ECE 2230: Computer Systems Engineering
Machine Problem 5

Spring 2022

Due: 11:59 pm, Monday, April 19

1 Introduction

The goal of this machine problem is to design a binary-search-tree module and implement two insertion methods using a modular design similar to the design for the `list.c` module from MP2. In particular, we provide one header file that contains key definitions of the data structures that are needed for the interface and the prototype definitions for the functions that are the interfaces to the BST module, `bst.h`. You will need to create internal functions inside `bst.c`. The new function's scope should be limited to `bst.c`. Also, expand upon the provided `lab5.c` file with test drivers. Be sure to also submit a makefile that correctly builds your program if you make any modifications.

Three additional documents should be included. The first required document is a **test plan** that describes details of your implementation and how you plan to verify that the code works correctly. The second required document is a **test log** that demonstrates how you verified that the code works correctly. The third document describes your **performance evaluation**, and the details are described below.

2 Interface specifications

The tree must have a header with type `bst_t`. The header stores pointers to memory blocks based on keys with type `bst_key_t`. For testing purposes, use keys that are non-negative integers. Here is an example of the structure definitions, but you may need to modify some of the details of the structure to suit your design.

```
enum balanceoptions {BST, AVL};
typedef void *data_t;
typedef int bst_key_t;


typedef struct bst_node_tag {
        data_t data_ptr;
        bst_key_t key;
        int height;
        struct bst_node_tag *left;
        struct bst_node_tag *right;
} bst_node_t;

typedef struct bst_tag {
        bst_node_t *root;
        int size;
        int num_recent_rotations;
        int policy; // must be one of the balance options
        int num_recent_key_comparisons;
} bst_t;
```

3 BST interface

The following are functions required for the BST interface

`data_t bst_access (bst_t *, bst_key_t);` – Find the tree element with the matching key and return a pointer to the data block that is stored in this node in the tree. If the key is not found in the tree, then return `NULL`.

`bst_t *bst_construct (int tree_policy);` – Create the header block for the tree and save the tree's `policy` in the header block. The tree's `policy` must be either equal to `BST` or `AVL`, and the labels must be defined with an `enum` as shown above. Initialize the root pointer to `NULL`. The size stores the current number of keys in the tree. The `num_recent_key_comparisons` stores the number of key comparisons during the most recent access, insert, or remove. Use Standish's definitions for the number of comparisons even if your implementation is slightly different. That is, there is one comparison to determine if the key is found at the current level and if the key is not found one more comparison to determine if the next step is to the left or right. Do not count checks for `NULL` pointers. The `num_recent_rotations` stores the number of tree rotations during the most recent access, insert, or remove.

`void bst_destruct (bst_t *);` – Free all items stored in the tree including the memory block with the data and the `bst_node_t` structure. Also frees the header block.

`int bst_insert (bst_t *, bst_key_t, data_t);` – Insert the memory block pointed to by `data_t` into the tree with the associated key. The function must return 0 if the key is already in the tree (in which case the data memory block is replaced). The function must return 1 if the key was not already in the tree but was instead added to the tree. The insertion function should use the basic insertion method if the tree's policy is `BST`, and must balance the tree to maintain the `AVL` property for each node if the tree's policy is `AVL`.

`data_t bst_remove (bst_t *, bst_key_t);` – *Note: you are not required to implement the remove function for this assignment.* Create a function that does not work and just returns a `NULL` pointer, so that the `lab5.c` drivers will compile correctly. Implementing the remove function is an optional assignment. This function removes the item in the tree with the matching key. Return the pointer to the data memory block and free the `bst_node_t` memory block. If the key is not found in the tree, return `NULL`. If the tree's policy is `AVL`, then ensure all nodes have the AVL property.

`int bst_size(bst_t *);` – Return the number of keys in the tree.

`int bst_key_comparisons (bst_t *);` – Return `num_recent_key_comparisons`, the number of key comparisons for the most recent call to bst_access, `bst_insert`, or `bst_remove`.

`int bst_rotations (bst_t *);` – Return `num_recent_rotations`, the number of rotations for the most recent call to `bst_insert` or `bst_remove`.

`int bst_int_path_len(bst_t *);` – Return the internal path length of the tree

4 Testing

Test your module extensively and write a detailed description in your **test plan/log**. Make sure to test special cases such as boundary conditions. These tests should be added as drivers to the file `lab5.c` and documented in your test plan. For example, use the "`-u`" driver specified below to construct example trees that show you code is correct.

4.1 Provided test drivers

Four different test drivers are included in `lab5.c`. The first driver is a unit driver to run tests in which a specific list of keys are inserted into the tree. Three additional drivers examine the successful and unsuccessful access times for trees with shapes that are optimum, random, and poor (if not balanced). Compile `lab5.c` and run "`lab5 -help`" to see a list of options.

1. `lab5 -u 0` This driver allows you to specify a list of keys to insert into the tree (and, optionally, a list of which of those keys to then remove from the tree). You can add additional unit driver cases to specify specific trees you want to build. To specify the tree, make an integer array with the list of keys. For example, here is a list that builds a tree starting with key 100 as the root.

```
const int ins_keys[] = {100, 50, 125, 25, 75, 65, 60, 70, 110, 120, 115, 122};
```

2. `lab5 [-o -r -p] -w levels -t trials` The driver tests bst_insert and bst_access. An optimal tree is built with the number of levels in the tree equal to levels. If trials is greater than zero, for each trial a random key is generated and bst_access is used to search for the key. The average number of successful and unsuccessful searches is printed. You specify one of `-o, -r,` or `-p` to make the initial shape of the tree optimal, random, or poor. Verify that the expected number of searches predicted by the theory matches the measured performance from your program to three significant digits when run with 1,000,000 trials.

Example start to a test script (use with both `-f bst` and `-f avl`)

1.
```
lab5 -u 0                          // unit driver with custom tests 0 through 3
lab5 -o -w 5 -t 0 -v               // tests inserts only and prints tree
lab5 -r -w 5 -t 0 -v -s 1          // same with random tree
lab5 -p -w 5 -t 0 -v -s 2          // same with poor tree
lab5 -o -w 20 -t 1000000           // tests inserts and accesses
lab5 -r -w 20 -t 1000000           // same with random tree
lab5 -p -w 20 -t 1000000           // same with poor insertion order
```

5 Performance evaluation

For your performance evaluation, discuss the data collected for the number of successful and unsuccessful searches, and compare to the expected values as developed in the textbook by Standish. Consider both the optimal and random trees as generated by the provided drivers. Also, discuss the performance you would expect for a worst case tree. Describe how your implementation supports the claim that the successful search time has a complexity class $\mathcal{O}(\log n)$.

6 Optional MP Replacement: Deletions (75 points)

Implement the remove function for both deletions that do not involve any rebalancing, and with rebalancing that maintains the AVL property. Extend the testing document to demonstrate that all of the options work correctly. Extend the performance analysis to demonstrate the performance of all combinations of options. Discuss the advantages and disadvantages of each combination and support your claims with data collected from your test drivers and the supplied test drivers.

Grading: The optional assignment will be graded separately from MP5, and is worth up to a maximum of 75 points. At the end of the semester if you have a grade for one of the MP's that is below 75 points, the score for this optional assignment can be used to replace that grade. Note that all MP scores are based on a 100 point scale, so the optional assignment can raise a score for an MP to at most 75 out of 100 points.

If you are interested in the optional assignment see me to discuss details for the due date for this additional optional assignment. This optional assignment does not replace MP5. Complete MP5 as assigned. Use the equilibrium driver to test if deletions are working correctly.

The unit driver (-u) can be extended to specify the key or keys to remove. Make another array with a list of keys. Leave this array empty until you implement the remove function.

```
const int del_keys[] = {100, 110};
```

lab5 –e –w levels –t trials - This driver tests a random sequence of inserts and removes. The initial tree is generated randomly with the number of levels equal to levels. Then for each trial a random key is generated and the probably the key is in the tree is approximately 0.5. With probably 0.5 the key is inserted (or replaced) in the tree and with probability 0.5 the key is removed from the tree (if it is found in the tree).

Example tests (do for both -f bst and -f avl):

```
lab5 -e -w 5 -t 10 -v -s 2 // tests random removes
lab5 -e -w 20 -t 100000 // exercise tree
```

7 Submission

Command line arguments must be used to modify parameters for the test drivers, and options for the tree module. See the comments in lab5.c for the command line options and their meanings.

Compress all code, test plan, and optional performance evaluation into a ZIP file, and submit the ZIP file to Canvas by the deadline. **Your last submission is the one that will be graded.**

See the ECE 2230 Programming Guide for additional requirements that apply to all programming assignments. Work must be completed by each individual student, and see the course syllabus for additional policies.