Singly Linked List Implementation of the List ADT

objectives

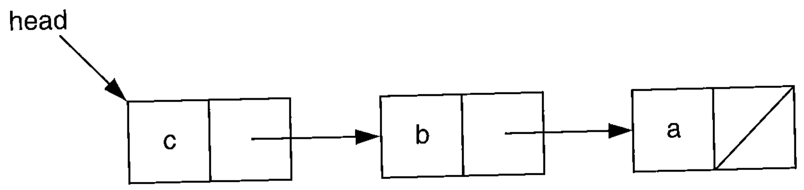
In this laboratory you

- implement the List ADT using a singly linked list.
- learn how to implement linked data structures using C++ pointers.
- examine how a fresh perspective on insertion and deletion can produce more efficient linked list implementations of these operations.
- learn how to use C++ inner classes.
- analyze the efficiency of your singly linked list implementation of the List ADT.

ADT Overview

In Laboratory 3, you created an implementation of the List ADT using an array to store the list data items. Although this approach is intuitive, it is not terribly efficient either in terms of memory usage or time. It wastes memory by allocating an array that is large enough to store what you estimate to be the maximum number of data items a list will ever hold. In most cases, the list is rarely this large and the extra memory simply goes unused. In addition, the insertion and deletion operations require shifting data items back and forth within the array, a very inefficient task.

In this laboratory, you implement the List ADT using a singly linked list. This implementation allocates memory data-item-by-data-item as data items are added to the list. In this way, you only allocate memory when you actually need it. Because memory is allocated over time, however, the data items do not occupy a contiguous set of memory locations. As a result, you need to link the data items together to form a linked list, as shown in the following figure.



Equally important, a linked list can be reconfigured following an insertion or deletion simply by changing one or two links.

C++ Concepts Overview

Linked data structures: To implement a linked list, we need a mechanism to connect the individual data items together; pointers are the C++ method. A pointer refers to a specific location in memory where another object is stored[1]. Each pointer has a specific type that limits the type of the objects it is allowed to refer to.

Each element in the list is called a node and contains the data item and a pointer to the next node. The last node in the list has no successor, which raises the problem of what we should do with the node's pointer when there is no successor. The standard solution in computer science is to have a special value called null, which indicates that the pointer doesn't point to an object.

In contrast to the array-based list implementation, the default linked list constructor does not allocate any memory for data items; individual nodes are created as they are needed. In a similar manner, the destructor must destroy each of the nodes, one-by-one, to ensure that all dynamically allocated memory is returned to the memory manager.

Inner classes: The singly-linked list is composed of two classes, the List class and the ListNode class. There are two choices about where to declare the ListNode:

1. Using the techniques that we have learned, we can create the ListNode as a separate ADT. This becomes problematic because the List needs access to the

[1]Technically, a pointer can refer to objects, built-in data types, or even functions. When we say "object" when talking about what a pointer references, we intend it in the most generic sense.

ListNode, but no other classes should be allowed access. Restricting access to the ListNode can be accomplished by declaring the List class to be a friend of the ListNode class and placing the entire ListNode implementation inside the private section of the ListNode.

2. A more elegant solution involves declaring the ListNode class inside the private section of the List class. That provides encapsulation against external access, but still provides the List class with proper access.

List ADT Specification

Data items

The data items in a list are of generic type DataType.

Structure

The data items form a linear structure in which list data items follow one after the other, from the beginning of the list to its end. The ordering of the data items is determined by when and where each data item is inserted into the list and is *not* a function of the data contained in the list data items. At any point in time, one data item in any nonempty list is marked using the list's cursor. You travel through the list using operations that change the position of the cursor.

ListNode Operations

```
ListNode( const DataType& nodeData, ListNode* nextPtr )
```

Requirements:
None

Results:
Constructor. Creates an initialized ListNode by setting the ListNode's data item to the value `nodeData` and the ListNode's next pointer to the value of `nextPtr`.

List Operations

```
List ( int ignored = 0 )
```

Requirements:
None

Results:
Constructor. Creates an empty list. The parameter is provided for call compatibility with the array implementation and is ignored.

`List ( const List& other )`

Requirements:
None

Results:
Copy constructor. Initializes the list to be equivalent to the `other` List.

`List& operator= ( const List& other )`

Requirements:
None

Results:
Overloaded assignment operator. Sets the list to be equivalent to the `other` List and returns a reference to this object.

`~List ()`

Requirements:
None

Results:
Destructor. Deallocates (frees) the memory used to store the nodes in the list.

`void insert ( const DataType& newDataItem ) throw ( logic_error )`

Requirements:
List is not full.

Results:
Inserts `newDataItem` into the list. If the list is not empty, then inserts `newDataItem` after the cursor. Otherwise, inserts `newDataItem` as the first (and only) data item in the list. In either case, moves the cursor to `newDataItem`.

`void remove () throw ( logic_error )`

Requirements:
List is not empty.

Results:
Removes the data item marked by the cursor from the list. If the resulting list is not empty, then moves the cursor to the data item that followed the deleted data item. If the deleted data item was at the end of the list, then moves the cursor to the beginning of the list.

`void replace ( const DataType& newDataItem ) throw ( logic_error )`

Requirements:
List is not empty.

Results:
Replaces the data item marked by the cursor with `newDataItem`. The cursor remains at `newDataItem`.

```
void clear ()
```

Requirements:
None

Results:
Removes all the data items in the list.

```
bool isEmpty () const
```

Requirements:
None

Results:
Returns `true` if the list is empty. Otherwise, returns `false`.

```
bool isFull () const
```

Requirements:
None

Results:
Returns `true` if the list is full. Otherwise, returns `false`. (Note: the implementation notes discuss the issue of what it means to say that a list of dynamically allocated nodes is full.)

```
void gotoBeginning () throw ( logic_error )
```

Requirements:
List is not empty.

Results:
Moves the cursor to the beginning of the list.

```
void gotoEnd () throw ( logic_error )
```

Requirements:
List is not empty.

Results:
Moves the cursor to the end of the list.

```
bool gotoNext () throw ( logic_error )
```

Requirements:
List is not empty.

Results:
If the cursor is not at the end of the list, then moves the cursor to mark the next data item in the list and returns `true`. Otherwise, returns `false`.

```
bool gotoPrior () throw ( logic_error )
```

Requirements:
List is not empty.

Results:
If the cursor is not at the beginning of the list, then moves the cursor to mark the preceding data item in the list and returns `true`. Otherwise, returns `false`.

```
DataType getCursor () const throw ( logic_error )
```

Requirements:
List is not empty.

Results:
Returns the value of the data item marked by the cursor.

```
void showStructure () const
```

Requirements:
None

Results:
Outputs the data items in the list. If the list is empty, outputs "Empty list". Note that this operation is intended for testing/debugging purposes only. It only supports list data items that are one of C++'s predefined data types (`int`, `char`, and so forth) or other data structures with an overridden ostream `operator<<`.

Implementation Notes

Pointers: There are two main operations associated with pointers: 1) access the object being pointed to (dereference the pointer), and 2) point to an object by obtaining the object's address. A pointer is dereferenced by prefixing the pointer name with the `*` operator. To obtain the address of an object, prefix the object name with the `&` operator.

When we dereference a pointer to a class instance, we usually are trying to access member data. The syntax for doing so is `(*pointer).member`. The pointer is first dereferenced to access the object, then the specific member in the object is accessed. Parentheses are needed because the '.' operator has higher precedence than the `*` operator. Dereferencing pointers to access object members is so common that C++ provides shortcut syntax: `pointer->member`.

Null pointers in C++ are indicated by the value 0. In C, null pointers are indicated by the identifier NULL. Many individuals learned C, then C++, and have carried over the habit of using NULL. This usually works, but can cause hard-to-identify problems in certain situations, so we recommend using 0 for null at all times.

The C++ compiler provides every object with a pointer to itself named `this` and can be used like any other pointer. We can uniquely identify an object using `this` because only one object can occupy a given memory location. We compare `this` to the address of other objects in the overloaded assignment operator in order to avoid trying to copy from ourselves because the first step is usually to delete the old data to make room for the new data. If we are copying from ourselves, deleting the old data also deletes the new data.

Inner classes: The ListNode class is declared inside the List class. For example,

```
class List {
      ...
    private:
      class ListNode {
          ...
     };
};
```

The practical consequence of having an inner class is that the scope resolution operator must include both the outer and inner class names. For example, when defining the ListNode constructor, the function should look like the following:

```
template <typename DataType>
List<DataType>::ListNode::ListNode(const DataType& nodeData,
                                   ListNode* nextPtr)
{
    // Your implementation here
}
```

Creating new ListNode objects: The ListNode class constructor is used to add nodes to the list. The statement below, for example, creates the first node in the list with `newDataItem` as its data member.

```
head = new ListNode<DataType>(newDataItem, 0);
```

Note: The null pointer indicates that the node has no successor.

The `new` operator allocates memory for a linked list node and calls the ListNode constructor, passing both the data item to be inserted (`newDataItem`) and a pointer to the next node in the list (`0`). Finally, the assignment operator assigns a pointer to the newly allocated node to `head`, thereby completing the creation of the node.

Running out of dynamically allocated (heap) memory: The memory manager does not have an infinite amount of memory available. If a process makes a very large number of memory requests (using the `new` operator) without returning enough memory (using `delete`), it is possible to run out of memory in the heap. Approaches to determining whether there is still available memory are generally non-trivial and implementation dependent. We wish to keep the linked implementation of the List interface-compatible with the array implementation, so we need to keep the `isFull` member function. Because your applications will probably not require much memory, we recommend that your implementation of the `isFull` function assume that there is always more memory available and always return `true`. Warning: although useful for the purposes of this book, this is not an appropriate solution in many cases.

Step 1: Implement the operations in the List ADT using a singly linked list. Each node in the linked list should contain a list data item (`dataItem`) and a pointer to the node containing the next data item in the list (`next`). Your implementation should also maintain pointers to the node at the beginning of the list (`head`) and the node containing the data item marked by the cursor (`cursor`). Base your implementation on the declarations from the file *ListLinked.h*. An implementation of the showStructure operation is given in the file *show5.cpp*. Please insert the contents of *show5.cpp* into your *ListLinked.cpp* file.

Step 2: Save your implementation of the List ADT in the file *ListLinked.cpp*. Be sure to document your code.

Compilation Directions

Compile *test5.cpp*. As in previous cases with templated classes, the test program directly includes the class definition (implementation in the *.cpp* file), so it is not necessary to compile the class separately.

Testing

Test your implementation of the List ADT using the test program in the file *test5.cpp*. This program allows you to interactively test your implementation of the List ADT using the following commands.

Command	Action
+x	Insert data item *x* after the cursor.
–	Remove the data item marked by the cursor.
=x	Replace the data item marked by the cursor with data item *x*.
@	Display the data item marked by the cursor.
N	Go to the next data item.
P	Go to the prior data item.
<	Go to the beginning of the list.
>	Go to the end of the list.
E	Report whether the list is empty.
F	Report whether the list is full.
C	Clear the list.
Q	Quit the test program.

Step 1: Download the online test plans for Lab 5.

Step 2: Complete Test Plan 5-1 by adding test cases that check whether you correctly implemented the List ADT operations.

Step 3: Execute Test Plan 5-1. If you discover mistakes in your implementation of the List ADT, correct them and execute your test plan again.

Step 4: Activate Test 1 by changing the value of LAB5_TEST1 from 0 to 1 in the *config.h* file. The second test changes the data type being used by the List from a character to an integer.

Step 5: Recompile the test program. Note that recompiling this program will compile your implementation of the List ADT to produce an implementation for a list of integers.

Step 6: Replace the character data in your test plan with integer values to create Test Plan 5-2.

Step 7: Complete and execute your Test Plan 5-2 using the revised test program. If you discover mistakes in your implementation of the List ADT, correct them and execute Test Plan 5-2 again.

Programming Exercise 1

List data items need not be one of C++'s built-in types. The following declaration, for example, represents a slide show presentation as a list of slides

```
List<Slide> slideShow;
```

where each slide is an object in the Slide class outlined here.

```
class Slide
{
  public:
    static const int HEIGHT = 10,          // Slide dimensions
                     WIDTH  = 36;
    void display () const;                 // Display slide and pause
  private:
    char image [HEIGHT][WIDTH];            // Slide image
    int pause;                             // Seconds to pause after
                                           //  displaying slide
    friend istream& operator>> (istream& in, Slide& slide);
    friend ostream& operator<< (ostream& out, const Slide& slide);
};
```

Step 1: Using the program shell given in the file *slideshow.cs* as a basis, create a program that reads a list of slides from a file and displays the resulting slide show from beginning to end. Your program should pause for the specified length of time after displaying each slide. It then should clear the screen (by scrolling, if necessary) before displaying the next slide.

Assume that the file containing the slide show consists of repetitions of the following slide descriptor,

Time
Row 1
Row 2
...
Row 10

where Time is the length of time to pause after displaying a slide (in seconds) and Rows 1–10 form a slide image (each row is 36 characters long).

Step 2: Test your program using Test Plan 5-3 and the slide show in the file *slides.dat*.

Programming Exercise 2

In many applications, the order of the data items in a list changes over time. Not only are new data items added and existing ones removed, but data items are repositioned within the list. The following List ADT operation moves a data item to the beginning of a list.

```
void moveToBeginning () throw ( logic_error )
```

Requirements:
List is not empty.

Results:
Removes the data item marked by the cursor from the list and reinserts the data item at the beginning of the list. Moves the cursor to the beginning of the list.

Step 1: Implement the operation described above and add it to the file *ListLinked.cpp*. A prototype for this operation is included in the declaration of the List class in the file *ListLinked.h*.

Step 2: Activate Test 2 in the test program *test5.cpp* by changing the definition of LAB5_TEST2 from 0 to 1 in *config.h* and recompiling.

Step 3: Complete Test Plan 5-4 by adding test cases that check whether your implementation of the moveToBeginning operation correctly processes requests for a number of scenarios. The test program uses `M` to execute the moveToBeginning operation.

Step 4: Execute Test Plan 5-4. If you discover mistakes in your implementation of the moveToBeginning operation, correct them and execute your test plan again.

Programming Exercise 3

Sometimes a more effective approach to a problem can be found by looking at the problem a little differently. Consider the following List ADT operation.

```
void insertBefore ( const DataType& newDataItem )
     throw ( logic_error )
```

Requirements:
List is not full.

Results:
Inserts `newDataItem` into a list. If the list is not empty, then inserts `newDataItem` immediately before the cursor. Otherwise, inserts `newDataItem` as the first (and only) data item in the list. In either case, moves the cursor to `newDataItem`.

You can implement this operation using a singly linked list in two very different ways. The obvious approach is to iterate through the list from its beginning until you reach the node immediately before the cursor and then to insert `newDataItem` between this node and the cursor. A more efficient approach is to copy the data item pointed to by the cursor into a new node, to insert this node after the cursor, and to place `newDataItem` in the node pointed to by the cursor. This approach is more efficient because it does not require you to iterate through the list searching for the data item immediately before the cursor.

Step 1: Implement the insertBefore operation using the second (more efficient) approach and add it to the file *ListLinked.cpp*. A prototype for this operation is included in the declaration of the List class in the file *ListLinked.h*.

Step 2: Activate Test 3 in the test program in *test5.cpp* by changing the definition of LAB5_TEST3 from 0 to 1 in *config.h* and recompiling.

Step 3: Complete Test Plan 5-5 by adding test cases that check whether your implementation of the insertBefore operation correctly handles insertions into single-data item lists and empty lists. The test program uses # to execute the insertBefore operation.

Step 4: Execute Test Plan 5-5. If you discover mistakes in your implementation of the insertBefore operation, correct them and execute your test plan again.

Analysis Exercise 1

Given a list containing N data items, develop worst-case, order-of-magnitude estimates of the execution time of the following List ADT operations, assuming they are implemented using a linked list. Briefly explain your reasoning behind each estimate.

insert O( )
Explanation:

remove O( )
Explanation:

gotoNext O( )
Explanation:

gotoPrior O( )
Explanation:

Analysis Exercise 2

Part A

Programming Exercise 3 introduces a pair of approaches for implementing an insertBefore operation. One approach is straightforward, whereas the other is somewhat less obvious but more efficient. Describe how you might apply the latter approach to the remove operation. Use a diagram to illustrate your answer.

Part B

The resulting implementation of the remove operation has a worst-case, order of magnitude performance estimate of O(N). Does this estimate accurately reflect the performance of this implementation? Explain why or why not.