ECE 36800 Assignment #7

```
Original Due: 1:00 PM, Tuesday,  November 11
    Extended: 1:00 PM, Thursday, November 13
```

Description

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. 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. 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 counter-clockwise 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.

Tree node structure

The tree node structure (*Tnode*) is defined in the file *hbt.h*.

```
typedef struct _Tnode {
        int key: 29, balance: 3;
        struct _Tnode *left;
        struct _Tnode *right;
} Tnode;
```

The *left* and *right* fields store the left and right child nodes of a *Tnode*, respectively. An int, which has 32 bits, is used to store two bit fields: The bit field key uses 29 bits to store a signed integer value between *HBT_MIN* and *HBT_MAX* (inclusive), and the bit field balance uses 3 bits to store the difference in the height of the left sub-tree (*left*) and the height of the right sub-tree (*right*).

A 29-bit signed integer can be as large as $2^{28} - 1 = 268435455$ (= *HBT_MAX*) and as small as $-2^{28} = -268435456$ (= *HBT_MIN*). Both *HBT_MAX* and *HBT_MIN* are defined in hbt.h as follows:

```
#define HBT_MAX 268435455
#define HBT_MIN -268435456
```

A 3-bit (signed) field can be as large as 22 − 1 = 3 and as small as −4. Therefore, the bit field balance can store the balance of a height-balanced tree properly. Moreover, it can also store an (in)balance of 2 or −2 at a node when the tree becomes temporarily unbalanced. A correctly implemented height-balanced tree insertion or deletion routine should not allow the balance of a node to go below −2 or above 2.

You should include this file in your other .h and .c files that you will develop for this assignment. This file will be provided when we compile your submission. You do not have to submit this file. In your other .h and .c, you should not define other structures. No other user-defined structures are allowed in this assignment. If you have other user-defined structure(s) in your .h and/or .c files, your submission will not be graded.

Executable and Input/Output Format

There are two options that the executable a7 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

./a7 -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 (sizeof(int) + sizeof(char)) bytes, with the first sizeof(int) bytes being an int and the next sizeof(char) byte being a char.

The int is the key. The value stored in an int is guaranteed to be in the range of *HBT_MIN* and *HBT_MAX* (inclusive). In other words, the bit field key in a *Tnode* can always store the given int properly even though the bit field key has only 29 bits, whereas an int stored in a file has 32 bits.

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 × (sizeof(int)+sizeof(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. We cannot just write into a file 29 bits stored in the bit field key. So, we store the 29-bit key as an int in the output file.

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*.

All other more significant bits in the char should be 0. Therefore, a numerical value of 2 or 3 (in decimal) for the binary pattern stored in the char means there is a left child. A numerical value of 1 or 3 (in decimal) means that there is a right child. A numerical value of 0 means that there are no child nodes.

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 height-balanced 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

./a7 -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.

Grading

This assignment will be graded based on both the correctness and memory safety of your implementation. Each test case is worth 2 points for the build part and 1 point for the evaluation part and will be graded on an all-or-nothing basis (i.e., your output must exactly match the expected output). Even minor differences in formatting may cause your solution to fail a test case.

Submissions that do not terminate within a 10-minute time limit will receive a score of zero. You may submit to Gradescope as many times as you'd like before the deadline. Only your active submission (by default, the most recent one) will be graded. While the autograder score is generally a good indicator of your final grade, we reserve the right to add additional test cases after the submission deadline.

Submission Instructions

You must submit the following files to Gradescope under Assignment 7:

1. All source files and header files used in your implementation.
2. A Makefile that compiles your code to an executable named a7.