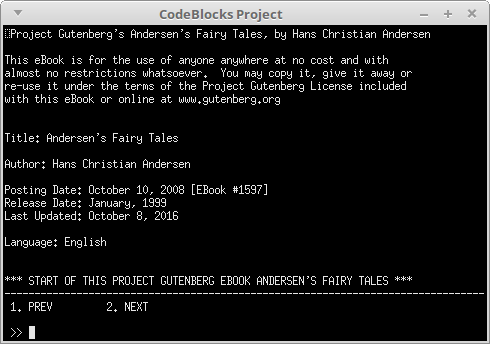
**Linked list project**

CS 250, Spring 2017

[](https://github.com/Rachels-Courses/CS250-Data-Structures/blob/master/Assignments/Projects/Project%2001%20-%20Linked%20Lists/images/LinkedListScreenshot.png)

**Setup**

Make sure to download the starter code, which contains the shell for the doubly linked list.

**Turn-in**

Upload your code to your class repository on GitHub, as well as turning in your code in the class Dropbox.

Make sure to include **all source files: .cpp, .hpp, .h, etc.**. (Project files / solution files are not required.)

**Group work policy**

* This project is a **solo effort**.
* You can brainstorm with others, but you cannot code together, share code, etc.

**Starter project**

The starter project contains quite a few code files. You will only be responsible for modifying **LinkedList.cpp**.

The project contains the **cuTEST** framework and **Tester.hpp** for testing, as well as a **BookProgram** object and **Page** object - these load a book into the program and allow you to read the book from the terminal. There is also **CPP-Utilities**, which contain string utilities and menu utilities -- you don't have to do anything with these.

**Linked list basics**

A linked list is a type of structure that only allocates memory for new objects as-needed. Unlike the Dynamic Array, it doesn't pre-allocate large chunks of memory and resize as needed. Instead, each time a new item is pushed into the list, memory for a new "Node" is allocated.

A linked list requires two different classes: A **Node** and a **List**.

**Node**

A Node contains the data itself, as well as pointers. In a doubly-linked-list, the Node contains pointers to the **next node** and the **previous node**. In a singly-linked-list, the node only contains a pointer to the next node.

Using these "ptrNext", "ptrPrev" nodes, we can traverse the list by updating a "traversal" pointer... starting at the first node, and step-by-step going to each node's "ptrNext".

**List**

The list class itself contains the standard Push, Insert, Pop, etc. functions. Instead of storing an array as a private member variable, instead it stores **pointers** to nodes, usually the first node and the last node.

The list keeps track of the itemCount (amount of items that have been pushed in), as well as the memory addresses of the first item in the list, and the last item in the list.

When a list is empty, ptrFirst and ptrLast should be pointing to **nullptr**.

For functions like Push, there will be different sets of code that will be executed depending on whether the list is empty, or whether there is already something in the list. We use an if statement to check the list's state and decide how to go about adding a new item.

If a list is empty, then ptrFirst and ptrLast should both be pointing to nullptr.

**Memory allocation**

As we add new items to the linked list, each new element needs memory allocated.

To do this, create a local pointer variable within the function where new memory needs to be allocated.

Node<T>\* ptrNew;

And allocate the data...

ptrNew = new Node<T>;

And you can set up the data as well.

ptrNew->data = newItem;

You won't need to call **delete** in the same function; the **Pop** function will be responsible for freeing any allocated memory.

**List traversal**

When traversing the list, we need to make a traversal pointer, which begins at the first item, and steps through each node via each node's *ptrNext* pointer.

We cannot **randomly access data** in a linked list because it is not implemented with an array. Because we only keep track of the first and last item with pointers, we have to *traverse* the list to find some item at position *n*.

First, we create a pointer:

Node<T>\* ptrCurrent;

**We do not allocate memory with this pointer!** This pointer is meant to point at existing data.

To begin at the beginning of the list, you need to point this to the first item:

ptrCurrent = m\_ptrFirst;

And to step to the next item, make use of *ptrNext*:

ptrCurrent = ptrCurrent->ptrNext; // Go to the next item

If you want to go to item *n* in the list, then you'll just have to do the above step *n* times (via a loop.)

You can traverse a few ways. If you have a specific position you want to stop at, use an integer counter.

If you want to traverse over a full list, you can loop *while the traversal pointer is not nullptr.*

**Common errors**

**New node pointer vs. Traversal pointer**

It is common to mix up utilizing pointers for list traversal with utilizing pointers for making new nodes. Know whether you're traversing a list, or creating a new node!

When traversing, you **will not allocate new memory**!

**Linked list specs**

Some of the methods have been implemented already, but you will need to fill in others.

**void Push( const T& newItem )**

When a list is first made, ptrFirst and ptrLast should be pointing to **nullptr**. We will use this fact to check whether our list is empty prior to adding an item.

**Adding the first item**

* Allocate memory using the **m\_ptrFirst** pointer.
* Set m\_ptrFirst's **data** field to the data passed in as the parameter (newItem).
* Make sure to set m\_ptrLast to also point to m\_ptrFirst.
* Increment **m\_itemCount**.

**Adding additional items**

* Allocate memory for a new Node.
* Set up the new node's **data** field to the data passed in as the parameter (newItem).
* Set the new node's ptrPrev to the current m\_ptrLast.
* m\_ptrLast's new ptrNext value should be this new node.
* Update m\_ptrLast to point to the new node.
* Increment **m\_itemCount**.

**bool Insert( int index, const T& newItem )**

Insert will still have to make sure the *index* is valid. If it isn't valid, simply return false. A valid index is >= 0, and <= m\_itemCount.

If the index is valid, and happens to be at the end of the list, call **Push** and pass in the newItem. Make sure to return true.

Otherwise, we are going to have to find this position. You will need a **traversal** pointer (see the Linked List section above).

Once your traversal node is pointing to the node at position *index*, now it's time to allocate some new memory (via a DIFFERENT pointer.)

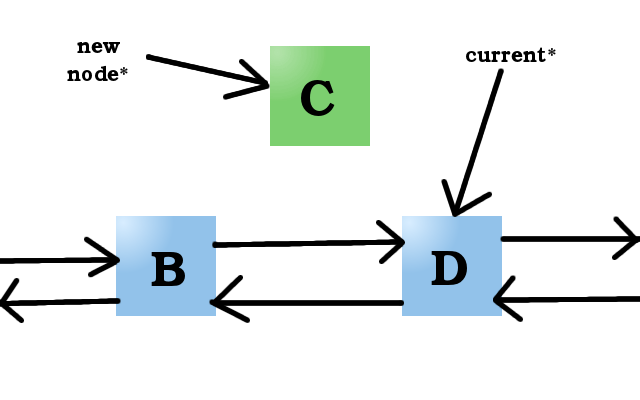
1. Create a new Node pointer
2. Allocate memory for the new node
3. Set the new node's **data**

This new node is going to be inserted between two nodes. To do this, you will need to update the **ptrNext** pointer of the previous item, and the **ptrPrev** pointer of the next item.

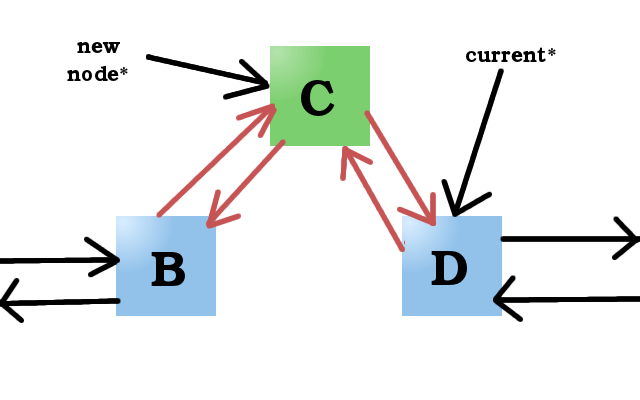
We also have to update *ptrNext* and *ptrPrev* of the new node.

1. Set the new node's *ptrNext* to our traversal pointer.
2. Set the new node's *ptrPrev* to our traversal pointer's *ptrPrev*.
3. Set the previous node's *ptrNext* to the new node.
4. Set the traversal node's *ptrPrev* to the new node.

**Before insert**

[](https://github.com/Rachels-Courses/CS250-Data-Structures/blob/master/Assignments/Projects/Project%2001%20-%20Linked%20Lists/images/beforeinsert.png)

A new node has been created, and the current\* (traversal) pointer is pointing at item *index*.

[](https://github.com/Rachels-Courses/CS250-Data-Structures/blob/master/Assignments/Projects/Project%2001%20-%20Linked%20Lists/images/afterinsert.png)

B, C, and D's *ptrPrev* and/or *ptrNext* pointers have been updated to include C in-between.

**void Extend( const LinkedList& other )**

Extend can utilize the *Push* function to push all elements from the *other* list onto the local list.

Use a traversal pointer to go through all of the *other* list's items and push onto the local list.

**bool Pop()**

This will remove the last item of the list, but the functionality will be different based on whether the list has 1 item or more than 1 item.

**Empty list**

Do nothing; return false.

You can tell the list is empty if *m\_ptrFirst* and/or *m\_ptrLast* is pointing to nullptr.

**One item in list**

If there is only one item in the list, you will delete that only item. Both m\_ptrFirst and m\_ptrLast should both be pointing at the single item, so delete via one of these pointers.

Afterward, make sure to update m\_ptrFirst and m\_ptrLast to both now point to *nullptr*.

Make sure to decrement m\_itemCount and return true.

**Multiple items in list**

You will need to use a traversal pointer to get to the 2nd-to-last item of the list.

You can get the 2nd-to-last item via m\_ptrLast 's *ptrPrev* node.

Node<T>\* penultimate = m\_ptrLast->ptrPrev;

1. Free the memory at m\_ptrLast.
2. Update the m\_ptrLast pointer to point to the 2nd-to-last item.
3. Update the new m\_ptrLast's *ptrNext* item to a nullptr.
4. Decrement m\_itemCount.
5. Return true.

**bool Remove( int index )**

First, check for valid index. Return false if invalid.

Next, if there is only one item in the list, or if the remove index is the last item of the list, just use Pop() to get rid of it.

Otherwise, locate the item to be removed using a traversal pointer.

Once you have the item, you will update the pointers (in a reverse of the insert functionality):

1. The previous item's ptrNext will now point to the next item.
2. The next item's ptrPrev will now point to the previous item.
3. Free the memory that your traversal pointer is pointing to.
4. Decrement m\_itemCount.
5. Return true.

**T Get( int index ) const**

Use a traversal technique to return the **data** of the Node at the given position.

If the index is invalid, just return a T constructor:

if ( index < 0 || index >= m\_itemCount )

{

// Return a new T item...

return T();

}

**LinkedList& operator=( const LinkedList& other )**

To set one linked list equal to another, you should make sure to free all the data first.

1. Use Free() to clear out all memory in the local object.
2. Traverse through every element of the *other* list, using Push() to push each element onto the local list.
3. Return a pointer to this item.

return (\*this);

**bool operator==( const LinkedList& other )**

This should return true of both lists are the same...:

* If both lists have different sizes, return false.
* Use **two** traversal pointers to traverse through both lists concurrently in a loop.
  + Check the data values of both traversal items. If they don't match, return false.
  + Otherwise, have both traversal items go to the next Node.
  + Keep going until you've covered both lists fully.

If you have traversed the lists and it never hit a *return false* statement, then that must mean the lists match, so return *true*.

**Grading rubric**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Grading Rubric | | | |
|  | Name: |  | | |
|  | Assignment: | CS 250, Linked List HW Project | | |
|  |  |  |  |  |
|  | Breakdown | | | |
|  | Item | Description | Total % | Your Score |
|  | Builds & Runs |  | 50.00% |  |
|  | Insert |  | 7.00% |  |
|  | Remove |  | 7.00% |  |
|  | Extend |  | 6.00% |  |
|  | operator= |  | 6.00% |  |
|  | Push |  | 5.00% |  |
|  | Pop |  | 5.00% |  |
|  | Get |  | 5.00% |  |
|  | operator== |  | 5.00% |  |
|  | Clean code |  | 2.00% |  |
|  | No logic errors |  | 2.00% |  |
|  | Totals |  |  |  |
|  |  |  | 100.00% | 0.00% |