sortCOP 3502C Programming Assignment # 5

Binary Search Tree and Sorting

Read all the pages before starting to write your code!

Deliverable:

Write all the code in a single main.c file and upload the main.c file. Please include the following commented lines in the beginning of your code to declare your authorship of the code:

```
/* COP 3502C Assignment 5
This program is written by: Your Full Name */
```

Compliance with Rules: UCF Golden rules apply towards this assignment and submission. Assignment rules mentioned in syllabus, are also applied in this submission. The TA and Instructor can call any students for explaining any part of the code in order to better assess your authorship and for further clarification if needed.

Caution!!!

Sharing this assignment description (fully or partly) as well as your code (fully or partly) to anyone/anywhere is a violation of the policy. I may report to office of student conduct and an investigation can easily trace the student who shared/posted it. **Also, getting a part of code from anywhere will be considered as cheating.**

Deadline:

See the deadline in Webcourses. An assignment submitted by email will not be graded and such emails will not be replied according to the course policy.

What to do if you need clarification on the problem?

Write an email to the TA and put the course teacher in the cc for clarification on the requirements. I will also create a discussion thread in webcourses, and I highly encourage you to ask your question in the discussion board. Maybe many students might have same question as you. Also, other students can reply, and you might get your answer faster.

How to get help if you are stuck?

According to the course policy, all the helps should be taken during office hours. There Occasionally, we might reply in email.

Theater Loyalty Program

Objective

Practice implementing a binary search tree.
Practice coming up with a functional breakdown for a large program.
Practice efficiently updating all necessary components of a data structure after each update.

Background Story

Our theater is now done experimenting with pie shaped projectors (turned out to be a big fail!)

Instead, they are going to copy the Universal Cinema Foundation and start a rewards program. Each guest gets 1 loyalty point for each dollar they spend at the theater. Over time, guests may gain loyalty points, use loyalty points to redeem prizes or query the number of loyalty points they have. On occasion, a guest may get very upset at the theater (maybe they frequently show movies that he does not like), and request to be removed from the loyalty program. One strange request the theater wants the program to handle is a query of how many users have names that come alphabetically before a particular user.

Since the theater knows you are learning about binary search trees in class, they would like for you to implement this functionality via a binary search tree of nodes, where the nodes are **compared via the name of the customer stored in the node, in alphabetical order.**

Problem

Write a program that reads in input corresponding to various changes and queries to the theater's loyalty program and prints out corresponding messages for each of the input commands. Here is an informal list of each of the possible commands in the input:

(1) Add Loyalty Points to a particular customer.
(2) Subtract Loyalty Points from a particular customer.
(3) Delete a particular customer.
(4) Search for a particular customer in the binary search tree. If the customer is found, report both their number of loyalty points and their depth in the tree (distance from the root in # of links to follow.)
(5) Count the number of customers whose names come alphabetically before a particular customer.

At the very end of the input, your program should store pointers to each struct storing customer data and sort that data by loyalty points, from highest to lowest, breaking ties alphabetically. (For two customers with the same number of loyalty points, the one whose name comes first alphabetically should be listed first.) This data should be sorted via Quick Sort.

Input

The first line of input contains a single positive integers: ***n*** (***n*** $\leq$ 300,000), the number of commands to process.

The next ***n*** lines will each contain a single command. **Note: The commands will be such that the resulting binary search tree will never exceed a height of 100.**

Here is the format of each of the possible input lines:

Command 1
`add <name> <points>`

`<name>` will be a lowercase alphabetic string with no more than 25 characters.
`<points>` will be a positive integer less than or equal to 100.

This command adds the customer with name <name> into the tree. If the customer is already in the system, it will increase the points of the customer with the given points.

Command 2
`sub <name> <points>`

`<name>` will be a lowercase alphabetic string with no more than 25 characters.
`<points>` will be a positive integer less than or equal to 100.

Note: if a customer has fewer points than is specified in this command to subtract, then just subtract the total number of points they have instead.

Command 3
`del <name>`

`<name>` will be a lowercase alphabetic string with no more than 25 characters.

Delete the customer with the name `<name>` from the binary search tree. No action is taken if the customer isn't in the tree to begin with.

Command 4
`search <name>`

`<name>` will be a lowercase alphabetic string with no more than 25 characters.

This will search for the customer with the name `<name>` and report both the number of loyalty points the customer has and the depth of the node in the tree storing that customer, if the customer is in the tree.

Command 5
`count_smaller <name>`

`<name>` will be a lowercase alphabetic string with no more than 25 characters.
This will calculate the number of names in the binary search tree that come alphabetically before `<name>`.

Output

For each input command, output a single line as described below:

Commands 1 and 2
Print out a single line with the format:

```
<name> <points>
```

where `<name>` is the name of the customer who added or subtracted points and `<points>` is the new total number of points they have.

<u>Command 3</u>
If the customer in question wasn't found in the binary search tree, output the following line:

```
<name> not found
```

If the name is found, output a line with the following format:

```
<name> deleted
```

where `<name>` is the name of the customer being deleted. (Of course, delete the node storing that customer from the tree!) **If you are deleting a node with two children, please replace it with the maximum node in the left subtree. This is to ensure there is one right answer for each test case.**

<u>Command 4</u>
If the customer in question wasn't found in the binary search tree, output the following line:

```
<name> not found
```

If the name is found, output a line with the following format:

```
<name> <points> <depth>
```

where `<name>` is the name of the customer being searched, `<points>` is the number of loyalty points they currently have and `<depth>` is the distance of the node the customer in question was found in from the root node of the tree.

<u>Command 5</u>
For this command, just print a single integer on a line by itself representing the number of names in the binary search tree that come before `<name>`, alphabetically. (Note: Because we require a run time of O(h), where h is the height of the tree, this is likely the most challenging command to process.)

After all commands in the input have been processed, create an array to store pointers to each struct storing customer data. Then, sort that array by customer loyalty points from highest to lowest, breaking ties by the names in alphabetical order as previously described. Finally, print out one line per customer in this sorted order with the format:

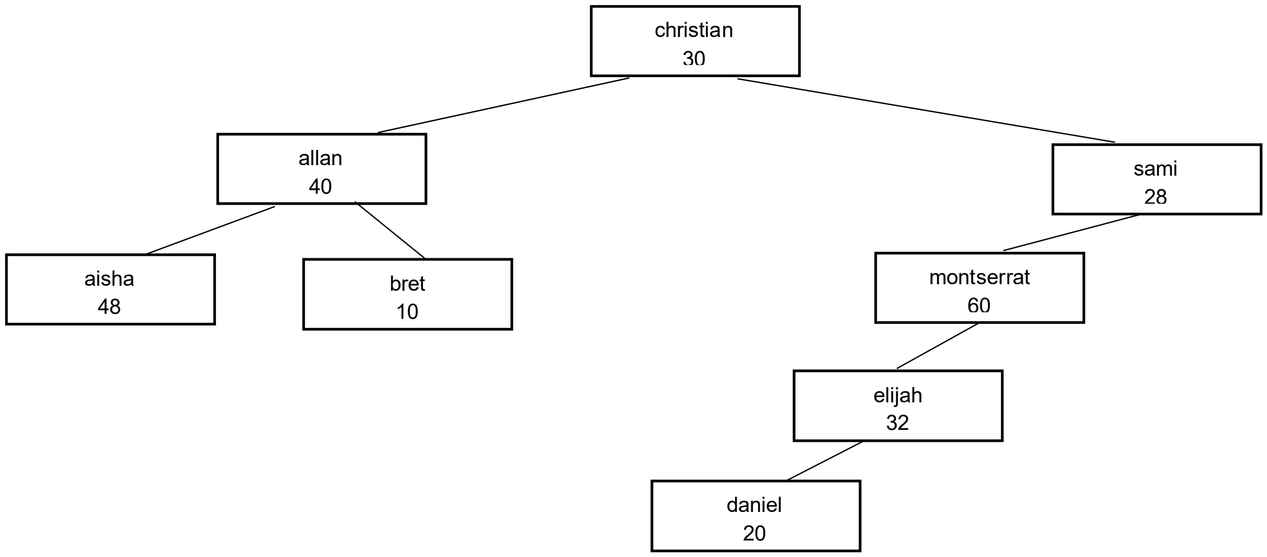
```
<name> <points>
```

where `<name>` is the name of the customer and `<points>` is the number of loyalty points they have at the end of the set of input commands.

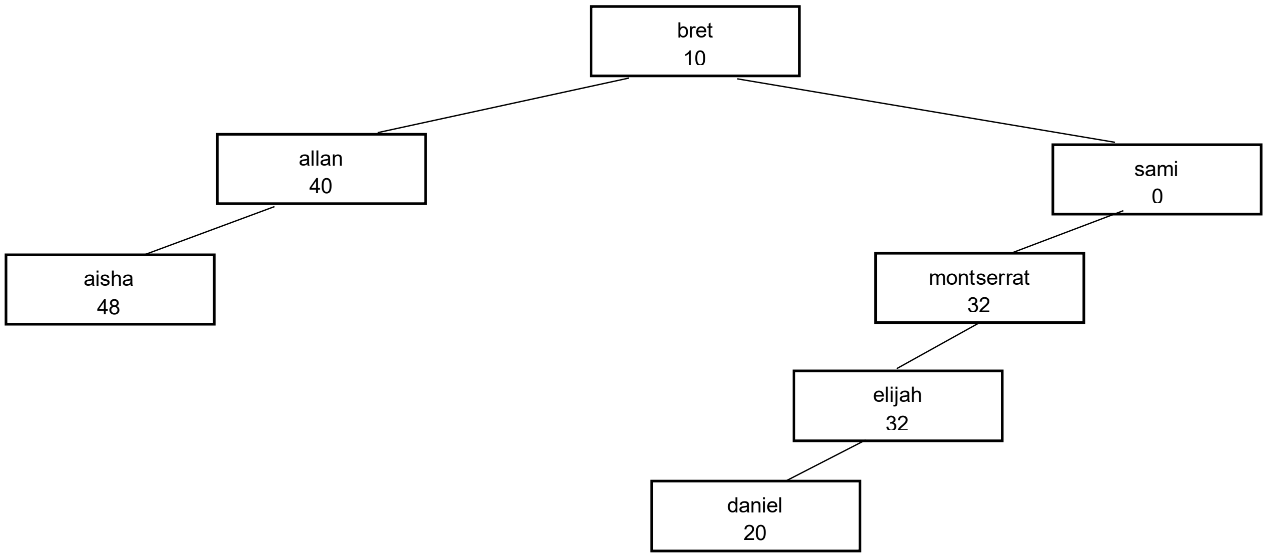
Sample Input	Sample Output
18	christian 30
add christian 30	allan 40
add allan 40	aisha 45
add aisha 45	sami 28
add sami 28	montserrat 60
add montserrat 60	elijah 32
add elijah 32	bret 10
add bret 10	aisha 48
add aisha 3	daniel 20
add daniel 20	sami 0
sub sami 30	christian deleted
del christian	montserrat 32
sub montserrat 28	bret 10 0
search bret	daniel 20 4
search daniel	christian not found
search christian	christian not found
sub christian 20	6
count_smaller sami	sami deleted
del sami	aisha 48
	allan 40
	elijah 32
	monserrat 32
	daniel 20
	bret 10

Sample Explanation

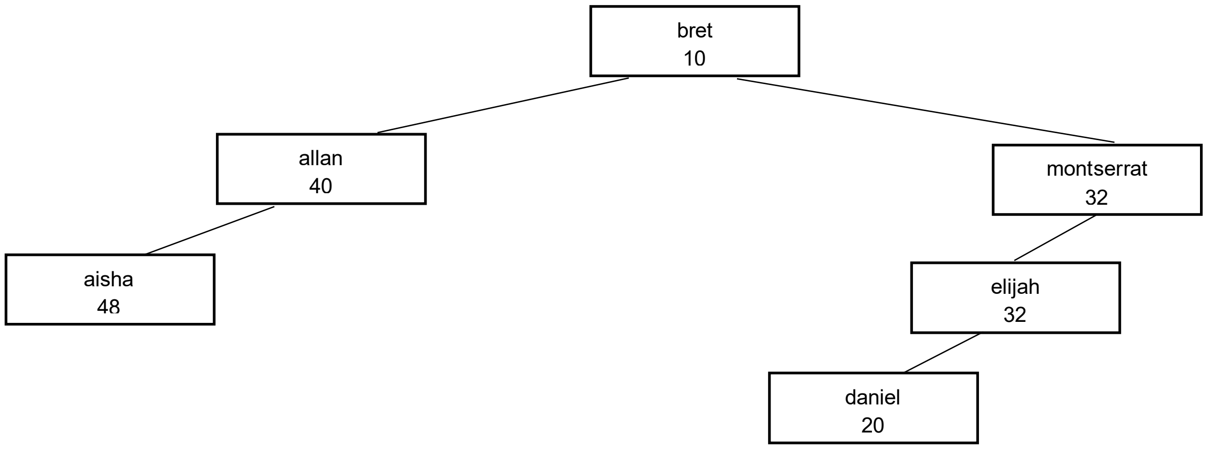
Right before the first sub command, here is a picture of the tree (without all information stored in each node):



After sami loses all of her points, christian is deleted (with the physical node being replaced by bret), and montserrat loses some points, our new tree structure is



After sami is deleted, we have the final tree structure as follows:



Finally, after we copy these nodes into an array of pointers to struct and sort as specified, the array should be ordered as follows:

(aisha, 48), (allen, 40), (elijah, 32), (montserrant, 32), (daniel, 20), (bret, 10)

Structs to Use

Please use the following #define and two structs in your code:

```
#define MAXLEN 25

typedef struct customer {
    char name[MAXLEN+1];
    int points;
} customer;

typedef struct bstnode {
    customer* custPtr;
    int size;
    struct bstnode* left;
    struct bstnode* right;
} bstode;
```

Note: the size variable in the bstnode will store the total number of nodes in the subtree rooted at that node, including itself. These will have to be updated accordingly during each insert and delete operation. Their main purpose is to allow for an O(h) run-time for command number 5, where h is the height of the tree. Use the exact name of the struct and struct properties mentioned above. Do not change their name.

Implementation Requirements/Run Time Requirements

1. A binary search tree of nodes of type bstnode will be used to store the data as commands are processed.

2. The run-time for processing each of the commands should be O(h), where h is the current height of the tree. For command 5, if you can do it with O(n), you can get 60% credit for that command. However, to receive full credit for that command, you need to achieve this in O(h). You can take help size property of the nodes if you want to get O(h) run-time for that command. However, you can get the result in O(n), without using the size property.

3. There should be two compare functions, one to be used by the binary search tree functions and a different one to be used by the sorting functions.

4. Only one full copy of each customer struct should exist. The array to be sorted should be of type `customer**`, storing an array of pointers to structs.

5. The sort implemented at the end of the program must be Quick Sort implementation based on our class.

6. You should free all the memory to receive full credit

7. Your code must compile and execute on the Codegrade system.

Few Hints:

- Carefully go thorough the sample input and output and understand the problem clearly
- Use strcmp function to compare strings.
- Work for one command at a time and test it. Obviously, start with search and insertion and after inserting couple of customer, print them in in-order traversal to check whether they are inserted properly or not
- You may need to update the size during insertion and deletion.
- For the depth, you can calculate it as you process the searching.
- The above implementation requirements and structs will help you as well.
- Various functions discussed in the delete operation will help you as well.
- Count_smaller with O(h) will be challenging for some of you. My recommendation would be do the O(n) version first like the sum of node function discussed in the class and add some condition to get the correct count. The O(n) version does not need the size property and implementation would be much easier than O(h). The size properly might get messed up during deletion. So relying on that could be risky if you want to do O(h) version of Count_smaller right away. So, I highly encourage you to do O(n) first and see whether you can match the numbers or not. Then, if you have time then go to the o(h) and see if you can work on it to receive 100%.

<u>Some Steps to check your output AUTOMATICALLY in a command line in repl.it or other system:</u>
You can run the following commands to check whether your output is exactly matching with the sample output or not.
<u>Step1:</u> Copy the sample output to sample_out.txt file and move it to the server
<u>Step2:</u> compile your code using typical gcc and other commands.

//if you use math.h library, use the -lm option with the gcc command. Also, note that scanf function returns a value depending on the number of inputs. If you do not use the returned value of the scanf, gcc command may show warning to all of the scanf. In that case you can use "-Wno-unused-result" option with the gcc command to ignore those warning. So the command for compiling your code would be:

# gcc main.c leak_detector_c.c -Wno-unused-result -lm

Step3: **Execute your code and pass the sample input file as a input and generate the output into another file with the following command**
$ ***./a.out < sample_in.txt > out.txt***

Step4: Run the following command to compare your out.txt file with the sample output file

```
$cmp out.txt sample_out.txt
```

The command will not produce any output if the files contain exactly same data. Otherwise, it will tell you the first mismatched byte with the line number.

Step4(Alternative): Run the following command to compare your out.txt file with the sample output file

```
$diff -y out.txt sample_out.txt
```

The command will not produce any output if the files contain exactly same data. Otherwise, it will tell you the all the mismatches with more details compared to cmp command.
# diff -c myout1.txt sample_out1.txt //this command will show ! symbol to the unmatched lines.