**Objectives**

- Understand how the AVL tree works
- Give you further practice with C and data structures

In this assignment, you will implement AVL tree and a set of functions associated with AVL tree. For simplicity, we make the following assumptions:

1. Each item of an AVL tree contains an integer key and an integer value.
2. No AVL tree contains duplicate items. Two items (k1, v1) and (k2, v2) are duplicates iff k1=k2 and v1=v2 hold.
3. An AVL tree may contains multiple items with the same key and the number of duplicate keys is a constant.

A template file named MyAVLTree.c is provided. MyAVLTree.c contains the type definitions of AVL tree and AVL tree node as well as some basic functions. You can add your own helper functions and auxiliary data structures for better performance in terms of time complexity.

You need to implement the following functions:

1. AVLTree *CreateAVLTree(const char *filename). This function creates an AVL tree by reading all the items from a text file or from the standard input (keyboard) depending on the argument filename. If filename is "stdin", this function will read all the items from the standard input. Otherwise, it will read all the items from a text file with filename as its full path name. (**2 marks**)

   An input text file contains zero or more items where each item is of the form (key, value). Any characters such as white space between two adjacent items are ignored. For example, the following sample file contains 10 items:

   (2, 50) (4, 30) (9, 30) (10, 400) (-5, -40)
   (7, 20) (19, 200) (20, 50) (-18, -200) (-2, 29)

   Similarly, when reading from the standard input, each input line may have zero or more items, separated by one or more white space characters. An empty line indicates the end of input.

   In case of an error in the input, this function will print the error and your program terminates.

You may assume that the input does not contain duplicate items and thus this function does not need to check for duplicate items.

The time complexity of this function cannot be higher than O(n logn), where n is the size of the resulting AVL tree. If your time complexity is higher, you will get 0 mark for this function. You may assume that each call to a C built-in function takes O(1) time.

2. AVLTree *CloneAVLTree(AVLTree *T). This function creates an identical copy (clone) of the input AVL tree T, and returns a pointer to the clone tree. (**1 mark**)

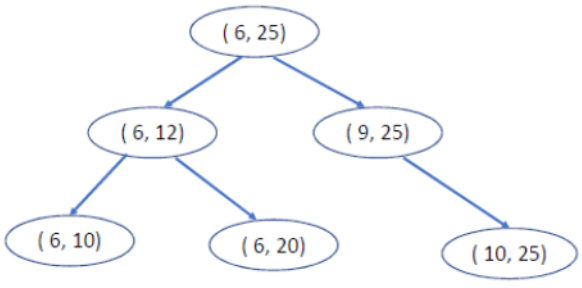
   The time complexity of this function cannot be higher than O(n), where n is the size of T. If your time complexity is higher, you will get 0 mark for this function.

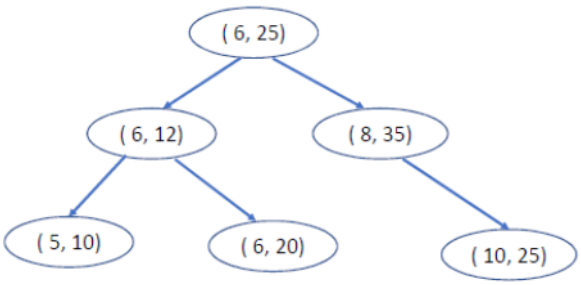
3. AVLTree *AVLTreesUnion(AVLTree *T1, AVLTree *T2). This function computes the union tree of two AVL trees T1 and T2 and returns a pointer to the union tree. The union tree of two AVL trees T1 and T2 is an AVL tree that contains all the items of both T1 and T2 without duplicate items. Assume that neither T1 nor T2 contains duplicate items. Note that this function does not make any change to T1 and T2. (**2 marks**)

   The time complexity of this function cannot be higher than O((m+n)log(m+n)), where m and n are the sizes of T1 and T2, respectively. If your time complexity is higher, you will get 0 mark for this function.

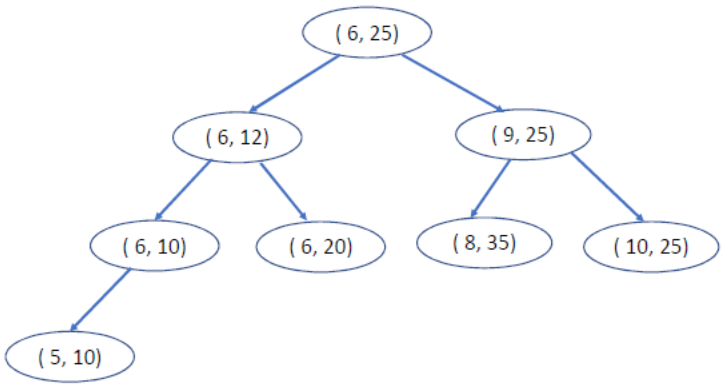
   **Bonus marks:** A correct tree union function with the time complexity O(m+n) will be awarded **1 bonus mark**, where m and n are the sizes of T1 and T2, respectively.

   An example: consider the following two AVL trees T1 and T2:



The union tree of T1 and T2 is shown as follows:



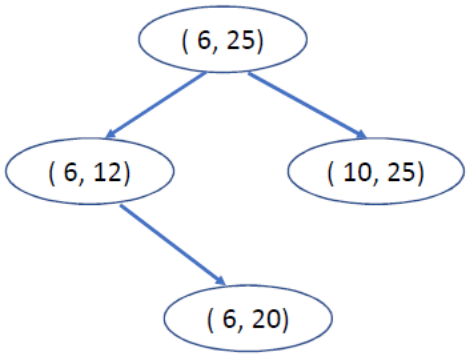

Note that in general the union tree may not be unique with respect to shape (structure) depending on how it is constructed.

4. AVLTree *AVLTreesIntersection(AVLTree *T1, AVLTree *T2). This function computes the intersection tree of two AVL trees T1 and T2 and returns a pointer to the intersection tree. The intersection tree of two AVL trees T1 and T2 is an AVL tree that contains all the items that appear in both T1 and T2. Assume that neither T1 nor T2 contains duplicate items. Note that this function does not make any change to T1 and T2. (**2 marks**)

   The time complexity of this function cannot be higher than O(m+n+k log k), where m and n are the sizes of T1 and T2, respectively, and k the size of the intersection tree. If your time complexity is higher, you will get 0 mark for this function.

   **Bonus marks:** A correct tree intersection function with the time complexity O(m+n) will be awarded **1 bonus mark**, where m and n are the sizes of T1 and T2, respectively.

   An example: consider the previous two AVL trees T1 and T2. The intersection tree is shown as follows:




   Note that in general the intersection tree may not be unique with respect to shape (structure) depending on how it is constructed.

5. int InsertNode(AVLTree *T, int k, int v). If the item (k, v) exists in the tree, this function simply returns 0 without adding the new item (k, v) to the tree. Otherwise, it inserts the new item (k, v) into the AVL tree T, increases the tree size by one and returns 1. (**0.5 mark**)

   The time complexity of this function cannot be higher than O(log n), where n is the size of T. If your time complexity is higher, you will get 0 mark for this function.

6. int DeleteNode(AVLTree *T, int k, int v). If the item (k, v) exists in the AVL tree T, this function deletes the node containing this item, decreases the tree size by one and returns 1. Otherwise, it returns 0 only. (**1 mark**)

   The time complexity of this function cannot be higher than O(log n), where n is the size of T. If your time complexity is higher, you will get 0 mark for this function.

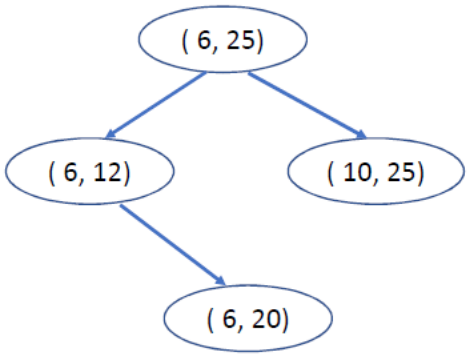
7. AVLTreeNode *Search(AVLTree *T, int k, int v). This function search for the item (k, v) in the AVL tree T. If the item is found, it returns a pointer to the node containing the item. Otherwise, it returns NULL. (**0.5 mark**)

   The time complexity of this function cannot be higher than O(log n), where n is the size of T. If your time complexity is higher, you will get 0 mark for this function.

8. void FreeAVLTree(AVLTree *T). This function frees up the heap space occupied by the AVL tree T. (**0.5 mark**)

   The time complexity of this function cannot be higher than O(n), where n is the size of T. If your time complexity is higher, you will get 0 mark for this function. You may assume that each call to free() takes O(1) time.

9. void PrintAVLTree(AVLTree *T). This function prints all the items and their heights stored in the AVL tree T sorted in non-decreasing order of keys on the standard output (screen). Each item is denoted by (key, value) with one item per line. For example, consider the following AVL tree:




   The output of PrintAVLTree is:

   (6, 12), 1
   (6, 20), 0
   (6, 25), 2
   (10, 25), 0

Your output can be different as long as it makes sense.

The time complexity of this function cannot be higher than O(n), where n is the size of T. If your time complexity is higher, you will get 0 mark for this function. You may assume that each call to a built-in C function takes O(1) time. (**0.5 mark**)

For each function, analyze its time complexity, and put the time complexity analysis as comments before the function. For the time complexity of each function, you just need to give the time complexity of major components (loops) and the total time complexity. You may assume that each call to a built-in C function takes constant (O(1)) time.