

Data Structures and Algorithms

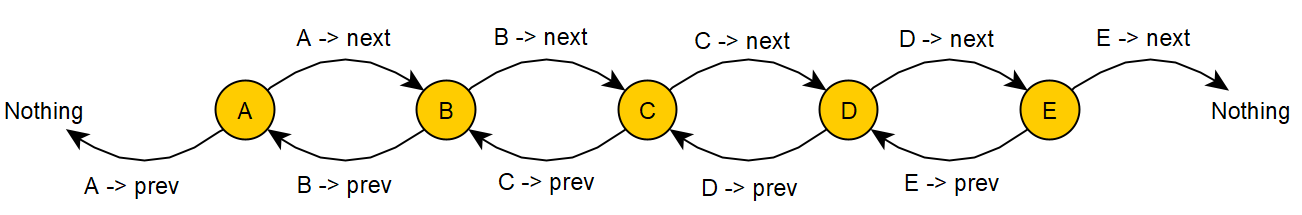
**Lab 3: DList.h**

**The Scenario**

You’ve mastered Dynamic Arrays/Vectors. Now it’s time to tackle your next challenge… There is a dragon that you must slay. The most feared dragon in all the land. You ready your armor and your trusty stallion. You even put your earbuds in with some epic music playing. You head to the local Blacksmith—appropriately named **Visual Studio**—and the blacksmith tells you that you that anyone skilled in the art of the sword knows they must craft their own sword and shield to defeat the dragon. Your sword is the **Doubly-Linked List**, and your shield is the **Iterator**. Today, you will implement the functionality for both of these.

***DList:***

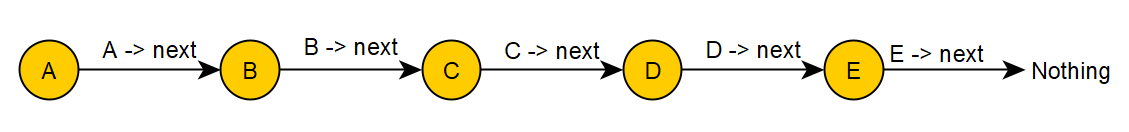
The Doubly-Linked List, or **DList**, is a commonly used data structure. Here is an illustration of a Doubly-Linked List with 5 nodes (A through E):



As you can see, a **node** is represented by a circle and consists of three components. A pointer leading from it to the **next** node, one to the **previous** node, and the **data** **itself** stored in each node. The data can be anything from a single integer to an array of chars. Here, the data is a single character (**char**. A, B, C, etc.) You can also see that the first and last Nodes of a linked list each point to nothing. The pointers still exist in each node—they simply have nothing to point to. This is represented in C++ as a null pointer, or **nullptr**.

To add a node between **B** and **C**, you would have to adjust **B -> next** to point to the new node along with **C -> prev**. The new node would also have to have its pointers set to point at B and C as well. i.e. newNode -> next and newNode -> prev.

On a side note, not all lists are Doubly Linked Lists. Here is the same list as a Singly Linked List:



As you can see, each node only consists of two components here. The node/data itself, and a pointer to the next node. There is no way to get back to the previous node once you have left it in a singly-linked list. In later labs, we will explore this problem further in the form of Binary Trees.

***Iterator:***

An **Iterator** is an object designed to traverse data. In this lab, we will focus on how it relates to lists, but it can be applied to many different data structures. The iterator provides functionality such as access to the current node, the data in the current node, and the ability to move forward and backwards in your list. An Iterator’s primary function, much as the name implies, is to **iterate** through a list.

**What To Do…**

Open DList.h. There will be instructions written in the comments on what is expected. Below is the gist of each function and variable.

***Variables:***

**data** The data stored in the current node. The data type “Type” is a generic declaration that allows for any type of data to be stored by the program in each node.

**\*next** A pointer to the next node. (The \* means pointer.)

**\*prev** A pointer to the previous node in the list.

**\*mCurr** A pointer to the current node in the list. Used in Iterator to give access to the current node, its data, etc. You can think of it like a bookmark in the list to show where the iterator is.

**\*mHead** A pointer to the first element of the DList. Must be reset if a node is added to the beginning of the DList.

**\*mTail** A pointer to the last element of the DList. Must be reset if another node is added to the end of the Dlist.

***Functions:***

**Node Constructor** Takes in the data as well as the next and previous nodes and sets each node’s component appropriately.

**Pre-fix ++ overload** i.e. ++iter. Advances the iterator through the list and returns the current Iterator(the invoking object). All of the ++ and -- operators will involve mCurr.

**Post-fix ++ overload** i.e. iter++. This version has a dummy variable passed in (an int). This plays no role in the code you will write, but is needed to specify that we’re overloading the post-fix version of ++. Advances the iterator through the list and returns a temporary Iterator pointing at the previous element.

**Pre-fix** -- **overload** i.e. --iter. Moves the iterator backwards and returns the current Iterator. See Pre-fix -- overload.

**Post-fix** -- **overload** i.e. iter--. See Post-fix ++ overload.

**\* operator overload** Overloads the \* operator to give access to the data stored in the node where the iterator is. i.e. \*

**DList default constructor** Sets **mHead**, **mTail**, and **mSize** to appropriate default, empty values.

**DList destructor** Cleans up all dynamically allocated memory. (Remember the principles of deep-copy vs. shallow-copy.) Associated with Clear.

**DList copy constructor** Constructs a new DList that is a copy of the one passed in. (Remember, deep-copy vs. shallow-copy)

**DList = overload** Assignment operator. This method will cycle through the parameter DList, \_assign, constructing a copy. If there is currently data, clear it out first. Associated with the copy constructor

**Copy (optional)** You may choose to do the Assignment Op0erator and/or the Copy Constructor **Recursively**. Recursive code can be much shorter and simpler to write and involves calling the function from within itself. (i.e. calling Copy from within itself.).

**AddHead** Adds a new first element to the front of the DList and updates mHead. Similar to a push\_front function.

**AddTail** Adds a new last element to the end of the DList and updates mTail. Similar to a push\_back function.

**Clear** Clear the list out of all dynamic memory and reset it to its default size. It will be as if the copy constructor had been called.

**Clear (optional)** An optional helper method to call from Clear should you decide to write Clear recursively. You will call it from the above version of Clear as well as within itself.

**Insert** Insert a piece of data into a node placed right before where the passed in iterator is stationed. Remember edge-cases.

**Erase** Erase the element the iterator passed in is stationed at. The iterator will be pointing to the node after the one erased.

**Begin** Create a new iterator at the first node of the list and return it.

**End** Create a new iterator at the last node of the list and return it.

**Tips, Tricks, and Resources**

When working with lists, trees, and other data structures involving pointers it is nearly **always** beneficial to have a pencil and paper handy. Draw out each node and figure out how to redirect the arrows before converting that into code. **REMEMBER: HOURS OF PLANNING CAN SAVE WEEKS OF PROGRAMMING!**

You can use the arrow multiple times in one statement. i.e. in the DList above, **A -> next -> prev** would be equivalent to **B -> prev.**

With the pre and post-fix versions of ++ and --, the only huge difference comes when setting a value equal. i.e. **newIter = ++iter** vs. **newIter = iter++**.

One side note: As a general rule of thumb, Lists are more memory-efficient, while Arrays are more processor-efficient. Whereas each node of a list can be stored anywhere in memory, arrays require a contiguous piece of memory to be used. This gives arrays random-access capability, so you don’t have to iterate from the beginning of the list each time you want a certain element. This tradeoff of memory versus processor efficiency is common in Computer Science.

**Plagiarism**

Plagiarism and Academic Dishonesty are considered a **very** serious offense in this class and can have a range of consequences including suspension, and in very serious cases, expulsion. If you either share your code or copy someone else’s code, you will be given a **0** on your lab and can face further disciplinary action.

In other words, don’t cheat please!