

CS241 Project 4: Sentiment Analysis with Binary Search Trees

Overview: Sentiment analysis, in natural language processing or computational linguistics, is a set of techniques used to extract subjective information from text, usually in the sense of figuring out whether a piece of text is expressing a positive, neutral, or negative opinion about something. In this project, you will write a simple sentiment analysis program to try to figure out what sort of opinion is being expressed in a collection of movie reviews. (More about sentiment analysis:

https://en.wikipedia.org/wiki/Sentiment_analysis)

The sentiment analysis algorithm for this project is very simple. You are given a collection of movie reviews in text format. You will use a symbol table (a.k.a. dictionary or map) to keep track of how often certain words appear in the reviews. That is, your symbol table will store an association between words (the keys), and the frequencies with which they appear in the movie reviews (the value).

You are provided with a text file containing movie reviews written by people who have seen various movies. Each person has rated the movie on a scale from 0-4, where 2 is neutral (so the person gives the movie a score of 0 when they absolutely hate it, 1 when they dislike it, 2 when they're indifferent, 3 when they like it, and 4 when they love it). Each person also has written a text review of the movie as well.

You can see in the text files that each review is contained on one line of the file. The first item on each line is the rating score from 0-4, and then the rest of the line is the review. Notice that all punctuation is removed, and all words are lowercase (this makes your life easier).

You will write a program that reads this file and stores all the words in a binary search tree. In the BST, each word carries with it some additional information: the frequency that the word appears in all the movie reviews, as well as the total score (the sum) of all the movie reviews in which that word appears. This additional information will be needed in our sentiment analysis algorithm. This means your binary tree will represent a Symbol Table ADT, because each word (a string) will be associated with two integers (a frequency and a total score), but all searching/inserting/deleting is done based on the word (the key attribute).

After you read in all the movie reviews and create a binary search tree from the words (and print out some statistics about the tree), you will read in a file of what are called *stop words*. In natural language processing, stop words are words that are filtered out before any text processing happens, usually because these words carry very little meaning. Usually stop words are just a large collection of common English words, like "the," "is," "a," etc. These words are in the file stopwords.txt, one per line.

To get some practice using the built-in C++ data types, you will store the stop words using a C++ set (#include <set> to access this). Under the hood, this data type uses a binary search tree as well (which you don't have to know about to use it). You can create a new set of strings by using

```
set<string> stopwords;
```

To put something into this set, use the insert() function:

```
stopwords.insert(word);
```

To check if a word is in a set, use the count() function (it will return 0 if the word isn't there and 1 if it is)

```
if (stopwords.count(word) == 0) // this word is not a stop word
```

To remove a word from a set (though you won't need that for this project), use `remove()`:

```
stopwords.remove(word);
```

More information: <http://en.cppreference.com/w/cpp/container/set>

After you remove the stop words from your binary search tree of word frequencies (and print a bunch of statistics again), you will have the user type in **new** movie reviews, until they type the word "quit". For each review, you will calculate a number between 0-4 indicating the sentiment of the review, where 2 is neutral (0-1 are negative; 3-4 are positive).

What you need to do

Step 1: Write the `FreqBST` class. This class represents a Symbol Table that stores an association between a word from a movie review (the *key*) and two additional pieces of information about that word: the **number of times that word appears in all the movie reviews**, and the **total score of all the movie reviews in which that word appears** (the *values*). Remember that in BSTs, all the searching/inserting/removing happens based on comparisons done with the keys, and the values are just along for the ride (they don't factor into the BST algorithms, except as extra information carried along in each node).

The `WordNode` struct is already defined for you; it holds a word and its associated frequency and total score. Each node of the BST will be this struct.

Your BST class should have the methods named in the `FreqBST.h` file. Some additional information about them:

- `addOrUpdateScore`: This function takes a word and a movie score (a rating from 0-4). It should be called on each word within each movie review to update the BST. This function should insert the given word into the BST with the given score (and a frequency of 1), **unless** the word already appears in the tree, in which case it will **update** the total score and frequency appropriately. Use the standard BST insert algorithm, with some extra code to handle the case when you call this function on a word that's already in the tree (in which case you'll update the information in a `WordNode` instead of adding a new one).

To be clear: you should only "walk" the tree once in this function to add/update the `WordNode`. Just use the BST search algorithm to find the appropriate `WordNode` in the tree.

If the `WordNode` *is not* found (you reach a null pointer where it should be), that means it's the first time you've seen this word in any movie review, so create a new `WordNode` and attach it to the BST (here, frequency should be 1, because you've only seen the word once, and `totalScore` should be the corresponding movie score).

If the `WordNode` *is* found, then you've already encountered this word in the BST, so update the `WordNode` with the new information (increase frequency by 1, since you've now seen the word again, and increase `totalScore` by whatever the new score is).

Note: You do not need any special handling if a word appears multiple times in the same review. Just

call `addOrUpdateScore` for every word in the review.

- `remove`: This function removes a word (and its total score & frequency) from the tree. Use the standard BST deletion algorithm. To ensure everyone does this the same way, if a node has two children, replace the node with the inorder successor (like the book does). This function will be called on all the stop words to remove them.
- `getAverageWordScore`: This function calculates the average *sentiment* for a word. This is defined as the average of all the movie scores in which this word appears. This should be very easy to calculate, as every `WordNode` contains the total of all the scores and the frequency already (just divide one by the other to get the average sentiment). Use the standard BST search algorithm to find the right `WordNode`. If this function is called on a word that is not in the BST, return 2 (the “neural” score).
- `printFrequencyTable()`: This function does an inorder traversal of the tree and prints, for each word, its total score, and frequency.
- `printPreorder/inorder/postorder`: These functions are mostly here for debugging. They just do the standard traversals of the BST and print all the words on one line.
- `countWords()`: returns a count of all the words in the BST. Write this recursively.
- `height()`: returns the height of the BST. Write this recursively.

Note that you will likely have to add some private methods to this class to support your recursive functions.

Step 2: Write some sanity checks in `main.cpp` to make sure the BST works. You can call your tree traversal functions and `printFrequencyTable` to make sure.

Step 3: Write code in `main` to process the movie reviews text file. This will be a simple loop that iterates through the movie reviews file, calling `addOrUpdateScore` once for every word in every review. If a word appears more than once in a review, you will call `addOrUpdateScore` multiple times on that word (which is OK). I suggest testing with the small file, then try the big one.

Step 4: Write code to handle the stop words. Read in the stop words file, remove them from the BST, and save them in a set for later (you’ll need them to filter them out of the sentiment analysis algorithm in the next step).

Step 5: Write code to calculate sentiments for new movie reviews. There is already a loop in the code that asks the user to type in a new review. For each new review, get the average word score for each word in the review (ignoring stop words). If a word appears more than once in the review, it’s OK to call `getAverageWordScore` more than once. Then take the average of all the average word scores; this is your sentiment!

Hints, tips, and guidelines

- There is a constant defined in `main` called `PRINT_TREES`. Use this to control when your code prints the tree traversals and frequency table (just toggle the value from `true` to `false`). This output is useful for

debugging, but is probably too much output for the larger movie reviews file.

- Your output should match mine to the best of your abilities. In particular, make sure to print the messages asked for as you calculate the sentiment score for a new movie review in Step 5.
- I highly recommend not trying to write your own algorithm for removing/deleting a node from a binary search tree. Instead, use the code I handed out in class (check the website for it as well). You can also copy the code from the handout for search() and insert() as well, to use in your getAverageWordScore and addOrUpdateScore functions.
- Feel free to add private helper functions to the FreqBST class --- this is frequently done for recursive functions that need a WordNode* argument. For instance, to implement printPreorder, you'll have the normal void printPreorder() function in the public section, and a void printPreorder(WordNode * node) helper function in the private section. Note: We put these helper functions in the private section because we don't want users of the class to have direct access to them.

A Complete Example

The reviews-small.txt file contains the following movie reviews:

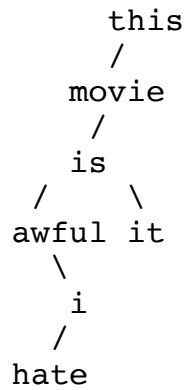
```
0 this movie is awful i hate it
2 this movie is ok i think its not bad
4 this movie is the best thing ive ever seen i love it
```

This means there are three reviews (one per line). The first person gave a score of zero, the second one a two, and the third one a four.

Your goal is to loop over the file, as you see each word, call addOrUpdateScore with word and the corresponding movie score from that line. For instance, after processing the entire first line, you will have the following WordNodes in your BST:

```
word=this, freq=1, totalScore=0
word=movie, freq=1, totalScore=0
word=is, freq=1, totalScore=0
word=awful, freq=1, totalScore=0
word=i, freq=1, totalScore=0
word=hate, freq=1, totalScore=0
word=it, freq=1, totalScore=0
```

(all those WordNodes would be arranged in a BST, organized by word, like this:)

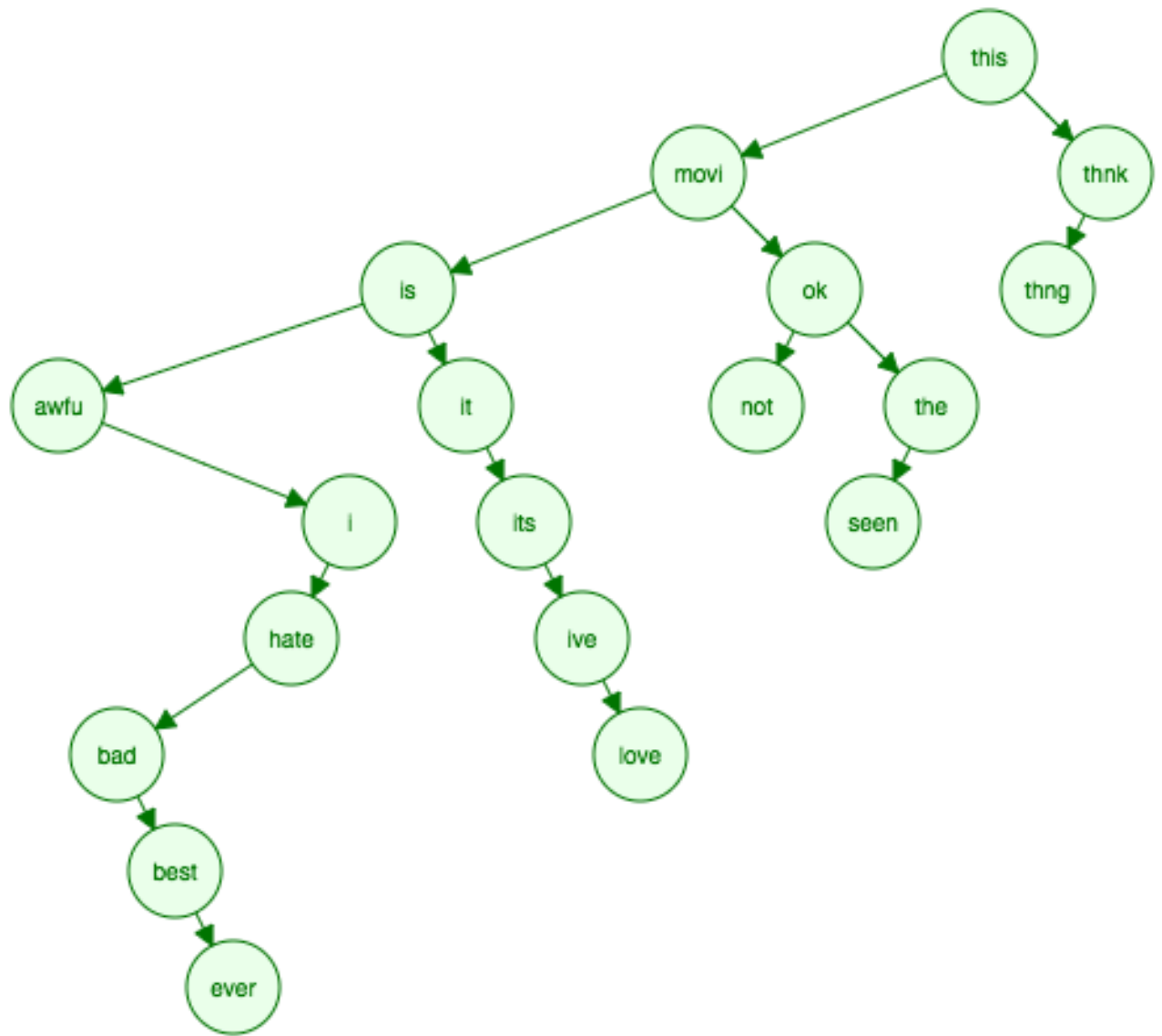


After processing the 2nd line, you'd have:

word=this, freq=2, totalScore=2
word=movie, freq=2, totalScore=2
word=is, freq=2, totalScore=2
word=awful, freq=1, totalScore=0
word=i, freq=2, totalScore=2
word=hate, freq=1, totalScore=0
word=ok, freq=1, totalScore=0
word=think, freq=1, totalScore=2
word=its, freq=1, totalScore=2
word=not, freq=1, totalScore=2
word=bad, freq=1, totalScore=2

Then you'd process the third line, inserting & updating nodes of the tree.

At this point (before removing stopwords), the BST looks like this: (the tool I used to draw this picture only shows 4 letters per word for conciseness; you should be able to figure out what they are).



Then process stopwords.txt and call remove() on all the stopwords. This will delete some nodes from the tree.

Follow along on small-reviews-out-complete.txt for the test cases.