CSE 220 – Assignment 5 – Data Structures Options

Topics:

- C/C++ Syntax
- Pointers
- Functions
- Dynamic Allocation of Memory
- Data Structures: Linked Lists
- Object Orientation

Overall Specifications:

You are to create a Linked List or Binary Search Tree data structure from scratch. This structure should be Templated so that any data could be stored within it.

For a maximum of B on specifications – you can make your structure non-templated – meaning it stores only the data required for the problems.

The primary goal of this assignment is to make a Linked List that you can use and re-use. The Linked List itself is the majority of the specifications grade.

Specifications Scoring Breakdown:

Templated Linked List/BST + Problem – 100% of specifications

Templated Linked List/BST only – 80% of specifications

Untemplated Linked List/BST + Problem – 80% of specifications

Untemplated Linked List/BST only – 70% of specifications

** BIG GIANT NOTE – TEMPLATES AND FILES **

When you use a templated type in C++ ALL templated code must be done in the .h file. This means that ALL of your methods will be defined in the .h file as well as your class.

You should still forward declare Classes above then Methods below.

Your .h file should have your LinkedList/BST class and your Node class and the method definitions for both. Remember to use your :: operator correctly.

Feel free to use friendship if needed.

Option 1

Zombie Conga Party! - Linked Lists

You will create a Linked List Class and a Node Class

The Linked List should contain the following methods in its public interface:
NOTE: T stands for the template data type, so T data means the variable of type T (whatever T is)

- Constructor
- Destructor
- AddToFront(T data) – create a node containing T data and add it to the front of the list
- AddToEnd(T data) – create a node containing T data and add it to the end of the list
- AddAtIndex(T data, int index) – create a node containing T data and add it to the list at index. The new node containing the data will be the #index node in the list. Return boolean for success or failure (optional: you could also return an integer with failure codes since this method can fail multiple ways)
- RemoveFromFront() – Delete first item and return its contents
- RemoveFromEnd() – Delete last item and return its contents
- RemoveTheFirst(T data) – find first instance of T data and remove it
- RemoveAllOf(T data) – find each instance of T data and remove it
- ElementExists(T data) – Returns a T/F if element exists in list
- Find(T data) – Look for data in the list, return a pointer to its node
- IndexOf(T data) – returns an index of the item in the list (zero-based)
- RetrieveFront – returns the data contained in the first node, *does not delete it*
- RetrieveEnd – returns the data contained in the last node, *does not delete it*
- Retrieve(int index) – returns the data contained in node # index, *does not delete* it, returns null if index is out of bounds or data does not exist
- PrintList – Loop through each node and print the contents of the Node
- Empty – Empty out the list, delete everything
- Length – How many elements are in the list

More methods private or public should be created as needed to facilitate the functionality of the Interface methods. If you feel your list needs more functionality, feel free to create it.

*As per usual, I have **not** made a perfect representation of the parameters you might need, etc. I have made suggestions where appropriate, but you might need more depending on how you code things.*

Node Class

- Constructor
- Destructor
- Getters & Setters

The node class should be fairly rudimentary. Ideally, it should be templated so you can store anything in it.

Description:

You are going to create a silly Zombie Conga Line using your Linked List.

Each node is going to store a Zombie in it. Zombies can be Red, Yellow, Green, Blue, Magenta and Cyan.

Every turn you will randomly generate a Zombie object and an action. You will then perform that action using the Zombie object as the parameter for that action.

Actions:

- Engine!
  - This zombie becomes the first Zombie in the conga line
- Caboose!
  - This zombie becomes the last zombie in the conga line
- Jump in the Line!
  - This zombie joins the conga line at position X where X <= length of the linked list
- Everyone Out!
  - Remove all matching zombies from the linked list
- You're done!
  - Remove the first matching zombie from the linked list
- Brains!
  - Generate two more matching Zombies and add one to the front, one to the end and one to the middle.
- Rainbow Brains!
  - Add this zombie to the front, then add one of each zombie color to the end of the conga line.

Every 5 rounds remove the Head zombie and Tail zombie.

Setting up the List:

Set up the initial Conga Line by running these actions:

1. Run a Rainbow Brains! Action
2. Run a random number (between 2 and 5) of Brains actions

User Interface:

Ask the user how many rounds they want to run.

Then generate that many random actions and fulfill them.

If the conga line every empties completely due to an action tell the user that the Party is Over.

Once the number of rounds has finished. Ask the user if they want to continue the party or end.

If they choose to continue ask them for a new number of rounds to run.

Output:

Each round you'll output the entire Linked List. You can represent each zombie as a single character corresponding to their color (R, Y, G, B, M, C).

Have fun, make silly messages and color the output if you like!

You'll show the zombie generated and the action generated.

Then you'll show the outcome of the action.

Spelling it out for you:

You should create the following classes:

1. LinkedList
2. Node
3. Zombie

Your Nodes should store Zombies.

Your LinkedList is made of Nodes.

There is no inheritance involved here!

Hints:

Build robust constructors and use them! Your Node and your Zombie will benefit highly from good constructors.

Do yourself a favor and overload the == operator in your Zombie. It will make life easier!

Overload the cout for your Zombie. It'll make your printList method easier.

Sample output:

```
Round: 0
The Zombie Party keeps on groaning!
Size: 16 :: [R] [M] [G] [R] [R] [G] [G] [R] [M] [B] [Y] [M] [C] [G] [M] [R]
B zombie jumps in the front of the line! (ENGINE)
The conga line is now:
Size: 17 :: [B] [R] [M] [G] [R] [R] [G] [G] [R] [M] [B] [Y] [M] [C] [G] [M] [R]
******************


Round: 1
The Zombie Party keeps on groaning!
Size: 17 :: [B] [R] [M] [G] [R] [R] [G] [G] [R] [M] [B] [Y] [M] [C] [G] [M] [R]
M zombie jumps in the front of the line! (ENGINE)
The conga line is now:
Size: 18 :: [M] [B] [R] [M] [G] [R] [R] [G] [G] [R] [M] [B] [Y] [M] [C] [G] [M] [R]
******************


Round: 2
The Zombie Party keeps on groaning!
Size: 18 :: [M] [B] [R] [M] [G] [R] [R] [G] [G] [R] [M] [B] [Y] [M] [C] [G] [M] [R]
C zombie pulls up the rear! (CABOOSE)
The conga line is now:
Size: 19 :: [M] [B] [R] [M] [G] [R] [R] [G] [G] [R] [M] [B] [Y] [M] [C] [G] [M] [R] [C]
******************


Round: 3
The Zombie Party keeps on groaning!
Size: 19 :: [M] [B] [R] [M] [G] [R] [R] [G] [G] [R] [M] [B] [Y] [M] [C] [G] [M] [R] [C]
Y zombie brought a whole party itself! (RAINBOW BRAINS!)
The conga line is now:
Size: 26 :: [Y] [M] [B] [R] [M] [G] [R] [R] [G] [G] [R] [M] [B] [Y] [M] [C] [G] [M] [R] [C] [R] [G] [B] [Y] [M]
[C]
******************


Round: 4
The Zombie Party keeps on groaning!
Size: 26 :: [Y] [M] [B] [R] [M] [G] [R] [R] [G] [G] [R] [M] [B] [Y] [M] [C] [G] [M] [R] [C] [R] [G] [B] [Y] [M]
[C]
Y zombie brings its friends to the party (BRAINS!)
The conga line is now:
Size: 29 :: [Y] [Y] [M] [B] [R] [M] [G] [R] [R] [G] [G] [R] [M] [B] [Y] [Y] [M] [C] [G] [M] [R] [C] [R] [G] [B]
[Y] [M] [C] [Y]
******************
```

NOTE:

You don't have to use my messages, I just made them on a whim. Do however follow my lead and put a label showing what action the message represents.

Don't forget to output the list before and after the action.

Option 2:

Word Frequency Analysis – Binary Search Tree

You will create a Binary Search Tree Class and a Node Class/Struct

The Binary Search Tree should contain the following methods in its public interface:

- Constructor
- Destructor
- Insert(T data) – create a node containing T data and add it to the tree (REMEMBER: NO DUPLICATES)
- Remove (T data) – find instance of T data and remove it
- ElementExists(T data) – Returns a T/F if element exists in tree
- Find(T data) – Look for data in the list, return a pointer to its node
- ToArray – Create an array from the contents of the tree and return it
- Empty – Empty out the list, delete everything
- Length – How many elements are in the list

More methods private or public should be created as needed to facilitate the functionality of the Interface methods. If you feel your tree needs more functionality, feel free to create it.

Note: I'm leaving some of these vague so that you can customize your tree a bit. For example, insert often returns a pointer to the node that was created when you add the element to the tree – or if it was a duplicate, it returns a pointer to the node where the element was found. Sometimes insert just returns a Boolean stating that the operation was successful or not. Etc.

Description

The goal will be to parse a large text file and determine the word frequency of every word in the text file.

You will use file input to read in, analyze and output statistics about the file. You should provide an adequate user interface to accomplish the following tasks:

- Read in a text file and analyze it. The text files will come from the Gutenberg Project (http://www.gutenberg.org/) so you should be able to possibly analyze an entire novel.
- Provide the user a summary of the analysis including:
  - Total words
  - Total unique words
  - The 5 most and least frequently used words
    - Hint: there are multiple ways of doing this, some more brute force then others
- Allow the user to input a word and retrieve the frequency of that word
- Allow the user to output the frequency analysis to a file for review
  - Bonus Opportunity: Give the option to sort alphabetically or by frequency

You will use a Binary Search Tree to accomplish these goals (and any other algorithms or data structures needed to accomplish those goals - i.e.: if you need a stack for certain algorithms, you may use it ... if you need a linked list, you may use it ... *but remember the star of the show is the BST*)

Your word counts should ignore capitalization, so **the**, **The**, **THE**, and **tHe** all increase the count for the word "the" by one. For purposes of this assignment, a word is any consecutive string of letters and the apostrophe character, so **don't** counts as a single word, and **best-selling** counts as two words: **best** and **selling**. Notice that a blank space will not necessarily occur between two words. Numbers such as **27** and **2/3** will NOT be counted as words.

NOTES:

Literally the hardest part of this assignment is parsing the words.

The Node SHOULD NOT store the frequency.

Make a WordEntry class to store in your Nodes. *The WordEntry should store the word and the frequency.*

Grading of Programming Assignment

The TA will grade your program following these steps:
(1) Compile the code. If it does not compile, If it does not compile you will receive a U on the **Specifications** in the Rubric.

(2) The TA will read your program and give points based on the points allocated to each component, the readability of your code (organization of the code and comments), logic, inclusion of the required functions, and correctness of the implementations of each function.

Rubric:

	Levels of Achievement						
Criteria	A	B	C	D	E	U	F
Specifications Weight 50.00%	100 % The program works and meets all of the specifications.	85 % The program works and produces the correct results and displays them correctly. It also meets most of the other specifications.	75 % The program produces mostly correct results but does not display them correctly and/or missing some specifications	65 % The program produces partially correct results, display problems and/or missing specifications	35 % Program compiles and runs and attempts specifications, but several problems exist	20 % Code does not compile and run. Produces excessive incorrect results	0 % Code does not compile. Barely an attempt was made at specifications.
Code Quality Weight 20.00%	100 % Code is written clearly	85 % Code readability is less	75 % The code is readable only by someone who knows what it is supposed to be doing.	65 % Code is using single letter variables, poorly organized	35 % The code is poorly organized and very difficult to read.	20 % Code uses excessive single letter identifiers. Excessively poorly organized.	0 % Code is incomprehensible
Documentation Weight 15.00%	100 % Code is very well commented	85 % Commenting is simple but solid	75 % Commenting is severely lacking	65 % Bare minimum commenting	35 % Comments are poor	20 % Only the header comment exists identifying the student.	0 % Non existent
Efficiency Weight 15.00%	100 % The code is extremely efficient without sacrificing readability and understanding.	85 % The code is fairly efficient without sacrificing readability and understanding.	75 % The code is brute force but concise.	65 % The code is brute force and unnecessarily long.	35 % The code is huge and appears to be patched together.	20 % The code has created very poor runtimes for much simpler faster algorithms.	0 % Code is incomprehensible

What to Submit?

You are required to submit your solutions in a compressed format (.zip). Zip all files into a single zip file. Make sure your compressed file is labeled correctly - lastname_firstname5.zip.

For this home assignment, the compressed file MUST contain the following:

- Makefile
- README.txt – instructions for using the makefile
- Your code files:
    - lastname_firstname_assn4.cpp
    - lastname_bst.h / lastname_list.h
    - If you didn't template, also include the .cpp files for your .h files
    - Any other files that you created/need
- any other code/library files you create or use for the sake of the assignment

No other files should be in the compressed folder.

If multiple submissions are made, the most recent submission will be graded, even if the assignment is submitted late.

Where to Submit?

All submissions must be electronically submitted to the respected homework link in the course web page where you downloaded the assignment.

Academic Integrity and Honor Code.

You are encouraged to cooperate in study group on learning the course materials. However, you may not cooperate on preparing the individual assignments. Anything that you turn in must be your own work: You must write up your own solution with your own understanding. If you use an idea that is found in a book or from other sources, or that was developed by someone else or jointly with some group, make sure you acknowledge the source and/or the names of the persons in the write-up for each problem. When you help your peers, you should never show your work to them. All assignment questions must be asked in the course discussion board. Asking assignment questions or making your assignment available in the public websites before the assignment due will be considered cheating.

*The instructor and the TA will **CAREFULLY** check any possible proliferation or plagiarism. We will use the document/program comparison tools like MOSS (Measure Of Software Similarity: http://moss.stanford.edu/) to check any assignment that you submitted for grading. The Ira A. Fulton Schools of Engineering expect all students to adhere to ASU's policy on Academic Dishonesty. These policies can be found in the Code of Student Conduct:*

http://www.asu.edu/studentaffairs/studentlife/judicial/academic_integrity.htm

ALL cases of cheating or plagiarism will be handed to the Dean's office. Penalties include a failing grade in the class, a note on your official transcript that shows you were punished for cheating, suspension, expulsion and revocation of already awarded degrees.