CS202 Data Structures

Assignment 2

On building your own search engine

Due: 11pm on Tuesday, 13th March, 2018 (on LMS)

Late Submission Policy

This assignment can be submitted till 11pm on Friday, 16th March, 2018 with a 10% penalty per day.

In this assignment, you will implement a small-scale *search engine* using different data structures. You will implement this search engine using *trees*, *binary search trees,* and *AVL trees*. This will help you in appreciating the tradeoffs (e.g., running times vs. memory requirements) associated with using different data structures.

This assignment is more intense than the first assignment and contains three tasks so start early!

The course policy about plagiarism is as follows:

1. Students must not share actual program code with other students.
2. Students cannot copy code from the Internet.
3. Students must be prepared to explain any program code they submit.
4. Students must indicate any assistance they received.
5. All submissions are subject to automated plagiarism detection.

Students are strongly advised that any act of plagiarism will be reported to the Disciplinary Committee.

Task-1: [25 Points]

In Task-1, you will write a program that searches for files on your personal computer efficiently (*and obviously without using the Windows search feature*). Your program will read the contents of a portion of your Windows directory from a text file. Your program will save the directory contents in a tree data structure. Then, the user of your program can search for exact matches of any file or directory name.

The basic layout of this directory tree class is provided to you in *tree.h.* You may add other functions to this header file.

Input Format:

The input text file may contain a directory list in the following format:

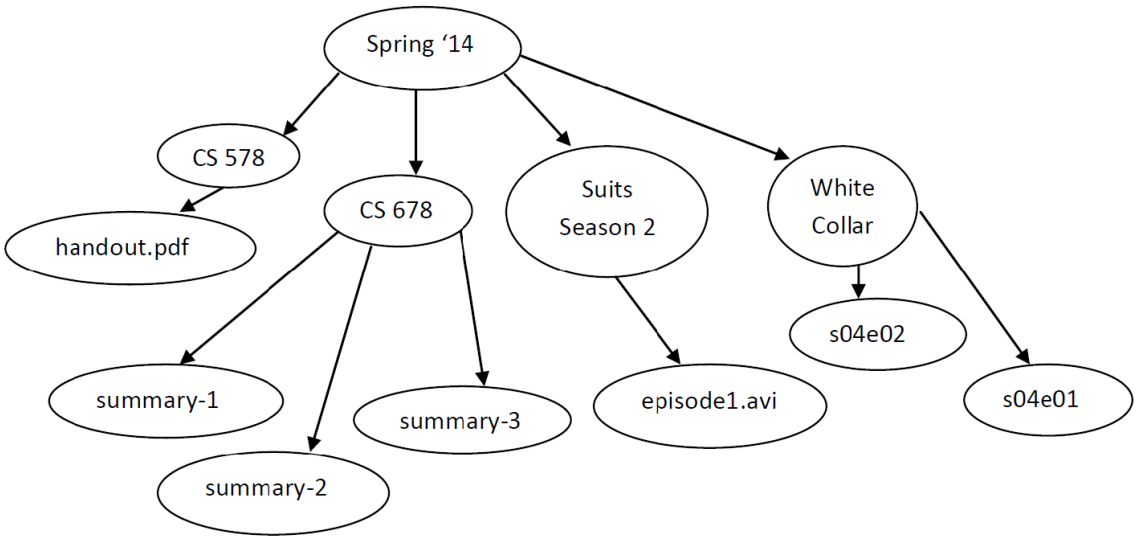
```
\Spring '17\CS 578
\Spring '17\CS 678
\Spring '17\Suits Season 2
\Spring '17\White Collar
\Spring '17\CS 578\handout.pdf
\Spring '17\CS 678\summary-1.docx
\Spring '17\CS 678\summary-2.docx
\Spring '17\CS 678\summary-4.docx
\Spring '17\CS 678\summary-5.docx
\Spring '17\Suits Season 2\episode1.avi.
\Spring '17\Suits Season 2\torrent.txt.txt
\Spring '17\White Collar\s04e01.avi
\Spring '17\White Collar\s04e02.avi
```

For simplicity, in this assignment we assume that there are no duplicate file names or directory names.

As part of the assignment, you will have to first generate a text file of all the directories on your own PC. This can be easily done by opening the command prompt on your windows machine and then typing the command "**dir /s /b > input.txt**". This will create a file named input.txt in the current directory. Transfer this file to your mars account or Linux machine so that your program may use it.

Directory Tree:

Below is how the tree data structure corresponding to the above sample input looks like on paper. You should program the tree by using the sibling pointers as taught in the class.



The constructor for the tree class expects the filename as the input parameter. The program expects the specified file to contain the directory contents as mentioned earlier. Inside the constructor, it will read the directories from that file and load into memory in a tree data structure. Your tree should support two operations:

1) **Locate**: Takes the name of the file to search for, and returns the complete path of the file in a vector i.e., If the user enters "**s04e02**" the returned vector would contain **"s04e02" "White Collar"** "**Spring'14"** in this order (root folder is the last element of the vector). You should also print out the time taken for the search.

2) **Lowest Common Ancestor:** Given names of two different files or directories, your function should return the *Lowest Common Ancestor* (LCA) for them. For example, the LCA for "**summary-2**"and "**summary-3**" is "**CS 678**".

3) **Move Folder (Bonus Task):** Given names of two different directories, your function should move one of the entire directory to the other. You can use test_bonus.cpp to check your solution. For sake of simplicity, you can assume the leaf nodes are directories. Secondly, you can not move an ancestor director to any of its descendant directory.

Task-2: [50 points]

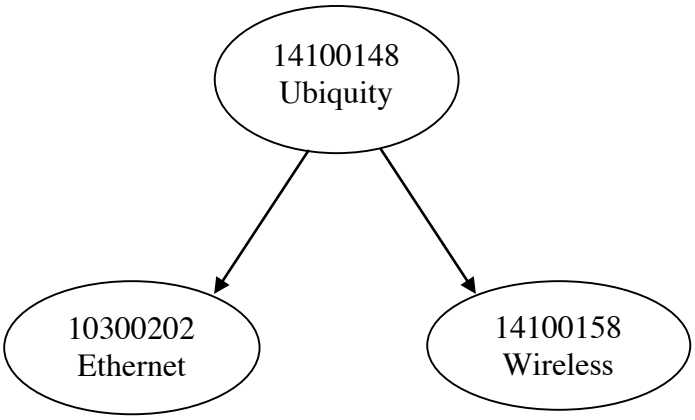
In this task, you will write a program that implements the functionality of a **Binary Search Tree (BST)**. The nodes in the BST are to be arranged according to the relative magnitude of the keys. The basic layout of this BST class is provided to you in *bst.h*

Input Format:

Each line includes a value (a string) followed by its unique key (which can be a string, integer, or a float).'~' Marks the end of a value. For example:

```
Ubiquity~ 14100148
Wireless~ 14100158
Ethernet~ 10300202
```

The corresponding BST would look like:



Requirements:

1. Your BST should support **insertion** of a **key-value pair**. Insertion should not violate any properties of a BST and should be with respect to the key. The **insert** function in bst.cpp calls **insertHelper.** You are required to implement **insertHelper** using a recursive algorithm. In it's current form, the **insertHelper** function calls **balance(node<T>* p).** Write all your code above the call to **balance(node<T>* p)**. You do not need to change **balance(node<T>* p)** at this point.
2. BST should support **deletion** of a **key-value pair**. Deletion should properly free the memory and rearrange the tree to maintain its properties. The **delete_node** function in bst.cpp calls the **remove** function**.** You are required to implement **remove** using a recursive algorithm. Similar to **insertHelper**, the **remove** function calls **balance(node<T>* p).** Write all your code above the call to **balance(node<T>* p)**. You should also implement **findmin(Node<T>* p)** and **removemin(Node<T>* p)** functions which find and delete the smallest Node from a subtree respectively.
3. BST should support the **search** operation. Given a **key,** it should return a pointer to the node that holds that **key**.
4. BST should have a function that returns the **height** of the tree in O(1) time. Every time you call the **insert** or **delete_node**, the tree's height may change. The **balance(node<T>* p)** is there to ensure that the height of every node is updated after

an **insert** or **delete** operation. The **balance(node<T>* p)** function accomplishes this using the **fixHeight(node<T>* p)** function which you need to implement. The **height(node<T>* p)** function must also be implemented.

At this point test2.cpp should pass all tests.

By adding just a few more functions to our current implementation of the BST, we can create an AVL tree:

1. For starters, you need to write code for **rotateLeft(node<T>* p)** and **rotateRight(node<T>* p).** Both functions should return the pointers to the new root of the tree, around which the subtrees were rotated. Remember to fix the heights of the subtrees.
2. You should call appropriate rotations in the **balance(node<T>* p)** function depending on the 4 cases discussed in class.

   At this point test3.cpp should pass all tests.

Task-3: [25 points]

In this part of the assignment, you get two Binary search trees as input and you have to perform union operation on the trees. You have to perform this operation in O(m+n), where there are m nodes in tree 1 and n nodes in tree 2, and return a balanced tree as output. (You are not allowed to call insert operation in this part.)

You can use test_union.cpp to check your solution.

(Assuming we have set a = {1,2,3,5} and set b = {1,3,8,10}. So a union b would be {1,2,3,5,8,10}.)