CS201: Data Structures and Algorithms

Assignment 2

Version 3d

Introduction

You will compare the performance of green binary search trees versus AVL trees by reading in a corpus of text, storing the word and phrases therein into a search tree, and then performing operations on the resulting tree. A green tree is just like a regular binary search tree except that it can store duplicates in an efficient way. Your AVL tree will efficiently store duplicates as well. However, the plain binary search tree module upon which your green and AVL trees are based will not.

The corpus will be stored in a file; phrases will be delineated by double quotes while tokens will be sequences of non-quote, non-whitespace characters. Whitespace consists primarily of the characters space, tab, and newline.

Commands for manipulating the tree composed of corpus entities will be stored in a file as well. Here is a list of commands your application should handle:

`i W`	insert word or phrase `W` into the tree
`d W`	delete word or phrase `W` from the tree
`f W`	report the frequency of word or phrase `W`
`s`	show the tree
`r`	report statistics

All words that are inserted into a tree should be composed of letters from the ASCII character set `[a-z]`; phrases may possibly contain spaces as well. Both words and phrases should be *cleaned*. In the case of words, cleaning means all undesirable characters are to be removed and uppercase characters translated to lowercase. For example, the word `Joy's.` would be rendered as `joys`. The same is true for phrases, but after cleaning, any leading or trailing whitespace should be removed and any contiguous whitespace in the interior should be replaced by a single space. For example, the phrase `"  by  ,    and by"` would become `"by and by"`. Do not insert empty words or phrases or phrases with only spaces into the tree.

For green and AVL trees, inserting a word already in the tree would increase its frequency count by one. When deleting a word from the tree, its frequency count should be reduced by one. If the frequency count goes to zero, the corresponding node should be removed from the tree.

When showing the tree, display the nodes with a breadth-first (left-first) traversal. All nodes at a given level should be on the same line of output. The level number should precede the display of the nodes at that level. Display each node according to the following format:

- an equals sign if the node is a leaf, followed by
- the node value, followed by
- a frequency count (if the count is greater than one), followed by
- a balance factor, either `+` or `-` (if the balance factor is not zero), followed by
- a parenthesized display of the parent, followed by
- an `X` if the node is the root, an `L` if the node is a left child, and an `R` otherwise

Your display function must run in linear time. Note that you will need to add a *displayBSTdecorated* method to your BST class in order to add the leaf, root, left child, and right child decorations to the level order display of the tree. You will need an AVL-specific and private display function (call it *adisplay*), which you will passed to the BST constructor. The *adisplay* function will handle the frequency count and balance factor decorations. It will also handle the actual display of the generic value by using the original display function passed to the AVL constructor.

Note also that the parent of the root is itself. Green tree nodes do not have balance factors. Here is an example of a AVL tree:

```
0: beta(beta)X
1: =alpha(beta)L =gamma(beta)R
```

Note that each value occurs once in the tree above. Note also, that this could also be a green tree display. Here is another:

```
0: beta-(beta-)X
1: =alpha(beta-)L gamma+(beta-)R
2: =delta[2](gamma+)L
```

If the balance factor is a negative one, a minus sign is printed. If the balance factor is a positive one, a plus sign is printed. The corpus to generate such a tree might look like:

```
beta alpha
  gamma
      delta
```

The words should be ordered within a tree in case-insensitive lexicographic ordering. Suppose the corpus was:

```
The "quickity quick" brown fox
     jumped over the girl
         and her lazy, lazy dog.
```

found in a file named *data*. Suppose the file *commands* holds the command:

```
s
```

Then a display of a green tree generated by this corpus would look like this:

```
$ trees -g data commands
0: the[2](the[2])X
1: quickity quick(the[2])L
2: brown(quickity quick)L
3: =and(brown)L fox(brown)R
4: =dog(fox)L jumped(fox)R
5: girl(jumped)L over(jumped)R
6: =her(girl)R =lazy[2](over)L
```

When the corpus is inserted into a AVL tree, the resulting structure would look like this:

```
$ trees -r data commands
0: jumped(jumped)X
1: fox(jumped)L quickity quick+(jumped)R
2: brown(fox)L girl-(fox)R over+(quickity quick+)L =the[2](quickity quick+)R
3: =and(brown)L =dog(brown)R =her(girl-)R =lazy[2](over+)L
```

The display of an empty tree should generate the following line of output:

```
EMPTY
```

The statistics to be reported are:

- the number of duplicates in the tree
- the number of nodes in the tree
- the minimum depth, which is the distance from the root to the closest node with a null child pointer
- the maximum depth, which is the distance from the root to the furthest node with a null child pointer

If the root had a null child, the minimum depth would be 0. Here is an example statistics report:

```
Duplicates: 2
Nodes: 8
Minimum depth: 2
Maximum depth: 4
```

Here is an example frequency report:

```
Frequency of albatross: 5
```

All output must be formatted as shown. Each line of output should have no leading whitespace, no trailing whitespace (except the mandatory newline), no interstitial whitespace except spaces, and no more than one space in a row.

Commands

The commands will be read from a free format text file; individual tokens may be separated by arbitrary amounts of whitespace. For example, these three file contents are all legal and equivalent:

```
    i spongebob
    i "Bikini Bottom"
    f Patrick
    s
```

or

```
    i spongebob i "Bikini Bottom" f Patrick s
```

or

```
    i spongebob i
        "Bikini Bottom" f

        Patrick

    s
```

Error handling

You should ignore, but report an attempt to delete something that does not exist in the tree. Thus you ought to be able to randomly generate a large number commands and have your application run without failing. The error message printed when attempting to delete a value not in the tree should have the form:

```
    Value x not found.
```

There should be no quotes around the value. Normally, one would print error messages to *stderr*, but for testing purposes, print them to *stdout*.

Program organization

The tree portion of your code should be composed of three modules:

- `http://beastie.cs.ua.edu/cs201/assign-bst.html`.
- `http://beastie.cs.ua.edu/cs201/assign-gst.html`.
- `http://beastie.cs.ua.edu/cs201/assign-avl.html`.

The *GST* module and *AVL* modules should take advantage of *BST* objects as much as possible. For example, a *GST* object should encapsulate a *BST* object, much like a queue might encapsulate a singly-linked list. The same is true of a *AVL* object. For both *GST* and *AVL*, their size methods should return the number of nodes via the *BST* size method. Another example is that a *GST* delete method should simply make a call to *deleteBST*. An *AVL* delete method, on the other hand, would make a call to *swapToLeafBST*, followed by a call to a deletion fixup routine, and finally followed by a call to *pruneLeafBST*.

Both *GST* and *AVL* will need to wrap generic values with frequency count fields, while *AVL* will need to wrap in additional fields for quickly calculating the balance factor. For example, a *GST* value object will hold a `void *` pointer to a generic value, an `int` field to hold the frequency count, a display pointer to point to the original display function, a comparator pointer to point to the original comparator, and a freeing pointer to point to the original freeing function. As such, the constructors *newGST* and *newAVL* will need the same argument types as the first two arguments to the *newBST* constructor (or you could just pass in the tree object).

The only local includes a *BST* module should have are *bst.h* and *queue.h* (needed by the BST display debug method).

Compliance

The *swapToLeafBST* function should prefer swapping with a successor over swapping with a predecessor.

The *pruneLeafBST* should not decrement the size of the BST. Only the *deleteBST* method should decrement the count. Therefore, the *deleteAVL* method will need to adjust the BST size. The *insertGST* and *deleteGST* methods should just use *insertBST* and *deleteBST*, respectively.

The insertion and deletion fixup routines for AVL trees must follow the *pseudocode* found on the Beastie website.

The display debug method of a *BST* should use a queue.

If a newline is to be printed, there can be no preceding qhitespace. No lines of output are indented.

Note that during some tests, your *BST*, your *GST*, and your *AVL* module will be replaced. In others, your modules will be tested in isolation, so do not add any additional public interface methods or includes.

You are required to use the *test2* drop box prior to submission.

Reuse your code from previous assignments. You should not need to modify it much (other than the new display method for BST).

Program invocation

Your program will process a free-format corpus of text and a free-format file containing an arbitrary number of commands. The name of the corpus and the file of commands will be passed to the interpreter as a command line arguments. Switching between the two tree implementations is to be accomplished by providing the command line options -g (green tree) and -r (AVL tree). Here is an example call to your interpreter:

```
trees -g corpus commands
```

where *corpus* is a file of text and *commands* is the name of a file which contains a sequence of commands. In executing this call, you would read the words found in *corpus*, store them into a simple binary search tree, and the process the sequence of commands found in *commands*. Both the corpus and the commands file may be empty.

Your executable should handle a '-v' option. With this option, your executable should print the author's name and immediately exit with a zero error code, performing no other work.

If no `-g` or `-r` function is given, you application should assume a AVL tree.

Either the corpus or the commands may be empty, possibly both.

Program output

All output should go to the console (stdout). When processing commands, only the result should be echoed; the command should not be echoed.

The *insert* and *delete* commands do not have a printable result and therefore should be processed silently.

Documentation

All code you hand in should be attributed to the author. Comment sparingly but well. Do explain the purpose of your program. Do not explain obvious code. If code is not obvious, consider rewriting the code rather than explaining what is going on through comments.

Project compilation

You must implement your modules in C99. You must provide a file named *makefile*, which responds properly to the commands:

```
make
make test
make clean
make valgrind
```

The `make` command compiles your program, which should compile cleanly with no warnings or errors at the highest level of error checking (the `-Wall and -Wextra options`). The `make test` command should test your program and the `make clean` command should remove object files and the executable.

The compilation command must name the executable *trees* (not *trees.exe* for you poor Cygwin users). You may develop on any system you wish but your program will be compiled and tested on a Linux system. Only the most foolish students would not thoroughly test their implementations on a Linux system before submission. Note: depending on where you develop your code, uninitialized variables may have a tendency to start with a value of zero. Thus, a program with uninitialized variables may work on your system but fail when I run it.

The correctness and efficiency of your makefile will be tested. You must have the correct dependencies and your makefile should not perform any unnecessary compilations.

Grading

You must pass all tests for your program to be graded.

Handing in the application

For preliminary testing, run a `make clean` and then send me all the files in your directory by running the command:

```
    submit cs201 lusth test2
```

For your final submission, run a `make clean` and use the command:

```
    submit cs201 lusth assign2
```

Again, your implementation may be developed on other hardware and operating systems, but it must also compile and run cleanly and correctly on a Linux system.