MPCS 51100 - Fall 2018
PSet 2
Binary Search Trees

Due Date: October 22, 2018 @ 5:30pm on Canvas

Introduction

- In this assignment, you will be implementing a variety of short programs in C that deal with binary search trees.
- You should begin by cloning the github repository at `https://github.com/jtramm/51100_pset2`. The repository contains an API and functional prototypes that you will be filling in, as well as a testing framework. When you submit your code, you must maintain the same directory structure and filenames (e.g., p1.c, p2.c) so that your code can be compiled and run by the grader easily.
- In your top directory, you must populate the README.txt file to list your name, the assignment, and a discussion of any shortcomings your code may have. You will receive more partial credit if you are able to clearly identify faults or problems with your code, rather than letting us find them ourselves.
- Your code should be able to be compiled with the included makefile. That is, using gcc and with the following flags: `gcc -Wall -O2 -std=gnu99`. Unlike Problem Set 1, you are allowed to use the C99 standard.
- Many of the problems in this assignment have default test main functions that run a series of tests on your functions. You must retain these tests in your final solutions.
- If you are unable to implement the program correctly or fully, you may still receive partial credit, so turn in what you have done, and specify in the README where things may be problematic. If you have questions regarding expectations on the homework, please post your questions to Slack.
- Due date and extension policies are as prescribed by the course syllabus.
- The academic integrity policy for this assignment is as described in the course syllabus.

1 Binary Search Tree Verification (Recursive) - p1.c (10 points)

Write a recursive function to verify that a binary tree (of integers) is a proper binary search tree (BST). That is, your function should verify that:

1. the left subtree of a node contains only nodes with keys less than the node's key
2. the right subtree of a node contains only nodes with keys greater than the node's key
3. both the left and right subtrees are binary search trees themselves

Your function should assume a tree with distinct integer data (key) values and a node type of:

```
typedef struct tree_node{
    struct tree_node * left;
    struct tree_node * right;
    int data;
} Tree_node;
```

and have a prototype of:

```
int is_tree_recursive(Tree_node * tree, int min, int max);
```

Assume that empty children nodes are set to `NULL`, and that the minimum and maximum possible key values are exclusively bounded by the `INT_MIN` and `INT_MAX` macros defined in your system's `limits.h` header. Your function should return a non-zero value if the binary tree qualifies as a proper binary search tree, and zero if it does not.

For this problem you are provided a framework starting point that includes several test cases.

2 Binary Search Tree Verification (Iterative) - p2.c (15 points)

While recursive algorithms are often elegant and useful for theoretical analysis, in certain cases they can yield poor performance, sometimes making them impractical for large use cases. In such cases, iterative methods may instead be required.

Repeat the previous problem, but this time using an iterative method instead of recursion. Assume the same node types and parameters as the previous problem, with a functional prototype of:

```
int is_tree_iterative(Tree_node * tree);
```

While there are a number of ways of accomplishing this task, your implementation will utilize in-order tree traversal. Use a Stack data structure to implement the traversal, built on top of the Vector data structure that you implemented in Problem Set 1 (p9.c) with only slight modification:

```
typedef struct{
    Tree_node ** data;
    int size;
    int capacity;
} Vector;
```

where we are now storing an array of pointers to Tree_nodes. This will also require small modifications to your get(), append(), and init() Vector functions to accommodate the new data type. With those changes made, you should define your Stack type as:

```
typedef struct{
    Vector * v;
} Stack;
```

and implement push(), pop(), and init_stack() functions to complete your basic Stack type.

In p2.c, you have been given a framework and some test trees to build your code off of. Begin by importing and modifying your Vector code from p9.c in Problem Set 1.

3 Binary Search Tree Comparison - p3.c (10 points)

Given the same node structure from the previous two problems, write a recursive function:

```
int is_same(Tree_node * tree1, Tree_node * tree2);
```

that determines if two binary search trees are identical; that is, that they have the same values, and the values are arranged in the same hierarchy. Your function will return a non-zero value if the trees are the same, and zero if not. You have been given a framework in p3.c with a few test cases to build off of.

4 Dictionary (Naive BST) - p4.c (15 points)

Implement a dictionary of word/definition pairs using an unbalanced binary search tree as the underlying data structure. That is, the "key" for your binary search tree is the word itself, which can be numerically compared to other keys using the `strncmp()` C function from `string.h`. The "value" is the definition (in sentence form) of the word. Your program should accept command line input interactively as:

1. `$ add word "definition"`

   Adds the given word and associated definition to the dictionary. Print success or failure message to screen.

2. `$ import file file_name`

   Adds one or more entries stored in a text file. Print success or failure message to screen.

3. `$ delete word`

   Remove word from dictionary. Print success or failure message to screen.

4. `$ find word`

   Find definition associated with input word and print to screen.

5. `$ print`

   Show the contents of the dictionary in key order.

6. `$ clear`

   Deletes all entries in the dictionary and associated allocations. Print success or failure messsage to screen.

7. `$ quit`

   Quits the program and returns to the command line. Program state is not saved.

Handle input errors in a reasonable way (e.g. the program shouldn't crash if the input is incorrect, a key cannot be found, a duplicate key is added, etc.). You can assume words are always given in lowercase format only. Note that you are required to use a binary search tree, and you are not allowed to call any other sorting functions (e.g., quicksort) as part of your implementation. For the "delete" and "clear" functions, you must be sure to free any dynamically allocated memory associated with deleted nodes to ensure there are no memory leaks. Adding a new word, deleting a word, and finding a word should all be accomplished in $\mathcal{O}(\log n)$ time for the average case, and $\mathcal{O}(n)$ in the worst case.

In p4.c, you will find a framework to build your code off of. You have also been given a test file, `dictionary.txt`, to aid you in debugging your code. You are welcome to add in additional helper functions, or change the existing functional interfaces if you desire. Assumptions limiting the length of user inputs for word and definition lengths are as defined in p4.c by macros.

5 Dictionary (Self Balancing BST) - p5.c (15 points)

After publishing your program from Problem 4, you receive critical feedback from a user that claims the "find" command always runs for them in $\mathcal{O}(n)$ time rather than $\mathcal{O}(\log n)$ time. Upon investigating, you find that the most common use case for your user is for them to "import" a dictionary file that is already in alphabetical order, or to "add" in keys that are already alphabetically ordered. The user never uses the "delete" option.

Redo your code from Problem 4 to fix your user's issue, such that importing an alphabetized file into BST format and adding new entries no longer results in $\mathcal{O}(n)$ lookups. You should accomplish this by implementing a Red-Black self balancing binary search tree as discussed in class. Your self balancing tree should provide for all of the same operations as were required for Problem 4, with the exception that you do not need to be able to delete individual elements.

In the worst case, your program should execute "find" commands in $\mathcal{O}(\log n)$ time. You have been given a framework in p5.c to build your code off of, and a large alphabetized test dictionary is available in "alphabetized_dictionary.txt".

You have been given a framework to build off of in p5.c. Many functions, such as `find()`, can be directly imported from your solution to problem 4. Additionally, you are encouraged to break up the self balancing functionality of the `add_word()` function into a number of smaller helper functions as you see fit. Similar to problem 4, you are welcome to change the framework's functional interfaces if you like.

6 Dictionary Analysis - p6_naive.c & p6_balanced.c (5 points)

Start by copying your finished code from problems 4 and 5 into the p6_naive.c and p6_balanced.c files. You will be modifying your code from those problems slightly to add in timers to several functions. You should alter your codes so they no longer require interactive user inputs, but rather only execute the required functions in order to measure the amount of time they take to complete. Namely, we want to know for both your naive code and balanced code:

1. How long it takes (in seconds) to import the large ($> 83,000$ line) alphabetized dictionary file.
2. How long it takes (on average) to find a definition given a random (valid) word, in terms of microseconds (us) per word lookup.

For the first part, you will only need to time your import function once for each method. For the second part, you will want to perform a large number of randomized lookups. As you'll need to sample valid key input values, you should write a function to allocate and fill an array for all the words in your dictionary file. Then, you can randomly generate integer indices into this array to obtain your random sample key values. Test the amount of time it takes to lookup 100,000 random words for each method on your computer, and report the time per lookup in microseconds for each method. Be sure to disable any print statements in your `find()` function, as we don't want to measure the amount of time it takes to write things to `stdout`.

Report your four timing results in your `README.txt` file. Also include a brief discussion of the results and if they are surprising or not.

Note that you may run into unexplained segmentation faults when loading or using the "alphabetized_dictionary.txt" file with your naive dictionary program if you are using recursive techniques. This is because recursive calls on your binary tree may result in $\mathcal{O}(n)$ recursive calls being loaded onto your stack rather than $\mathcal{O}(\log n)$ calls, which for large problem sizes can cause stack overflow issues. Using the `-O2` compiler optimization flag can help make recursive calls more efficient. If you still run into segmentation faults or other stack overflow issues when running your naive dictionary with the large "alphabetized_dictionary.txt" file, try halving the size of the dictionary file (multiple times if necessary on your computer) for your timing studies, or setting the stack limit to unlimited if on a unix based system (`ulimit -s unlimited`).

7 Related Binary Searches - p7.c (10 points)

Imagine m arrays of floating point values where the i'th array contains n_i values. Each array is sorted from smallest to largest.

Given a search value p as input, design and implement an $\mathcal{O}(m \log |n|)$ algorithm that fills a results array of m ints representing the lower bounding index of p in each array. That is, the indices returned should correspond to the index in each array whose value is less than or equal to p. You may assume that p falls within the max and min values of each array, and that that values n_i vary little between arrays, so that $|n|$ can be taken as a representative average value. Your function should use the following `Grid` type and have the prototype:

```
typedef struct{
    float ** arrays;
    int m;   /* Number of arrays */
    int * n; /* Length of each array */
} Grid;

void grid_search( float p, Grid grid, int * results );
```

You have been given a framework and a test case in to build off of in p7.c You are free to use any C standard library functions you like.

8 Improved Related Binary Searches - p8.c (10 points)

Attempt to redo Problem 7 by designing and implementing a more efficient algorithm that is of lower time complexity than $\mathcal{O}(m \log |n|)$. Assume that the arrays will be searched many times (with many different values of p) so as to amortize the cost of any initial data restructuring, which should not factor into the cost of your time complexity analysis. You are allowed to allocate additional memory in your initial data restructuring routine. In your README.txt file, define the search time efficiency and memory usage requirements of your improved algorithm. Your improved method may involve additional data structures and/or helper functions. So, you are allowed to add to the `Grid` type, but things should meet the following interfaces:

```
typedef struct{
    float ** arrays;
    int m;   /* Number of arrays */
    int * n; /* Length of each array */
    /* Add Additional Fields Here */
} Grid;

void init_improved_grid_search( Grid * grid );
void improved_grid_search( float p, Grid grid, int * results );
```

where your `init_improved_grid_search()` function is called once to prepare any acceleration structures, and then any number of searches can be performed without having to initialize again. You have been given a framework and a test case in to build off of in p8.c. You are free to use any C standard library functions you like.

9 Code Cleanliness & Coding Habits (10 points)

- Your code should be submitted with the same filenames and directory structure as the cloned github repository, all contained within a single zip file.
- We should be able to compile all problems using the included makefile with gcc.

- We are not grading to any particular coding standard and we do not have any strict requirements in terms of style. However, your code should be human readable, which means it should be very clean, well commented, and easy for the grader to understand.