CS 341 – Fall 2024
Assignment #4 – Hop Dawg
Due: 11/1/2024

Have you ever wondered how on your smart device (PC/Phone/Tablet/etc.) that a screen of apps can be displayed in order of most recently used? We know that there is only so much screen real estate and it makes sense from a usability aspect that we display those apps in groupings based upon their usage. One such solution for handling this task is to use a Doubly Linked List (DLL). A DLL is a Linked List in which each Node in the list is linked to both its predecessor as well as its successor. This means that you can traverse the list both forwards and backwards. The method for doing this is to maintain a list of the apps by their usage and to move or insert apps into the slots provided. Your forth assignment will allow you to explore this concept of a DLL and explore how ordering and insertions can be done on such a data structure. This will build upon your existing knowledge of C++ and help reinforce the topics of Inheritance, Polymorphism, Pointers, and Memory Management that we have discussed in lecture.

For this project you are given a large data set and have been tasked with inserting new numbers into the data set…but how!? You really have two options at your disposal: you could simply insert the new number onto the head, or front, of the list OR you could insert the number onto the tail, the end, of the list (*we will ignore the third option which would allow you to "randomly" insert the number into the existing list, for now…*). For simplicity sake, we will leverage your familiarity with Arrays and assume that when building your DLL that inserts will be Tail Inserts, meaning that we will append the element to the end of the DLL.

Once you have built your DLL you will need to insert new numbers into the existing List. Blue IV (our loyal and faithful CEO) does not simply want the values inserted into the DLL – instead, his desire is for the elements to be inserted such that the insert can **only** be executed when the location is found where the number is followed by a larger value (think pseudo-sorted).

For example, given the following list:

```
15, 7, 10, 44, 54
```

And given the following number to insert:

```
34
```

With our two existing options (Head Insertion/Tail Insertion) we have the following two scenarios:

1. If you start at the head of the DLL: '34' is greater than '15', '7', '10' but not '44', so we would need to insert it between the '10' and '44' in the existing DLL.
2. If we start at the tail: '34' is not larger than '54' and it is also not larger than '44', but it is larger than '10', so we would insert it between the '10' and '44' in the list. In this case, we came to the same conclusions….what next!?

Now comes the catch…Blue IV has instructed us that we MUST select the optimal insertion, meaning the insertion that will require the **least** amount of hops. We define a "hop" to be measured by the number of comparisons needed in order to insert the number into the existing DLL. In our above example, Option #1 requires three (3) hops - while Option #2 requires only two (2) hops. Thus, we should insert the number '34' from the Tail of the DLL and not the Head, and thus we are left with the following DLL:

```
15, 7, 10, 34, 44, 54
```

Note: If the steps required are the same (e.g., both head and tail insertion require two (2) hops) we will use the default of Tail Insertion.

Note: We will assume that if the value is equal to (==) the existing number in the DLL we will always place the new number AFTER the existing number.

Blue IV has given you two data files that you will need to load (`data.txt` and `sorted.txt`) and one insertion file (`inserts.txt`). You will need to run your program twice – **once** for `data.txt` and **once** for `sorted.txt`. In both instances you should insert the data elements found in the file `inserts.txt`. He would then like two output text files (`output.txt` and `sortedOutput.txt`) along with a Hop Count for **each** execution – the Hop Count should be stored in a `README.txt` file. We will assume that both input files contain only space delimited Integer entries.

A sample scenario is shown below:

```
Inserting 15...
Inserting 7...
Inserting 10...
Inserting 44...
Inserting 54...
15<-->7<-->10<-->44<-->54

                        (Inserting 34, 36, 1, 2, 2)

15<-->7<-->10<-->34<-->44<-->54
15<-->7<-->10<-->34<-->36<-->44<-->54
1<-->15<-->7<-->10<-->34<-->36<-->44<-->54
1<-->2<-->15<-->7<-->10<-->34<-->36<-->44<-->54
1<-->2<-->2<-->15<-->7<-->10<-->34<-->36<-->44<-->54

Total Number of Big Dawg Hops: 7
```

A few notes about the specific requirements of the program:

- The program should use proper Class structure and hierarchy – we will outline this in lecture.
  - `LinkedList` → `DoublyLinkedList`
  - `Node` → `LinkedNode`
- You should include a Constructor/Destructor in every Class.
  - Remember the impact of the `virtual` keyword!
- Your Driver (`driver.cpp`) should handle all File I/O.
  - No File I/O should be in any of your Classes.
- You must store your nodes (`Node`) and DLL on the Heap.
  - No Memory Leaks!
    - Use Valgrind: `valgrind --log-file=valgrind.txt A4.exe`

Development Process:

You **may** (but are not required to) work with a partner (groups of two (2) students) on this assignment. If you choose to do so, you are both expected to work on and maintain a single GitHub repository – be sure to add the appropriate collaborators. This will allow you to practice your skills of Forking, Cloning, and Merging using Git. Please let me know if you do plan to work with a partner on this project – if you do not notify me I will assume you are working independently on this assignment.

For the development of your code: you should create a class named `LinkedList`. From there we will create another Class called `DoubleLinkedList`. This Class will lean on `LinkedList` in order to accomplish its task. You will use OO pillar of Polymorphism to support this behavior. You will also need to create a `Node` Class and then inherit from it into a `LinkedNode` Class that will represent each element of the DLL. The `Node` Class will be linked (no pun intended) to the `LinkedList` Class via Aggregation (Composition). Tying this all together you will need to write a `driver` that will test this hierarchy and provide the necessary functionality as described earlier as outlined by Blue IV. Your driver program should allow for the user to input a space delimited text file (`.txt`) to be loaded and built into a DLL and then allow for the input of a second file (`.txt`) that will insert the new values into the DLL at the appropriate position with the final resulting list being output to another text file (`.txt`). Each Class should have a Header/Source file (`.h/.cpp`). Please include the Hop Count for each test in a `README.txt` file. Finally, you will need to create a `makefile` that properly links all of this code together and creates an executable named **A4.exe**

Phase I – Due: 10/22/2024

For Phase I you are tasked with the responsibility of creating a `Node` class. We will later use this `Node` class in conjunction with our `LinkedNode` and even use it in a later assignment! You are responsible for implementing the Header file (`Node.h`) provided to you. Create a driver program to test your newly created `Node` class.

Upon successful completion and testing of your `Node` you are ready to move on to Phase II.

Phase II – Due: 10/24/2024

For Phase II you are tasked with the responsibility of creating a `LinkedList`. Here you will create a singularly `LinkedList`…meaning that you only maintain "forward" links. In order to accomplish this you will need to create a `LinkedNode` class that inherits from your `Node` class in Phase I with the following additional behavior:

```
Public:
      LinkedNode(int data, LinkedNode * nextLinkedNode);
      LinkedNode * getNextLinkedNode();
      void setNextLinkedNode(LinkedNode * nextLinkedNode);
      bool hasNextLinkedNode();

Private:
      LinkedNode * nextLinkedNode_;
```

Once a `LinkedNode` has been created we can then create our `LinkedList` class. For this phase we will only focus on the following methods:

```
Public:
      bool isEmpty();
      int getLength();
      void insert(int element);
      void printList();

Private:
      LinkedNode * head_;
      LinkedNode * tail_;
```

You can then modify your driver to create an instance of a `LinkedList` and populate it with some sample "nodes" and then print the `LinkedList` out in the console.

```
Inserting 15...
Inserting 7...
Inserting 10...
15-->7-->10
Length: 3
```

Upon successful completion and testing of your `LinkedList` class you are ready to move on to Phase III.

Phase III – Due: 10/26/2024

For Phase III we are going to modify our `LinkedNode` class to now take into consideration links that are maintained both "forward" and "backward." This will prepare our code for Phase IV where we will create our `DoublyLinkedList` class. In order to accomplish this phase you will need to modify/add the following to your `LinkedNode`:

```
Public:
      LinkedNode(int data, LinkedNode * nextLinkedNode, LinkedNode *
      prevLinkedNode);
      LinkedNode * getPrevLinkedNode();
      void setPrevLinkedNode(LinkedNode * prevLinkedNode);
      bool hasPrevLinkedNode();

Private:
      LinkedNode * prevLinkedNode_;
```

Upon successful completion and testing of your `LinkedNode` class you are ready to move on to Phase IV.

Phase IV – Due: 10/28/2024

For Phase IV we are going to implement our `DoublyLinkedList`. A `DoublyLinkedList` will contain all the *goodness* of our `LinkedList` class (inheritance) but now will take advantage of both forward and backward traversals of our list. In order to accomplish this phase you will need to create the `DoublyLinkedList` class with the following methods:

```
virtual void printList();
void insertLinkedNode(LinkedNode * node, int data);
void insertAfterLinkedNode(LinkedNode * node, int data);
void insertBeforeLinkedNode(LinkedNode * node, int data);
```

You should now be able to create an instance of a `DoublyLinkedList` and produce the following output from your driver:

```
Inserting 15...
Inserting 7...
Inserting 10...
15<-->7<-->10
Length: 3
```

Upon successful completion and testing of your `DoublyLinkedList` class you are ready to move on to Phase V.

Phase V – Due: 11/1/2024

For the final phase of this project you are tasked with the responsibility of implementing the previously mentioned Hop Count optimization into the program. Once you have completed this task you have completed the assignment! Sample completed output of this phase is shown above.

```
Inserting 15...
Inserting 7...
Inserting 10...
Inserting 44...
Inserting 54...
15<-->7<-->10<-->44<-->54

                      (Inserting 34, 36, 1, 2, 2)

15<-->7<-->10<-->34<-->44<-->54
15<-->7<-->10<-->34<-->36<-->44<-->54
1<-->15<-->7<-->10<-->34<-->36<-->44<-->54
1<-->2<-->15<-->7<-->10<-->34<-->36<-->44<-->54
1<-->2<-->2<-->15<-->7<-->10<-->34<-->36<-->44<-->54

Total Number of Big Dawg Hops: 7
```

I will be grading what is located in the **master branch** of your GitHub repository. It is strongly recommended that you commit and push often! If you opt to work on this assignment with a partner both members must maintain the group code in their own repositories. Please make use of the Git feature of leaving descriptive messages along with your commits. Be sure to add me to any repository as a collaborator so I can view and grade your submissions. Failure to do so will result in a 0 on the assignment as I will have nothing to grade!

Submission:

All assignments must be submitted on Butler GitHub (github.butler.edu).

The directory structure of the repository must contain the following files:

- **driver.cpp**
- **LinkedList.h**
- **LinkedList.cpp**
- **DoubleLinkedList.h**
- **DoubleLinkedList.cpp**
- **LinkedNode.h**
- **LinkedNode.cpp**
- **Node.h**
- **Node.cpp**
- **makefile**
- **README.txt**
- **sortedOutput.txt**
- **output.txt**

Each source file (`.h`/`.cpp`) **must** include the Honor Pledge and digital signature – in the case of a partner submission both digital signatures should be present.