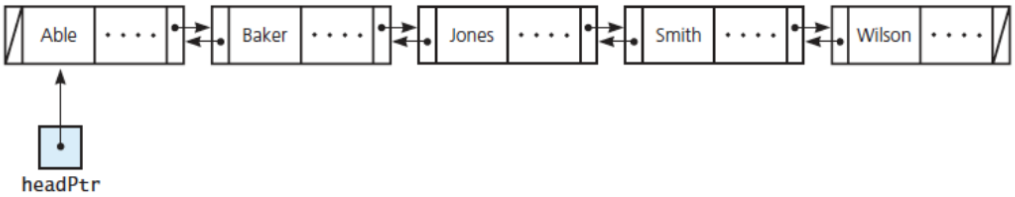
Project 3: Doubly-Linked List

Dr. Oyewole Oyekoya

Due March 31, 2020 by 5PM

Your objective for this project is to implement a Doubly-Linked List.



Part 1:

Define a class DoubleNode that is capable of holding an item of any arbitrary type ItemType. As a node of a Doubly-Linked list, it should additionally contain two pointers that respectively point to other objects of type DoubleNode.

The following methods will be required of your DoubleNode class, but feel free to add methods as you see fit:

1. Default Constructor
2. Parameterized Constructor(s)

   Hint: You can use default parameters as you did in Project 1 to require the use of only a single parameterized constructor.
3. `ItemType getItem() const`
4. A method that allows you to set the next node of the DoubleNode.
5. A method that allows you to set the previous node of the DoubleNode

Entitle your header (.hpp) file *DoubleNode.hpp*, and entitle your implementation file (.cpp) *DoubleNode.cpp*.

Part 2:

Define a class DoublyLinkedList that is a demonstration of the Doubly-Linked List concept discussed in class. It should contain a head pointer to a DoubleNode of any arbitrary type ItemType, and it should contain a member that keeps track of its size.

Hint: If you get stuck on the way to design DoublyLinkedList, take a look at the LinkedBag header from Project 2.

The following methods are required of your DoublyLinkedList class:

1. Default Constructor
2. Copy Constructor
3. Destructor
4. `bool insert(const ItemType& item, const int& position)`, which is intended to insert item at index position in your list

   Note: Let the list be 1 indexed unlike arrays, which are 0 indexed.
5. `bool remove(const int& position)`, which is intended to remove the node at index position
6. `int getSize() const`, which is intended to return the number of the nodes in the list
7. `DoubleNode<ItemType> *getHeadPtr() const`, which is intended to return a copy of the hearPtr
8. `DoubleNode<ItemType> *getAtPos(const int& pos) const`, which returns a pointer to the node located at pos
9. `bool isEmpty() const`, which returns whether the list is empty
10. `void clear()`, which clears the list
11. `void display() const`, which prints the contents of the list in order
12. `void displayBackwards() const`, which prints the contents of the list backwards.
13. `DoublyLinkedList<ItemType> interleave(const DoublyLinkedList<ItemType>& a_list)`, which alters the calling list to be the interleaved list of the original and parameter lists

    Example: Define the calling list as a set of ordered nodes, $L1 = \{4, 2, 8, 5, 8\}$, and define the list that is passed as a parameter as the set of ordered nodes, $L2 = \{5, 1, 8, 4, 5, 9\}$. L1.interleave(L2) should yield the set $\{4, 5, 2, 1, 8, 8, 5, 4, 8, 5, 9\}$. In other words, to create the interleaved list, first add a node from L1, then L2, and then repeat. If there are any nodes left over in L1 or L2 exclusively, append them to the end of the list.

Entitle you header (.hpp) file *DoublyLinkedList.hpp*, and entitle your implementation file (.cpp) *DoublyLinkedList.cpp*.

Testing

You must always implement and test you programs **INCREMENTALLY!!!**

What does this mean?

- Implement and test one method at a time.
- For each class:
  1. Implement one function/method and test it thoroughly (multiple test cases + edge cases if applicable)
  2. Implement the next function/method and test in the same fashion.

How do you do this?

Write your own **main()** function to test your classes. In this course you will never submit your test program, but you must always write one to test your classes. Choose the order in which you implement your methods so that you can test incrementally (i.e. implement mutator functions before accessor functions). Sometimes functions depend on one another. If you need to use a function you have not yet implemented, you can use **stubs**: a dummy implementation that always returns a single value for testing (don't forget to go back and implement the stub!!! If you put the word STUB in a comment, some editors will make it more visible.

Grading Rubric:

- **Correctness 80%** (distributed across unit testing of your submission)
  - A submission that implements all required classes and/or functions but does not compile will receive 40 points total (including documentation and design).
- **Documentation 10%**
- **Style and Design 10%** (proper naming, modularity, and organization)

Submission:

You will submit **the following files:**

- DoubleNode.hpp
- DoubleNode.cpp
- DoublyLinkedList.hpp
- DoublyLinkedList.cpp

Your project must be submitted on Gradescope.
Although Gradescope allows multiple submissions, it is not a platform for testing and/or debugging and it should not be used for that. You MUST test and debug your program locally. Before submitting to Gradescope you MUST ensure that your program compiles (with g++) and runs correctly on the Linux machines in the labs at Hunter (see detailed instructions on how to upload, compile and run your files in the "Programming Rules" document). That is your baseline, if it runs correctly there it will run correctly on Gradescope, and if it does not, you will have the necessary feedback (compiler error messages, debugger or program output) to guide you in debugging, which you don't have through Gradescope. "But it ran on my machine!" is not a valid argument for a submission that does not compile. Once you have done all the above you submit it to Gradescope.