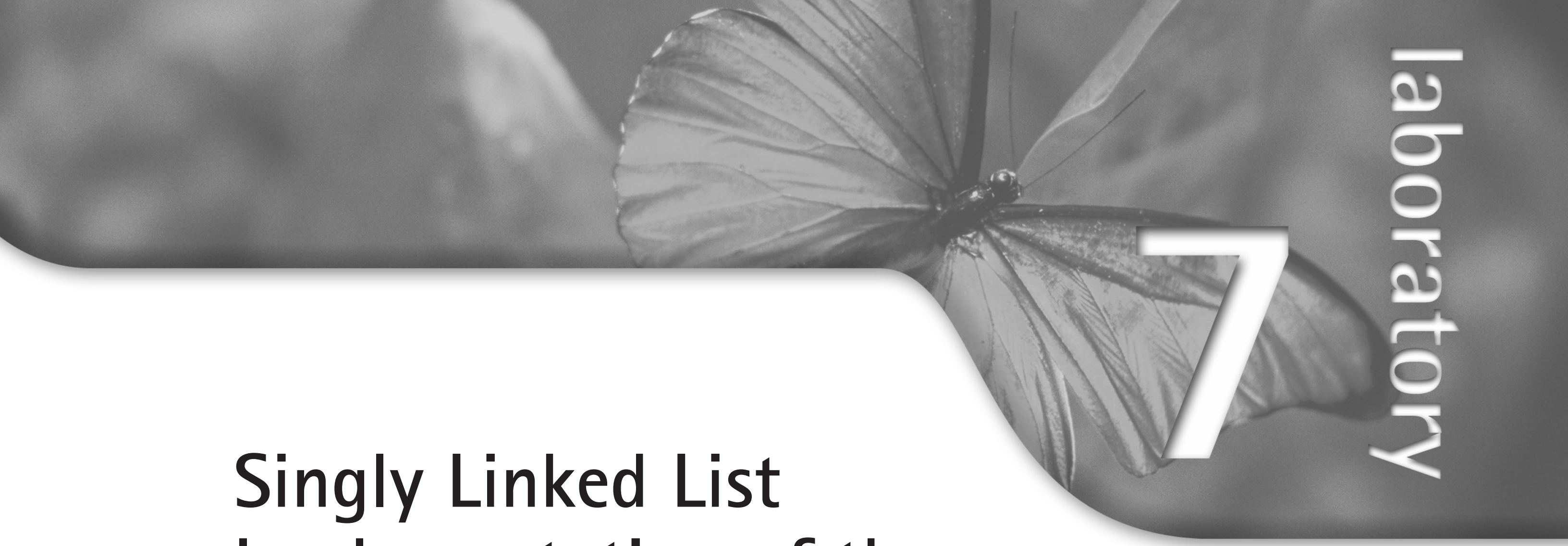

Singly Linked List Implementation of the List ADT

objectives

In this laboratory you will:

- Implement the List ADT using a singly linked list
- Create a program that displays a slide show
- Examine how a fresh perspective on insertion and deletion can produce more efficient linked list implementations of these operations
- Analyze the efficiency of your singly linked list implementation of the List 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 time-consuming 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. Equally important, a linked list can be reconfigured following an insertion or deletion simply by changing one or two links.

List ADT

Data Items

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

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.

Operations

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

Requirements:
None

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

```
~List ()
```

Requirements:
None

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

```
void insert ( const DT &newDataItem ) throw ( bad_alloc )
```

Requirements:
List is not full.

Results:
Inserts `newDataItem` into a 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 a 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 DT &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 a list.

```
bool isEmpty () const
```

Requirements:
None

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

```
bool isFull () const
```

Requirements:
None

Results:
Returns `true` if a list is full. Otherwise, returns `false`.

```
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 a 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 a list, then moves the cursor to mark the preceding data item in the list and returns `true`. Otherwise, returns `false`.

```
DT getCursor () const throw ( logic_error )
```

Requirements:
List is not empty.

Results:
Returns a copy of the data item marked by the cursor.

```
void showStructure () const
```

Requirements:
None

Results:
Outputs the data items in a list. If the list is empty, outputs “Empty list”. Note that this operation is intended for testing/debugging purposes only. It supports only list data items that are one of C++’s predefined data types (`int`, `char`, and so forth).

Laboratory 7: Cover Sheet

Name ______________________________ Date ____________________

Section ______________________________

Place a check mark in the *Assigned* column next to the exercises your instructor has assigned to you. Attach this cover sheet to the front of the packet of materials you submit following the laboratory.

Activities	**Assigned:** Check or list exercise numbers	**Completed**
Prelab Exercise		
Bridge Exercise		
In-lab Exercise 1		
In-lab Exercise 2		
In-lab Exercise 3		
Postlab Exercise 1		
Postlab Exercise 2		
Total		

Laboratory 7: Prelab Exercise

Name ______________________________ Date ____________________

Section ______________________________

Your linked list implementation of the List ADT uses a pair of classes, ListNode and List, to represent individual nodes and the overall list structure, respectively. If you are unfamiliar with this approach to linked lists, read the discussion in Laboratory 5.

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 also should 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 following declarations from the file *listlnk.h*. An implementation of the `showStructure` operation is given in the file *show7.cpp*.

```
template < class DT >          // Forward declaration of the List class
class List;

template < class DT >
class ListNode                 // Facilitator class for the List class
{
  private:

    // Constructor
    ListNode ( const DT &nodeData, ListNode *nextPtr );

    // Data members
    DT dataItem;       // List data item
    ListNode *next;    // Pointer to the next list node

  friend class List<DT>;
};

//-----------------------–

template < class DT >
class List
{
  public:

    // Constructor
    List ( int ignored = 0 );

    // Destructor
    ~List ();

    // List manipulation operations
    void insert ( const DT &newData ) throw ( bad_alloc );    // Insert after cursor
    void remove () throw ( logic_error );                     // Remove data item
    void replace ( const DT &newData ) throw ( logic_error ); // Replace data item
    void clear ();                                            // Clear list

    // List status operations
    bool isEmpty () const;                                    // List is empty
    bool isFull () const;                                     // List is full

    // List iteration operations
    void gotoBeginning () throw ( logic_error );              // Go to beginning
    void gotoEnd () throw ( logic_error );                    // Go to end
    bool gotoNext () throw ( logic_error );                   // Go to next data
                                                              //  item
    bool gotoPrior () throw ( logic_error );                  // Go to prior item

    DT getCursor () const throw ( logic_error );              // Return item

    // Output the list structure — used in testing/debugging
    void showStructure () const;

  private:

    // Data members
    ListNode<DT> *head,      // Pointer to the beginning of the list
                 *cursor;    // Cursor pointer
};
```

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

Laboratory 7: Bridge Exercise

Name ______________________________ Date ____________________

Section ______________________________

Check with your instructor whether you are to complete this exercise prior to your lab period or during lab.

The test program in the file *test7.cpp* 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: Compile and link the test program. Note that compiling this program will compile your linked list implementation of the List ADT (in the file *listlnk.cpp*) to produce an implementation for a list of characters.

Step 2: Complete the following test plan by adding test cases that check whether your implementation of the List ADT correctly determines whether a list is empty and correctly inserts data items into a newly emptied list.

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

Test Plan for the Operations in the List ADT

Test Case	*Commands*	*Expected Result*	*Checked*
Insert at end	`+a +b +c +d`	a b c **d**	
Travel from beginning	`< N N`	a b **c** d	
Travel from end	`> P P`	a **b** c d	
Delete middle data item	`-`	a **c** d	
Insert in middle	`+e +f +f`	a c e f **f** d	
Remove last data item	`>-`	**a** c e f f	
Remove first data item	`<-`	**c** e f f	
Display data item	`@`	Returns `c`	
Replace data item	`=g`	**g** e f f	
Clear the list	`C`	Empty list	

Note: The data item marked by the cursor is shown in **bold**.

Step 4: Change the list in the test program from a list of characters to a list of integers by replacing the declarations for `testList` and `testDataItem` with

```
List<int> testList(8);     // Test list
int testDataItem;          // List data item
```

Step 5: Recompile and relink 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 (‘a’-‘g’) in your test plan with integer values.

Step 7: Execute your revised test plan using the revised test program. If you discover mistakes in your implementation of the List ADT, correct them and execute your revised test plan again.

Laboratory 7: In-lab Exercise 2

Name ________________________________ Date ____________________

Section ________________________________

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 a list and reinserts the data item at the beginning of the list. Moves the cursor to the beginning of the list.

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

Step 2: Activate the 'M' (move) command in the test program in the file *test7.cpp* by removing the comment delimiter (and the character 'M') from the lines beginning with "//M".

Step 3: Complete the following test plan by adding test cases that check whether your implementation of the `moveToBeginning` operation correctly processes attempts to move the first data item in a list and also moves within a single-data item list.

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

Test Plan for the `moveToBeginning` Operation

Test Case	*Commands*	*Expected Result*	*Checked*
Set up list	`+a +b +c +d`	a b c **d**	
Move last data item	`M`	**d** a b c	
Move second data item	`N M`	**a** d b c	
Move third data item	`N N M`	**b** a d c	

Note: The data item marked by the cursor is shown in **bold**.

Laboratory 7: In-lab Exercise 3

Name ______________________________ Date ____________________

Section ______________________________

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 DT &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 *listlnk.cpp*. A prototype for this operation is included in the declaration of the List class in the file *listlnk.h*.

Step 2: Activate the '#' (insert before) command in the test program in the file *test7.cpp* by removing the comment delimiter (and the character '#') from the lines beginning with "//#".

Step 3: Complete the following test plan by adding test cases that check whether your implementation of the `insertBefore` operation correctly handles insertions into single data item lists and empty lists.

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

Test Plan for the `insertBefore` Operation

Test Case	*Commands*	*Expected Result*	*Checked*
Set up list	`+a +b +c`	a b **c**	
Insert in middle	`#d`	a b **d** c	
Cascade inserts	`#e`	a b **e** d c	
Insert after head	`P #f`	a **f** b e d c	
Insert as head	`P #g`	**g** a f b e c	

Note: The data item marked by the cursor is shown in **bold**.