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 either Merge Sort or 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 19 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 19 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 19 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 19 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 19 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.

Command 3
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.**

Command 4
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.

Command 5
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:

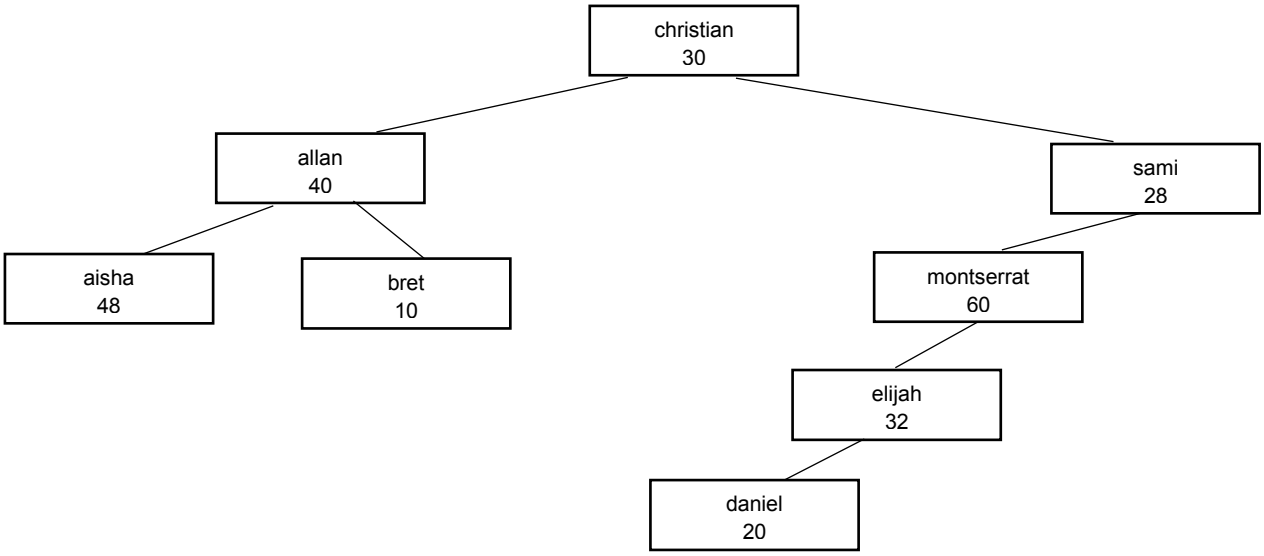
```
<name> <points>
```

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.

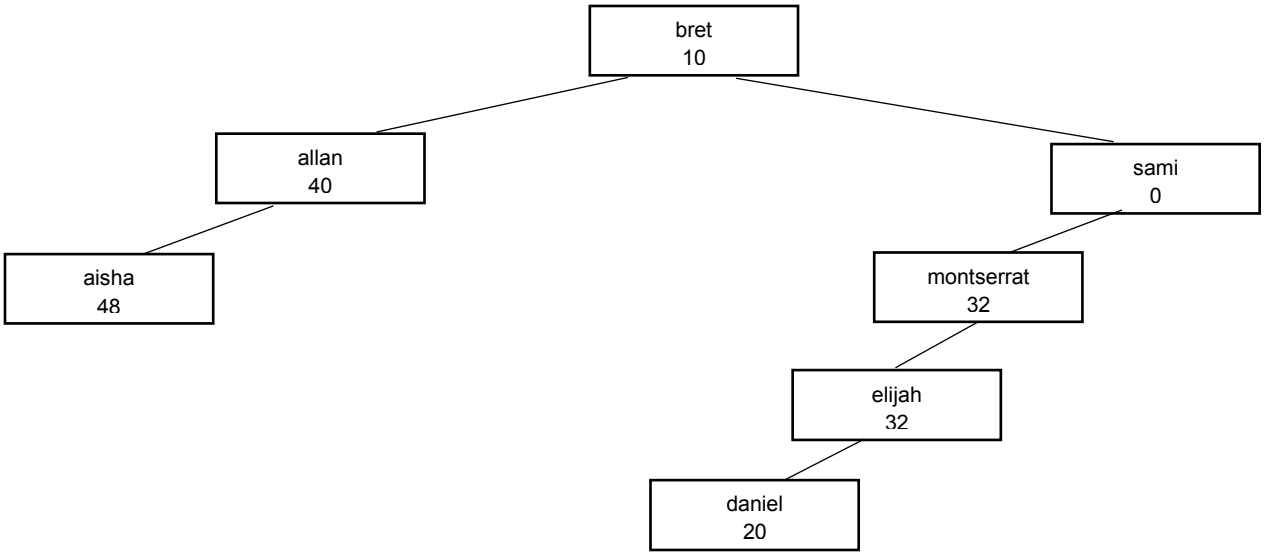
Sample Input	Sample Output
18 add christian 30 add allan 40 add aisha 45 add sami 28 add montserrat 60 add elijah 32 add bret 10 add aisha 3 add daniel 20 sub sami 30 del christian sub montserrat 28 search bret search daniel search christian sub christian 20 count_smaller sami del sami	christian 30 allan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

Sample Explanation

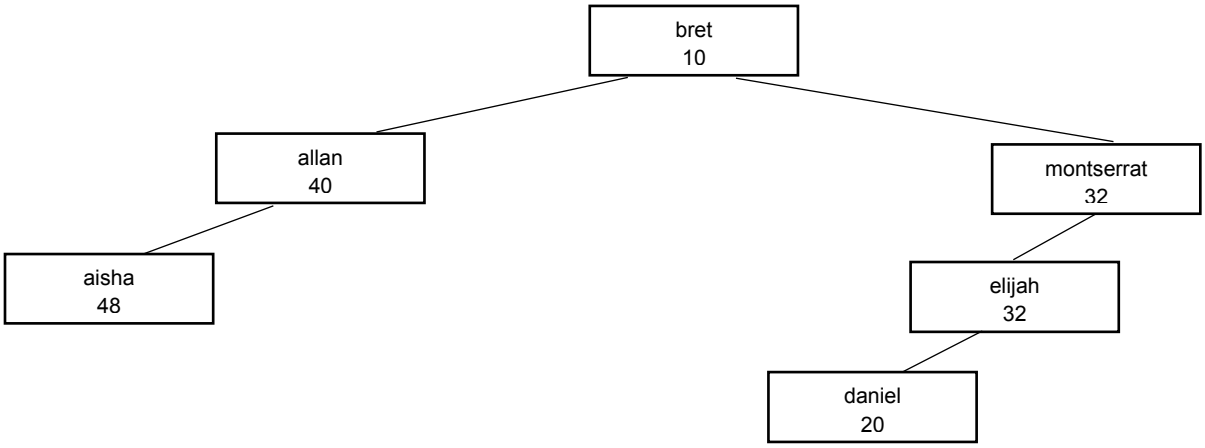
Right before the first sub command, here is a picture of the tree (without all information stored in each node):



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 19

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

typedef struct treenode {
    customer* cPtr;
    int size;
    struct treenode* left;
    struct treenode* right;
} treenode;
```

Note: the size variable in the treenode 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.

Implementation Requirements/Run Time Requirements

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