Project #03: AVL Trees

Complete By: Saturday, October 19th @ 11:59pm
Assignment: AVL class
Policy: Individual work only, late work *is* accepted (see "Policy" section for more details)
Submission: via Gradescope --- limited to 12 submissions

Assignment

For the last couple weeks we have been building an AVL class --- it's time to put all the pieces together. The assignment is to create a templated **avltree<TKey, TValue>** class in the file "avl.h" which properly balances trees as part of the insert function. When completed, **search** and **insert** must be O(lgN) operations; **size** and **height** must be O(1) operations. Recall that we began modifying insert in HW #07 to compute the heights; in HW #08 you were asked to implement the left and right rotate functions. Your test cases were created as part of lab week #07.

The major change to make now is to modify **insert** so that as you walk up the tree updating heights, you also check to see if the AVL condition is broken. If so then rotate to fix --- either one rotation for AVL cases 1 and 4, or 2 rotations for AVL cases 2 and 3. Recall that in class (day 20) we discussed a **_RotateToFix** function that insert could call to fix the tree; this function in turn calls _LeftRotate and _RightRotate.

You will need to implement the following public functions in your avltree class, exactly as shown here. You cannot add nor remove parameters, or change types:

```
template<typename TKey, typename TValue>
class avltree
{
private:
  .
  .
  .

public:
  avltree()
  avltree(avltree& other)

  virtual ~avltree()          // destructor:
  int size()
  int height()

  void clear()                // clear:

  TValue* search(TKey key)
  void insert(TKey key, TValue value)

  int distance(TKey k1, TKey k2)    // distance:

  void                 inorder()
  std::vector<TKey>    inorder_keys()
  std::vector<TValue>  inorder_values()
  std::vector<int>     inorder_heights()

};
```

Note that three functions have been added for the purposes of project #03: a **destructor** to free memory, a **clear** function to empty a given tree, and a **distance** function that returns the path length between 2 keys. Your avltree class must now free all allocated memory; more details in the next section.

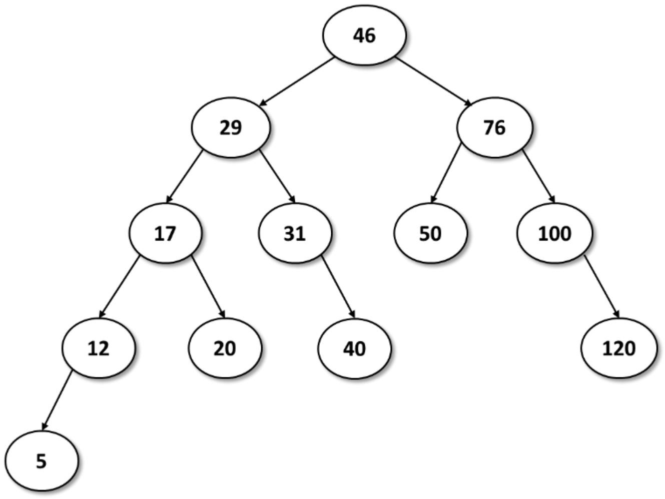
Assignment details

Your avlclass must now contain a **destructor** that properly frees all memory allocated by the avltree class. On submission we'll check using **valgrind**, so you should do the same as part of your testing; recall that destructors and valgrind were introduced in lab during week 05.

Your avlclass must also contain a **clear** function that empties the tree of all nodes, properly freeing the memory. Note that you *cannot* call the destructor to do this; destructors in C++ are special functions that should only be called by the compiler. Instead, what you should do is create a private helper function that both the destructor and clear call. When clear returns, the root should be nullptr and the size 0.

Finally, add a function **distance(k1, k2)** that returns the "distance" between the keys k1 and k2. The distance is defined as the length of the minimum path between k1 and k2. For example, consider the following tree ---------> The distance(5, 100) = 6. The distance from 31 to 12 is 3. And the distance from 46 to 12 = 3. The reverse are also true, e.g. the distance(100, 5) = 6.

If k1 or k2 is not in the tree, then the distance is defined as -1. If k1 = k2, then the distance is 0.



Programming Environment

You are free to program in any environment you want; final submissions will be collected using Gradescope. If you want to use Codio, a project has been created named "**cs251-project03-avl**". The environment will contain catch and valgrind for testing purposes.

In Codio, use "**make catch**" to build your Catch-based "test.cpp" program. Then "**make run**" to run. If you want to run your program with valgrind to check for memory leaks / pointer errors, do "**make valgrind**".

Requirements

Your avltree class must be templated, and follow the approach we have been discussing in class (and building in the HW and Labs). Rebalancing must occur during insert, both search and insert must be O(lgN), and size and height must be O(1). This implies your private NODE structure must contain a height field that is properly maintained by the insert and the rotate functions.

Additional program requirements:

1. *No other data structures may be used to implement the AVL tree itself --- i.e. build your tree using a Struct NODE with Key, Value, Height, Left and Right pointers. You can add a Parent pointer if you want, but nothing else should be needed nor added. You can use a stack to walk up the tree and update the heights, etc., but the tree itself must be built using NODEs and pointers.*

2. *Use std::vector to capture and return the data needed by the* ***inorder*** *testing functions. Likewise, if you need an additional data structure to implement your distance function, by all means use one.*

Program submission

Submit your final avl class file --- avl.h" --- to Gradescope for grading under **Project 03 - AVL**. Your work will receive a tentative grade for correctness, and then will undergo manual review for commenting, readability, approach, and adherence to requirements. Note that you'll be limited to a total of 12 submissions; your tentative score will be the highest value earned from your submissions.

Commenting/readability/approach will count for 10% of project score. Adherence to requirements may count anywhere from 10% to 100%.

Policy

Late work *is* accepted. You may submit as late as 24 hours after the deadline for a penalty of 10%. After 24 hours, no submissions will be accepted.

All work submitted for grading *must* be done individually. While we encourage you to talk to your peers and learn from them (e.g. your "iClicker teammates"), this interaction must be superficial with regards to all work submitted for grading. This means you *cannot* work in teams, you cannot work side-by-side, you cannot submit someone else's work (partial or complete) as your own. The University's policy is available here:

https://dos.uic.edu/conductforstudents.shtml .

In particular, note that you are guilty of academic dishonesty if you extend or receive any kind of unauthorized assistance. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing your program to another student, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. It is also considered academic dishonesty if you click someone else's iClicker with the intent of answering for that student, whether for a quiz, exam, or class participation. Academic dishonesty is unacceptable, and penalties range from a letter grade drop to expulsion from the university; cases are handled via the official student conduct process described at https://dos.uic.edu/conductforstudents.shtml .