CS201: Data Structures and Algorithms

Assignment 2, Version 8

Introduction

You will compare the performance of vanilla binary search trees versus red-black 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. The corpus will be stored in a file; the commands for manipulating the tree will be stored in a file as well. You will implement the following operations for both kinds of trees:

- *insert* which inserts a value into the tree (or updates a frequency count)
- *delete* which decreases a frequency count (or removes a value from the tree)
- *find* which reports the frequency of a value in the tree

These operations should maintain *bst* ordering. You will also implement an interpreter which processes requests to manipulate the trees. Your interpreter should handle the following commands:

`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

Legal tree values are simply words from the character set [a-z]. When reading a word, you should read a whitespace delimited token and then delete from it all non-letter characters. You should then convert all upper-case letters to lower case. For example, the word `Joy's.` would be rendered as `joys`. Do not insert empty strings into the tree.

Phrases will be delineated by double quotes and will consist of all the characters between the opening and closing quotes. The same processing that is applied to words should be applied to phrases, as well. In addition, all spans of whitespace (spaces, tabs, and newlines) should be replaced with a single space. For example, the phrase `"Girls' night out!"` would be rendered as `girls night out` As with words, do not insert empty strings into the tree. When displaying values, regardless of whether they started out as tokens or double quoted strings, display the double quotes.

Inserting a word already in the tree would increase its frequency count by one. When deleting a word from the tree, you should reduce its frequency count by one. If the frequency count goes to zero, you should remove its corresponding node completely 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, which may include a frequency count (if the count is greater than one) and a color, followed by
- a parenthesized display of the parent's value (which again may include a frequency count and color), followed by
- a `-` if the node is the root, a `-l` if the node is a left child, and a `-r` otherwise

Your display function must run in linear time. Note that the parent of the root is itself. For the vanilla *bst*, all nodes are considered black. Here is an example of a *red-black* tree:

Here is an example of a completely balanced *red-black* tree:

```
0: "beta"-B("beta"-B)-
1: ="alpha"-R("beta"-B)-l ="gamma"-R("beta"-B)-r
```

Here is an example of a less-than-perfectly-balanced *red-black* tree:

```
0: "beta"-B("beta"-B)-
1: ="alpha"-B("beta"-B)-l "gamma"-B("beta"-B)-r
2: ="delta"-R("gamma"-B)-l
```

The corpus to generate such a tree might look like:

```
beta alpha
  gamma
```

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

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

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

```
s
```

Then a display of a *vbst* generated by this corpus would look something like:

```
$ bstrees -v data commands
0: "the"-2("the"-2)-
1: "quick"("the"-2)-l
2: "brown"("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 red-black tree, the resulting structure might look like:

```
$ bstrees -r data commands
0: "jumped"-B("jumped"-B)-
1: "fox"-R("jumped"-B)-l "quick"-R("jumped"-B)-r
2: "brown"-B("fox"-R)-l "girl"-B("fox"-R)-r "over"-B("quick"-R)-l ="the"-2-B("quick"-R)-r
3: ="and"-R("brown"-B)-l ="dog"-R("brown"-B)-r ="her"-R("girl"-B)-r ="lazy"-2-R("over"-B)-l
```

An empty tree should have the following display:

```
0:
```

The statistics to be reported are:

- the number of words/phrases 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 1. Here is an example statistics report:

```
Words/Phrases: 10
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, with each line having no preceding whitespace. There should be no trailing whitespace other than the terminating newline.

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 via *stderr*, an attempt to delete a word that does not exist in the tree. Thus you ought to be able to randomly generate a large number commands and have your interpreter run without failing.

Program organization

The tree portion of your code should be composed of three modules: *bst.c*, *vbst.c*, and *rbt.c*.

The *bst* module should implement the following functions:

- `bst *newBST(void (*display)(FILE *,void *),int (*comparator)(void *,void *));`
- `bstNode *insertBST(bst *tree,void *value);   //returns inserted node`
- `int findBST(bst *tree,void *value);          //returns 0 if not found`
- `bstNode *findBSTNode(bst *tree,void *value); //returns 0 if not found`
- `bstNode *swapToLeafBSTNode(bstNode *n);      //returns leaf node holding the original value`
- `void pruneBSTNode(bst *tree,bstNode *n);     //disconnects n from tree`
- `int sizeBST(bst *tree);               ;      //returns the number of nodes in the tree`
- `void statisticsBST(bst *tree,FILE *fp);      //displays min and max depth`
- `void displayBST(FILE *fp,bst *tree);       //displays tree, calls display function to display node value`

This module holds the functions that are common to both vanilla binary search trees and red-black trees. Note: your bst module must assume that all inserted values are unique. Your *vbst* and your *rbt* modules will handle duplicate values. Those modules will also keep track of the number of values (including duplicates) in the tree.
Here is a conforming *bst.h* file:

```
/*** common binary search tree class ***/

#include <stdio.h>

#ifndef __BST_INCLUDED__
#define __BST_INCLUDED__

typedef struct bstNode
    {
    struct bstNode *left;
    struct bstNode *right;
    struct bstNode *parent;
    void *value;
    } bstNode;

typedef struct bst
    {
    bstNode *root;
    int size;
    void (*display)(FILE *,void *);
    int (*compare)(void *,void *);
    } bst;

extern bst *newBST(void (*)(FILE *,void *),int (*)(void *,void *));
```

```
extern bstNode *insertBST(bst *,void *);
extern int findBST(bst *,void *);
extern bstNode *findBSTNode(bst *,void *);
extern bstNode *swapToLeafBSTNode(bstNode *);
extern void pruneBSTNode(bst *,bstNode *);
extern int sizeBST(bst *);
extern void statisticsBST(bst *,FILE *);
extern void displayBST(FILE *,bst *);
extern void checkBST(bst *);                //optional
#endif
```

The *vbst* module and *rbt* modules should take advantage of *bst* objects as much as possible. For example, a *vbst* object should encapsulate a *bst* object, much like your queue encapsulates a singly-linked list. The same is true of a *rbt* object. Another example is that a *vbst* delete method should simply make a call to *swapToLeafBST* followed by a call to *pruneBSTNode*. An *rbt* delete method would make a call to *swapToLeafBST* followed by a call to a deletion fixup routine followed by a call to *pruneBSTNode*.

Both *vbst* and *rbt* will need to wrap generic values with frequency count fields, while *rbt* will need to wrap in an additional color field. For example, a *vbst* 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, and a comparator pointer to point to the original comparator. As such, the constructors *newVBST* and *newRBT* will need the same argument types as the *newBST* constructor. are:

Here is a conforming *rbt.h* file:

```
/*** red-black binary search tree class ***/

#include <stdio.h>
#include "bst.h"

#ifndef __RBT_INCLUDED__
#define __RBT_INCLUDED__
typedef struct rbt
    {
    bst *tree;
    void (*display)(FILE *,void *);
    int (*compare)(void *,void *);
    int size;
    int words;
    } rbt;

extern rbt *newRBT(void (*)(FILE *,void *),int (*)(void *,void *));
extern void insertRBT(rbt *,void *);
extern int findRBT(rbt *,void *);
extern void deleteRBT(rbt *,void *);
extern int sizeRBT(rbt *);
extern int wordsRBT(rbt *);
extern void statisticsRBT(rbt *,FILE *);
extern void displayRBT(FILE *,rbt *);
extern void checkRBT(rbt *);               //optional
#endif
```

Your *vbst.h* file should be similar. Some of these methods will be wrappers for the similarly name *bst* methods, while others will add some functionality. Note that during some tests, your *bst*, your *vbst*, and your *rbt* module will be replaced, so don't add any additional public interface methods to the `.h` files. Note also that if you need a stack, you should use your stack class from assignment zero. Likewise if you need a queue (and you do need a queue), use your queue class from assignment zero.

Compliance

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

The insertion and deletion fixup routines for red-black trees must follow the pseudocode found on the Beastie website.

The error message printed when attempting to delete a value not in the tree should have the form similar to:

```
Value "x" not found.
```

There can be no whitespace other than a newline after the last printable character of each line on any output. No lines are indented.

You are required to use the *test2* drop box in order to ensure your modules can be replaced without error.

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 -v (vanilla bst) and -r (red-black tree). Here is an example call to your interpreter:

```
bstrees -v 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.

If the program adds an additional argument, all output goes to the file indicated by that argument:

```
bstrees -r corpus commands outputfile
```

Program output

All output should go to the console (stdout), unless otherwise directed. 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
```

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 *bstrees* (not *bstrees.exe* for you 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 results

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.