CS201: Data Structures and Algorithms

Assignment 2

Version 3

Introduction

You will compare the performance of green 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. A green tree is just like a regular binary search tree except that it can store duplicates in an efficient way. Your red-black tree will efficiently store duplicates as well. However, the plain binary search tree module upon which your green and red-black 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 red-black 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, 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. Green tree nodes do not have a color. Here is an example of a 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
      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 something like:

```
$ trees -g data commands
0: the-2(the-2)-
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 red-black tree, the resulting structure might look like:

```
$ trees -r data commands
0: jumped-B(jumped-B)-
1: fox-R(jumped-B)-l quickity quick-R(jumped-B)-r
2: brown-B(fox-R)-l girl-B(fox-R)-r over-B(quickity quick-R)-l =the-2-B(quickity 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
```

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

```
EMPTY
```

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. 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: *bst.c*, *gt.c*, and *rbt.c*.

The *BST* module

This module holds the functions that are common to both green binary search trees and red-black trees. Note: your *BST* module must assume that all inserted values are unique. Note also, that your BST module is different from the last assignment as you will not need the *key* field and the tree nodes will be made public.

Here is a conforming *bst.h* file:

```
#ifndef __BST_INCLUDED__
#define __BST_INCLUDED__

#include <stdio.h>

typedef struct bstnode BSTNODE;

extern void    *getBSTNODE(BSTNODE *n);
extern void    setBSTNODE(BSTNODE *n,void *value);
extern BSTNODE *getBSTNODEleft(BSTNODE *n);
extern void    setBSTNODEleft(BSTNODE *n,BSTNODE *replacement);
extern BSTNODE *getBSTNODEright(BSTNODE *n);
extern void    setBSTNODEright(BSTNODE *n,BSTNODE *replacement);
extern BSTNODE *getBSTNODEparent(BSTNODE *n);
extern void    setBSTNODEparent(BSTNODE *n,BSTNODE *replacement);

typedef struct bst BST;

extern BST *newBST(
    void (*)(FILE *,void *),           //display
    int (*)(void *,void *),            //comparator
    void (*)(BSTNODE *,BSTNODE *));    //swapper
extern void    setBSTroot(BST *t,BSTNODE *replacement);
extern BSTNODE *getBSTroot(BST *t);
extern BSTNODE *insertBST(BST *t,void *value);
extern BSTNODE *findBST(BST *t,void *value);
extern BSTNODE *deleteBST(BST *t,void *value);
extern BSTNODE *swapToLeafBST(BST *t,BSTNODE *node);
extern void    pruneLeafBST(BST *t,BSTNODE *leaf);
extern int     sizeBST(BST *t);
extern void    statisticsBST(FILE *fp,BST *t);
extern void    displayBST(FILE *fp,BST *t);
#endif
```

The BST and BSTNODE methods should all be placed in bst.c

Here are some of the behaviors your methods should have. This listing is not exhaustive; you are expected, as a computer scientist, to complete the implementation in the best possible and most logical manner.

- *newBST* - The constructor is passed three functions, one that knows how to display the generic value stored in a node, one that can compare two generic values, and one that knows how to swap the two generic values held by *BSTNODE*s (the *swapper* function is used by *swapToLeafBST*).
- *setBSTroot* - This method updates the root pointer of a *BST* object. It should run in constant time.
- *deleteBST* - This method is implemented with a call to the swap-to-leaf method followed by a call to the prune-leaf method, returning the pruned node. It should run in linear time for a green tree and logarithmic time for a red-black tree.
- *findBST* - This method returns the node that holds the searched-for value. If the value is not in the tree, the method should return null. It should run in linear time for a green tree and logarithmic time for a red-black tree.
- *swapToLeafBST* - This method takes a node and recursively swaps its value with its predecessor's (or successor's) until a leaf node holds the original value. It calls the *BST*'s swapper function to actually accomplish the swap, sending the two nodes whose values need to be swapped. If the swapper function is NULL, then the method should just swap the values as normal.
- *sizeBST* - This method returns the number of nodes currently in the tree. It should run in amortized constant time.
- *statisticsBST* - This method should display the number of nodes in the tree as well as the minimum and maximum heights of the tree. It should run in linear time.
- *displayBST* - This method should run in linear time.
- *getBSTNODE* - This method should return the value stored in the given search tree node.

The *GT* module and *RBT* modules should take advantage of *BST* objects as much as possible. For example, a *GT* object should encapsulate a *BST* object, much like a queue might encapsulate a singly-linked list. The same is true of a *RBT* object. For both *GT* and *RBT*, their size methods should return the number of nodes via the *BST* size method. Another example is that a *GT* delete method should simply make a call to *deleteBST*. An *RBT* 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 *GT* 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 *GT* 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 *newGT* and *newRBT* will need the same argument types as the first two arguments to the *newBST* constructor.

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

The red-black tree module

Here is a conforming *rbt.h* file:

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

#ifndef __RBT_INCLUDED__
#define __RBT_INCLUDED__

#include <stdio.h>

typedef struct rbt RBT;

extern RBT *newRBT(
    void (*)(FILE *,void *),           //display
    int (*)(void *,void *));           //comparator
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(FILE *,RBT *);
extern void displayRBT(FILE *,RBT *);

#endif
```

Here are some of the behaviors your methods should have. This listing is not exhaustive; you are expected, as a computer scientist, to complete the implementation in the best possible and most logical manner.

- *newRBT* - The constructor is passed two functions, one that knows how to display the generic value to be stored and one that can compare two of these generic values.
- *findRBT* - This method returns the frequency of the searched-for value. If the value is not in the tree, the method should return zero.
- *sizeRBT* - This method returns the number of nodes currently in the tree. It should run in amortized constant time.
- *wordsRBT* - This method returns the number of words (including duplicates) currently in the tree. It should run in amortized constant time.
- *statisticsRBT* - This method should display the number of words/phrases (including duplicates) and then call the BST display method.

Some of these methods will be wrappers for the similarly named *BST* methods, while others will add some functionality. You will need a private function to swap values. It should look something like:

```
function swapper(BSTNODE *a,BSTNODE *b)
    {
    RBTVALUE *ra = getBSTNODE(a);
    RBTVALUE *rb = getBSTNODE(b);

    /* swap the keys stored in the RBT value objects */
    void *vtemp = ra->value;
    ra->value = rb->value;
    rb->value = vtemp;

    /* swap the counts stored in the RBT value objects */
    int ctemp = ra->count;
    ra->count = rb->count;
    rb->count = ctemp;

    /* note: colors are NOT swapped */
    }
```

The *swapper* function is passed as the third argument to the *BST* constructor.

The only local includes a *RBT* module should have are *rbt.h* and *bst.h*.

The green tree module

Here is a conforming *gt.h* file:

```
/*** green binary search tree class ***/

#ifndef __GT_INCLUDED__
#define __GT_INCLUDED__

#include <stdio.h>

typedef struct gt GT;

extern GT *newGT(
    void (*)(FILE *,void *),           //display
    int (*)(void *,void *));           //comparator
extern void insertGT(GT *,void *);
extern int findGT(GT *,void *);
extern void deleteGT(GT *,void *);
extern int sizeGT(GT *);
extern int wordsGT(GT *);
extern void statisticsGT(FILE *,GT *);
extern void displayGT(FILE *,GT *);

#endif
```

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

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

Note that during some tests, your *BST*, your *GT*, and your *RBT* 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 as much as possible with as little modification as possible.

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 (red-black 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.

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

```
trees -r corpus commands outputfile
```

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

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
```

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.