Recursion with Linked Lists

objectives

In this laboratory you will:

- Examine how recursion can be used to traverse a linked list in either direction
- Use recursion to insert, delete, and move data items in a linked list
- Convert recursive routines to iterative form
- Analyze why a stack is sometimes needed when converting from recursive to iterative form

Overview

Recursive functions, or functions that call themselves, provide an elegant way of describing and implementing the solutions to a wide range of problems, including problems in mathematics, computer graphics, compiler design, and artificial intelligence. Let's begin by examining how you develop a recursive function definition, using the factorial function as an example.

You can express the factorial of a positive integer n using the following iterative formula:

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1$$

Applying this formula to 4! yields the product $4 \times 3 \times 2 \times 1$. If you regroup the terms in this product as $4 \times (3 \times 2 \times 1)$ and note that $3! = 3 \times 2 \times 1$, then you find that 4! can be written as $4 \times (3!)$. You can generalize this reasoning to form the following recursive definition of factorial:

$$n! = n \cdot (n - 1)!$$

where 0! is defined to be 1. Applying this definition to the evaluation of 4! yields the following sequence of computations.

$$\begin{aligned} 4! &= 4 \cdot (\,3!\,) \\ &= 4 \cdot (\,3 \cdot (\,2!\,)\,) \\ &= 4 \cdot (\,3 \cdot (\,2 \;\; (\,1!\,)\,)\,) \\ &= 4 \cdot (\,3 \cdot (\,2 \;\; (\,1 \cdot (\,0!\,)\,)\,)\,) \\ &= 4 \cdot (\,3 \cdot (\,2 \;\; (\,1 \cdot (\,1\,)\,)\,)\,) \end{aligned}$$

The first four steps in this computation are recursive, with $n!$ being evaluated in terms of $(n - 1)!$. The final step $(0! = 1)$ is not recursive, however. The following notation clearly distinguishes between the **recursive step** and the nonrecursive step (or **base case**) in the definition of $n!$.

$$n! = \begin{cases} 1 & \text{if } n = 0 \text{ (base case)} \\ n \bullet (n-1)! & \text{if } n > 0 \text{ (recursive step)} \end{cases}$$

The following `factorial()` function uses recursion to compute the factorial of a number.

```
long factorial ( int n )
// Computes n! using recursion.
{
    long result;   // Result returned

    if ( n == 0 )
        result = 1;                    // Base case
    else
        result = n * factorial(n-1);   // Recursive step
    return result;
}
```

Let's look at the call `factorial(4)`. Because 4 is not equal to 0 (the condition for the base case), the `factorial()` function issues the recursive call `factorial(3)`. The recursive calls continue until the base case is reached—that is, until n equals 0.

```
factorial(4)
     ↓ RECURSIVE STEP
    4*factorial(3)
          ↓ RECURSIVE STEP
          3*factorial(2)
                ↓ RECURSIVE STEP
                2*factorial(1)
                      ↓ RECURSIVE STEP
                      1*factorial(0)
                            ↓ BASE CASE
                            1
```

The calls to `factorial()` are evaluated in the reverse of the order they are made. The evaluation process continues until the value 24 is returned by the call `factorial(4)`.

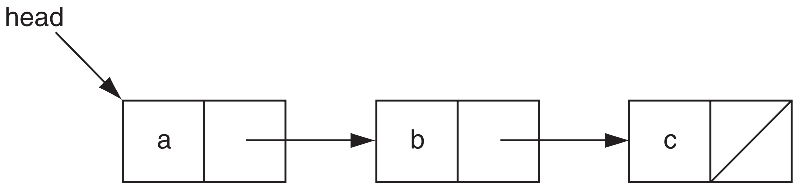
```
factorial(4)
       ↑ RESULT 24
    4*factorial(3)
             ↑ RESULT 6
          3*factorial(2)
                   ↑ RESULT 2
                2*factorial(1)
                         ↑ RESULT 1
                      1*factorial(0)
                              ↑ RESULT 1
                            1
```

Recursion can be used for more than numerical calculations, however. The following pair of functions traverse a linked list, outputting the data items encountered along the way.

```
template < class DT >
void List<DT>:: write () const

// Outputs the data items in a list from beginning to end. Assumes that
// objects of type DT can be output to the cout stream.

{
    cout << "List : ";
    writeSub(head);
    cout << endl;
}

// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

template < class DT >
void List<DT>:: writeSub ( ListNode<DT> *p ) const

// Recursive partner of the write() function. Processes the sublist
// that begins with the node pointed to by p.
```

```
{
    if ( p != 0 )
    {
       cout << p->dataItem;       // Output data item
       writeSub(p->next);         // Continue with next node
    }
}
```

The role of the `write()` function is to initiate the recursive process, which is then carried forward by its recursive partner the `writeSub()` function. Calling `write()` with the linked list of characters



yields the following sequence of calls and outputs "`abc`".

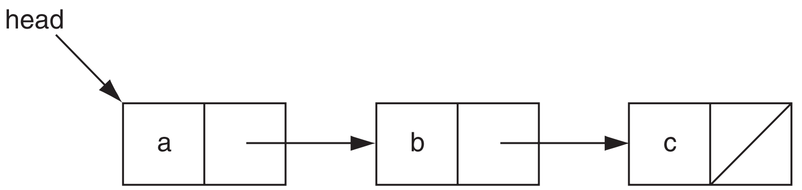
`writeSub(head)`
↓ RECURSIVE STEP
Output 'a' `writeSub(p->next)`
↓RECURSIVE STEP
Output 'b' `writeSub(p->next)`
↓RECURSIVE STEP
Output 'c' `writeSub(p->next)`
↓ BASE CASE
No output

Recursion also can be used to add nodes to a linked list. The following pair of functions insert a data item at the end of a list.

```
template < class DT >
void List<DT>:: insertEnd ( const DT &newDataItem )

// Inserts newDataItem at the end of a list. Moves the cursor to
// newDataItem.

{
    insertEndSub(head,newDataItem);
}

// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

template < class DT >
void List<DT>:: insertEndSub ( ListNode<DT> *&p,
                               const DT &newDataItem )

// Recursive partner of the insertEnd() function. Processes the
// sublist that begins with the node pointed to by p.

{
    if ( p != 0 )
       insertEndSub(p->next,newDataItem);    // Continue searching for
```

```
    else                                        // end of list
    {
       p = new ListNode<DT>(newDataItem,0);   // Insert new node
       cursor = p;                              // Move cursor
    }
}
```

The `insertEnd()` function initiates the insertion process, with the bulk of the work being done by its recursive partner, the `insertEndSub()` function. Calling `insertEnd()` to insert the character '`!`' at the end of the following list of characters:



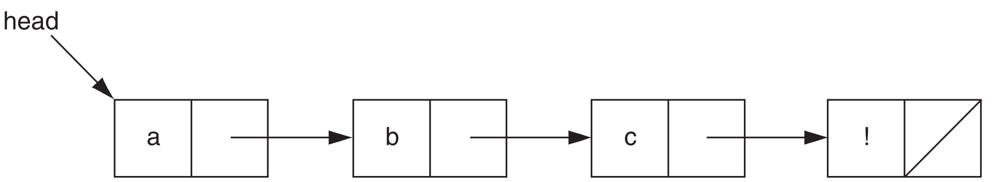
yields the following sequence of calls.

insertEndSub(head)
↓RECURSIVE STEP
insertEndSub(p->next)
↓RECURSIVE STEP
insertEndSub(p->next)
↓RECURSIVE STEP
insertEndSub(p->next)
↓BASE CASE
Create a new node containing '`!`'

On the last call, `p` is null and the statement

```
p = new ListNode<LE>(newDataItem,0);    // Insert new node
```

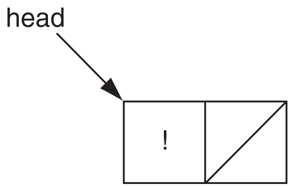
is executed to create a new node containing the character '`!`'. The address of this node is then assigned to `p`. Because `p` is passed using call by reference, this assignment changes the next pointer of the last node in the list ('`c`') to point to the new node, thereby producing the following list:



Calling `insertEnd()` to insert the character '`!`' into an empty list results in a single call to the `insertEndSub()` function.

insertEndSub(head)
↓RECURSIVE STEP
Create a new node containing '`!`'

In this case, assigning the address of the newly created node to `p` changes the list's `head` pointer to point to this node.



Note that the `insertEnd()` function automatically links the node it creates into either an existing list or an empty list without the use of special tests to determine whether the insertion changes a node's `next` pointer or the list's `head` pointer. The key is that parameter `p` is passed using call by reference.

Laboratory 10: 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 10: Prelab Exercise

Name ____ Date ________________________

Section ____

We begin by examining a set of recursive functions that perform known tasks. These functions are collected in the file *listrec.cs*. You can execute them using the test program in the file *test10.cpp*.

Part A

Step 1: To complete this laboratory, you need to use some of the functions from your singly linked list implementation of the List ADT. Complete the partial implementation of the List ADT in the file *listrec.cs* by adding the following functions from the linked list implementation you developed in Laboratory 7:

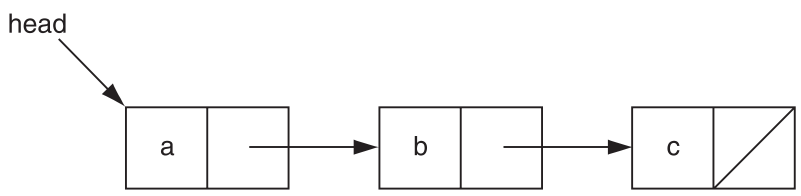
- The constructor for the ListNode class.
- The List class constructor, destructor, `insert()`, `clear()`, and `showStructure()` functions. Add any other functions that these depend on.

Prototypes for these functions are included in the declaration of the List class in the file *listrec.h*. Add prototypes for any other functions as needed.

Step 2: Save the resulting implementation in the file *listrec.cpp*.

Step 3: Activate the calls to the `write()` and `insertEnd()` functions in the test program in the file *test10.cpp* by removing the comment delimiter (and the characters '`PA`') from the lines beginning with "`//PA`".

Step 4: Execute the `write()` and `insertEnd()` functions using the following list.



Step 5: What output does `write()` produce?

Step 6: What list does `insertEnd()` produce?

Step 7: Execute these functions using an empty list.

Step 8: What output does `write()` produce?

Step 9: What list does `insertEnd()` produce?

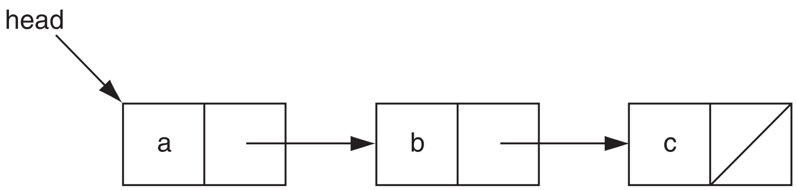
Part B

One of the most common reasons to use recursion with linked lists is to support traversal of a list from its end back to its beginning. The following pair of functions outputs each list data item twice, once as the list is traversed from beginning to end and again as it is traversed from the end back to the beginning.

```
template < class DT >
void List<DT>:: writeMirror () const

// Outputs the data items in a list from beginning to end and back
// again. Assumes that objects of type DT can be output to the cout
// stream.

{
    cout << “Mirror : “;
    writeMirrorSub(head);
    cout << endl;
}

// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

template < class DT >
void List<DT>:: writeMirrorSub ( ListNode<DT> *p ) const

// Recursive partner of the writeMirror() function. Processes the
// sublist that begins with the node pointed to by p.

{
    if ( p != 0 )
    {
       cout << p->dataItem;         // Output data item (forward)
       writeMirrorSub(p->next);     // Continue with next node
       cout << p->dataItem;         // Output data item (backward)
    }
}
```

Step 1: Activate the call to the `writeMirror()` function in the test program in the file *test10.cpp* by removing the comment delimiter (and the characters ‘PB’) from the lines beginning with “//PB”.

Step 2: Execute the `writeMirror()` function using the following list.



Step 3: What output does `writeMirror()` produce?

Step 4: Describe what each statement in the `writeMirrorSub()` function does during the call in which parameter `p` points to the node containing `'a'`.

Step 5: What is the significance of the call to `writeMirrorSub()` in which parameter `p` is null?

Step 6: Describe how the calls to `writeMirrorSub()` combine to produce the “mirrored” output. Use a diagram to illustrate your answer.

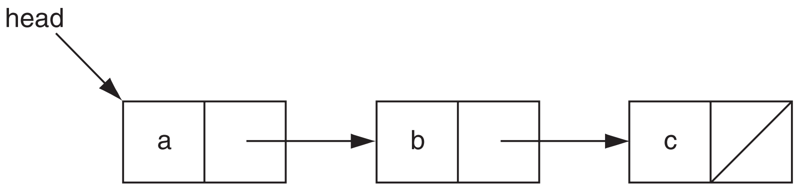
Part C

The following pair of functions reverse a list by changing each node's `next` pointer. Note that the pointers are changed on the way back through the list.

```
template < class DT >
void List<DT>:: reverse ()

// Reverses the order of the data items in a list.

{
    reverseSub(0,head);
}

// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

template < class DT >
void List<DT>:: reverseSub ( ListNode<DT> *p, ListNode<DT> *nextP )

// Recursive partner of the reverse() function. Processes the sublist
// that begins with the node pointed to by nextP.

{
    if ( nextP != 0 )
    {
       reverseSub(nextP,nextP->next);   // Continue with next node
       nextP->next = p;                 // Reverse link
    }
    else
       head = p;                        // Move head to end of list
}
```

Step 1: Activate the call to the `reverse()` function in the test program by removing the comment delimiter (and the characters '`PC`') from the lines beginning with "`//PC`".

Step 2: Execute the `reverse()` function using the following list.



Step 3: What list does `reverse()` produce?

Step 4: Describe what each statement in the `reverseSub()` function does during the call in which parameter `p` points to the node containing '`a`'. In particular, how are the links to and from this node changed as a result of this call?

Step 5: What is the significance of the call to `reverseSub()` in which parameter `p` is null?

Step 6: Describe how the calls to `reverseSub()` combine to reverse the list. Use a diagram to illustrate your answer.

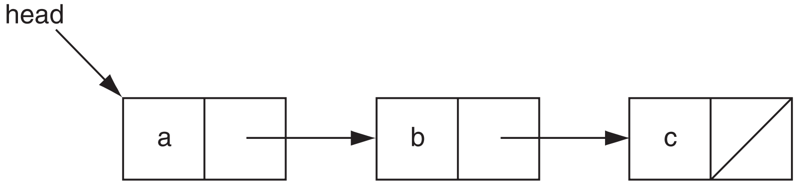
Part D

In the Overview, you saw how you can use recursion in conjunction with call by reference to insert a node at the end of a list. The following pair of functions use this technique to delete the last node in a list.

```
template < class DT >
void List<DT>:: deleteEnd ()

// Deletes the data item at the end of a list. Moves the cursor to the
// beginning of the list.

{
    deleteEndSub(head);
    cursor = head;
}

// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

template < class DT >
void List<DT>:: deleteEndSub ( ListNode<DT> *&p )

// Recursive partner of the deleteEnd() function. Processes the
// sublist that begins with the node pointed to by p.

{
    if ( p->next != 0 )
       deleteEndSub(p->next);   // Continue looking for the last node
    else
    {
       delete p;                // Delete node
       p = 0;                   // Set p (link or head) to null
    }
}
```

Step 1: Activate the call to the `deleteEnd()` function in the test program by removing the comment delimiter (and the characters 'PD') from the lines beginning with "//PD".

Step 2: Execute the `deleteEnd()` function using the following list.



Step 3: What list does `deleteEnd()` produce?

Step 4: What is the significance of the calls to the `deleteEndSub()` function in which `p->next` is not null?

Step 5: Describe what each statement in `deleteEndSub()` does during the call in which `p->next` is null. Use a diagram to illustrate your answer.

Step 6: What list does `deleteEnd()` produce when called with a list containing one data item? Describe how this result is accomplished. Use a diagram to illustrate your answer.

Part E

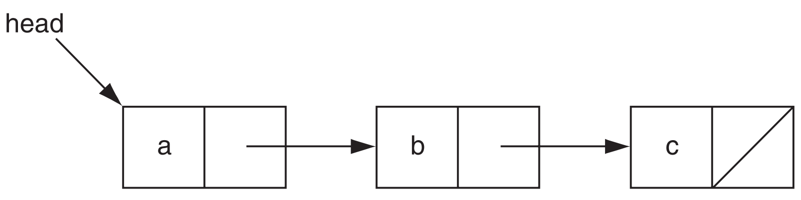
The following pair of functions determine the length of a list. These functions do not simply count nodes as they move through the list from beginning to end (as an iterative function would). Instead, they use a recursive definition of length in which the length of the list pointed to by pointer `p` is the length of the list pointed to by `p->next` (the remaining nodes in the list) plus one (the node pointed to by `p`).

$$length(\mathrm{p}) = \begin{cases} 0 & \text{if p = 0 (base case)} \\ \text{length(p->next)} + 1 & \text{if p} \neq 0 \text{ (recursive step)} \end{cases}$$

```
template < class DT >
int List<DT>:: getLength () const

// Returns the number of data items in a list.

{
    return getLengthSub(head);
}

// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

template < class DT >
int List<DT>:: getLengthSub ( ListNode<DT> *p ) const

// Recursive partner of the getLength() function. Processes the sublist
// that begins with the node pointed to by p.

{
    int result;   // Result returned

    if ( p == 0 )
       result = 0;                             // End of list reached
    else
       result = ( getLengthSub(p->next) + 1 ); // Number of nodes after
                                               // this one + 1
    return result;
}
```

Step 1: Activate the call to the `getLength()` function in the test program by removing the comment delimiter (and the characters '`PE`') from the lines beginning with "`//PE`".

Step 2: Execute the `getLength()` function using the following list.



Step 3: What result does `getLength()` produce?

Step 4: What is the significance of the call to the `getLengthSub()` function in which parameter `p` is null?

Step 5: Describe how the calls to `getLengthSub()` combine to return the length of the list. Use a diagram to illustrate your answer.

Step 6: What value does the `getLength()` function return when called with an empty list? Describe how this value is computed. Use a diagram to illustrate your answer.

Laboratory 10: Bridge Exercise

Name ______________________________ Date ____________________

Section ______________________________

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

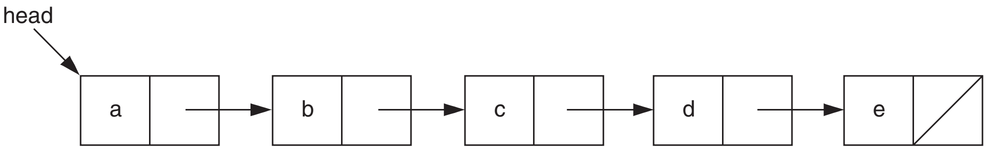
Part A

The following pair of functions perform some unspecified action.

```
template < class DT >
void List<DT>:: unknown1 () const

// Unknown function 1.

{
    unknown1Sub(head);
    cout << endl;
}

// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

template < class DT >
void List<DT>:: unknown1Sub ( ListNode<DT> *p ) const

// Recursive partner of the unknown1() function.

{
    if ( p != 0 )
    {
       cout << p->dataItem;
       if ( p->next != 0 )
       {
          unknown1Sub(p->next->next);
          cout << p->next->dataItem;
       }
    }
}
```

Step 1: Activate the call to the `unknown1()` function in the test program in the file *test10.cpp* by removing the comment delimiter (and the characters '`BA`') from the lines beginning with "`//BA`".

Step 2: Execute the `unknown1()` function using the following list.



Step 3: What output does `unknown1()` produce?

Step 4: Describe what each statement in the `unknown1Sub()` function does during the call in which parameter `p` points to the node containing ‘`a`’.

Step 5: Describe how the calls to `unknown1Sub()` combine to output the list. Use a diagram to illustrate your answer.

Part B

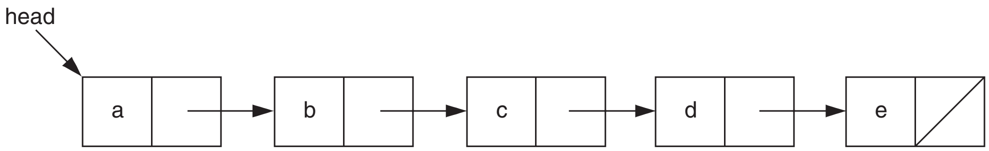
The following pair of functions perform yet another unspecified action.

```
template < class DT >
void List<DT>:: unknown2 ()
// Unknown function 2.
{
    unknown2Sub(head);
}

// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
template < class DT >
void List<DT>:: unknown2Sub ( ListNode<DT> *&p )
// Recursive partner of the unknown2() function.
{
    ListNode<DT> *q;

    if ( p != 0  &&  p->next != 0 )
    {
       q = p;
       p = p->next;
       q->next = p->next;
       p->next = q;
       unknown2Sub(q->next);
    }
}
```

Step 1: Activate the call to the `unknown2()` function in the test program by removing the comment delimiter (and the characters '`BB`') from the lines beginning with "`//BB`".

Step 2: Execute the `unknown2()` function using the following list.



Step 3: What list does `unknown2()` produce?

Step 4: Describe what each statement in the `unknown2Sub()` function does during the call in which parameter `p` points to the node containing '`a`'. In particular, what role does the fact that `p` is passed using call by reference play in this call?

Step 5: Describe how the calls to `unknown2Sub()` combine to restructure the list. Use a diagram to illustrate your answer.

Laboratory 10: In-lab Exercise 1

Name ______________________________ Date ____________________

Section ______________________________

Although recursion can be an intuitive means for expressing algorithms, there are times you may wish to replace recursion with iteration. This replacement is most commonly done when analysis of a program's execution reveals that the overhead associated with a particular recursive routine is too costly, either in terms of time or memory usage.

Part A

Replacing recursion in a routine such as the `getLength()` function (Prelab Exercise, Part E) is fairly easy. Rather than using recursive calls to move through the list, you move a pointer of type `ListNode*` from node to node. In the case of the `getLength()` function, this iterative process continues until you reach the end of the list.

The `reverse()` function (Prelab Exercise, Part C) presents a somewhat more challenging problem. The iterative form of this routine moves a set of pointers through the list in a coordinated manner. As these pointers move through the list, they reverse the links between pairs of nodes, thereby reversing the list itself.

Step 1: Create an implementation of the `reverse()` function that uses iteration, in conjunction with a small set of pointers, in place of recursion. Call this function `iterReverse()` and add it to the file *listrec.cpp*. A prototype for this function is included in the declaration of the List class in the file *listrec.h*.

Step 2: Activate the call to the `iterReverse()` function in the test program in the file *test10.cpp* by removing the comment delimiter (and the characters '`1A`') from the lines beginning with "`//1A`".

Step 3: Prepare a test plan for the `iterReverse()` function that covers lists of different 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 `iterReverse()` function, correct them and execute your test plan again.

Test Plan for the `iterReverse` Operation

Test Case	*List*	*Expected Result*	*Checked*

Part B

The `writeMirror()` function (Prelab Exercise, Part B) presents an even greater challenge. The iterative form of this routine uses a stack to store pointers to the nodes in a list. This stack is used in concert with an iterative process of the following form:

```
Stack<ListNode<DT>*> tempStack;   // Stack of pointers
ListNode *p;                      // Iterates through list

Set p to the head of the list.    // Traverse list from
while ( p != 0 ) do               // first to last
{
    tempStack.push(p);
    Process the list node pointed to by p (if necessary).
    Advance p to the next node in the list.
}

while ( !tempStack.isEmpty() ) do // Traverse list from
{                                 // last to first
    p = tempStack.pop();
    Process the list node pointed to by p.
}
```

Step 1: Create an implementation of the `writeMirror()` function that uses iteration, in conjunction with a stack, in place of recursion. Call the resulting function `stackWriteMirror()` and add it to the file *listrec.cpp*. A prototype for this function is included in the declaration of the List class in the file *listrec.h*. Base your `stackWriteMirror()` function on one of your implementations of the Stack ADT from Laboratory 5.

Step 2: Activate the call to the `stackWriteMirror()` function in the test program by removing the comment delimiter (and the characters '`1B`') from the lines beginning with "`//1B`".

Step 3: Prepare a test plan for the `stackWriteMirror()` function that covers lists of different 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 `stackWrite-Mirror()` function, correct them and execute your test plan again.

Test Plan for the `stackWriteMirror` Operation

Test Case	*List*	*Expected Result*	*Checked*