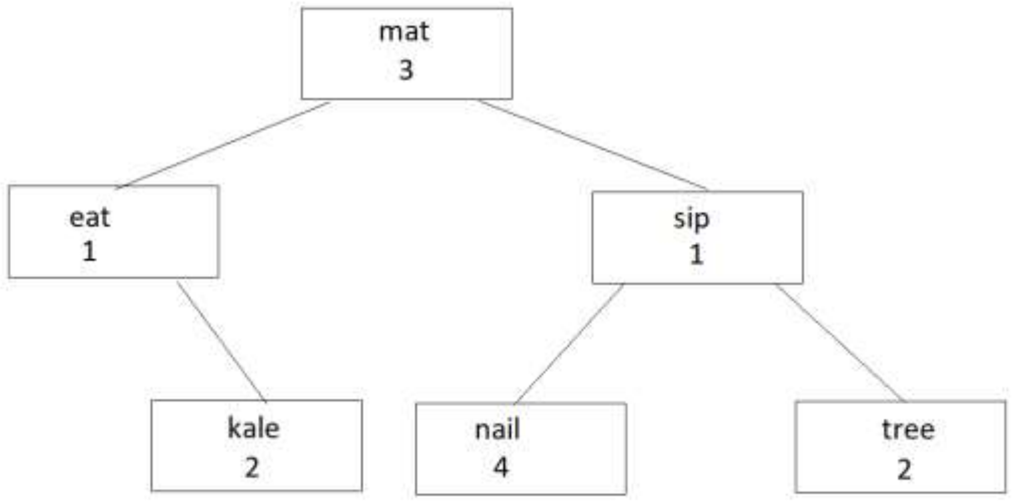
Computer Science I Program #5: Word Sorting, Searching (Binary Search Trees)

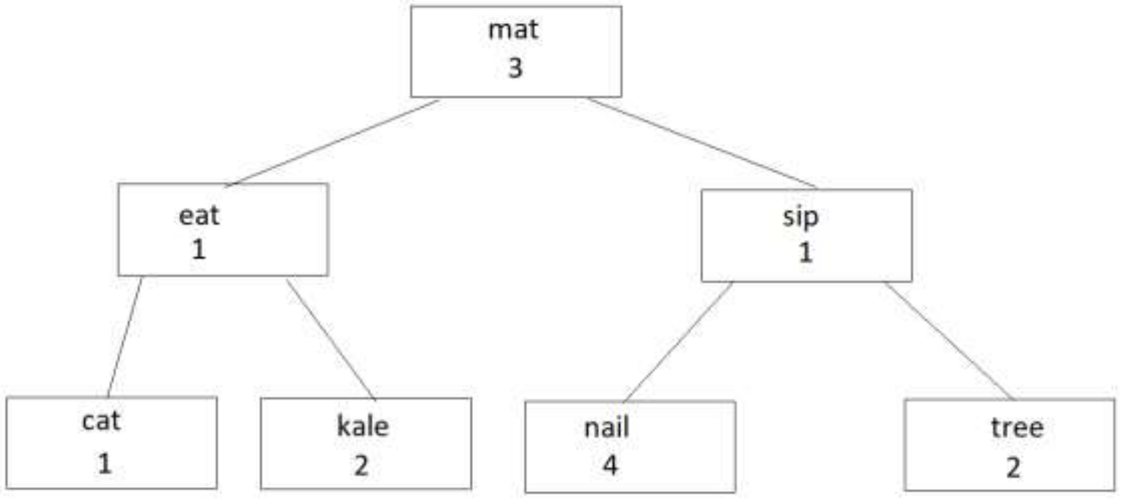
Please check Webcourses for the Due Date

Read all the pages before starting to write your code

You would like to create a tool that allows you to analyze the frequency of words in various works, while you practice your binary tree skills for COP 3502. You've decided to write a program that reads in words, one by one and stores them in a binary search tree, organized in alphabetical order. You will have one node per each unique word, and in that node, you'll store the word with its frequency. Here is an example picture of a small tree:

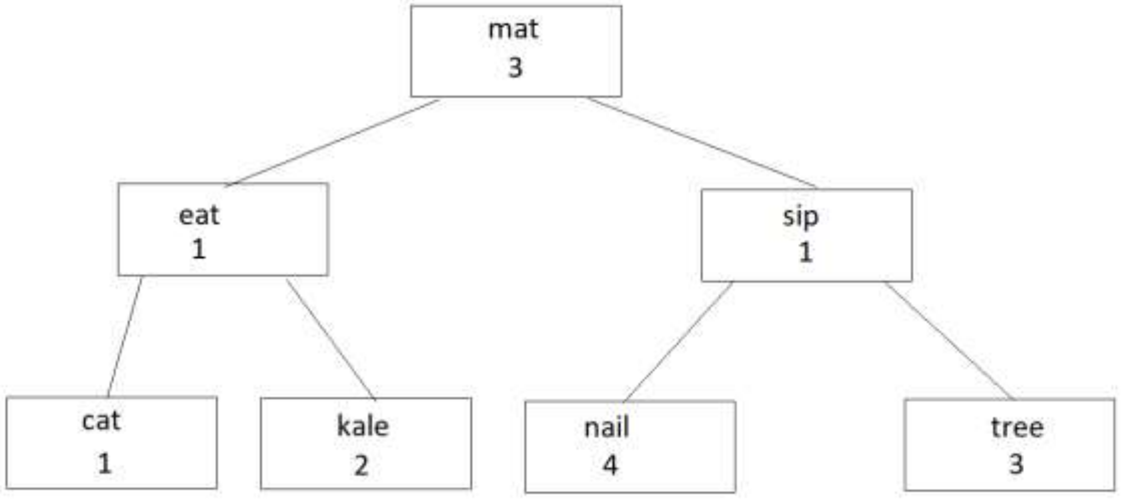


In this tree, the word "mat" is stored in the root and it has appeared 3 times, the word eat is to its left and has appeared once. "kale" which comes before "mat" and after "eat" is the right child of "eat" and has appeared twice. If we were to insert "cat" in this tree, the new picture would look like this:



Since “cat” was not previously in the tree, a new physical node is created, storing cat and showing that it has been seen once.

Subsequently, if we were to insert “tree” into this tree, the new picture would look like this:



Notice that since “tree” was a word that was already processed, no new node is created. Instead, 1 is added to the frequency value stored at the node.

Assignment Part I: Building the Tree, Answering Queries

To prove that you are building the tree correctly, in the middle of the process of reading in some words, you may have to handle some queries. A query is a word, and your program must determine how many times the word has appeared so far **AND** the depth of the node storing the word (depth of a node is how many links must be followed from the root node to get to it). You should store this value in the node structure. If a word has not appeared, you should answer -1 for both questions.

After completing reading in all the words and answering the various queries, your program will move to Part II, described below.

Assignment Part II: Building a Sorted List of Words

For statistical purposes, you want to create a sorted list of words by frequency, from most frequent to least frequent. If two words appear the same number of times, break the tie by alphabetical order, with words that come first alphabetically being listed first.

What you need to do is copy all of the words from the tree into an array of structs, each of which stores both the word and the frequency. (If you are clever about it, you'll only have one copy of each struct that gets used in BOTH the tree and the array!)

Then, write either a Quick Sort or a Merge Sort and sort the array according to the directions given. Finally, output the list of words with frequencies in the desired order.

The Input (to be read in from standard input)

The first line of input will contain a single positive integer, n ($n \leq 200000$). This will represent the total number of words to insert into the tree and the number of queries.

The input follows on the next n lines, one line for each action. Each of these lines will start with an integer, a, representing the action to be performed. This integer is guaranteed to be either 1 or 2. If this integer is equal to 1, that means the action to be performed is an insertion. If this integer is equal to 2, then that means the action to be performed is a query. In both cases, the integer is followed by a space and then followed by a string of 1 to 20 lowercase letters, representing the word for the action. It is guaranteed that at most 100,000 of these actions will be of type 1 **and that the tree that is created by following all of these actions will never exceed a height of 100.**

The Output (to be printed to standard output)

For each action of type 2, print a single line with two space separated integers: the number of times the queried word has already appeared and the depth of the node storing the word in the tree as previously defined.

After handling all of the queries, output the final list of words and frequencies sorted in the order designated, outputting each word, followed by a space, followed by its frequency on a line by itself.

Sample Input

```
23
1 mat
1 eat
1 sip
2 mat
2 eat
2 apple
1 mat
1 kale
1 nail
1 kale
1 tree
1 mat
2 mat
2 nail
1 nail
1 nail
1 tree
1 tree
2 tree
1 cat
2 cat
1 nail
2 nail
```

Sample Output

```
1 0
1 1
-1 -1
3 0
1 2
3 2
1 2
4 2
nail 4
mat 3
tree 3
kale 2
cat 1
eat 1
sip 1
```

Implementation Restrictions/ Run-Time/Memory Restrictions

1. You must create a binary search tree of structs, where each struct stores a string (can be statically allocated) and an integer (its frequency). It's also recommended that a third integer, the depth of the node, be stored in the struct. Though each field of the struct will be statically allocated, the structs themselves should be dynamically allocated when needed. (Thus, the node struct should comprise of three pointers - pointers to the left and right substrees and a pointer to a struct that stores the necessary information described above.)

2. For full credit, your algorithm must perform insertions into the binary search tree in O(h) time, where h represents the height of the tree at the time of the insertion.

3. For full credit, you must have appropriately designed functions. **In particular, any correct solution which is fully coded in main will NOT receive full credit.**

4. You create an array of pointers to the structs described above for the second portion of solving the problem.

5. You must have a stand-alone sorting function that takes in the array from requirement #4 and sorts it. You may use auxiliary functions.

6. The algorithm the sorting function implements must be either Quick Sort or Merge Sort.

7. You must only declare your array variable **INSIDE** your case loop.

8. You must free all dynamically allocated memory for full credit.

Deliverables

You must submit one file over WebCourses:

1) A source file, *wordsort.c*. Please use stdin, stdout. There will be an automatic 10% deduction if you read input from a file or write output to a file.

2) A file describing your testing strategy, *lastname_Testing.doc(x) or lastname_Testing.pdf*. This document discusses your strategy to create test cases to ensure that your program is working correctly. If you used code to create your test cases, just describe at a high level, what your code does, no need to include it.

3) Files *wordsort.in* and *wordsort.out*, storing both the test cases you created AND the corresponding answers, respectively.