ECE250 Project 1 Design Document Matthew Li (m739li)

For this project, I designed two classes: Node and LinkedList. It also includes a header file for the Node class in order to keep the code separated and organized.

Node

In the Node class, I have member variables value (double), name (string), and next (Node *). The next variable is needed to point to the next Node in the linked list, or assigned to NULL if the current Node is the last Node. I have 7 member functions in my Node class including the constructor and destructor.

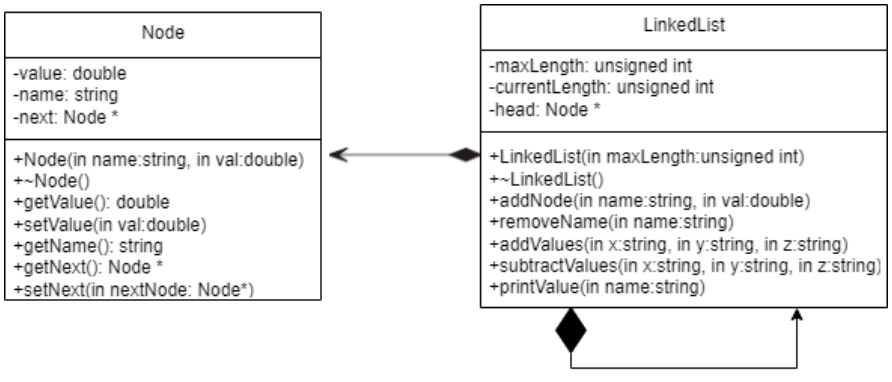
Constructor: Each Node has a specific value, a unique name, and a pointer to the next node in the list. The name and value, which are a string and double respectively, are passed in as parameters to the constructor, and will always have these parameters, so I assign the values to the variables immediately. The initial value for the pointer to the next node is NULL because there is no assigned next node yet.
Destructor: A default destructor is used for this class because each individual node is deleted through the destructor of LinkedList.
getValue(): Returns a double which represents the value of the current node, takes in 0 parameters.
setValue(): No return value, takes in 1 parameter (double) which represents the new value of the current node.
getName(): Returns a string which represents the name of the current node, takes in 0 parameters.
getNext(): Returns a pointer to a node which represents the next node in the list, takes in 0 parameters.
setNext(): No return value, takes in 1 parameter (Node *) which represents the new next node in the list.

Linked List

In the LinkedList class, I have member variables maxLength(unsigned int), currentLength(unsigned int), and head (Node *). The head variable stores and updates the head if needed in order to have a starting node to iterate through the list, or assigned to NULL if there are no nodes in the list. I have 7 member functions in my LinkedList class including the constructor and destructor. Each function's algorithm other than the constructor and destructoris explained in the "Asymptotic Upper Bound Runtime" section.

Constructor: Each LinkedList consists of a head, a maximum length, and a current length. The maximum length of the LinkedList, an unsigned integer, is passed in as a parameter to the constructor, and will always have this parameter, so I assign this value to the variable "maxLength" immediately. When creating a LinkedList, I assigned the head to the value NULL, and the current length to the value 0 since there are no Nodes in the list yet.
Destructor: When the destructor is called for LinkedList, I start at the head of the list, and iterate through the entire list until I reach the end, storing the next node and deleting the current one. In the event the list is empty, my loop condition is to check if the head is NULL, in which case I would exit the loop immediately and not delete any nodes.
addNode(): No return value, takes in two parameters (string and double) which represent the name and value of the node to add to the list.
removeName: No return value, takes in one parameter (string) which represents the name of the node to remove.
addValues(): No return value, takes in three parameters (all strings) which represent the two names of the nodes to add and the name of the node to store the added value.
subtractValues(): No return value, takes in three parameters (all strings) which represent the two names of the nodes to add and the name of the node to store the added value.
printValue(): No return value, takes in one parameter (string) which represents the name of the node to print.

UML Diagram



Asymptotic Upper Bound Runtime

In the runtimes below, n represents the number of variable names in the list.

CRT: O(1) - This command assigns the variable “list” to a new LinkedList with a maximum length of val. This takes one clock cycle to complete, thus has a runtime of O(1).

DEF: O(n) - This command calls the member function “addNode()” of the LinkedList class with the name and value of the new node. addNode() first checks if the list is not empty and the current length doesn’t match the maximum length of the list. Since the runtime of this command must be O(n), I can only iterate through the list once, starting at the head, until the current Node’s “next” pointer is NULL. For each node, I check if the name of each node matches the name of the new node. If a duplicate name is detected at any time, it immediately outputs “failure”, then returns to the main function. If none of the names match the name of the new node, I increment the current length of the list and set the current node’s “next” pointer to the new node.

ADD: O(n) - This command calls the member function “addValues()” of the LinkedList class with the two node names to add and the last node name to store the added value. Since the runtime of this command must be O(n), I can only iterate through the list once. addValues() iterates through the list, starting at the head, checking if the name of each node matches any of the three names passed into the function. If either of the two node names to add are found, I store the value in a variable, and set a flag to true indicating that the variable name was found in the list. If the name to store the added value is found, the pointer to the node is stored in a different variable. If at any time all three names are found, I output “success”, change the value of the pointer to the sum of the other two variables and return to the main function. If none of the names are found at the end, I output “failure” and return to the main function.

SUB: O(n) - This command calls the member function “subtractValues()” of the LinkedList class with the two node names to subtract and the last node name to store the subtracted value. This function works exactly the same as the ADD function, except if all three names are found, the new value is the subtraction of the two values rather than the sum. This has a runtime of O(n) since I am only iterating through the list once and the other conditions are run in constant time.

REM: O(n) - This command calls the member function “removeName()” of the LinkedList class with the name of the node to remove. removeName() first checks if the list is not empty, then checks if the name of the head matches the name of the node to remove. If it does, I set head to be the next node and return to the main function. If not, then I iterate through the list, checking if the name of the next node matches the name of the node to remove. Since the runtime of this command must be O(n), I can only iterate through the list once. If at any time the name matches, I decrease the current length, set the previous node’s “next” pointer to be the next node, delete the current node, and return to the main function. If I iterate through the whole list and never find the name, I output “failure” and return to the main function.

PRT: O(n) - This command calls the member function “printValue()” of the LinkedList class with the name of the node to print. Since the runtime of this command must be O(n), I can only iterate through the list once. printValue() iterates through the list, checking if the name of the current node matches the name of the node to print. If at any time it matches, I output the value of the node and return to the main function. If I iterate through the whole list and never find the name, I output that the variable was not found, and return to the main function. This has a runtime of O(n) since I only iterate through the list once, and the other conditions are run in constant time.

END: O(n) - Immediately breaks out of the while loop that accepts input. This takes one clock cycle to complete, thus has a runtime of O(1).