ECE 2230: Computer Systems Engineering **Fall 2018**
Machine Problem 5 **Due: 11:59 pm, Tuesday, October 30**

The goal of this machine problem is to design and implement a binary search tree (BST) module.

Use a modular design similar to the design for the **`list.c`** module from MP2, or the **`mem.c`** module from MP4. In particular, develop 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. Also, expand upon the provided **`lab5.c`** file with new unit test drivers. Be sure to also submit a **`makefile`** that builds your program.

Two additional documents should be submitted. One is a **test plan** that describes details of your implementation and demonstrates, with a test script, how you verified that the code works correctly. The verification should include detailed prints from your program to show that you program operates correctly. The second document describes your **performance evaluation**, and the details are described below.

Interface specifications

The BST should have a header **`bst_t`**, and should store pointers to memory blocks based on a key 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 will need to modify some of the details of the structure to suit your design.

```
enum balanceoptions {BST, AVL, TWOTHREET};

typedef void *data_t;
typedef int bst_key_t;
typedef struct bst_node_tag {
      data_t data_ptr;
      bst_key_t key;
      struct bst_node_tag *left;
      struct bst_node_tag *right;
} bst_node_t;
typedef struct bst_tag {
      bst_node_t *root;
      int tree_policy;        // must be a balanceoptions
      int tree_size;
      int num_recent_key_comparisons;
} bst_t;
```

The following functions are required.

`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_policy` in the header block. The `tree_policy` must be one of the labels defined with an `enum` as shown above. **While the definition allows for multiple types of trees, you are only required to implement the BST option.** Initialize the root pointer to null. The `tree_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.

`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 BST (in which case the data memory block is replaced). The function must return 1 if the key was not already in the BST but was instead added to the tree. **You are required to implement the BST policy only**. If you elect to implement more than one type of tree, then the insertion method should conform to the **`tree_policy`** that is defined when the tree was initially constructed.

`data_t bst_remove (bst_t *, bst_key_t);`

Remove 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. **You are required to implement the BST policy only**. If you elect to implement more than one type of tree, then the deletion method should conform to the **`tree_policy`** that is defined when the tree was initially constructed.

`int bst_size(bst_t *);`

Return the number of keys in the BST.

`int bst_stats (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_int_path_len(bst_t *);`

Return the internal path length of the tree

In addition you should make at least two debugging functions. See below for examples of these functions.

```
void bst_debug_print_tree(bst_t *);
void bst_debug_validate(bst_t *);
```

Testing and performance evaluation

Test your library extensively and write a detailed description in your **test plan**. 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. Use the "**`-u`**" unit driver specified below to construct example trees that show your code is correct.

Provided test drivers

Five different test drivers are included in **`lab5.c`**. Three drivers examine the successful and unsuccessful access times for trees with shapes that are optimum, random, and poor. The fourth driver provides an example of designing a unit driver that allows you to specify keys to insert into the tree and keys to delete. The fifth driver, called equilibrium, builds a random tree and then performs insertions and deletions for a large number of trials. Compile **`lab5.c`** and run "**`lab5 –help`**" to see a list of options.

1. `lab5 [-o –r –p] –w levels –t trials`
   The driver tests `bst_insert` and `bst_access`. An initial 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.

2. `lab5 –u 0`
   This driver allows you to specify a list of keys to insert into the tree and a list of which of those keys to then remove from the tree. This tests both `bst_insert` and `bst_remove`. You can build multiple unit drivers each with two arrays to specify the tree you want to build and the keys you want to delete. To specify the initial 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};
```

   To specify the key or keys to remove, make another array with a list of keys. For example, this list results in the root key being removed from the tree, followed by the key 110.

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

3. `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 start to a test script

```
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 1   // same with poor tree
lab5 -o –w 16 –t 50000      // tests inserts and accesses
lab5 -r –w 16 –t 50000      // same with random tree
lab5 -p –w 16 –t 50000      // same with poor tree
lab5 –u x                   // unit test x with removes
lab5 –e –w 5 –t 10 –v –s 2  // tests random removes
lab5 –e –w 20 –t 100000     // exercise the tree
```

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 O(log n).

Notes

Command line arguments should be used to modify parameters for the test drivers and any options for the BST. See **`lab5.c`** for the arguments that are already defined.

Here is a crude but simple way to print a tree (you need another function to call this function with a pointer to the root of the tree).

```
void ugly_print(bst_node_t *N, int level) {
    if (N == NULL) return ;
    ugly_print(N->right, level+1) ;
    for (int i=0; i<level; i++) printf("     ");  /* 5 spaces */
    printf("%5d\n", N->key);                       /* field width is 5 */
    ugly_print(N->left, level+1);
}
```

Here is a function to partially validate the tree. Add **`#include <limits.h>`** for definitions of **`INT_MIN`** and **`INT_MAX`**.

```
void bst_debug_validate(bst_t *T)
{
    int size = 0;
    assert(bst_debug_validate_rec(T->root, INT_MIN, INT_MAX, &size) == TRUE);
    assert(size == T->size);
}
int bst_debug_validate_rec(bst_node_t *N, int min, int max, int *count)
{
    if (N == NULL) return TRUE;
    if (N->key <= min || N->key >= max) return FALSE;
    assert(N->data_ptr != NULL);
    *count += 1;
    return bst_debug_validate_rec(N->left, min, N->key, count) &&
        bst_debug_validate_rec(N->right, N->key, max, count);
}
```

Verify your program has no memory leaks. See the ECE 223 Programming Guide for additional requirements that apply to all programming assignments. For code to be accepted for grading, it must pass the five test drivers that are included in **`lab5.c`**.

All code, a makefile, a test script, and a test log must be submitted to the ECE assign server. You submit by email to ece_assign@clemson.edu. Use as subject header ECE223-1,#5. When you submit to the assign server, verify that you get an automatically generated confirmation email within a few minutes. If you don't get a confirmation email, your submission was not successful. You must include your files as attachments. Your email must be formatted as plain text (not html). You can make more than one submission but we will only grade the final submission. A re-submission must include all files. You cannot add files to an old submission.

Work must be completed by each individual student, and see the course syllabus for additional policies.

Optional additional assignment for MP5

Implement extensions to the BST module to support either AVL or Two-three trees. 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 70 points. At the end of the semester if you have a grade for one of the MP's that is below 70 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 70 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 for a BST.