAL if the keys are equal.

Operations (provided by Unsorted List ADT):

MakeEmpty
Function: Initializes list to empty state.
Preconditions: None.
Postcondition: List is empty.

Boolean IsFull
Function: Determines whether list is full.
Precondition: List has been initialized.
Postcondition: Function value = (list is full).

int GetLength
Function: Determines the number of elements in list.
Precondition: List has been initialized.
Postcondition: Function value = number of elements in list.

RetrieveItem (ItemType& item, Boolean& found)
Function: Retrieves list element whose key matches item’s key (if present).
Preconditions: List has been initialized.
Key member of item is initialized.
Postconditions: If there is an element someItem whose key matches item’s key, then found = true and item is a copy of someItem; otherwise found = false and item is unchanged.
List is unchanged.

InsertItem (ItemType item)
Function: Adds item to list.
Preconditions: List has been initialized.
List is not full.
item is not in list.
Postcondition: item is in list.
DeleteItem (ItemType item)

Function: Deletes the element whose key matches item’s key.

Preconditions: List has been initialized.
Key member of item is initialized.
One and only one element in list has a key matching item’s key.

Postcondition: No element in list has a key matching item’s key.

ResetList

Function: Initializes current position for an iteration through the list.

Precondition: List has been initialized.

Postcondition: Current position is prior to first element in list.

GetNextItem (ItemType & item)

Function: Gets the next element in list.

Preconditions: List has been initialized.
Current position is defined.
Element at current position is not last in list.

Postconditions: Current position is updated to next position.
item is a copy of element at current position.

Because we do not know the makeup of the key member in the ItemType, we must pass an entire object of ItemType as the parameter to both RetrieveItem and DeleteItem. Notice that the preconditions for both operations state that the key member of the parameter item is initialized. RetrieveItem fills in the rest of the members of item if a list component with the same key is found, and DeleteItem removes from the list the component whose key matches that of item.

The specifications of the operations are somewhat arbitrary. For instance, we specified in the preconditions of DeleteItem that the element to delete must exist in the list and must be unique. We could also specify an operation that does not require the element to be in the list and leaves the list unchanged if the item is not present. This decision is a design choice. If we were designing a specification for a specific application, then the design choice would be based on the requirements of the problem. In this case, we made an arbitrary decision. In the exercises, you are asked to examine the effects of different design choices.

The operations defined in this specification are a sufficient set to create and maintain an unsorted list of elements. Notice that no operation depends on the type of the items in the structure. This data independence makes the Unsorted List ADT truly abstract. Each program that uses the Unsorted List ADT defines ItemType within the
context of the application and provides a comparison member function defined on two
items of type ItemType.

**Application Level**

The set of operations provided for the Unsorted List ADT may seem rather small and
primitive. In fact, this set of operations gives you the tools to create other special-pur-
pose routines that require a knowledge of ItemType. For instance, we have not
included a print operation. Why? Because to write a print routine, we must know what
the data members look like. The user (who does know what the data members look like)
can use the GetLength, ResetList, and GetNextItem operations to iterate through
the list, printing each data member in turn. In the code that follows, we assume that the
user has defined a member function for ItemType that prints the data members of one
item. We also assume that the Unsorted List ADT is itself implemented as a class with
the operations as member functions.

```cpp
void PrintList(std::ofstream& dataFile, UnsortedType list)
// Pre:  list has been initialized.
//       dataFile is open for writing.
// Post: Each component in list has been written to dataFile.
//       dataFile is still open.
{
  int length;
  ItemType item;
  list.ResetList();
  length = list.GetLength();
  for (int counter = 1; counter <= length; counter++)
  {
    list.GetNextItem(item);
    item.Print(dataFile);
  }
}
```

Note that we defined a local variable length, stored the result of
list.GetLength() in it, and used the local variable in the loop. We did so for effi-
ciency reasons: The function is called only once, saving the overhead of extra function
calls.

Another operation that depends on the application reads data (of type ItemType)
from a file and creates a list containing these elements. Without knowing how the list is
implemented, the user can write a function CreateListFromFile, using the operations
specified in the Unsorted List ADT. We assume a function GetData, which accesses the
individual data members from the file and returns them in item.

```cpp
void CreateListFromFile(std::ifstream& dataFile, UnsortedType& list)
// Pre:  dataFile exists and is open.
```
Chapter 3: ADT Unsorted List

// Post: list contains items from dataFile.
//       dataFile is in the fail state due to end-of-file.
//       Items read after the list becomes full are discarded.
{
  ItemType item;
  list.MakeEmpty();
  GetData(dataFile, item); // Reads one item from dataFile.
  while (dataFile)
  {
    if (!list.IsFull())
      list.InsertItem(item);
    GetData(dataFile, item);
  }
}

In these two functions we have made calls to the list operations specified for the Unsorted List ADT, creating and printing a list without knowing how the list is implemented. At an application level, these tasks are logical operations on a list. At a lower level, these operations are implemented as C++ functions that manipulate an array or other data-storing medium holding the list's elements. Multiple functionally correct ways are available to implement an abstract data type. Between the user picture and the eventual representation in the computer's memory, intermediate levels of abstraction and design decisions are possible. For instance, how is the logical order of the list elements reflected in their physical ordering? We address questions like this as we now turn to the implementation level of our ADT.

Implementation Level

The logical order of the list elements may or may not mirror the way that we actually store the data. If we implement a list in an array, the components are arranged so that the predecessor and the successor of a component are physically before and after it. Later in this chapter, we introduce a way of implementing a list in which the components are sorted logically rather than physically. However, the way that the list elements are physically arranged certainly affects the way that we access the elements of the list. This arrangement may have implications for how efficient the list operations are. For instance, there is nothing in the specification of the Unsorted List ADT that requires us to implement the list with the elements stored in random order. If we stored the elements in an array, completely sorted, we could still implement all the Unsorted List operations. Does it make a difference if the items are stored unsorted or sorted? We answer the question in Chapter 4.

There are two ways to implement a list that preserves the order of the list items; that is, that stores the elements physically in such a way that, from one list element, we can access its logical successor directly. We look at both in this chapter. The first is a
3.2 Abstract Data Type Unsorted List

sequential array-based list representation. The distinguishing feature of this implementation is that the elements are stored sequentially, in adjacent slots in an array. The order of the elements is implicit in their placement in the array.

The second approach is a linked-list representation. In a linked implementation, the data elements are not constrained to be stored in physically contiguous, sequential order; rather, the individual elements are stored “somewhere in memory,” and their order is maintained by explicit links between them. Figure 3.1 shows the difference between an array and a linked list.

Before we go on, let’s establish a design terminology that we can use in our algorithms, independent of the eventual list implementation.

**List Design Terminology**

Assuming that location “accesses” a particular list element,

- **Node(location)** refers to all data at location, including implementation-specific data.
- **Info(location)** refers to the user’s data at location.
- **Info(last)** refers to the user’s data at the last location in the list.
- **Next(location)** gives the location of the node following Node(location).

What, then, is location? For an array-based implementation, location is an index, because we access array slots through their indexes. For example, the design statement

```c++
Print element Info(location)
```

means “Print the user’s data in the array slot at index location”; eventually it might be coded in C++ as

```c++
list.info[location].Print(dataFile);
```

When we look at a linked implementation later in the chapter, the translation is quite different but the algorithms remain the same. That is, our code implementing the operations changes, but the algorithms do not. Thus, using this design notation, we define implementation-independent algorithms for our Unsorted List ADT.
But what does Next(location) mean in an array-based sequential implementation? To answer this question, consider how we access the next list element stored in an array: We increment the location, which is the index. The design statement

\[\text{Set location to Next(location)}\]

might, therefore, be coded in C++ as

```cpp
location++; // location is an array index.
```

We have not introduced this list design terminology just to force you to learn the syntax of another computer “language.” Rather, we want to encourage you to think of the list, and the parts of the list elements, as abstractions. We have intentionally made the design notation similar to the syntax of function calls to emphasize that, at the design stage, the implementation details can be hidden. A lower level of detail is encapsulated in the “functions” Node, Info, and Next. Using this design terminology, we hope to record algorithms that can be coded for both array-based and linked implementations.

**Data Structure** In our implementation, the elements of a list are stored in an array of class objects.

```cpp
ItemType info[MAX_ITEMS];
```

We need a length data member to keep track of both the number of items we have stored in the array and the location where the last item was stored. Because the list items are unsorted, we place the first item put into the list into the first slot, the second item into the second slot, and so forth. Because our language is C++, we must remember that the first slot is indexed by 0, the second slot by 1, and the last slot by MAX_ITEMS - 1. Now we know where the list begins—in the first array slot. Where does the list end? The array ends at the slot with index MAX_ITEMS - 1, but the list ends in the slot with index length - 1.

Is there any other information about the list that we must include? Both operations ResetList and GetNextItem refer to a “current position.” What is this current position? It is the index of the last element accessed in an iteration through the list. Let’s call it currentPos. ResetList initializes currentPos to -1. GetNextItem increments currentPos and returns the value in info[currentPos]. The ADT specification states that only ResetList and GetNextItem affect the current position. Figure 3.2 illustrates the data members of our class UnsortedType.

```cpp
#include "ItemType.h"
```

// File ItemType.h must be provided by the user of this class.
// ItemType.h must contain the following definitions:
// MAX_ITEMS: the maximum number of items on the list
// ItemType: the definition of the objects on the list
// RelationType: (LESS, GREATER, EQUAL)
// Member function ComparedTo(ItemType item), which returns
// LESS, if self "comes before" item
// GREATER, if self "comes after" item
// EQUAL, if self and item are the same

class UnsortedType
{
public:
  UnsortedType();
  void MakeEmpty();
  bool IsFull() const;
  int GetLength() const;
  void RetrieveItem(ItemType& item, bool& found);
  void InsertItem(ItemType item);
  void DeleteItem(ItemType item);
  void ResetList();
  void GetNextItem(ItemType& item);
private:
  int length;
  ItemType info[MAX_ITEMS];
  int currentPos;
};

Now let’s look at the operations that we have specified for the Unsorted List ADT.
Constructor Operations

Our CRC card didn’t have a constructor, but we know that a class should have one. We said earlier that a constructor is often a language-level operation. C++ provides a language-level construct called a class constructor that does this initialization automatically when a variable of the class is declared. A class constructor is a member function having the same name as the class but having no return type. A constructor’s purpose is to initialize class members, and if necessary, to allocate resources (usually memory) for the object being constructed. Like any other member function, a constructor has access to all members, public and private, both data members and function members. Like all member functions, class constructors can have an empty parameter list (called a default class constructor) or have one or more parameters.

We included a class constructor in the class declaration above. Now, what must the UnsortedType class constructor do? The postcondition states that the list is empty. Any array cannot be “empty”; the slots exist. A list, however, consists of only those values that we have stored in the array, that is, from location zero through location length – 1. An empty list, then, is one where the length is 0.

UnsortedType::UnsortedType()
{
    length = 0;
}

Notice that we do not have to do anything to the array that holds the list items to make a list empty. If length is zero, the list is empty. If length is not zero, we must have stored items in the array through the length – 1 position, covering up what was there. What is present in the array from the length position to the end is of no interest to us. This distinction is very important: The list is between positions 0 and length – 1; the array is between positions 0 and MAX_ITEMS – 1.

C++ Rules for Using Class Constructors

C++ has intricate rules governing the use of constructors. The following guidelines are especially pertinent:

1. A constructor cannot return a function value, so the function is declared without a return value type. Although not necessary, return statements with no expressions are allowed at the end of a constructor. Thus return; is legal in a constructor, but return 0; is not.
2. Like any other member function, a constructor may be overloaded. Hence, a class may provide several constructors. When a class object is declared, the compiler chooses the appropriate constructor based on the number and data types of the parameters to the constructor, just as in any other call to overloaded functions.
3. Arguments to a constructor are passed by placing the argument list immediately after the name of the class object being declared:

    SomeType myObject(argument1, argument2);
3.2 Abstract Data Type Unsorted List

4. If a class object is declared without a parameter list, as in the statement

```cpp
SomeType myObject;
```

then the effect depends on the constructors (if any) provided by the class. If the class has no constructors, the compiler generates a default constructor that does nothing. If the class does have constructors, then the default (parameterless) constructor is invoked if there is one. If the class has constructors but no default constructor, a syntax error occurs.

5. If a class has at least one constructor, and an array of class objects is declared as in the statement

```cpp
SomeType myObject[5];
```

then one of the constructors must be the default (parameterless) constructor. This constructor is invoked for each element in the array.

---

**Observer Operations** The observer function `IsFull` checks whether `length` is equal to `MAX_ITEMS`.

```cpp
bool UnsortedType::IsFull() const
{
    return (length == MAX_ITEMS);
}
```

The body of the observer member function `GetLength` is also just one statement.

```cpp
int UnsortedType::GetLength() const
{
    return length;
}
```

So far we have not used our special design terminology. The algorithms have all been one (obvious) statement long. The next operation, `RetrieveItem`, is more complex. The `RetrieveItem` operation allows the list user to access the list item with a specified key, if that element exists in the list. `item` (with the key initialized) is input to this operation; `item` and a flag (`found`) are returned. If the key of `item` matches a key in the list, then `found` is true and `item` is set equal to the element with the same key. Otherwise, `found` is false and `item` remains unchanged. Notice that `item` is used for both input to and output from the function. Conceptually, the key member is input; the other data members are output because the function fills them in.

To retrieve an element, we must first find it. Because the items are unsorted, we must use a linear search. We begin at the first component in the list and loop until
either we find an item with the same key or there are no more items to examine. Recognizing a match is easy: item.ComparedTo(info[location]) returns EQUAL. But how do we know when to stop searching? If we have examined the last element, we can stop. Thus, in our design terminology, we continue looking as long as we have not examined Info(last). Our looping statement is a while statement with the expression (moreToSearch AND NOT found). The body of the loop is a switch statement based on the results of function ComparedTo. We summarize these observations in the following algorithm:

```
RetrieveItem
Initialize location to position of first item
Set found to false
Set moreToSearch to (have not examined Info(last))
while moreToSearch AND NOT found
    switch (item.ComparedTo(Info(location)))
        case LESS :
        case GREATER : Set location to Next(location)
            Set moreToSearch to (have not examined Info(last))
        case EQUAL : Set found to true
            Set item to Info(location)
```

Before we code this algorithm, let’s look at the cases where we find the item in the list and where we examine Info(last) without finding it. We represent these cases in Figure 3.3 in an honor roll list. First, we retrieve Sarah. Sarah is in the list, so moreToSearch is true, found is true, and location is 1. That’s as it should be (see Figure 3.3a). Next, we retrieve Susan. Susan is not in the list, so moreToSearch is false, found is false, and location is equal to length (see Figure 3.3b).

Now we are ready to code the algorithm, replacing the general design notation with the equivalent array notation. The substitutions are straightforward except for initializing location and determining whether we have examined Info(last). To initialize location in an array-based implementation in C++, we set it to 0. We know we have not examined Info(last) as long as location is less than length. Be careful: Because C++ indexes arrays starting with 0, the last item in the list is found at index length - 1. Here is the coded algorithm:

```
void UnsortedType::RetrieveItem(ItemType& item, bool& found)
// Pre: Key member(s) of item is initialized.
// Post: If found, item’s key matches an element’s key in the
//       list and a copy of that element has been stored in item;
//       otherwise, item is unchanged.
```

© Jones and Bartlett Publishers. NOT FOR SALE OR DISTRIBUTION
3.2 Abstract Data Type Unsorted List

{  
    bool moreToSearch;
    int location = 0;
    found = false;

    moreToSearch = (location < length);

    while (moreToSearch && !found)  
    {  
        switch (item.ComparedTo(info[location]))  
        {  
            case LESS :
            case GREATER : location++;  
                moreToSearch = (location < length);  
                break;
            case EQUAL  : found = true;
                item = info[location];  
                break;
        }
    }
}

Note that a copy of the list element is returned. The caller cannot access directly any data in the list.
Transformer Operations Where do we insert a new item? Because the list elements are unsorted by key value, we can put the new item anywhere. A straightforward strategy is to place the item in the \texttt{length} position and then to increment \texttt{length}.

This algorithm is translated easily into C++.

```
void UnsortedType::InsertItem(ItemType item)
// Post: item is in the list.
{
    info[length] = item;
    length++;
}
```

The \texttt{DeleteItem} function takes an item with the key member indicating which item to delete. This operation clearly has two parts: finding the item to delete and removing it. We can use the \texttt{RetrieveItem} algorithm to search the list: When \texttt{ComparedTo} returns \texttt{GREATER} or \texttt{LESS}, we increment \texttt{location}; when it returns \texttt{EQUAL}, we exit the loop and remove the element.

How do we "remove the element from the list"? Let's look at the example in Figure 3.4. Removing Judy from the list is easy, for hers is the last element in the list (see Figures 3.4a and 3.4b). If Bobby is deleted from the list, however, we need to move up all the elements that follow to fill in the space—or do we? If the list is sorted by value, we would have to move all elements up, as shown in Figure 3.4c. Because the list is unsorted, however, we can just swap the item in the \texttt{length - 1} position with the item being deleted (see Figure 3.4d). In an array-based implementation, we do not actually remove the element; instead, we cover it up with the element that previously followed it (if the list is sorted) or the element in the last position (if the list is unsorted). Finally, we decrement \texttt{length}.

Because the preconditions for \texttt{DeleteItem} state that an item with the same key is definitely present in the list, we do not need to test for the end of the list. This choice simplifies the algorithm so much that we give the code with no further discussion.

```
void UnsortedType::DeleteItem(ItemType item)
// Pre:  item's key has been initialized.
// Post: No element in the list has a key that matches item's.
{
    int location = 0;
```
3.2 Abstract Data Type Unsorted List

(a) Original list

(b) Deleting Judy

(c) Deleting Bobby (move up)

(d) Deleting Bobby (swap)

Figure 3.4 Deleting an item in an unsorted list
Chapter 3: ADT Unsorted List

```cpp
while (item.ComparedTo(info[location]) != EQUAL)
    location++;

info[location] = info[length - 1];
length--;
```

How do we make the list empty? We just set length to 0.

```cpp
void UnsortedType::MakeEmpty()
    // Post: List is empty.
{
    length = 0;
}
```

**Iterator Operations**

The `ResetList` function is analogous to the open operation for a file in which the file pointer is positioned at the beginning of the file so that the first input operation accesses the first component of the file. Each successive call to an input operation gets the next item in the file. As a consequence, `ResetList` must initialize `currentPos` to point to the predecessor of the first item in the list.

The `GetNextItem` operation is analogous to an input operation; it accesses the next item by incrementing `currentPos` and returning `Info(currentPos)`.

The `ResetList` function initializes `currentPos` to position it at the beginning of the list.

The `GetNextItem` operation retrieves the next item by incrementing `currentPos`.

Current `currentPos` remains undefined until `ResetList` initializes it. After the first call to `GetNextItem`, `currentPos` is the location of the last item accessed by `GetNextItem`. Therefore, to implement this algorithm in an array-based list in C++, `currentPos` must be initialized to `-1`. These operations are coded as follows:

```cpp
void UnsortedType::ResetList()
    // Post: currentPos has been initialized.
{
    currentPos = -1;
}
```

What would happen if a transformer operation is executed between calls to `GetNextItem`? The iteration would be invalid. We should add a precondition to prevent this from happening.
3.2 Abstract Data Type Unsorted List

```cpp
void UnsortedType::GetNextItem(ItemType& item)
// Pre:  ResetList was called to initialized iteration.
//       No transformer has been executed since last call.
//       CurrentPos is defined.
// Post: item is current item.
//       Current position has been updated.
{    
    currentPos++;
    item = info[currentPos];
}
```

ResetList and GetNextItem are designed to be used in a loop in the client program that iterates through all items in the list. The precondition in the specifications for GetNextItem protects against trying to access an array element that is not present in the list. This precondition requires that, before a call to GetNextItem, the element at the current position not be the last item in the list. Notice that this precondition places the responsibility for accessing only defined items on the client rather than on GetNextItem.

Here are the complete specification and implementation files for the array-based version of class UnsortedType with reduced documentation.

```cpp
// File "UnsortedType.h" contains the specification for the
class UnsortedType.
#include "ItemType.h"
// File ItemType.h must be provided by the user of this class.
// ItemType.h must contain the following definitions:
// MAX_ITEMS:     the maximum number of items on the list
// ItemType:      the definition of the objects on the list
// RelationType:  {LESS, GREATER, EQUAL}
// Member function ComparedTo(ItemType item) which returns
// LESS, if self "comes before" item
// GREATER, if self "comes after" item
// EQUAL, if self and item are the same
class UnsortedType
{
public:
    UnsortedType();
    void MakeEmpty();
    bool IsFull() const;
    int GetLength() const;
    void RetrieveItem(ItemType& item, bool& found);
    void InsertItem(ItemType item);
    void DeleteItem(ItemType item);
    void ResetList();
    void GetNextItem(ItemType& item);
private:
    int length;
    ItemType info[MAX_ITEMS];
    int currentPos;
};
```
// Implementation file for Unsorted.h

#include "unsorted.h"

UnsortedType::UnsortedType()
{
    length = 0;
}

bool UnsortedType::IsFull() const
{
    return (length == MAX_ITEMS);
}

int UnsortedType::GetLength() const
{
    return length;
}

void UnsortedType::RetrieveItem(ItemType& item, bool& found)
{
    bool moreToSearch;
    int location = 0;
    found = false;
    moreToSearch = (location < length);

    while (moreToSearch && !found)
    {
        switch (item.ComparedTo(info[location]))
        {
            case LESS    :
            case GREATER :
                location++;
                moreToSearch = (location < length);
                break;
            case EQUAL   :
                found = true;
                item = info[location];
                break;
        }
    }
}

void UnsortedType::MakeEmpty()
{
    length = 0;
}

void UnsortedType::InsertItem(ItemType item)
{
    info[length] = item;
    length++;
}

void UnsortedType::DeleteItem(ItemType item)
{
    int location = 0;
}
while (item.ComparedTo(info[location]) != EQUAL)
    location++;

info[location] = info[length - 1];
length--;
}

void UnsortedType::ResetList()
{
    currentPos = -1;
}

void UnsortedType::GetNextItem(ItemType& item)
{
    currentPos++;
    item = info[currentPos];
}

**Notes on the Array-Based List Implementation** In several of our list operations, we have declared the local variable `location`, which contains the array index of the list item being processed. The values of array indexes are never revealed outside of the list operations; this information remains internal to the implementation of the Unsorted List ADT. If the list user wants an item in the list, the `RetrieveItem` operation does not give the user the index of the item; instead, it returns a copy of the item. If the user wants to change the values of data members in an item, those changes are not reflected in the list unless the user deletes the original values and inserts the modified version. The list user can never see or manipulate the physical structure in which the list is stored. These details of the list implementation are encapsulated by the ADT.

**Test Plan** The class `UnsortedType` has a constructor and seven other member functions: `InsertItem` and `DeleteItem` (transformers); `IsFull`, `GetLength`, and `RetrieveItem` (observers); and `ResetList` and `GetNextItem` (iterators). Because our operations are independent of the type of the objects on the list, we can define `ItemType` to be `int` and know that if our operations work with these data, they work with any other `ItemType`. Here, then, is the definition of `ItemType` that we use in our test plan. We set the maximum number of items to 5. We include a member function to print an item of the class `ItemType` to an ofstream object (a file). We need this function in the driver program to see the value in a list item.

```cpp
// The following declarations and definitions go into file ItemType.h.
#include <fstream>

const int MAX_ITEMS = 5;
enum RelationType {LESS, GREATER, EQUAL};

class ItemType
{
    public:
```
Chapter 3: ADT Unsorted List

```cpp
ItemType();
RelationType ComparedTo(ItemType) const;
void Print(std::ofstream&) const;
void Initialize(int number);
private:
  int value;
};

// The following definitions go into file ItemType.cpp.
#include <fstream>
#include "ItemType.h"
ItemType::ItemType()
{
  value = 0;
}

RelationType ItemType::ComparedTo(ItemType otherItem) const
{
  if (value < otherItem.value)
    return LESS;
  else if (value > otherItem.value)
    return GREATER;
  else return EQUAL;
}

void ItemType::Initialize(int number)
{
  value = number;
}

void ItemType::Print(std::ofstream& out) const
// Pre:  out has been opened.
// Post:  value has been sent to the stream out.
{
  out << value << " ";
}
```

The preconditions and postconditions in our specification determine the tests necessary for a black-box testing strategy. The code of the functions determines a clear-box testing strategy. To test the ADT Unsorted List implementation, we use a combination of the two strategies. Because a precondition on all other operations is that the list has been initialized, we test the constructor by checking whether the list is empty initially (a call to GetLength returns 0).
GetLength, InsertItem, and DeleteItem must be tested together. That is, we insert several items and check the length; we delete several items and check the length. How do we know that InsertItem and DeleteItem work correctly? We write an auxiliary function PrintList that uses GetLength, ResetList, and GetNextItem to iterate through the list, printing out the values. We call PrintList to check the status of the list after a series of insertions and deletions. To test the IsFull operation, we insert four items and print the result of the test, and then insert the fifth item and print the result of the test. To test RetrieveItem, we search for items that we know are present in the list and for items that we know are not found in the list.

How do we choose the values used in our test plan? We look at the end cases. What are the end cases in a list? The item is in the first position in the list, the item is in the last position in the list, and the item is the only one in the list. We must be sure that DeleteItem can correctly delete items in these positions. We must also confirm that RetrieveItem can find items in these same positions and correctly determine that values that are less than the one in the first position or greater than the one in the last position are not found. Notice that this test plan involves a black-box strategy. That is, we look at the list as described in the interface, not the code.

These observations are summarized in the following test plan. The tests are shown in the order in which they should be performed.

<table>
<thead>
<tr>
<th>Operation to Be Tested and Description of Action</th>
<th>Input Values</th>
<th>Expected Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>print GetLength</td>
<td></td>
<td></td>
</tr>
<tr>
<td>InsertItem</td>
<td>5, 7, 6, 9</td>
<td>5 7 6 9</td>
</tr>
<tr>
<td>Insert item and print</td>
<td>1</td>
<td>5 7 6 9 1</td>
</tr>
<tr>
<td>RetrieveItem</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Retrieve 4 and print whether found</td>
<td>Item is not found</td>
<td></td>
</tr>
<tr>
<td>Retrieve 5 and print whether found</td>
<td>Item is found</td>
<td></td>
</tr>
<tr>
<td>Retrieve 9 and print whether found</td>
<td>Item is found</td>
<td></td>
</tr>
<tr>
<td>Retrieve 10 and print whether found</td>
<td>Item is not found</td>
<td></td>
</tr>
<tr>
<td>IsFull</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Invoke (list is full)</td>
<td>List is full</td>
<td></td>
</tr>
<tr>
<td>Delete 5 and invoke</td>
<td>List is not full</td>
<td></td>
</tr>
<tr>
<td>DeleteItem</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delete 1 and print</td>
<td>7 6 9</td>
<td></td>
</tr>
<tr>
<td>Delete 6 and print</td>
<td>7 9</td>
<td></td>
</tr>
</tbody>
</table>
What about testing GetLength, ResetList, and GetNextItem? They do not appear explicitly in the test plan, but they are tested each time we call the auxiliary function PrintList to print the contents of the list.

Here are the UML diagrams for ItemType, used in testing, and UnsortedType. These diagrams become part of the documentation for class UnsortedType. When a class contains an instance of another class, such as the array of class ItemType contained in the UnsortedType, we draw a solid diamond at the end of the line next to the containing class. An ampersand (&) following the parameter type on a method (function) heading indicates that the parameter is a reference parameter.

On the Web, the file listType.in contains the input to listDr.cpp (the test driver) that reflects this test plan; listType.out and listTest.screen contain the output.
3.3 Pointer Types

Logical Level

Pointers are simple—not composite—types, but they allow us to create composite types at run time. We describe the creation process, accessing function, and use of pointers in this chapter.

A pointer variable does not contain a data value in the ordinary sense; rather, it contains the memory address of another variable. To declare a pointer that can point to an integer value, you use the following syntax:

```c
int* intPointer;
```

The asterisk (*) as a postfix symbol on the type says that the variable being defined is a pointer to an object of that type: `intPointer` can point to a place in memory that can contain a value of type `int`. (Alternatively, the asterisk may serve as a prefix symbol on the variable name.) The contents of `intPointer`, as with all newly defined variables, are undefined. The following diagram shows memory after this statement has been executed. (For illustrative purposes, we assume that the compiler has allocated location 10 to `intPointer`.)
How do we get intPointer something to point to? One way is to use the prefix & operator, which is called the *address-of* operator. Given the declarations:

```c
int alpha;
int* intPointer;
```

the assignment statement

```c
intPointer = &alpha;
```

takes the address of alpha and stores it into intPointer. If alpha is at address 33, memory looks like this:

We have a pointer, and we have a place to which the pointer points, but how do we access that place? An asterisk (*) as a prefix to the pointer name accesses the place to which the pointer points. The asterisk is called the *dereference operator*. Let’s store 25 in the place to which intPointer points.

```c
*intPointer = 25;
```
Memory now looks like this:

Because `intPointer` points to `alpha`, the statement

```
*intPointer = 25;
```

represents *indirect addressing* of `alpha`; the machine first accesses `intPointer`, then uses its contents to find `alpha`. In contrast, the statement

```
alpha = 10;
```

involves *direct addressing* of `alpha`. Direct addressing is analogous to opening a post office box (P.O. Box 15, for example) and finding a package, whereas indirect addressing is analogous to opening P.O. Box 15 and finding a note telling you that your package is sitting in P.O. Box 23.

A second method for getting `intPointer` something to point to is called dynamic allocation. In the previous example, the memory space for both `intPointer` and `alpha` was allocated statically (at compile time). Alternatively, our programs can allocate memory dynamically (at run time).
To achieve dynamic allocation of a variable, we use the C++ operator `new`, followed by the name of a data type:

```cpp
intPointer = new int;
```

At run time, the `new` operator allocates a variable capable of holding an `int` value and returns its memory address, which is then stored into `intPointer`. If the `new` operator returns the address 90 as a result of executing the preceding statement, memory looks like this:

Variables created by `new` reside on the **free store** (or **heap**), a region of memory set aside for dynamic allocation. A dynamically allocated variable has no name and cannot be directly addressed. It must be indirectly addressed through the pointer returned by `new`.

Sometimes we want a pointer to point to nothing. By definition in C++, a pointer value of 0 is called the **null pointer**; it points to nothing. To help distinguish the null pointer from the integer value 0, `<cstddef>` contains the definition of a named constant `NULL` that we use instead of referring directly to 0. Let's look at a few more examples.

```cpp
bool* truth = NULL;
float* money = NULL;
```
When drawing pictures of pointers, we use a diagonal line from the upper right to the lower left to indicate that the value is NULL.

Let's examine memory after a few more pointer manipulations.

```cpp
class Point { public: int x, y; public: Point(int x, int y); }; int main() { Point p(2, 3); return 0; }```

When drawing pictures of pointers and the objects to which they point, we use boxes and arrows.

Any operation that can be applied to a constant or a variable of type `int` can be applied to `*intPointer`. Any operation that can be applied to a constant or a variable of type `float` can be applied to `*money`. Any operation that can be applied to a constant or a variable of type `bool` can be applied to `*truth`. For example, we can read a value into `*myMoney` with the following statement:

```cpp
std::cin >> *myMoney;
```

If the current value in the input stream is 99.86, then `*myMoney` contains 99.86 after the execution of the preceding statement.
Pointer variables can be compared for equality and assigned to one another as long as they point to variables of the same data type. Consider the following two statements:

```c
*myMoney = *money;
myMoney = money;
```

The first statement copies the value in the place pointed to by `money` into the place pointed to by `myMoney`.

The second statement copies the value in `money` into `myMoney`, giving the following configuration:

At this point, the location that holds the second copy of 33.46 cannot be accessed; no pointer points to it. This situation is called a memory leak, and the memory cells that can no longer be accessed are called garbage. Some programming languages, such as Java, provide garbage collection; that is, the run-time support system periodically goes through memory and reclaims the memory locations for which no access path exists.

To avoid memory leaks, C++ provides the operator `delete`, which returns to the free store a memory location allocated previously by the `new` operator. This mem-
ory may then be allocated again if necessary. The following code segment prevents our memory leak:

```c
delete myMoney;
myMoney = money;
```

The location originally pointed to by `myMoney` is no longer allocated. Note that `delete` does not delete the pointer variable, but rather the variable to which the pointer points.

Notice the difference between assigning one pointer to another (`myMoney = money`) and assigning the pointed-to locations (`*myMoney = *money`). *Always be careful to distinguish between the pointer and the object to which it points!*

**Application Level**

Have you passed an array as a parameter? If so, you have used a *pointer constant*. The name of an array without any index brackets is a constant pointer expression—namely, the base address of the array. Look at the following code segment:

```c
char alpha[20];
char* alphaPtr;
char* letterPtr;
void Process(char[]);

alphaPtr = alpha;
letterPtr = &alpha[0];
Process(alpha);
```

Here `alphaPtr` is assigned the constant pointer expression `alpha`; `letterPtr` is assigned the address of the first position in the array `alpha`. `alphaPtr` and `letterPtr` both contain the address of the first position in the array `alpha`. When the prototype for the function `Process` says that it takes a `char` array as a parameter, it means that it
expects a pointer expression (the base address of the array) as an actual parameter. The calling statement, therefore, sends the name of the array without any index brackets to the function.

Pointers can be used with other composite types as well.

```c
struct MoneyType
{
    int dollars;
    int cents;
};

MoneyType* moneyPtr = new MoneyType;
moneyPtr->dollars = 3245;
moneyPtr->cents = 33;
```

The arrow operator (->) provides a shortcut for dereferencing a pointer and accessing a struct or class member. That is, `moneyPtr->cents` is shorthand for `(moneyPtr).cents`. (The dereferencing operator has lower precedence than the dot operator, so the parentheses are necessary.)

In the next section we will use the idea of a pointer pointing to a variable of a composite type in which one of the data members is a pointer to another variable of the same composite type. Using this technique, we can build linked structures. One named pointer acts as the external pointer to the structure, and the structure is chained together by having a data member in each variable in the chain act as a pointer to the next one in the chain. The last variable in the chain has `NULL` in its pointer member.

**Implementation Level**

A pointer variable simply contains a memory address. The operating system controls memory allocation and grants memory to your program on request.

### 3.4 Implementing Class UnsortedType as a Linked Structure

For our first implementation we used an array. An array’s components are accessed by their position in the structure. Arrays, the first organizational structure a student usually learns, are used to implement other structures as we showed earlier in this chapter. They are available as a basic language construct in most high-level programming languages.

As we also said earlier, a linked structure is a collection of separate elements, with each element linked to the one that follows it in the list. We can think of a linked list as a chain of elements. The linked list is a versatile, powerful, basic implementation structure, and like the array, it is one of the primary building blocks for the more complicated structures. Teaching you how to work with links and linked lists is one of the important goals of this textbook. Not only can links be used to implement the more complicated classic structures, but they can also be used to create your own structures.
Now, let’s see how we might use the concept of dynamic storage allocation to implement a linked list.

**Linked Structures**

As we look at implementing a linked structure, let’s use the `InsertItem` function of the `UnsortedType` as our example. We can modify function `InsertItem` to allocate space for each new element dynamically.

---

### InsertItem

- Allocate space for new item
- Put new item into the allocated space
- Put the allocated space into the list

---

Implementing the first part of this operation is simple. We use the built-in C++ operator `new` to allocate space dynamically:

```cpp
// Allocate space for new item.
itemPtr = new ItemType;
```

The `new` operator allocates a block of memory big enough to hold a value of type `ItemType` (the type of data contained in the list) and returns the block’s address, which is copied into the variable `itemPtr`. Let’s say for the moment that `itemPtr` has been declared to be of type `ItemType*`. Now we can use the dereference operator (`*`) to put `newItem` into the space that was allocated:

```cpp
*itemPtr = newItem;
```

The third part of the `InsertItem` operation is to “put the allocated space into the list.” How do we do this? Let’s think for a minute what happens after we have inserted a
few characters. Space is allocated for each new element, and each character is put into the space. Figure 3.6 shows the results of calling InsertItem to add the characters ‘D’, ‘A’, ‘L’, and ‘E’ to the list.

Whoops—we see our data in the dynamically allocated space, but it’s not a list! There’s no order. Even worse, because we haven’t returned the pointers to the dynamically allocated space from function InsertItem, we have no way to access any of the elements any more. Clearly, the third part of the InsertItem operation needs to do something to fix this situation. Where can we store the pointers to our data?

One possibility that comes to mind is to declare the list as an array of pointers and to put the pointer to each new item into this array, as shown in Figure 3.7. This solution would keep track of the pointers to all the elements in the correct order, but it wouldn’t solve our original problem: we still need to declare an array of a particular size. Where else could we put the pointers? It would be nice if we could just chain all the elements together somehow, as shown in Figure 3.8. We call each element in this “linked” list a node.

This solution looks promising. Let’s see how we might use this idea to implement the list. First we insert the character ‘D’. InsertItem uses operator new to allocate
3.4 Implementing Class `UnsortedType` as a Linked Structure

We don’t want to lose the pointer to this element, so we need a data member in our list class in which to store the pointer to the top of the list (`listData`). Figure 3.9 illustrates the first `InsertItem` operation.

Now we call `InsertItem` to add the character ‘A’ to the list. `InsertItem` applies `new` to allocate space for the new element and puts ‘A’ into the space. Next we want to chain ‘A’ to ‘D’, the old top list element. We can establish a link between the two elements by letting one element “point” to the next; that is, we can store into each list element the address of the next element. To do this, we need to modify the list node type. Let’s make each node in the list contain two parts: `info` and `next`. The `info` member contains the list user’s data—a character, for instance. The `next` member contains the address of the next element in the list. A single node is pictured in Figure 3.10.

As you can see in the figure, the `next` member of each node points to the next node in the list. What about the `next` member of the last node? We cannot leave it unassigned. The `next` member of the last node in the list must contain some special value that is not a valid address. `NULL`, a special pointer constant available in `<cstddef>`, says, “This pointer doesn’t point to anything.” We can put `NULL` in the `next` member of the last node, to mark the end of the list. Graphically, we use a slash across the `next` member to represent a `NULL` pointer.

Earlier in the chapter we introduced a list design terminology that incorporated the abstraction of a node (see Figure 3.11).

In the array-based implementation `location` is an index; in our pointer-based implementation it must be a pointer to a record that contains both the user’s information and a pointer to the next node on the list.
Now let’s return to our `InsertItem` algorithm. We have allocated a node to contain the new element ‘A’ using operator `new` (Figure 3.12a). Let’s revert to our design terminology, and call the pointer `location`.

Set location to address of a new node of type `ItemType`  // Allocate space for new item

Then the new value, ‘A’ is put into the node (Figure 3.12b):  

Set `Info(location)` to `newItem`  // Put new item into the allocated space

Now we are ready to link the new node to the list. Where should we link it? Should the new node come before or after the node containing ‘D’? In the array-based
3.4 Implementing Class UnsortedType as a Linked Structure

Figure 3.12 The second InsertItem operation

(a) new(location)

(b) info(location) ← newElement

(c) Make next (location) point to List’s top node

(d) Make List point to the new node
implementation, we put the new item at the end because that was the easiest place to put it. What is the analogous place in the linked implementation? At the beginning of the list. Because the list is unsorted, we can put the new item wherever we choose, and we choose the easiest place: at the front of the list.

Linking the new node to the (previous) first node in the list is a two-step process:

Make Next(location) point to the stack's top node  // (Figure 3.12c)
Make listData point to the new node  // (Figure 3.12d)

Note that the order of these tasks is critical. If we changed the listData pointer before making Next(location) point to the beginning of the list, we would have lost access to the list nodes! (See Figure 3.13.) This situation is generally true when we are dealing with a linked structure: you must be very careful to change the pointers in the correct order, so that you do not lose access to any of the data.

Before we code this algorithm, let's see how the list data are declared. Remember that from the UnsortedType user's point of view, nothing has changed; the prototype for member function InsertItem is the same as it was for the array-based implementation.

```cpp
void Insert(ItemType newItem);
```

ItemType is still the type of data that the user wants to put in the list. Class UnsortedType, however, needs new definitions. It no longer is a class with an array member to hold the items; its member is listData, the pointer to a single node, the top of the list. The node to which listData points has two parts, info and next, which suggests a C++ struct or class representation. We choose to make NodeType a struct rather than a
3.4 Implementing Class UnsortedType as a Linked Structure

class, because the nodes in the structure are passive. They are acted upon by the member functions of UnsortedType.

We complete the coding of InsertItem within the context of class UnsortedType.

Class UnsortedType

In order to implement the Unsorted List ADT, we need to record two pieces of information about the structure in addition to the list of items. The GetLength operation returns the number of items in the list. In the array-based implementation, the length member defines the extent of the list within the array. Therefore, the length member must be present. In a link-based list we have a choice: we can keep a length member or we can count the number of elements each time the GetLength operation is called. Keeping a length member requires an addition operation each time InsertItem is called and a subtraction operation each time DeleteItem is called. Which is better? We cannot determine in the abstract; it depends on the relative frequency of use of the GetLength, InsertItem, and DeleteItem operations. Here, let’s explicitly keep track of the length of the list by including a length member in the UnsortedType class.

ResetList and GetNextItem require that we keep track of the current position during an iteration, so we need a currentPos member. In the array-based implementation, currentPos is an array index. What is the logical equivalent within a linked list? A pointer. Figure 3.14 pictures this structure.

Figure 3.14  Honor roll list with two items (ResetList has not been called)
Chapter 3: ADT Unsorted List

private:
    NodeType* listData;
    int length;
    NodeType* currentPos;

The function headings (prototypes) that complete the specification are identical to those for the array-based implementation. Remember that the specification of the class is the interface that the user sees. The interface documentation tells the user what the operations do and this doesn’t change with the implementation. However, in the linked implementation, our private data fields are different: currentPos and listData are pointers to nodes.

NodeType is a struct containing the user’s data, as well as a pointer to the next node. We alert the compiler that we are going to use a pointer to NodeType before we have defined NodeType by using the statement

```cpp
struct NodeType;
```

This statement, called a `forward declaration`, is analogous to a function prototype: the compiler is told the nature of an identifier before the identifier is fully defined. The definition of NodeType (shown below) comes later in either the header file or the implementation file.

```cpp
struct NodeType
{
    ItemType info;
    NodeType* next;
};
```

Let’s complete the implementation of **InsertItem** before we look at the other operations.

**Function InsertItem**

Here are the pieces of the detailed algorithm collected together.

```
InsertItem
Set location to address of a new node of type ItemType
Set Info(location) to item
Make Next(location) point to the list’s top node
Make listData point to the new node
```
3.4 Implementing Class UnsortedType as a Linked Structure

Table 3.1 Comparing node design notation to C++ Code

<table>
<thead>
<tr>
<th>Design Notation</th>
<th>C++ Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node(location)</td>
<td>*location</td>
</tr>
<tr>
<td>Info(location)</td>
<td>location-&gt;info</td>
</tr>
<tr>
<td>Next(location)</td>
<td>location-&gt;next</td>
</tr>
<tr>
<td>Set location to Next(location)</td>
<td>location = location-&gt;next</td>
</tr>
<tr>
<td>Set Info(location) to value</td>
<td>location-&gt;info = value</td>
</tr>
</tbody>
</table>

Table 3.1 summarizes the relationship between the design and the code terminology. Using this table, we can code function InsertItem.

We use a local variable, location, of type NodeType*. The first two tasks are simple:

```plaintext
// Allocate space for new item.
location = new NodeType;

// Put new item into the allocated space.
location->info = newItem;
```

*location is the way that we reference this node; it is a struct with two members. Because *location is a struct, we can access its members in the usual way—by adding a period, followed by the desired member name. So (*location).info refers to the info member of the struct on the free store pointed to by location. The parentheses are necessary because the dot operator has higher precedence than the dereferencing operator (*). Because we so often want to access the members of the struct or class to which a pointer points, C++ provides the -> operator, which both dereferences the pointer and accesses a member. So the expression (*location).info is equivalent to location->info. The shaded area in

![Figure 3.15](image-url)  

*Figure 3.15  Pointer dereferencing and member selection*
Chapter 3: ADT Unsorted List

Figure 3.15(a) corresponds to location, the shaded area in (b) corresponds to *location, and the shaded area in (c) corresponds to location->info. location->info is the same type as the user’s data, so we can assign newItem to it. So far, so good.

Now for the linking task:

Make Next(location) point to the list’s first node  // [Figure 3-12(c)]
Make listData point to the new node     // [Figure 3-12(d)]

Next(location) is the next member of the node pointed to by location. We can access it just as we accessed the info member: location->next. What can we put into this field? It is declared as a pointer, so we can assign it another value of the same pointer type. We want to make this member point to the list’s top node. Because we have a pointer to the list’s first node (listData), this assignment is simple.

location->next = listData;

Finally, we need to complete the linking by making listData point to the new node. listData is declared as a pointer, so we can assign another value of the same pointer type to it. Because we have a pointer to the new node (the local variable location), this assignment can be made:

listData = location;

Here is the complete function InsertItem.

```c
void UnsortedType::InsertItem(ItemType item)  // item is in the list; length has been incremented.
{
    NodeType* location;  // Declare a pointer to a node
    location = new NodeType;  // Get a new node
    location->info = item;  // Store the item in the node
    location->next = listData;  // Store address of first node
        // in next field of new node
    listData = location;  // Store address of new node
        // into external pointer
    length++;  // Increment length of the list
}
```
You have seen how this algorithm works on a list that contains at least one value. What happens if this function is called when the list is empty? Space is allocated for the new element and the element is put into the space (Figure 3.16a). Does the function correctly link the new node to the top of an empty list? Let’s see. The next member of the new node is assigned the value of listData. What is this value when the list is empty? It is NULL, which is exactly what we want to put into the next member of the last node of a linked list (Figure 3.16b). Then listData is reset to point to the new node (Figure 3.16c). So this function works for an empty list, as well as a list that contains at least one element.

From the List ADT implementor’s perspective, the algorithms for the rest of the list operations are very similar to the ones developed for the sequential (array-based) implementation.

### Constructor

To initialize an empty list we merely set listData (the external pointer to the linked list) to NULL and set length to 0. Here is the class constructor to implement this operation:

```cpp
UnsortedType::UnsortedType()  // Class constructor
{
    length = 0;
    listData = NULL;
}
```
Observer Operations

What about function IsFull? Using dynamically allocated nodes rather than an array, we no longer have an explicit limit on the list size. We can continue to get more nodes until we run out of memory on the free store. The ISO C++ Standard provides that the new operator in C++ throws a bad_alloc exception when there is no more space to allocate. The bad_alloc exception is defined in the <new> header. The exception needs to be handled because an unhandled exception carries the sentence of death for the program.

```cpp
bool UnsortedType::IsFull() const
// Returns true if there is no room for another ItemType
// on the free store; false otherwise.
{
    NodeType* location;
    try
    {
        location = new NodeType;
        delete location;
        return false;
    }
    catch(std::bad_alloc exception)
    {
        return true;
    }
}
```

As in the array-based implementation, the GetLength operation just returns the length data member.

```cpp
int UnsortedType::GetLength() const
// Post: Number of items in the list is returned.
{
    return length;
}
```

Function MakeEmpty

The MakeEmpty operation for a linked list is more complicated than its sequential list counterpart, because the dynamically allocated space used by the elements must be freed, one node at a time. Data member listData is deallocated when the list goes out of scope, but the nodes pointed to by listData do not. We must loop through the list returning the nodes to the free store with the delete operator. The easiest approach is just to unlink each successive node in the list and free it. How do we know when there are “more nodes in the list”? As long as listData is not NULL, the list is not empty. So the resulting condition on the loop is while(listData != NULL). We must also set length to zero.
3.4 Implementing Class UnsortedType as a Linked Structure

void UnsortedType::MakeEmpty()
// Post: List is empty; all items have been deallocated.
{
    NodeType* tempPtr;

    while (listData != NULL)
    {
        tempPtr = listData;
        listData = listData->next;
        delete tempPtr;
    }
    length = 0;
}

Function RetrieveItem

The algorithm for the linked implementation is the same as the one for the array-based implementation. Given the parameter item, we traverse the list looking for a location where item.ComparedTo(Info(location)) returns EQUAL. The difference is in how we traverse the list and how we access Info(location) to send it as a parameter to ComparedTo.

When we coded the array-based function, we directly substituted the index notation for the list notation. Can we directly substitute the pointer notation for the list notation in the linked list? Let’s try it and see. The algorithm follows; the substitutions are marked in boldface for emphasis.

```
RetrieveItem
Initialize location to listData
Set found to false
Set moreToSearch to (location != NULL)
while moreToSearch AND NOT found
    switch (item.ComparedTo(location -> info))
        case LESS :
        case GREATER :
            Set location to location -> next
            Set moreToSearch to (location != NULL)
        case EQUAL :
            Set found to true
            Set item to location -> info
```

Let’s look at this algorithm and be sure that the substitutions do what we want by examining the value of location at the end. There are two cases:
1. $location = NULL$. If we reach the end of the list without finding an item whose key is equal to $item$'s key, then the item is not in the list. $location$ correctly has the null pointer value (see Figure 3.17a).

2. $item.ComparedTo(location->info) = EQUAL$. In this case, we have found the element within the list and have copied it into $item$ (see Figure 3.17b).

We can now code this algorithm being reasonably confident that it is correct.

```cpp
void UnsortedType::RetrieveItem(ItemType& item, bool& found) {
    // Pre: Key member(s) of item is initialized.
    // Post: If found, item's key matches an element's key in the
    //       list and a copy of that element has been stored in item;
    //       otherwise, item is unchanged.

    bool moreToSearch;  // [Initialize to true]
    NodeType* location;  // [Initialize to listData]

    found = false;
    moreToSearch = (location != NULL);

    while (moreToSearch && !found) {
        moreToSearch = moreToSearch && (!location->info->Key().compareTo(item.Key()));
        if (moreToSearch) location = location->next;
    }
}
```
3.4 Implementing Class `UnsortedType` as a Linked Structure

```c
switch (item.ComparedTo(info[location]))
{
    case LESS :
    case GREATER : location = location->next;
                   moreToSearch = (location != NULL);
                   break;
    case EQUAL  : found = true;
                   item = location->info;
                   break;
}
```

**Function DeleteItem**

In order to delete an item, we must first find it. In Figure 3.18, we see that `location` is left pointing to the node that contains the item for which we are searching, the one to be removed. In order to remove it, we must change the pointer in the *previous* node. That is, the `next` data member of the previous node must be changed to the `next` data member of the one being deleted.

`location` now points to a node no longer needed, so we must deallocate it using the `delete` operator.

(a) Delete Lila

```
listData
  ↓
  Kate ─── Lila ─── Becca ─── John
```

(b) Delete Kate

```
listData
  ↓
  Kate ─── Lila ─── Becca ─── John
```

*Figure 3.18* Deleting an interior node and deleting the first node
Because we know from the specifications that the item to be deleted is in the list, we can change our search algorithm slightly. Rather than compare the item for which we are searching with the information in Info(location), we compare it with Info(location->next). When we find a match, we have pointers to both the previous node (location) and the one containing the item to be deleted (location->next). Note that removing the first node must be a special case because the external pointer to the list (listData) must be changed. Is removing the last node a special case? No. The next data member of the node being deleted is NULL and it is stored into the next data member of Node(location), where it belongs.

```cpp
void UnsortedType::DeleteItem(ItemType item)
// Pre: item's key has been initialized.
//      An element in the list has a key that matches item's.
// Post: No element in the list has a key that matches item's.
{
    NodeType* location = listData;
    NodeType* tempLocation;
    // Locate node to be deleted.
    if (item == listData->info)
    {
        tempLocation = location; // Save pointer to node.
        listData = listData->next; // Delete first node.
    }
    else
    {
        while (!(item==(location->next)->info))
        {
            location = location->next; // Delete node at location->next.
            tempLocation = location->next;
            location->next = (location->next)->next;
        }
        tempLocation = location->next;
        delete tempLocation; // Return node.
        length--;
    }
}
```

Note that we had to save the pointer to the node being deleted in order to return the node to available space.

**Functions ResetList and GetNextItem**

The Unsorted List ADT specification defines the “current position” to mean the location of the last item accessed during an iteration through the list. In the array-based implementation, ResetList initialized currentPos to -1, and GetNextItem incremented
currentPos and returned info[currentPos]. For currentPos, we used the special value -1 to mean “prior to the first item in the list.” In the linked implementation, what special pointer value can we use for “prior to the first item in the list”? We can use the special pointer value NULL. Therefore ResetList sets the currentPos member to NULL, and getNextItem sets currentPos either to listData (in case currentPos was NULL) or to currentPos->next (in case it was not NULL) before returning currentPos->info.

```cpp
void UnsortedType::ResetList()
// Post: Current position has been initialized.
{
    currentPos = NULL;
}
```

```cpp
void UnsortedType::GetNextItem(ItemType& item)
// Pre: No transformer has been executed since last call.
// Post: A copy of the next item in the list is returned.
//       When the end of the list is reached, currentPos
//       is reset to begin again.
{
    if (currentPos == NULL)
        currentPos = listData;
    else
        currentPos = currentPos->next;
    item = currentPos->info;
}
```

Recall from the specifications that the body of getNextItem is not required to check for running off the end of the list. That is the responsibility of the caller. A precondition of getNextItem is that the item at the current position is not the last in the list. However, when working with linked lists an error can cause the program to crash with no warning. So we changed the post condition to show that if the last item has been accessed the list begins again.

Here is the implementation file for the linked version of class UnsortedType.

```cpp
// This file contains the linked implementation of class
// UnsortedType.

#include "UnsortedType.h"

UnsortedType::UnsortedType()  // Class constructor
{
    length = 0;
    listData = NULL;
}
```

```cpp
bool UnsortedType::IsFull() const
```
Chapter 3: ADT Unsorted List

// Returns true if there is no room for another ItemType // on the free store; false otherwise.
{
    NodeType* location;
    try
    {
        location = new NodeType;
        delete location;
        return false;
    }
    catch(std::bad_alloc exception)
    {
        return true;
    }
}

int UnsortedType::GetLength() const
// Post: Number of items in the list is returned.
{
    return length;
}

void UnsortedType::MakeEmpty()
// Post: List is empty; all items have been deallocated.
{
    NodeType* tempPtr;
    while (listData != NULL)
    {
        tempPtr = listData;
        listData = listData->next;
        delete tempPtr;
    }
    length = 0;
}

void UnsortedType::InsertItem(ItemType item)
// item is in the list; length has been incremented.
{
    NodeType* location; // Declare a pointer to a node
    location = new NodeType; // Get a new node
    location->info = item; // Store the item in the node
    location->next = listData; // Store address of first node // in next field of new node
    listData = location; // Store address of new node into
3.4 Implementing Class UnsortedType as a Linked Structure

```cpp
//   external pointer
    length++; // Increment length of the list
}
void UnsortedType::RetrieveItem(ItemType& item, bool& found)
// Pre: Key member(s) of item is initialized.
// Post: If found, item's key matches an element's key in the
//       list and a copy of that element has been stored in item;
//       otherwise, item is unchanged.
{
    bool moreToSearch;
    NodeType* location;

    location = listData;
    found = false;
    moreToSearch = (location != NULL);

    while (moreToSearch && !found)
    {
        switch (item.ComparedTo(info[location]))
        {
            case LESS :
            case GREATER : location = location->next;
                moreToSearch = (location != NULL);
                break;
            case EQUAL  : found = true;
                item = location->info;
                break;
        }
    }
}
void UnsortedType::DeleteItem(ItemType item)
// Pre: item's key has been initialized.
//       An element in the list has a key that matches item's.
// Post: No element in the list has a key that matches item's.
{
    NodeType* location = listData;
    NodeType* tempLocation;

    // Locate node to be deleted.
    if (item == listData->info)
    {
        tempLocation = location;
        listData = listData->next; // Delete first node.
    }
```
Chapter 3: ADT Unsorted List

```cpp
} else {
    while (!(item == (location->next)->info))
        location = location->next;

    // Delete node at location->next
    tempLocation = location->next;
    location->next = (location->next)->next;
}
delete tempLocation;
length--;

void UnsortedType::ResetList()
    // Post: Current position has been initialized.
    {
        currentPos = NULL;
    }

void UnsortedType::GetNextItem(ItemType& item)
    // Post: A copy of the next item in the list is returned.
    //      When the end of the list is reached, currentPos
    //      is reset to begin again.
    {
        if (currentPos == NULL)
            currentPos = listData;
        else
            currentPos = currentPos->next;
        item = currentPos->info;
    }

Here are the UML diagrams for the linked class UnsortedType.
```
Note that the CRC card for the *UnsortedType* does not change because it contains no implementation information. However, the UML diagram used for maintenance documentation does change. It shows the data fields. Note also that the UML diagram for *ItemType* no longer contains a reference to *MAX_ITEMS*. The test plan developed for the array-based version is equally valid for the linked version.

### Lifetime of a Variable

The *lifetime* of a variable is the time during program execution when the variable has storage assigned to it.

- The lifetime of a global variable is the entire execution of the program.
- The lifetime of a local variable is the execution of the block in which it is declared.
- The lifetime of a dynamically allocated variable is from the time it is allocated until the time it is deallocated.
Class Destructors

We pointed out when we were designing the MakeEmpty function in the linked implementation, that listData is deallocated when the list goes out of scope, but not the nodes that listData points to. Therefore, we must include a class destructor. A class destructor is member function named the same as a class constructor except that the destructor has a tilde (~) in front of the type name. A destructor is implicitly invoked when a class object goes out of scope—for example, when control leaves the block in which an object is declared. We include the following prototype of the destructor in the public part of the class definition:

```cpp
~UnsortedType(); // Destructor.
```

and implement the function as follows:

```cpp
UnsortedType::~UnsortedType()
// Post: List is empty; all items have been deallocated.
{
    NodeType* tempPtr;

    while (listData != NULL)
    {
        tempPtr = listData;
        listData = listData->next;
        delete tempPtr;
    }
}
```

See how this class destructor differs from an operation that makes a list empty? The class destructor releases the space that was allocated to the array but does not set length to 0. Making a structure empty is a logical operation defined in an ADT specification, but a class destructor is an implementation level operation. Implementations that do not use dynamically allocated space usually do not need destructors. However, implementations that use dynamically allocated space almost always need a destructor.
3.5 Comparing Unsorted List Implementations

Now let’s compare the sequential and linked implementations of the Unsorted List ADT. In order to determine the Big-O notation for the complexity of these functions, we must first determine the size factor. Here we are considering algorithms to manipulate items in a list. Therefore the size factor is the number of items on the list: \( \text{length} \).

We look at two different factors: the amount of memory required to store the structure and the amount of work the solution does. An array variable of the maximum list size takes the same amount of memory, no matter how many array slots are actually used, because we need to reserve space for the maximum possible (\( \text{MAX_ITEMS} \)). The linked implementation using dynamically allocated storage space requires only enough space for the number of elements actually in the list at run time. However, each node element is larger, because we must store the link (the next member) as well as the user’s data.

We can use Big-O notation to compare the efficiency of the two implementations. Most of the operations are nearly identical in the two implementations. The class constructor, \( \text{IsFull} \), \( \text{Reset} \), and \( \text{GetNextItem} \) functions in both implementations clearly have \( O(1) \) complexity because they are not dependent on the number of items in the list. \( \text{MakeEmpty} \) is an \( O(1) \) operation for a sequential list but becomes an \( O(N) \) operation for a linked list in dynamic storage. The sequential implementation merely marks the list as empty, while the linked implementation must actually access each list element to free its dynamically allocated space.

\( \text{GetLength} \) is always \( O(1) \) in an array-based implementation, but we have a choice in the linked version. We chose to make it \( O(1) \) by keeping a counter of the number of elements we insert and delete. If we had chosen to implement \( \text{GetLength} \) by counting the number of elements each time the function is called, the operation would be \( O(N) \). The moral here is that you must know how \( \text{GetLength} \) is implemented in a linked implementation in order to specify its Big-O measure.

The \( \text{RetrieveItem} \) operations are virtually identical for the two implementations. Beginning at the first element, the algorithms examine one element after another until the correct element is found. Because they must potentially search through all the elements in a list, the loops in both implementations are \( O(N) \).

Because the list is unsorted, we can choose to put the new item into a directly accessible place; the last position in the array-based implementation or the front in the linked version. Therefore, the complexity of \( \text{InsertItem} \) is the same in both implementations: \( O(1) \). In both implementations, \( \text{DeleteItem} \) has \( O(N) \) because the list must be searched for the item to delete, even though the actual removal is \( O(1) \) in the linked version and \( O(N) \) in the array-based version.

Table 3.2 summarizes these complexities. We have replaced \( \text{length} \) with \( N \), the generic name for the size factor. For those operations that require an initial search, we break the Big-O into two parts: the search and what is done following the search.
Chapter 3: ADT Unsorted List

Table 3.2  Big-O Comparison of Sorted List Operations

<table>
<thead>
<tr>
<th></th>
<th>Array Implementation</th>
<th>Linked Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class constructor</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>MakeEmpty</td>
<td>O(1)</td>
<td>O(N)</td>
</tr>
<tr>
<td>IsFull</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>GetLength</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>ResetList</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>GetNextItem</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>RetrieveItem</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>InsertItem</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Find</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Insert</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Combined</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>DeleteItem</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Find</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Delete</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Combined</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
</tbody>
</table>

Summary

In this chapter, we created an abstract data type that represents a list. The Unsorted List ADT assumes that the list elements are not sorted by key. We have viewed it from three perspectives: the logical level, the application level, and the implementation level.

We have presented two quite different implementations of the class that encapsulates the Unsorted List ADT. The first is an array-based implementation in which the items in the list are stored in an array; the second is a linked-list implementation in which the items are stored in nodes that are linked together. We examined the C++ pointer data type, which allows the nodes in a linked list to be chained.

In order to make the software as reusable as possible, the specification of each ADT states that the user of the ADT must prepare a class that defines the objects to be in each container class. A member function ComparedTo that compares two objects of this class must be included in the definition. This function returns one of the constants in RelationType: LESS, EQUAL, GREATER. By requiring the user to provide this information about the objects on the list, the code of the ADTs is very general.

The operations on the two ADTs were compared using Big-O notation. Insertion into an unsorted list has O(1) in both implementations. Deletions from both have O(N). Searching in the unsorted list has O(N); searching in a sorted list has order O(log₂N) if a binary search is used.
Exercises

1. The Unsorted List ADT is to be extended with a Boolean member function, IsThere, which takes as a parameter an item of type ItemType and determines whether there is an element with this key in the list.
   a. Write the specifications for this function.
   b. Write the prototype for this function.
   c. Write the function definition using an array-based implementation.
   d. Write the function definition using a linked implementation.
   e. Describe this function in terms of Big-O.

2. Rather than enhancing the Unsorted List ADTs by adding a member function IsThere, you decide to write a client function to do the same task.
   a. Write the specifications for this function.
   b. Write the function definition.
   c. Write a paragraph comparing the client function and the member function (Exercise 1) for the same task.
   d. Describe this function in terms of Big-O.

3. An Unsorted Type ADT is to be extended by the addition of function SplitLists, which has the following specifications:

   SplitLists(UnsortedType list, ItemType item, UnsortedType& list1, UnsortedType& list2)

   Function: Divides list into two lists according to the key of item.
   Preconditions: list has been initialized and is not empty.
   Postconditions: list1 contains all the items of list whose keys are less than or equal to item’s key; list2 contains all the items of list whose keys are greater than item’s key.

   a. Implement SplitLists as an array-based member function of the Unsorted List ADT.
   b. Implement SplitLists as a linked member function of the Unsorted List ADT.

4. Implement SplitLists described in Exercise 3 as a client function.

5. The specifications for the Unsorted List ADT state that the item to be deleted is in the list.
Chapter 3: ADT Unsorted List

a. Rewrite the specification for `DeleteItem` so that the list is unchanged if the item to be deleted is not in the list.

b. Implement `DeleteItem` as specified in (a) using an array-based implementation.

c. Rewrite the specification for `DeleteItem` so that all copies of the item to be deleted are removed if they exist.

d. Implement `DeleteItem` as specified in (c) using an array-based implementation.

6. Redo Exercise 5(b) and 5(d), using a linked implementation.

7. a. Explain the difference between a sequential and a linked representation of a list.

b. Give an example of a problem for which a sequential list would be the better solution.

c. Give an example of a problem for which a linked list would be the better solution.

8. True or False? If you answer False, correct the statement.

a. An array is a random-access structure.

b. A sequential list is a random-access structure.

c. A linked list is a random-access structure.

d. A sequential list is always stored in a statically allocated structure.

Use the linked list pictured below for Exercises 9 through 12.

9. Give the values of the following expressions:

a. `ptr1->info`

b. `ptr2->next->info`

c. `listData->next->next->info`

10. Are the following expressions true or false?

a. `listData->next == ptr1`

b. `ptr1->next->info == 60`

c. `ptr2->next == NULL`

d. `listData->info == 25`
11. Decide whether the syntax of each of the following statements is valid or invalid. If it is valid, mark it OK; if it is invalid, explain what is wrong.
   a. listData->next = ptr1->next;
   b. listData->next = *(ptr2->next);
   c. *listData = ptr2;
   d. ptr2 = ptr1->next->info;
   e. ptr1->info = ptr2->info;
   f. ptr2 = ptr2->next->next;

12. Write one statement to do each of the following:
   a. Make listData point to the node containing 45.
   b. Make ptr2 point to the last node in the list.
   c. Make listData point to an empty list.
   d. Set the info member of the node containing 45 to 60.

If memory locations are allocated as shown in the second column of the following table, what is printed by the statements in the first column? Fill in the last column in the following table for Exercises 13 through 18. The exercise number is in the first column in comments.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Memory Allocated</th>
<th>What Is Printed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>int value;</td>
<td>value is assigned to location 200</td>
<td></td>
</tr>
<tr>
<td>value = 500;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>char* charPtr;</td>
<td>charPtr is at location 202</td>
<td></td>
</tr>
<tr>
<td>char string[10] = &quot;Good luck&quot;;</td>
<td>string[0] is at location 300</td>
<td></td>
</tr>
<tr>
<td>charPtr = string;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cout &lt;&lt; &amp;value; // Exercise 13</td>
<td>&amp; means &quot;the address of&quot;</td>
<td></td>
</tr>
<tr>
<td>cout &lt;&lt; value; // Exercise 14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cout &lt;&lt; &amp;charPtr; // Exercise 15</td>
<td>&amp; means &quot;the address of&quot;</td>
<td></td>
</tr>
<tr>
<td>cout &lt;&lt; charPtr; // Exercise 16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cout &lt;&lt; *charPtr; // Exercise 17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cout &lt;&lt; string[2]; // Exercise 18</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

13. An Unsorted List ADT is to be extended by the addition of a member function `Head`, which has the following precondition and postcondition:
   **Precondition**: list has been initialized and is not empty.
   **Postcondition**: return value is the last item inserted in the list.
Chapter 3: ADT Unsorted List

14. An Unsorted List ADT is to be extended by the addition of function Tail, which has the following precondition and postcondition:

Precondition: list has been initialized and is not empty.
Postcondition: return value is a new list with the last item inserted in the list removed.

a. Will this addition be easy to implement in the array-based class UnsortedType? Explain.

b. Will this addition be easy to implement in the linked implementation of class UnsortedType? Explain.

15. DeleteItem does not maintain the order of insertions because the algorithm swaps the last item into the position of the one being deleted and then decrements length. Would there be any advantage to having DeleteItem maintain the insertion order? Justify your answer.

16. Give a Big-O estimate of the run time for the functions you wrote in Exercises 5 and 6.

17. a. Change the specifications for the array-based Unsorted List ADT so that InsertItem throws an exception if the list is full.

b. Implement the revised specifications in (a).

18. Based on the following declarations, tell whether each statement below is syntactically legal (yes) or illegal (no).

```plaintext
int* p;
int* q;
int* r;
int a;
int b;
int c:
```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>f</td>
<td>k</td>
</tr>
<tr>
<td>b</td>
<td>g</td>
<td>l</td>
</tr>
<tr>
<td>c</td>
<td>h</td>
<td>m</td>
</tr>
<tr>
<td>d</td>
<td>i</td>
<td>n</td>
</tr>
<tr>
<td>e</td>
<td>j</td>
<td>o</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Yes</th>
<th></th>
<th>Yes</th>
<th></th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>p = new int;</td>
<td>b</td>
<td>q* = new int;</td>
<td>c</td>
<td>a = new int;</td>
</tr>
<tr>
<td>b</td>
<td>q = NULL;</td>
<td>c</td>
<td>a = *p;</td>
<td>d</td>
<td>p = r;</td>
</tr>
<tr>
<td>c</td>
<td>b = r;</td>
<td>e</td>
<td>q = b;</td>
<td>f</td>
<td>r = NULL;</td>
</tr>
<tr>
<td>d</td>
<td>q = &amp;c;</td>
<td>g</td>
<td>c = *p;</td>
<td>h</td>
<td>p = *a;</td>
</tr>
<tr>
<td>e</td>
<td>c = NULL;</td>
<td>i</td>
<td>delete b;</td>
<td>j</td>
<td>q = &amp;c;</td>
</tr>
<tr>
<td>f</td>
<td>q = new int;</td>
<td>g</td>
<td>c = *p;</td>
<td>h</td>
<td>p = *a;</td>
</tr>
<tr>
<td>g</td>
<td>q = &amp;c;</td>
<td>i</td>
<td>delete b;</td>
<td>j</td>
<td>q = &amp;c;</td>
</tr>
</tbody>
</table>
19. The following program has careless errors on several lines. Find and correct the errors and show the output where requested.

```cpp
#include <iostream>

int main ()
{
    int* ptr;
    int* temp;
    int x;

    ptr = new int;
    *ptr = 4;
    *temp = *ptr;
    cout << ptr << temp;
    x = 9;
    *temp = x;
    cout << *ptr << *temp; // output:_________________
    ptr = new int;
    ptr = 5;
    cout << *ptr << *temp; // output:_________________
    return 0;
}
```