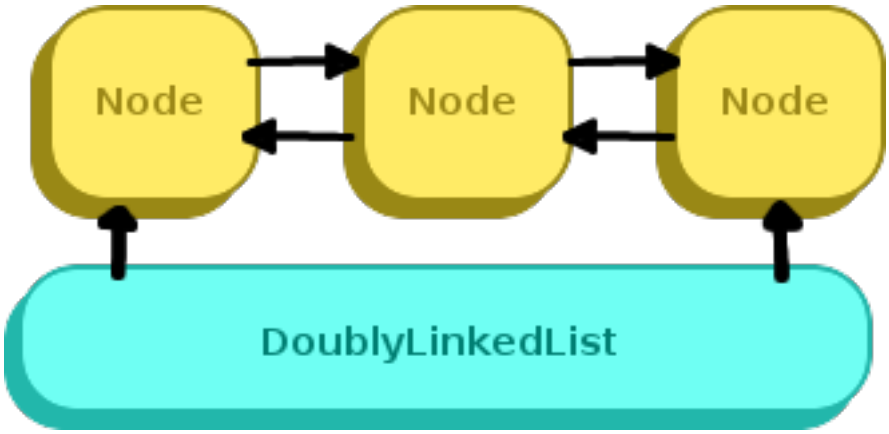
Project 1: Linked List



Information

Topics: Linked lists, pointers, basic program implementation

Turn in: All source files (.cpp and .hpp).

Starter files: Download on GitHub or D2L.

About

In this project, you will be implementing a Doubly Linked List.

There are two parts of the project: The tests, and an actual program. When you start the program, it will ask you which to run.

```
----------------------
1. Run tests
2. Run program
3. Exit
>>
```

For the first part of the assignment, you will work on implementing the DoublyLinkedList, using the tests to verify your work. For the second part, you will go into the `project1_program` and replace usage of the STL `list` with your DoublyLinkedList instead.

```
Project 1 - Linked List/
├─ project1_main.cpp ...  Contains main()
├─ project1_LinkedList.hpp ...  Class declaration
├─ project1_LinkedList.cpp ...  Class function definitions
├─ project1_Tester.hpp ...  Unit test functions
├─ project1_CustomerData.hpp ...  CustomerData, the data
│                                 stored in the LinkedList
├─ project1_CustomerData.cpp
├─ project1_program.hpp ...  The program functionality
├─ project1_program.cpp
├─ cuTEST/ ...  Unit test framework
│  ├─ Menu.hpp
│  ├─ StringUtil.hpp
│  ├─ TesterBase.hpp
│  └─ TesterBase.cpp
└─ CodeBlocks Project/ ...  Code::Blocks project file
   └─ VS2015 Project 1 ...  Visual Studio project files
```

Your project needs to be compiling as a C++11 project. To set this up in Code::Blocks, go to Settings, Compiler, then in the Compiler Flags pane, check the "Have g++ follow the C++11 ISO C++ language standard".

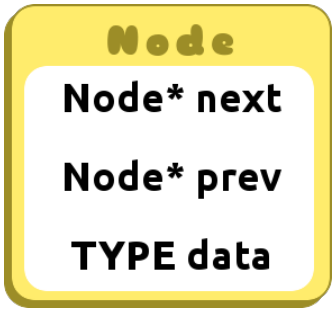
Linked List basics

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 LinkedList.

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.

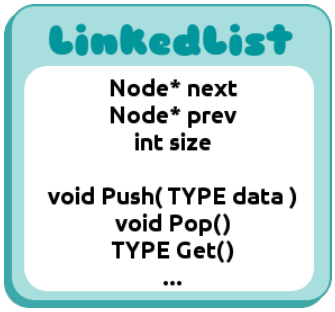


Using these previous and next pointers to nodes, we can traverse the list by updating a "traversal" pointer... starting at the first node, and step-by-step going to each node's next pointer.

When a Node is first created, its next and previous pointers should be pointing to `nullptr`.

LinkedList

The LinkedList class contains the main functionality of the list, such as Push to add new items, Pop to remove items, and Get to access items. The LinkedList also contains pointers to the first item in the list and, often, the last item in the list (though not required). The LinkedList is also responsible for keeping track of how many items have been added.



When a LinkedList is empty, whether when it is first created or if everything has been removed, the first and last pointers should be ponting to `nullptr`.

When the LinkedList is destroyed, its destructor should be responsible for freeing up all the memory allocated for the Node items.

Adding new items

When a Push or Insert function is called, the LinkedList needs to...

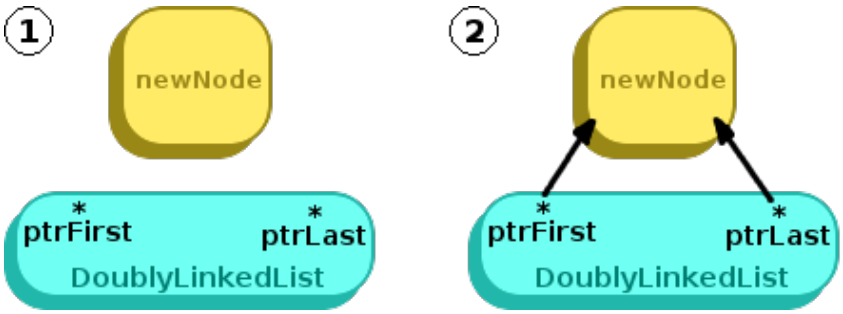
- **allocate memory for the new Node**
- **update the pointers of any existing Nodes that will be the new Node's neighbor**
- **update the pointer for the first or last item in the LinkedList.**

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, then allocate the memory for the Node, and set up the Node's data.

```
1 void Push( T data )
2 {
3     Node* newNode = new Node;
4     newNode->data = data;
5     m_itemCount++;
6     // ...
7 }
```

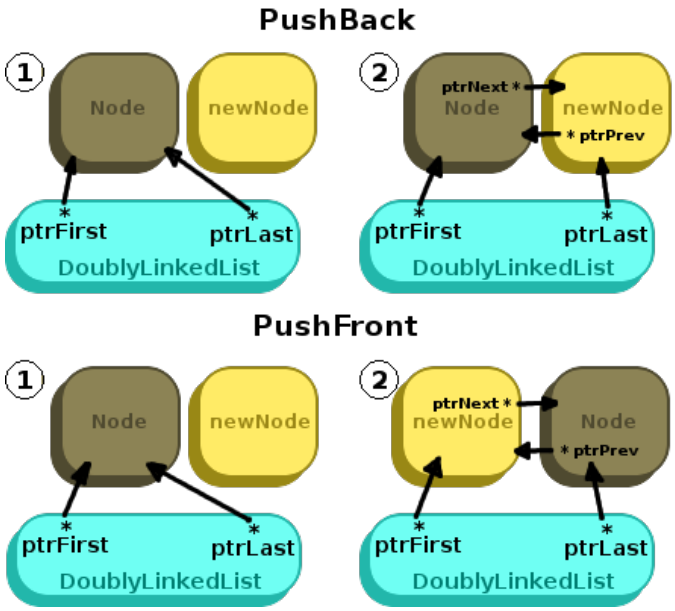
What we do with this newNode will then depend based on whether or not the LinkedList is empty, or if it contains at least one item.

Empty LinkedList:



Once a new Node is created, if our LinkedList is empty, the only step we have to do is update its *first* and *last* Node* pointers to the newNode that we've created.

Non-empty LinkedList:



If the LinkedList is **not** empty, there are more steps you will have to take care of. These steps also depend on whether you're writing a **Push Back** (insert as new end) or **Push Front** (insert as new beginning) function.

- PushBack:
  1. Set the current *last Node*'s next pointer to the new node.
  2. Set the new node's previous pointer to the current *last Node*.
  3. Update the *last Node* pointer of the LinkedList to point at the new Node.
- PushFront:
  1. Set the current *first Node*'s previous pointer to the new node.
  2. Set the new node's next pointer to the current *first Node*.
  3. Update the *first Node* pointer of the LinkedList to point at the new Node.

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

Removing items

When you remove an item from the LinkedList, you will have to update the pointers of the LinkedList, as well as any Nodes that were neighbors to the Node that was removed. Additionally, the two scenarios that have different behaviors here are if the LinkedList contains *only one item* (so you're removing the last item), or if it contains more than one item.

Pop when list is empty? If Pop is called and the list is already empty, the usual behavior is to ignore it.

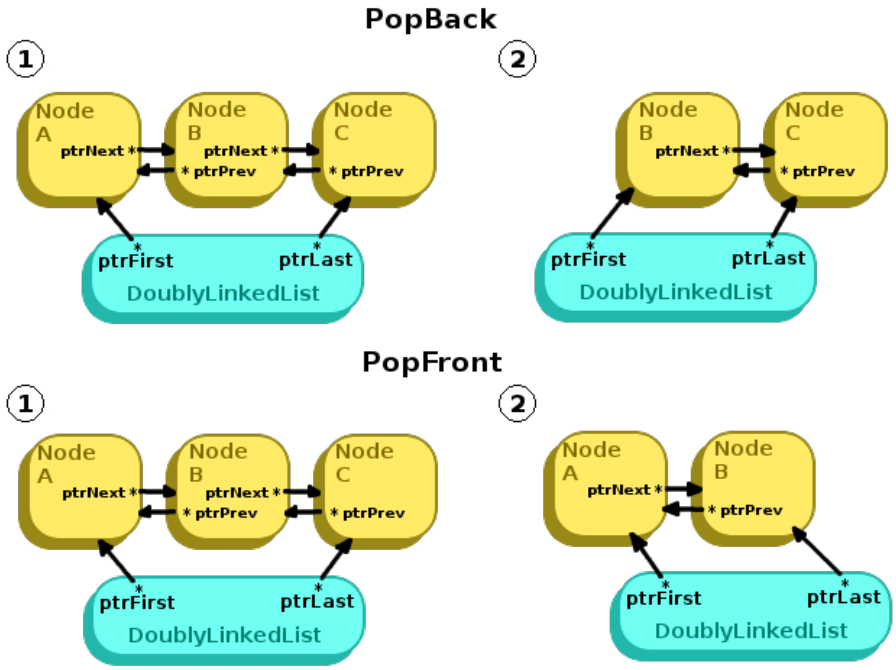
For example, if we were using the STL `list`, we can see on its documentation page (http://www.cplusplus.com/reference/list/list/pop_back/) its exception safety level:

> **Exception safety** If the container is not empty, the function never throws exceptions (no-throw guarantee). Otherwise, it causes *undefined behavior*.

Removing last item If you're removing the last item from the LinkedList, you will have to make sure to free the memory for the Node, but also update the LinkedList's Node pointers to the first and last item; these should both point at `nullptr`.

```
1  void Pop( T data )
2  {
3      // It is the last item
4      if ( m_ptrFirst == m_ptrLast )
5      {
6          delete m_ptrFirst;
7          m_ptrFirst = nullptr;
8          m_ptrLast = nullptr;
9          m_itemCount--;
10     }
11     // ...
12 }
```

Pop when we have more than one item in the list In our DoublyLinkedList we can Pop items off from the front of the list or the back of the list, so the way to implement this changes a bit for each.



- PopBack:

  1. Locate the second-to-last item in the LinkedList (With a DoublyLinkedList, you can simply get the last Node's *previous* item.)
  2. Update the second-to-last Node's next pointer to be `nullptr`.
  3. Free the memory pointed to by the last Node pointer
  4. Update the last Node pointer to point to the (formerly) second-to-last Node
  5. Decrease the item count

- PopFront:

  1. Locate the second item in the LinkedList (You can get the first Node's *next* item.)
  2. Update the second Node's previous pointer to be `nullptr`.
  3. Free the memory pointed to by the first Node pointer

4. Update the first Node pointer to point to the (formerly) second Node
5. Decrease the item count

Keeping track of the second Node
Your code might look ugly if you're trying to do these steps like

```
m_ptrLast->ptrPrev->ptrNext = nullptr;
```

Instead of working like this, simply make a local Pointer to keep track of your second-to-last or second item...

```
// PopBack
Node* ptrSecondToLast = m_ptrLast->m_ptrPrev;
ptrSecondToLast->ptrNext = nullptr;
// ...

// PopFront
Node* ptrSecond = m_ptrFirst->m_ptrNext;
ptrSecond->ptrPrev = nullptr;
// ...
```

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!

Doubly Linked List specs