COMP 285 (NC A&T, Fall '22) Homework 4 (100 pt)

Fun with Trees

Due 10/13/22 @ 11:59PM ET

Submitting

In order to submit this assignment, you must download your assignment as a ZIP file and upload it to Gradescope. There will be a written component and a coding component.

For the written component, you will write your answers in the corresponding `.txt` files.

For the coding component, you will write your solution in `answers.cpp`. We will use an autograder to test your solutions - the results of which can be seen after submitting to Gradescope. Note that you have unlimited tries to submit. In order to receive full credit, you must also provide documentation on your approach, see the commented sections in the `answers.cpp` file.

Question 1: Warm-Up with Trees/Recursion (10 pts)

a. (7 pt) Remembering from our in-class activity... we implemented a function that, given a binary tree, returned the maximum node value. As a warm up for this homework, write a function that instead returns the *minimum* node value.

For example, for the following tree, your algorithm would return 2 because it's the smallest node value. 

Write your solution in `answers.cpp`.

b. (3 pt) What is the Big-Oh runtime and space complexity of your `findMinOfTree` algorithm where N represents the number of nodes in the tree?

Write your answer in `q1.txt`.

Question 2: Meech Trees (20 pts)

Meech comes to you with the following claim. She says that she came up with a new kind of binary search tree, called *meechTree*, specifically for converting arrays into tree strucutres, allowing faster lookup, insertion, deletion operations!

More precisely, *meechTree* is a data structure that stores comparable elements in a *binary search tree*. She says that *meechTree* supports the following operations:

- **meechInsert(k)** inserts an item with value `k` into the *meechTree*, maintaining the BST property. It does not return anything. It runs in time `O(1)`.
- **meechSearch(k)** finds and returns a pointer to node with value `k`, if it exists in the tree. It runs in time `O(log (n))`.
- **meechDelete(k)** removes and returns a pointer to an item with value `k`, if it exists in the tree, maintaining the BST property. It runs in time `O(log (n))`.
- **meechTree()** initializes an empty tree.

Above, `n` is the number of items stored in the *meechTree*. She says that all these operations are deterministic, and that *meechTree* can handle arbitrary comparable objects.

a. (5 pt) You want to get a better understanding of the *meechTree*. To do so, you think it'd be helpful to write an algorithm that takes as input a vector of unsorted elements and creates a *meechTree* using the methods above. Write complete pseudocode for how to transform an unsorted input into a BST.

```
// Input: A is an unsorted vector of n comparable elements.
// Output: A binary search meech tree
algorithm makeMooseTree(A):
  // Your pseudocode here...
```

Write your answers in `q2.txt`.

b. (10 pt) Now that you understand how to make *meechTrees*, you consider the opposite. How can you go from a *meechTrees* to a sorted vector?

Hint: If you're having trouble, refresh your memory on tree traversal orders!

Write pseudocode of an algorithm that takes as input a *meechTree* and returns a sorted vector. To get you started, since the *meechTree* is still a kind of binary search tree, you can access the root of *meechTrees* by calling `meechTree.root()`.

```
// Input: T is a mooseTree with n elements.
// Input: A is the vectors where we should add elements into.
algorithm getElements(T, A):
  // Your pseudocode here...
```

Write your answers in `q2.txt`.

c. (5 pt) What is the Big-Oh runtime of your algorithm from part b where N is the number of nodes in the meechTree?

Write your answers in `q2.txt`.

Question 3: Spellcheck (30 points)

After Symphony's long afternoon reading *Catcher in the Rye*, she decides to create a spellchecking program so she keep a "dictionary" of words she comes across then look up whether certain spellings are valid. She also wants to support a feature enabling users to add custom words to the dictionary.

She can implement this functionality using a trie (pronunced "try"). A trie is a tree where each letter is a `char` and each path from the root to a leaf represents a valid word in the dictionary. The definition of a tree can be found in `TreeNode.h`. Note that it is a general N-ary Tree (*not* a Binary Tree).

- This trie spells out "ace", "acid", "act", “actor”, “be”, "beat", "bear", "cab", "cob"
- To look up whether a word in the dictionary, we simply traverse the trie to see whether the word is in the trie; to add a word in the dictionary, we insert it into the trie. Although we won't implement it in this homework, you can imagine how tries can also be used to implement autocomplete functionality.
- Note that the value in the root of the tree should be ignored when making words.
- Note that an extra node with a "$" character denotes when a word has ended. This is necessary to know we've reached the end of the word, otherwise, we'd be unable to represent both "act" and "actor" as above without also accidentally representing "ac" and "acto".

All of the below should be implemented in `answers.cpp` and you can see where it is tested in `main.cpp`'s main. You may assume the input parameters are not `nullptr`s and only valid dictionary tries will be passed in.

a. (10 pts) `isValidWord`

This method is the crux of the spellchecking functionality. Given a word and a dictionary, `isValidWord` returns whether the word is in the dictionary.

For example, if `dict` is the example trie above,

```
std::cout << isValidWord(dict1, "ace") << std::endl;  // 1
std::cout << isValidWord(dict1, "ac") << std::endl;  // 0
std::cout << isValidWord(dict1, "actor") << std::endl;  // 1
std::cout << isValidWord(dict1, "actors") << std::endl;  // 0
```

Write your `isValidWord` solution in `answers.cpp`.

b. (10 pts) `addWord`

This method adds a new word to the dictionary and returns whether the insertion actually happened. That is to say, if the word is already present, the method should return false and leave the trie unaltered.

Your trie should be "normalized," meaning that the added word should leverage the existing prefixes in the trie as much as possible. For example, adding "co" to the dictionary should not result in a complete separate branch, but rather only construct a '$' node under the existing '*' -> 'c' -> 'o' prefix.

Write your `addWord` solution in `answers.cpp`.

c. (10 pts) `dictEqual`

Now that dictionaries can be mutated, we want to be able to add a memory-location-agnostic comparator to see whether two dictionaries are equal. We consider two dictionaries to be equal iff they represent the same set of words, even though the ordering of children may differ or the memory locations of the nodes are not the same.

If `dict` is the example trie in the diagram above,

```
TreeNode<char>* dict1 = makeTrieExample1();
TreeNode<char>* dict2 = makeTrieExample2();
TreeNode<char>* dict3 = new TreeNode<char>('*');
TreeNode<char>* dict4 = new TreeNode<char>('*');
std::cout << dictEqual(dict1, dict2) << std::endl;  // 0, they are different
std::cout << dictEqual(dict3, dict4) << std::endl;  // 1, both are empty
std::cout << dictEqual(dict1, dict3) << std::endl;  // 0, one is empty and the other isn't
```

Hints

- You may consider using `std::unordered_map<char, TreeNode<char>*>`.

Write your `dictEqual` solution in `answers.cpp`.

Question 4: Interview Practice Target Sum (20 pts)

a. (15 pt) Given the root of a binary tree and an integer `targetSum`, return true if the tree has a root-to-leaf path such that adding up all the values along the path equals `targetSum`.

For example, for the following tree, if we passed in `targetSum = 26`, your algorithm would return `true` because of the path `2 -> 7 -> 6 -> 11` and `2 + 7 + 6 + 11 = 26`. However, if we passed in `targetSum = 7`, we would return `false` since there is not a path from root to leaf that sums to `7`. The fact that there is a `2 -> 5` path doesn't mean we return true; the path must be root to leaf.

Write your solution in `answers.cpp`.

b. (5 pt) What is the Big-Oh runtime of the algorithm from part a where N represents the number of nodes in the tree?

Write your answer in `q4.txt`.

Question 5: Applying to Jobs, Career Prep Module #3 (20 pts)

Thus far, we've polished our resumes and began prepping for behavorial interviews. Now it's time to actually apply to internships or full time opportunities (depending on your graduation year)!

Please apply to 10 engineering internships: software, hardware, industrial, etc. Submit screenshots of confirmation pages or emails to this form following the naming convention LastName_FirstName_Application# (e.g. Lucas_Chris_1, Lucas_Chris_2, ... Lucas_Chris_10).

A few things before you get started...

- If you'd like to run your resume by me one more time before submitting, please send it to me in a Word or Google doc!
- If you need somewhere to start, head over to the announcements on the course website for internship listings. You can also use Handshake, LinkedIn, or (obviously) Google.
- I highly recommend optimizing for job postings that only require submitting a resume and basic personal information (ie. forgo cover letters) - quantity is key!

What if I'm already committed?

If you already have committed to an internship, *huge congrats*! In lieu of applying to 10 jobs then, please submit 10 links to software, hardware, etc. engineering internships for your peers. They cannot be internships already flagged on the course website.

Note if you already have an internship offer or offers, but have yet to commit, I recommend applying to 10 jobs to increase your options. However, for this assignment, you may pursue either route: applying yourself or submitting links to internships for others.