CS 392 Spring 2024 Systems Programming — Homework 2

Preliminary

Concepts you need to understand for this assignment:

- C project file dependencies and Makefile;
- Binary search tree basic structure and algorithms;
- Dynamic allocation and memory leak;
- Void pointers and function pointers.

Your task is to write a Binary Search Tree (BST) that can store any type of data in its nodes, and related functionalities. A binary tree is a tree where each node has zero to two child nodes. We usually use left and right child to refer to the two child nodes. A binary search tree is a special binary tree, where a node's data is always greater than its left child's and smaller than its right child's.

File Structure

In your starter kit, you are provided with the following five files:

- `main.c` : for testing your code;
- `bstree.h` : declares data types and functions needed for BST;
- `bstree.c` : implements functions for BST;
- `utils.h` : declares type-specific functions needed;
- `utils.c` : implements type-specific functions.

The red colored files are the ones you need to complete.

In the following sections, we will walk through the files thoroughly. Please read them carefully before start your assignment.

1 Task 1: Utility Functions (`utils`)

It is clear that during construction of a BST we need to constantly compare data to decide if a new node is going left or right. However, the `data` field in `node_t` is a void pointer, meaning it can be any type of data. Therefore, when using the functions and types declared in `bstree.h`, we have to create type-specific functions. In this assignment, you will need to create type-specific functions for two types: `int` and `float`:

The first function is to compare two data:

```
1 int cmpr_int(void*, void*);
2 int cmpr_float(void*, void*);
```

If the first number is larger, return 1; if the second number is larger, return -1; otherwise return 0. In `cmpr_int()`, you'll compare two integers, while in `cmpr_float()`, you'll compare two float numbers. This function will be handy when you're calling `add_node()` function.

The section function is to print data:

```
void print_int(void*);
void print_float(void*);
```

where you print one node's data at a time by calling `printf()` with correct format.

In summary, the four functions you need to complete:

- `cmpr_int()`;
- `cmpr_float()`;
- `print_int()`;
- `print_float()`.

2 BST Types and Functions (`bstree.h`)

Based on the definition of BST, we declare a `node_t` type, used for all tree nodes:

```
typedef struct node {
    void*  data;
    struct node* left;
    struct node* right;
} node_t;
```

where `data` is a `void*`, since our goal is to create a BST that can store any type of data. Apparently, whenever you create a new node, for the `data` field, you'd have to use `malloc()` function to dynamically create a memory space for storing the data.

We also declare a `tree_t` type which includes a root that points to the root node, as well as three function pointers:

```
typedef struct tree {
    node_t* root;
    void (*add_node)(void*, size_t, struct tree*, int (*)(void*,void*));
    void (*print_tree)(node_t*, void (*)(void*));
    void (*destroy)(struct tree*);
} tree_t;
```

The function pointers also indicate the three major functions you're going to implement.

Task 2: Adding Nodes: `add_node`

The function you'll need to complete is declared as follows:

```
void add_node(void* , size_t, tree_t*, int (*)(void*,void*));
```

The arguments are used this way:

(1) `void*`: this is where you pass the address of a new data you want to add to the tree;

(2) `size_t`: indicates the number of bytes of the new data;
(3) `tree_t*`: points to the object of `tree_t` you created for testing;
(4) `int (*)(void*,void*)`: a function pointer pointing to a type-specific `cmpr` function in `utils.h`.

Inside this function, you need to first check whether `tree_t`'s `root` is `NULL`. If so, you should `malloc()` a new `node_t`, and connect `root` with that new node. Then to put the data in the node, you'd need to `malloc()` again to the `data` field in `node_t`, and copy the data over.

In summary, you need two `malloc()`'s every time you insert a new node: first time is for a new `node_t` object, while the second time is for the `data` field in this new object. **When there's a duplicate, insert it as the right child.**

Warning: when copy the data from the argument to the `data` field in the new node, you should use a loop to copy one byte at a time. Any type specific assignment (other than types that take only one byte) is prohibited. You are not allowed to use `sizeof()` to check data type and hard code data types.

Task 3: In-Order Printing the Tree: `print_tree`

You will complete `print_tree()` function to print the data in the BST from smallest to the biggest:

```
void print_tree(node_t*, void (*)(void*));
```

The first parameter points to the tree node where you start traversing the tree, while the second is a function pointer used to print type-specific data. You should pass one of the two `print` functions completed in `utils.h`.

You are free to use recursion or loops, but in this function it'll be much simpler to use recursion.

Task 4: Destroy the Tree: `destroy`

During `add_node()` we `malloc()`'ed a lot, so before exiting our program we need to free those spaces on the heap by calling `destroy()`:

```
void destroy(tree_t*);
```

where `tree_t*` argument is the pointer to the tree. One thing to pay attention to is how many `malloc()` we used in `add_node()`, then we should `free()` same amount of time. Also, for each node, what's the order of multiple `free()`'s?

Again, it'll be easier to do that in recursion. **You are free to create helper functions.**

Note

We will emphasize this again: because this is a generic BST, you **must not** use `if-else` to discriminate different data types in your code. For example, the following is not allowed:

```
if (sizeof(data) == 4) {...}
else {...}
```

3 Tester

To help you with testing, we provided a tester file `tester_bst_arm` for ARM-64 machines, and `tester_bst_x86` for X86-64 machines. In the following, we use `tester_bst_arm` as an example.

After downloaded the tester, please compile your code with it first:

```
$ gcc tester_bst_arm utils.c bstree.c
```

And then run `a.out` to see help manual:

```
$ ./a.out -h
```

To check memory leaks, use the following command:

```
$ valgrind --leak-check=full ./a.out -t all
```

4 Grading

The homework will be graded based on a total of 100 points.

- Task 1 (20 pts): 10 test cases in total, **2** points each;
- Task 2 (30 pts): 10 test cases in total, **3** points each;
- Task 3 (20 pts): 10 test cases in total, **2** points each;
- Task 4 (30 pts): 10 test cases in total, **3** points each;

After accumulating points from the testing above, we will inspect your code and apply deductibles listed below. The lowest score is 0, so no negative scores. Note that all the deduction rules are also applied to the helper function you created and called in that task.

- Task 1 `utils` (20 pts):
  - none;
- Task 2 `add_node` (20 pts):
  - **-20:** data is not dynamically allocated;
  - **-20:** directly dereferenced pointer with specific types (other than `char*`);
  - **-10:** used `sizeof()` on primitive types;
  - **-10:** did not call `compare` function in task 1, and/or did not call the function through the function pointer passed;
- Task 3 `print_tree` (20 pts):
  - **-20:** directly dereferenced pointer with specific types (other than `char*`);
  - **-10:** used `sizeof()` on primitive types;
  - **-10:** did not call `print` function in task 1, and/or did not call the function through the function pointer passed;
- Task 4 `destroy` (30 pts):

- **-30:** only root node is destroyed;
- **-20:** data is not destroyed.

► General (only deduct once):

- **-100:** the code does not compile, or executes with run-time error;
- **-100:** the code is generated by AI, and/or through reverse engineering on the tester;
- **-30:** memory leak through valgrind (only "definitely lost" category);
- **-10:** no pledge and/or name in C file.

Earlybird Extra Credit: 2% of extra credit will be given if the homework is finished two days before the deadline. For specific policy, see syllabus.

Deliverable

Submit the following files on Canvas. No need to zip.

(1) `bstree.c`;
(2) `utils.c`.