Data Structures and Algorithms

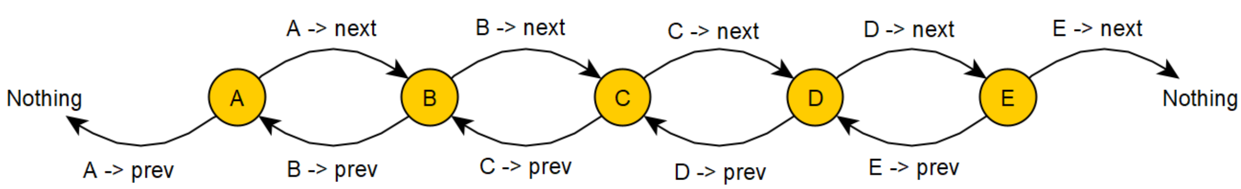
Lab 3: DList

Overview

Linked lists are the next data structure you will be creating. They are a non-contiguous sequence container, and will offer some advantages over vectors/DynArrays, namely a much more efficient way to remove data.

You will be creating a doubly-linked list, which allows for bi-directional traversal, as well as a much more efficient way to insert and remove data at any point in the sequence. Because the data is stored non-contiguously, this will require a good understanding of pointers and dynamic allocations.

Here is an illustration of a doubly-linked list with 5 nodes (A-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 **prev**ious 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.

Linked lists can be hard to mentally visualize. Do not be afraid to diagram out the steps needed to perform the necessary steps needed for the various methods.

Things to Review

- Dynamic memory
- Pointers
- Pointers to classes/structs
- Structs
- Explicitly calling non-default constructors

New Topics

- Advanced usage of pointers
- Nested classes/structs

Data Members

This class has three user-defined types you will be working with. Make sure to familiarize yourself with what data members are in each type, and what they represent.

Node

data	The value being stored (think of the "element" in an array)
next	A pointer to the next node in the list
prev	A pointer to the previous node in the list

Iterator

mCurr	A Node pointer representing the current position of the iterator

DList

mHead	A pointer to the "front" Node in the list
mTail	A pointer to the "back" Node in the list
mSize	The current number of Nodes allocated

Methods

Node Constructor

- Set the data members to the values of the parameters passed in

DList Constructor

- Set all data members to reflect that no nodes are currently allocated

AddHead

- Dynamically add a Node to the front of the list
- Update the **head** to point to the newly added node
- Update the **size** of the list
- *Don't forget to link the nodes together before updating the head*

AddTail

- Dynamically add a Node to the back of the list
- Update the **tail** to point to the newly added node
- Update the **size** of the list
- *Don't forget to link the nodes together before updating the tail*

Clear

- Free up the memory for all dynamically allocated nodes
- Set all of the data members back to their default state
- *Remember there are more nodes than just the head and tail*
    - This will require some form of loop

Destructor

- Free up the memory for all dynamically allocated nodes *(There's a method that does this)*

Begin

- Creates and returns an Iterator that points to the head node

End

- Creates and returns an Iterator that points to the Node after the last valid node in the list

Iterator Pre-Fix Increment (++)

- Moves the Iterator to the next node in the list and returns it

Iterator Post-Fix Increment (++)

- *Post-fix operators take in an int to allow the compiler to differentiate*
- Moves the Iterator to the next node in the list and return an Iterator to the original position
- *This will require a temporary variable*

Iterator Pre-Fix Decrement (--)

- Moves the Iterator to the previous node in the list and returns it

Iterator Post-Fix Decrement (--)

- *Post-fix operators take in an int to allow the compiler to differentiate*
- Moves the Iterator to the previous node in the list and return an Iterator to the original position
- *This will require a temporary variable*

Iterator Dereference (*)

- Return the data of the node the Iterator is pointing to

Insert

- ***Do not be afraid to diagram this!***
- Dynamically allocate a Node and insert it **in front** of the position of the passed-in Iterator
- There are three special cases for this method, depending on what the Iterator is storing
    - **Empty List**
        - Iterator will be storing a null pointer, so the list needs to be started
        - *There's a method to help with this*
    - **Head**
        - Iterator will be storing a pointer to the head of the list
        - *There's a method to help with this*
    - **Anywhere else**
        - Iterator is storing a pointer to another node (even the tail)
        - Link the nodes before and after the inserted nodes
        - This will require setting a total of four next/prev pointers
- In all cases, the passed-in Iterator should be updated to store the newly inserted node

Erase

- ***Do not be afraid to diagram this!***
- Delete the node stored in the passed-in Iterator
- This will require some pointers to be adjusted before the deletion
- In most of these cases, a temporary pointer will be required
- There are four special cases for this method, depending on what the Iterator is storing
    - **Empty List**
        - Iterator will be storing a null pointer
        - Since there is nothing to remove, the method can be exited
    - **Head**
        - Iterator will be storing a pointer to the head of the list
        - Will need to update the head pointer

    - **Tail**
        - Iterator will be storing a pointer to the tail of the list
        - Will need to update the tail pointer
    - **Anywhere else**
        - Iterator is storing a pointer to another node
        - This will require linking the nodes before and after the node to erase together
- In all cases, the passed-in Iterator should be updated to store the node **after** the erased node

Assignment Operator

- Assigns all values to match those of the object passed in
- Clean up existing memory before the deep copy *(There's a method that does this)*
- Deep copy the entire list
    - This requires some type of loop to move through the passed-in list
    - *Look at your other methods, as there are some that can make this very easy*
- If the size has not already been updated, shallow copy it

Copy Constructor

- Creates a copy of the object passed in
- Deep copy the entire list
    - This requires some type of loop to move through the passed-in list
    - *Look at your other methods, as there are some that can make this very easy*
- If the size has not already been updated, shallow copy it
- *Remember that data members are not automatically initialized in C++*