

ECE368 Programming Assignment 2

Due Friday, July 16, 2021, 11:59pm

You are required to implement a program to construct a height-balanced binary search tree (BST), starting from an empty tree, based on a sequence of insertion and deletion operations. You may have to perform rotation(s) to maintain the height-balanceness of the tree after each insertion or deletion. Note that height-balanceness is defined as we covered in the class. Given a node, its balance is the height of its left subtree minus the height of its right subtree. A tree is height-balanced when every node in the tree has a balance that is -1 , 0 , or 1 . As in the class, we will define the height of a tree recursively. An empty tree has a height of -1 . The height of a tree is 1 plus the maximum of the height of the left sub-tree and the height of the right sub-tree.

It is fine for you to use the code provided in the lecture notes for this assignment. However, you should re-organize the code. For example, you may want to write separate functions for clockwise rotation and counterclockwise rotation, and the insertion function will call the appropriate rotation functions.

Insertion

In this assignment, **we allow the insertion of a duplicate key**. In other words, the resulting height-balanced may have multiple nodes storing the same key value. Recall that insertion of a key requires you to search for a suitable location to insert a leaf node. To handle nodes storing the same key value, when you search for a leaf node location to insert the key, you should **always go left when you encounter a node storing the same key**. *This requirement is imposed for the grading purpose of this assignment.*

Deletion

When you are asked to delete a key, the tree should stay intact if the key is currently not in the tree. There may be multiple nodes containing the key that you want to delete. You should delete the first such node that you encounter in the search process. If the node you want to delete has two children, you should replace that node with its **immediate predecessor** (in an in-order traversal of the tree). Again, this requirement is imposed for the grading purpose of this assignment.

Deliverables

In this assignment, you are required to develop your own include file(s) and source file(s), which can be compiled with the following command:

```
gcc -std=c99 -pedantic -Wvla -Wall -Wshadow -O3 *.c -o pa2
```

It is recommended that while you are developing your program, you use the “-g” flag instead of the “-O3” flag for compilation so that you can use a debugger if necessary.

There are two options the executable `pa2` can accept. The `main` function should simply return `EXIT_FAILURE` if the argument count is incorrect or the options are invalid (see further details below).

Option "-b": Building a height-balanced BST

```
./pa2 -b operations input file tree output file
```

The option "-b" means that you are building a height-balanced BST (starting from an empty tree) based on the operations specified in operations input file.

Input file format

The operations input file is an input file in binary format. Every operation is specified by $(\text{sizeof}(\text{int}) + \text{sizeof}(\text{char}))$ bytes, with the first $\text{sizeof}(\text{int})$ bytes being an int and the next $\text{sizeof}(\text{char})$ byte being a char. The int is the key. The char is the operation. If it is an insertion of the specified key, the char is an ASCII character 'i'. If it is a deletion of the specified key, the char is an ASCII character 'd'.

If there are n keys to be inserted or deleted, the file size is $n \times (\text{sizeof}(\text{int}) + \text{sizeof}(\text{char}))$ bytes.

Output file format

The `tree_output_file` is an output file in binary format.

This file stores the **pre-order traversal** of the constructed height-balanced BST. Each non-NULL node is represented by an `int` and a `char`. The `int` is of course the key stored in the node.

The `char` is a binary pattern, with the least significant two bits capturing the types of branches that node has. At bit position 0 (the least significant bit position), a 0-bit means that the right branch of the node is NULL. A 1-bit means that the right branch of the node is a non-NULL node. At bit position 1 (the second least significant bit position), a 0-bit means that the left branch of the node is NULL. A 1-bit means that the left branch of the node is a non-NULL

Therefore, a binary value of 2 or 3 means there is a left child. A binary value of 1 or 3 means that there is a right child.

Output and return value of main function

If the given input file can be opened, your program should attempt to build a height-balanced BST. If the given input file cannot be opened, the program should print the value -1 (using the format "%d\n") to the `stdout` and return `EXIT_FAILURE`.

Now, suppose the given input file can be opened. If in the process of building the BST, your program encounters a problem in the input file (wrong format, for example) or a failure in memory allocation, you should still write to the output file the tree that has been constructed so far. You should print the value 0 (using the format "%d\n") to the `stdout` and return `EXIT_FAILURE`.

We will test your program with valid input files of reasonable sizes. Therefore, it is unlikely that you will have to print the value 0 to the `stdout`.

If you can successfully read the entire input file to build a tree, you should print the value 1 (using the format "%d\n") to the `stdout` and return `EXIT_SUCCESS`; your program should return `EXIT_FAILURE` otherwise.

We are not asking you to check whether you can write to the output file. We will use the option "-e" to evaluate that.

Option "-e": Evaluating a height-balanced BST

```
./pa2 -e tree_input_file
```

The option "-e" means that you are evaluating a tree specified in the tree input file.

The tree input file is actually of the same format as the output produced using the "-b" option. For the given tree input file, your program should print three integers to `stdout` using the format "%d,%d,%d\n", where the first integer indicates the validity of the input file, the second integer indicates whether the tree is a BST, and the third integer indicates whether the tree is a height-balanced tree.

If the input file cannot be opened, the first integer should be -1. If it can be opened, but of the wrong format, the first integer should be 0. If it can be opened and is of the correct format, the first integer should be 1.

If the input file is a valid tree, the second and third integers are meaningful (otherwise, their values are not important). If the tree is a BST, the second integer is 1; otherwise, it is 0. If the tree is height-balanced, the third integer is 1; otherwise, it is 0.

The main function should return `EXIT_SUCCESS` only if the input file is valid (i.e., the first integer of the terminal output is 1); your program should return `EXIT_FAILURE` otherwise.

Electronic Submission

The project requires the submission (electronically) of the C-code (source and include files) through Brightspace. You should create and submit a zip file called `pa2.zip`, which contains the `.h` and `.c` files. Your zip file should not contain a folder.

```
zip pa2.zip *.c *.h
```

You should submit `pa2.zip` to Brightspace.

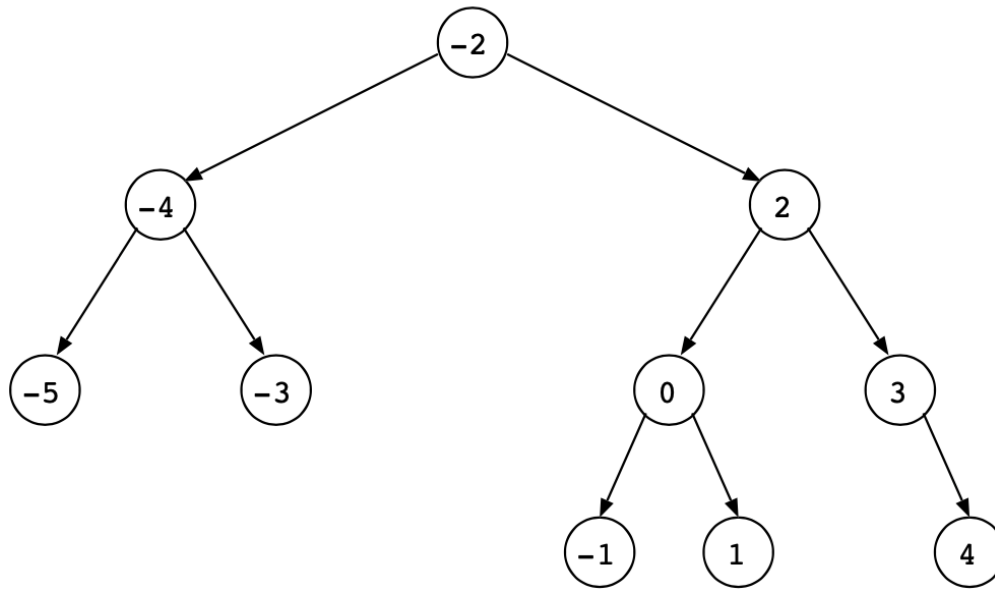
Grading

The assignment will be graded based on the two tasks performed by your program. The first task of constructing a height-balanced BST accounts for 70 points and the second task of evaluating a height-balanced BST accounts for 30 points. It is important all the files that have been opened are closed and all the memory that have been allocated are freed before the program exits. Memory errors or any errors reported by `valgrind` will result in a 40-point penalty.

What you are given

You are given 4 sample input files (ops0.b, ops1.b, ops2.b, ops3.b) and the corresponding output files for the trees constructed (tree0.b, tree1.b, tree2.b, and tree3.b).

The ops0.b file inserts 10 ascending keys -5, -4, ..., 3, 4. The following figure shows the corresponding height-balanced BST in tree0.b.



The files ops1.b and ops2.b contain the same insertion operations and a deletion operation: delete 2. In ops1.b, the “delete 2” operation occurs after the insertion of 2. Therefore, the final tree contains only 9 keys. In ops2.b, the “delete 2” operation occurs before the insertion of 2. Therefore, the tree2.b and tree0.b are the same.

While ops0.b, ops1.b, and ops2.b involve the insertion of distinct keys, ops3.b file involves the insertion and deletion of duplicate keys.

Besides these height-balanced BSTs, we also provide you three invalid height-balanced BSTs (invalidtree0.b, invalidtree1.b, and invalidtree2.b). For invalidtree0.b, it is neither a BST nor a height-balanced tree (the corresponding print out to the screen should be 1,0,0 when evaluated with the “-e” option). For invalidtree1.b, it is not a BST, but it is a height-balanced tree (the corresponding screen dump is 1,0,1). For invalidtree2.b, it is a BST, but it is not a height-balanced tree (the corresponding screen dump is 1,1,0).

To help you interpret these binary files, we also provide you the “equivalent” text files (ops0-3.txt, tree0-3.txt, and invalidtree0-2.txt).

For an operations input file, each (int) key is printed with “%d” format. The key is followed by a space, and then the (char) operation (‘i’ for insertion and ‘d’ for deletion).

For a tree file, each (int) key is again printed with "%d" format. The key is followed by a space, and then char '0' (the node has no children), '1' (the node has a right child and it has no left child), '2' (the node has no right child and it has a left child), or '3' (the node has a right child and a left child).

Additional information

You may want to write a program that allows you to convert from `tree0.b` to `tree0.txt` and a program that allows you to convert from `ops0.b` to `ops0.txt` (in fact, the two programs can be combined into one). Similarly, you may want to write a program that converts from `tree0.txt` (or `ops0.txt`) to `tree0.b` (or `ops0.b`).

This way, you can easily create other examples (operations input files and tree files) to test your programs.

You should write the evaluation part first. This part allows you to test your construction part. (As a good practice, you think of how to evaluate/test your program before you write your program). There are two components of the evaluation (BST and height-balancedness), I recommend that you write the two components separately.

For the construction part, I suggest that you write the insertion routine (with rotations) first. However, try to write rotation operations as functions (that could be used by the deletion routine later on). Then, you write the deletion routine. It is easier to write the deletion as a recursive function because you may have to perform rotations all the way to the root node. A recursive routine makes it easier to keep track of all the nodes encountered in the process of searching for the deleted node. Otherwise, you should use a stack to keep track of the nodes encountered in the search process.

You should also write the deletion routine in such a way that the height-balancing part can be easily isolated such that if necessary, you could comment out the height-balancing part and the routine is simply doing a regular deletion of a key from a BST (without worrying about height-balancing). This could be helpful when it comes to debugging. Moreover, in the worst-case scenario, you can also comment out this part and still have a somewhat functioning deletion routine in your final submission should you encounter problems in maintaining a height-balanced tree.

As you write the program, use the evaluation part to test your insertion and deletion routines. Check out the Brightspace website for any updates to these instructions.