Cooper Gilliland

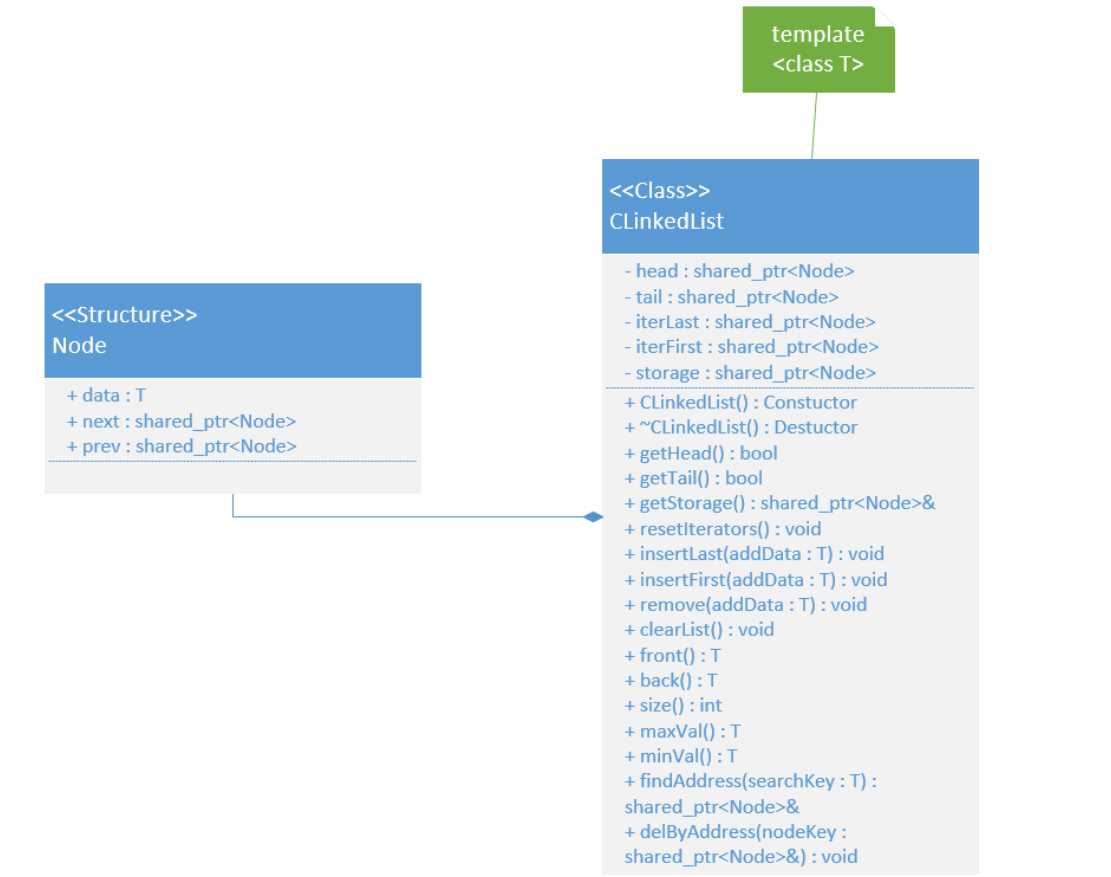
CSC382 – 6/3/17

Linked List Lab Report

Introduction: A linked list in C++ should be a class object that takes a templated data type. Within that class there should be a structure (ideally named Node) that contains a variable of the class type template, and two smart pointers that both point to Nodes. Ideally, these smart pointers will be named next and previous. Using supporting functions, the class should be able to add new nodes, delete nodes, and traverse the list.

The way nodes are added is by linking the smart pointers together. For example, if a node were added to the front it would do so by taking the new node's previous pointer and setting it equal to the list's front sentinel node. This is followed by setting the front sentinel's next pointer to equal the new node. Next, the front sentinel is set equal to the new node. Finally, the front sentinel's next pointer is set to NULL. Using the opposite pointers with the back sentinel will result in the new node being added to the other end of the current list.

Programmer's Guide: Following is a UML class diagram for the associated project. Afterword a description and use example of each class function for the linked list will be provided.



Within the private section of the CLinkedList class are five shared pointers that point to the addresses of object of the structure called Node (Node will be described farther down). The head and tail objects serve as the list's sentinel nodes. These will be on either end of all list elements, and exist to give the list easy points to start and end. The pointers named iterHead and iterTail serve as iterators for the list. They will be used to traverse the list's structure without necessarily modifying the contents of the list. Finally, the storage pointer is used to store the address to a specific node, if that node's address is being considered for further use.

The public section of the class contains a structure named Node. This Node structure contains a templated data type, and two shared pointers of type Node named next and previous. The purpose of all this is to allow the pointers to point to the addresses of other Nodes,

effectively linking those Nodes together. This is the heart of the Linked List, where the data is stored, and the ability to link data is created.

Following is a description of the meaning and operation of each class function, as well as a usage example of each.

- CLinkedList();

  This is the class constructor. It is never explicitly called. When a new class object is created, this will set the head and tail variables to NULL, and output a message to the console.

- ~CLinkedList();

  This is the class destructor. It is never explicitly called. When a class object goes out of scope, this will run, outputting a message to the console and deconstructing the object.

- bool getHead();

  This function is used to check if there are any items in the current list. It does so by checking the variable head. If head is not NULL, it will return true. Example: if (list.getHead() == true) {//list is not empty}

- bool getTail();

  This function is used to check if there are any items in the current list. It does so by checking the variable tail. If tail is not NULL, it will return true. Example: if (list.getTail() == true) {//list is not empty}

- shared_ptr<Node>& getStorage();

  This function only returns a reference to the variable storage. Example: cout<< list.getStorage(); will print the address of the storage node.

- void resetIterators();

This function resets iterators to proper positions after use. Example: list.resetIterators(); will set iterFirst equal to head, and iterLast equal to tail.

- void insertLast(T addData);

  This function will insert a new node into the back of the current list. Example: list.insertLast(12); will insert the integer 12 into the back of the current list.

- void insertFirst(T addData);

  This function will insert a new node into the back of the current list. Example: list.insertLast(117); will insert the integer 117 into the front of the current list.

- void remove(T addData);

  This function will take an argument and iterate through the current list, comparing the argument to each node's data. If a match is found, the function directs the adjacent node to point at each other, cutting out the current node for garbage collection. Example: list.remove(-144);

- void clearList();

  This function will iterate through the list from both ends simultaneously, stopping when both iterators have reached each other's current position As it runs, it releases memory, destroying the nodes in the process by setting them to point to NULL as it passes them. Example: list.clearList();

- T front();

  This function returns the first piece of data in the list. Example: cout << list.front(); will output the associated value to the console.

- T back();

  This function returns the last piece of data in the list. Example: cout << list.back(); will output the associated value to the console.

- int size();

  This function will return the size of the current list by iterating through the list, adding one to the total after each iteration. it will return the size as an integer and reset the iterators by calling the resetIterators() function. Example: cout << list.size(); will output the associated value to the console.

- T maxVal();

  The following function will iterate through the list searching for a maximum value. Once completed, it will return that value and reset the iterators by calling the resetIterators() function. Example: cout << list.maxVal(); will print the maximum value found in the list to the console.

- T minVal();

  The following function will iterate through the list searching for a minimum value. Once completed, it will return that value and reset the iterators by calling the resetIterators() function. Example: cout << list.minVal(); will print the minimum value found in the list to the console.

- shared_ptr<Node>& findAddress(T searchKey);

  The following function will search through the current list comparing user input to the stored data. When a match is found the function returns the memory address of the first matching node, and stores the address in the storage variable as a redundancy. It then

resets the iterators by calling the resetIterators() function. Example: list.findAddress(12); will find the first instance of 12 in the list and store that memory address.

- void delByAddress(shared_ptr<Node>& nodeKey);

  This function takes a pointer to a memory address and compares it to the addresses of the current list's nodes. If a match is found, the function directs the adjacent nodes to point at each other, cutting out the current node for garbage collection. It then resets the iterators by calling the resetIterators() function. Example: list.delByAddress(list.getStorage()); will take the address from the storage variable and compare it to each node’s memory address. If a matching address is found, the node is deleted.