Doubly Linked List Implementation of the List ADT

objectives

In this laboratory you will:

- Implement the List ADT using a doubly linked list
- Create an anagram puzzle program
- Reverse a linked list
- Analyze the efficiency of your doubly linked list implementation of the List ADT

Overview

The singly linked list implementation of the List ADT that you created in Laboratory 7 is quite efficient when it comes to insertion and movement from one node to the next. It is not nearly so efficient, however, when it comes to deletion and movement backward through the list. In this laboratory, you create an implementation of the List ADT using a circular, doubly linked list. This implementation performs most of the List ADT operations in constant time.

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 `dataItem` 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 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 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 9: 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 9: Prelab Exercise

Name ______________________________ Date ____________________

Section ______________________________

Each node in a doubly linked list contains a pair of pointers. One pointer points to the node that precedes the node (`prior`) and the other points to the node that follows the node (`next`). The resulting ListNode class is similar to the one you used in Laboratory 7.

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

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

    // Data members
    DT dataItem;        // List data item
    ListNode *prior,    // Pointer to the previous data item
             *next;     // Pointer to the next data item

  friend class List<DT>;
};
```

In a circular, doubly linked list, the nodes at the beginning and end of the list are linked together. The next pointer of the node at the end of the list points to the node at the beginning, and the prior pointer of the node at the beginning points to the node at the end.

Step 1: Implement the operations in the List ADT using a circular, doubly linked list. Base your implementation on the class declarations in the file *listdbl.h*. An implementation of the `showStructure` operation is given in the file *show9.cpp*.

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

Laboratory 9: 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 *test9.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: Prepare a test plan for your implementation of the List ADT. Your test plan should cover the application of each operation to data items at the beginning, middle, and end of lists (where appropriate). A test plan form follows.

Step 2: 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*

Laboratory 9: In-lab Exercise 1

Name ______________________________ Date ____________________

Section ______________________________

Lists can be used as data members in other classes. In this exercise, you will create an implementation of the Anagram Puzzle ADT described below using lists of characters to store both the solution to the puzzle and the current puzzle configuration.

Anagram Puzzle ADT

Data Items

Alphabetic characters.

Structure

The characters are arranged linearly. If rearranged properly they spell a specified English word.

Operations

`AnagramPuzzle ( char answ[], char init[] )`

Requirements:
Strings `answ` and `init` are nonempty and contain the same letters (but in a different order).

Results:
Constructor. Creates an anagram puzzle. String `answ` is the solution to the puzzle and string `init` is the initial scrambled letter sequence.

`void shiftLeft ()`

Requirements:
None

Results:
Shifts the letters in a puzzle left one position. The leftmost letter is moved to the right end of the puzzle.

`void swapEnds ()`

Requirements:
None

Results:
Swaps the letters at the left and right ends of a puzzle.

`void display ()`

Requirements:
None

Results:
Displays an anagram puzzle.

`bool solved ()`

Requirements:
None

Results:
Returns `true` if a puzzle is solved. Otherwise returns `false`.

The following code fragment declares a puzzle in which the word “yes” is scrambled as “yse”. It then shows how the puzzle is unscrambled to form “yes”.

```
AnagramPuzzle enigma(“yes”,”yse”);   // Word is “yes”, start w/ “yse”
enigma.shiftLeft();                  // Changes puzzle to “sey”
enigma.swapEnds();                   // Changes puzzle to “yes”
```

Rather than having the solution to the puzzle encoded in the program, your puzzle program allows the user to solve the puzzle by entering commands from the keyboard.

Step 1: Complete the anagram puzzle program shell given in the file *puzzle.cs* by creating an implementation of the Anagram Puzzle ADT. Base your implementation on the following declaration.

```
class AnagramPuzzle
{
  public:

    AnagramPuzzle( char answ[], char init[] );   // Construct puzzle
    void shiftLeft();                            // Shift letters left
    void swapEnds();                             // Swap end letters
    void display();                              // Display puzzle
    bool isSolved();                             // Puzzle solved

  private:

    // Data members
    List<char> solution,   // Solution to puzzle
               puzzle;     // Current puzzle configuration
};
```

Use your circular, doubly linked list implementation of the List ADT to represent the lists of characters storing the puzzle’s solution and its current configuration.

Step 2: Test your anagram puzzle program using the puzzles given in the following test plan.

Test Plan for the Anagram Puzzle Program

Test Case	*Checked*
Puzzle word “yes”, scrambled as “yse”	
Puzzle word “right”, scrambled as “irtgh”	

Laboratory 9: In-lab Exercise 2

Name ______________________________ Date ____________________

Section ______________________________

A list can be reversed in two ways: either you can relink the nodes in the list into a new (reversed) order, or you can leave the node structure intact and exchange data items between pairs of nodes. Use one of these strategies to implement the following List ADT operation.

```
void reverse ()
```

Requirements:
None

Results:
Reverses the order of the data items in a list. The cursor does not move.

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

Step 2: Activate the 'R' (reverse) command in the test program in the file *test9.cpp* by removing the comment delimiter (and the character 'R') from the lines that begin with "//R".

Step 3: Prepare a test plan for the reverse operation that covers lists of various lengths, including lists containing a single data item. A test plan form follows.

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

Test Plan for the Reverse Operation

Test Case	*Commands*	*Expected Result*	*Checked*

Laboratory 9: In-lab Exercise 3

Name ______________________________ Date __________________

Section ______________________________

In many list applications you need to know the number of data items in a list and the relative position of the cursor. Rather than computing these attributes each time they are requested, you can store this information in a pair of data members that you update whenever you insert data items, remove data items, or move the cursor.

Step 1: Add the following data members (both are of type `int`) to the List class declaration in the file *listdbl.h* and save the result in the file *listdbl2.h*.

`size`:	The number of data items in a list.
`pos`:	The numeric position of the cursor, where the list data items are numbered from beginning to end, starting with 0.

Step 2: Modify the routines in your circular, doubly linked list implementation of the List ADT so that they update these data members whenever necessary. Save your modified implementation in the file *listdbl2.cpp*.

Step 3: If you are to reference the `size` and `pos` data members within applications programs, you must have List ADT operations that return these values. Add prototypes for the following operations to the List class declaration in the file *listdbl2.h*.

```
int getLength () const
```

Requirements:
None

Results:
Returns the number of data items in a list.

```
int getPosition () const throw ( logic_error )
```

Requirements:
List is not empty.

Results:
Returns the position of the cursor, where the list data items are numbered from beginning to end, starting with 0.

Step 4: Implement these operations and add them to the file *listdbl2.cpp*.

Step 5: Modify the test program in the file *test9.cpp* so that the routines that incorporate your changes (in *listdbl2.cpp*) are included in place of those you created in the Prelab.

Step 6: Activate the '#' (length and position) command by removing the comment delimiter (and the character '#') from the lines that begin with "//#".

Step 7: Prepare a test plan for these operations that checks the length of various lists (including the empty list) and the numeric position of data items at the beginning, middle, and end of lists. A test plan form follows.

Step 8: Execute your test plan. If you discover mistakes in your implementation of these operations, correct them and execute your test plan again.

Test Plan for the Length and Position Operations

Test Case	*Commands*	*Expected Result*	*Checked*