CSCI243 - Mechanics of Programming

Homework 5 - Binary Search Trees

Due Date

7/25/2017 at 11:59pm

Goals and Objectives

This assignment gives you experience in dynamic allocation of memory and the manipulation of dynamically allocated, self-referential structures.

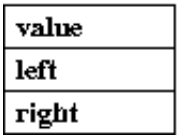
Overview

This program will read integer values from stdin and build a binary search tree with those values. The program requires a command line argument that will specify how many integers to read. After reading all of the values, they must be echoed back to stdout. The integers will then be inserted into a binary search tree in the order that they were entered. Finally, the tree will be traversed with *preorder*, *inorder*, and *postorder* traversals.

For this assignment you will work with a tree node structure that contains the following fields:

Field	Meaning
value	The number being stored at this node
left	Pointer to the left child of the node
right	Pointer to the right child of the node

This gives us a tree node that looks something like this:



You will use the following structure definition:

```
typedef struct TreeNode {
    int data;
    struct TreeNode* left;
    struct TreeNode* right;
} TreeNode;
```

You will implement a main program that gets input from the console, creates a binary search tree by inserting each input into the tree and traverses the final tree using pre-order, in-order and post-order traversals. Your implementation must have three required functions: one to insert a new node into a binary search tree, one that will traverse a tree using a parameterized specification for the traversal type, and a routine to clean up the tree.

Your solution *must use dynamically allocated memory* for each of the nodes in the tree. Your solution must have a root node with a linked structure containing all the inserted nodes in the tree. You will receive a zero for this assignment if you do not use a dynamically allocated linked structure.

Background

For a review on binary search trees, visit http://en.wikipedia.org/wiki/Binary_search_tree.

For a review on the various tree traversals, visit http://en.wikipedia.org/wiki/Tree_traversal.

Retrieving Source Code

Use the following command to retrieve the `bst.h` header file; do not modify this file.

```
get csci243 hw5
```

Program Specifications

Your `main` method must be in a file named `bst.c`. You should implement all the required functions, as well as any supporting functions in this file.

The prototype for the tree building method is:

```
void build_tree(TreeNode** root, const int elements[], const int count)
```

The parameter `root` is *a pointer to the pointer of the root node of the tree*. The parameter `elements` is a pointer to an array of integers and the parameter `count` is the number of elements in the array.

Sometimes called a 'handle', the `root` pointer is passed in *by value*, and the function must then allocate space for the tree and set the *pointee* of the `root` pointer to the address of the space allocated for storing the root node. The handle must always have valid storage; a null value for the pointee indicates an empty tree.

The prototype for the tree traversal method is:

```
void traverse(const TreeNode* root, const TraversalType type)
```

Where `root` is a pointer to the root node of the tree and `type` specifies which of the three traversals to do. The `traverse` function will traverse the entire tree and print each node's data at the time specified by the traversal `type`.

The traversal types are specified in an enumerated data type:

```
typedef enum {
    PREORDER,
    INORDER,
    POSTORDER
} TraversalType;
```

The prototype for the cleanup method is:

```
void cleanup_tree(TreeNode* root);
```

Where `root` is a pointer to the root node of the tree. It is responsible for deallocating all the nodes that were created in `build_tree`.

The `build_tree` Routine

The `build_tree` function takes these arguments:

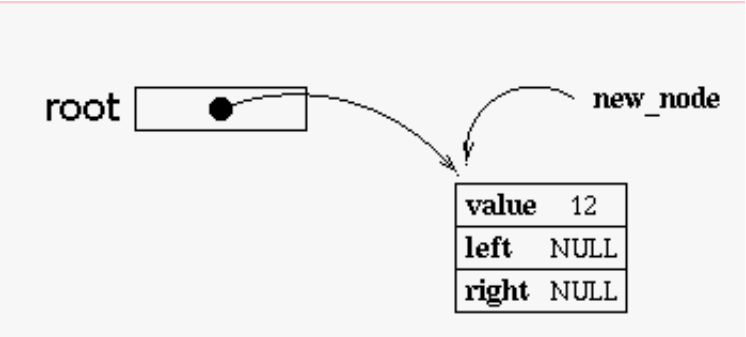
1. the pointer to the variable that contains the pointer to the root node of the tree (i.e., a pointer to a pointer to the root node),
2. (a pointer to) the array of numbers to be inserted into the tree, and
3. the count of how many numbers to insert in the tree.

The routine must create a new node for each provided number in the array, and then insert this new node in the proper position in the binary search tree, updating whatever pointer is necessary.

Keep in mind that a binary search trees cannot contain duplicate keys. If the new number matches a value that is already in the tree, *do not insert the duplicate value*.

If you have created a node that turns out to be a duplicate of one that already exists in the tree, make sure that you free the space allocated to the duplicate node prior to continuing.

To keep track of a binary search tree, you must have a variable, say `root`, that contains the address of the root node in the tree (i.e., a pointer to the root node). When the tree is empty, the contents of `root` will be zero (a NULL, meaning that there is no root to the tree). Inserting the first node into the tree requires that the address of the first node be stored in `root`; this is the reason why the first argument is the address of *the* `root` *variable* rather than just the address of the root node itself. With this in mind, the tree with one element inserted as the root would look like this:



The `traverse_tree` Routine

The second routine you must implement is named `traverse`; it takes these arguments:

1. The address of a tree node; any traversal begins at this node. This is a pointer to the tree node itself, not a pointer to a pointer to the node.

2. A constant that specifies which type of traversal to perform. The values in this field translate to the following traversal order:

Constant	Traversal Type
0	PREORDER
1	INORDER
2	POSTORDER

Your `traverse` routine must implement a recursive tree traversal; the type of traversal is indicated by the second parameter. (Note that an iterative traversal is possible but is not an acceptable solution to this assignment.

The `cleanup_tree` Routine

The third routine you must implement is named `cleanup_tree`; it takes one argument.

1. The address of a tree node, which should be the root node when the function is initially called.

This routine should be called after all the traversals complete, but before the `main` function returns.

Part of your grade is based on proper memory management. Make sure to run your program through `valgrind` to verify.

The `main` Routine

Your program should be run with a second argument which specifies the number of integers to read. If there is no second argument, you should display the following message (with a newline at the end) to standard output and exit:

```
Usage: bst #
```

If there is a second argument, we guarantee it will be an integer. If the integer is not greater than 0 you should display the following message (with a newline at the end) to standard output and exit:

```
# must be greater than 0
```

If the number of integers is greater than 0, we guarantee you will be given that exact number from standard output. You should display the following prompt (with a newline at the end) and then read them from standard input. Here, `#` character should be replaced with the number from the command line.

```
Enter # integer values to place in tree:
```

After reading all the values, they should be displayed to standard output, one per line, starting with the prompt:

```
Input values:
```

After displaying the values, the program will perform the traversals in the following order: preorder, inorder, postorder. Before each traversal begins, display which traversal is executing to standard output followed by a colon (:) and a newline:

```
Preorder:
Inorder:
Postorder:
```

When performing the traversals, each value should be printed to standard output. Each value should be followed by a newline.

Sample Output

```
$ bst 7
Enter 7 integer values to place in tree:
6
1
42
3
24
18
5
Input values:
6
1
42
3
24
18
5
Preorder:
6
1
3
5
42
24
18
Inorder:
1
3
5
6
18
24
42
Postorder:
```

```
5
3
1
18
24
42
6
```

Valgrind output

This is a sample of the valgrind output that demonstrates memory is being managed properly. Note, that anything read from a redirected input file will not be displayed.

```
$ valgrind --leak-check=full bst 7 < input.3
==11623== Memcheck, a memory error detector
==11623== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==11623== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==11623== Command: bst 7
==11623==

### SAMPLE OUTPUT FROM ABOVE ###

==11623==
==11623== HEAP SUMMARY:
==11623==     in use at exit: 0 bytes in 0 blocks
==11623==   total heap usage: 8 allocs, 8 frees, 112 bytes allocated
==11623==
==11623== All heap blocks were freed -- no leaks are possible
==11623==
==11623== For counts of detected and suppressed errors, rerun with: -v
==11623== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

You may wonder why there were 8 allocations when there were only 7 elements in the BST. In our solution, the array of values that was stored from input was also allocated on the heap; you are not required to do this.

Helpful Hints

Debugging these routines can be difficult because it's hard to tell if `build_tree` works without having the traverse tree routines working. Similarly, it's hard to test the `traverse` routine without having a correct tree to try and traverse.

We suggest that you start coding the `traverse` function first, using just a single node to start. Once your `traverse` function works with a single node, manually link some nodes together in your code and call `traverse`. After you have verified that `traverse` is working move on and write the `build_tree` routine. Finally, you should work on the `cleanup_tree` routine and use `valgrind` to verify you are managing all memory properly.

Submission

Use the command below to submit your program:

```
try grd-243 hw5-1 bst.c revisions.txt
```

Grading

This project is worth a total of 100 points, distributed as follows:

Functionality:	70
Memory Management:	20
Style:	10

The style grade includes use of the `git` version control system, your authorship, consistent formatting and documentation following the course's coding standards.
