Project #05: Threaded AVL: avlt<KeyT, ValueT>

Complete By:
Bonus: Thursday 3/12 @ 11:59pm (+10%)
On-time: Saturday, 3/14 @ 11:59pm
Late: Sunday, 3/15 @ 11:59pm (-10%)

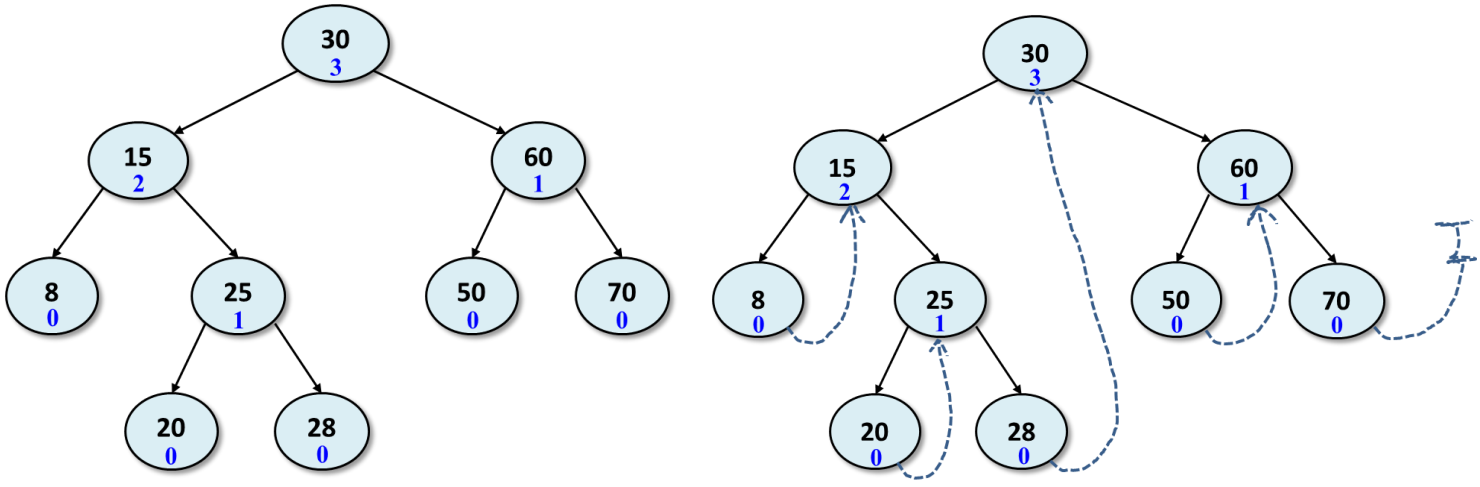
Assignment: "avlt.h" file

Policy: Individual work only, late work *is* accepted

Submission: "avlt.h" file via Gradescope; the first 12 submissions are free, each additional submission costs 1 point

Overview

As we've discussed in class, AVL trees guarantee that search and insert have worst-case O(lgN) time complexity. By threading the tree, we guarantee that an inorder traversal can be performed in O(N) time using O(1) space. The goal of this exercise is to put the two concepts together to form a threaded AVL tree class, **avlt<keyT, ValueT>**. On the left below is an AVL tree (the heights are shown within the node in blue); on the right is the threaded equivalent.



Assignment

The assignment is to implement a **threaded AVL tree** class **avlt<KeyT, ValueT>** in the file "avlt.h". The class should adhere to AVL balancing properties, storing (key, value) pairs in a balanced, threaded manner. Here's the start of the class definition:

```
template<typename KeyT, typename ValueT>
class avlt
{
private:
  struct NODE
  {
     KeyT    Key;
     ValueT  Value;
     NODE*   Left;
     NODE*   Right;
     bool    isThreaded;  // true => Right is a thread, false => non-threaded
     int     Height;      // height of tree rooted at this node
  };
```

You are required to use this NODE definition, since (a) AVL trees require a **Height** value in every node for efficiency reasons, and (b) an **isThreaded** field is needed to determine when the Right pointer is an actual pointer vs. a thread.

The vast majority of the necessary AVL class has already been implemented in the previous project #04, where you built a threaded binary search tree class bstt<KeyT, ValueT>. This project #04 code can be reused; if you didn't complete project #04, a solution will be provided for you to build upon. In this assignment you'll need to extend this earlier work in the following ways:

1. The **insert()** function will need to be modified to update the heights and rebalance the tree as necessary. Changes to insert(), AVL rotations, and AVL rotation cases have been discussed in class: lectures 18, 19 and 20 are particularly relevant (lectures are on course dropbox). Also relevant is HW #10, as well as the earlier HWs #8 and #9.
2. The left and right rotation functions will need to maintain threads as part of the rotation.
3. If necessary, modify the **copy constructor** to make an exact copy of the tree (likewise for **operator=**). If your copy constructor is calling insert() and then recursively traversing left and right, this will cause a different tree to be built (with lots of rotations) as it recursively traverses down the left-side of the tree --- and inserts N/2 values that are all < then the root. Figure out how to make an exact, threaded copy with *no* rotations.
4. Modify the **dump()** function to output the height as well; see "avlt.h" for more details, and an example at the end of the next section.
5. The addition of a **height()** function that returns the height of the tree by retrieving the height from the root node; O(1) time complexity. This function is already provided in "avlt.h".
6. The addition of **operator%()** function that returns the height stored in a given node; see "avlt.h" for more details.
7. The addition of a **range_search()** function; see "avlt.h" for more details. Pay attention to the

time complexity requirements.

As before, keep in mind that when traversing a threaded tree in a normal top-down fashion, a threaded pointer is equivalent to nullptr.

avlt<KeyT, ValueT>

For completeness, here's the "avlt.h" file that you are required to implement. Electronic copies of this file are available via Codio, and the course dropbox. You are required to use the **NODE** struct as given.

```
/*avlt.h*/

//
// Threaded AVL tree
//

#pragma once

#include <iostream>
#include <vector>

using namespace std;

template<typename KeyT, typename ValueT>
class avlt
{
private:
  struct NODE
  {
    KeyT   Key;
    ValueT Value;
    NODE*  Left;
    NODE*  Right;
    bool   isThreaded; // true => Right is a thread, false => non-threaded
    int    Height;     // height of tree rooted at this node
  };

  NODE* Root;  // pointer to root node of tree (nullptr if empty)
  int   Size;  // # of nodes in the tree (0 if empty)


public:
  //
  // default constructor:
  //
  // Creates an empty tree.
  //
  avlt()
  {
    Root = nullptr;
```

```
    Size = 0;
  }

  //
  // copy constructor
  //
  // NOTE: makes an exact copy of the "other" tree, such that making the
  // copy requires no rotations.
  //
  avlt (const avlt& other)
  {
    //
    // TODO
    //
  }

  //
  // destructor:
  //
  // Called automatically by system when tree is about to be destroyed;
  // this is our last chance to free any resources / memory used by
  // this tree.
  //
  virtual ~avlt()
  {
    //
    // TODO
    //
  }

  //
  // operator=
  //
  // Clears "this" tree and then makes a copy of the "other" tree.
  //
  // NOTE: makes an exact copy of the "other" tree, such that making the
  // copy requires no rotations.
  //
  avlt& operator=(const avlt& other)
  {
    //
    // TODO:
    //

    return *this;
  }

  //
  // clear:
  //
  // Clears the contents of the tree, resetting the tree to empty.
  //
  void clear()
  {
```

```
    //
    // TODO
    //
  }

  //
  // size:
  //
  // Returns the # of nodes in the tree, 0 if empty.
  //
  // Time complexity:  O(1)
  //
  int size() const
  {
    return Size;
  }

  //
  // height:
  //
  // Returns the height of the tree, -1 if empty.
  //
  // Time complexity:  O(1)
  //
  int height() const
  {
    if (Root == nullptr)
      return -1;
    else
      return Root->Height;
  }

  //
  // search:
  //
  // Searches the tree for the given key, returning true if found
  // and false if not.  If the key is found, the corresponding value
  // is returned via the reference parameter.
  //
  // Time complexity:  O(lgN) worst-case
  //
  bool search(KeyT key, ValueT& value) const
  {
    //
    // TODO
    //

    return false;
  }

  //
  // range_search
  //
  // Searches the tree for all keys in the range [lower..upper], inclusive.
```

```
// It is assumed that lower <= upper.  The keys are returned in a vector;
// if no keys are found, then the returned vector is empty.
//
// Time complexity: O(lgN + M), where M is the # of keys in the range
// [lower..upper], inclusive.
//
// NOTE: do not simply traverse the entire tree and select the keys
// that fall within the range.  That would be O(N), and thus invalid.
// Be smarter, you have the technology.
//
vector<KeyT> range_search(KeyT lower, KeyT upper)
{
  vector<KeyT>  keys;

  //
  // TODO
  //

  return keys;
}

//
// insert
//
// Inserts the given key into the tree; if the key has already been insert then
// the function returns without changing the tree.  Rotations are performed
// as necessary to keep the tree balanced according to AVL definition.
//
// Time complexity:  O(lgN) worst-case
//
void insert(KeyT key, ValueT value)
{
  //
  // TODO
  //
}

//
// []
//
// Returns the value for the given key; if the key is not found,
// the default value ValueT{} is returned.
//
// Time complexity:  O(lgN) worst-case
//
ValueT operator[](KeyT key) const
{
  //
  // TODO
  //

  return ValueT{ };
}
```

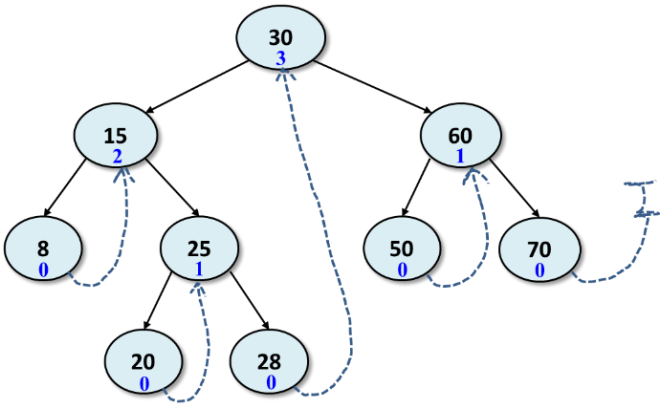
```
//
// ()
//
// Finds the key in the tree, and returns the key to the "right".
// If the right is threaded, this will be the next inorder key.
// if the right is not threaded, it will be the key of whatever
// node is immediately to the right.
//
// If no such key exists, or there is no key to the "right", the
// default key value KeyT{} is returned.
//
// Time complexity:  O(lgN) worst-case
//
KeyT operator()(KeyT key) const
{
  //
  // TODO
  //

  return KeyT{ };
}

//
// %
//
// Returns the height stored in the node that contains key; if key is
// not found, -1 is returned.
//
// Example:  cout << tree%12345 << endl;
//
// Time complexity:  O(lgN) worst-case
//
int operator%(KeyT key) const
{
  //
  // TODO
  //

  return -1;
}

//
// begin
//
// Resets internal state for an inorder traversal.  After the
// call to begin(), the internal state denotes the first inorder
// key; this ensure that first call to next() function returns
// the first inorder key.
//
// Space complexity: O(1)
// Time complexity:  O(lgN) worst-case
//
// Example usage:
//    tree.begin();
```

```
//    while (tree.next(key))
//      cout << key << endl;
//
void begin()
{
  //
  // TODO
  //
}

//
// next
//
// Uses the internal state to return the next inorder key, and
// then advances the internal state in anticipation of future
// calls.  If a key is in fact returned (via the reference
// parameter), true is also returned.
//
// False is returned when the internal state has reached null,
// meaning no more keys are available.  This is the end of the
// inorder traversal.
//
// Space complexity: O(1)
// Time complexity:  O(lgN) worst-case
//
// Example usage:
//    tree.begin();
//    while (tree.next(key))
//      cout << key << endl;
//
bool next(KeyT& key)
{
  //
  // TODO
  //

  return false;
}

//
// dump
//
// Dumps the contents of the tree to the output stream, using a
// recursive inorder traversal.
//
void dump(ostream& output) const
{
  output << "**************************************************" << endl;
  output << "********************* AVLT ***********************" << endl;

  output << "** size: " << this->size() << endl;
  output << "** height: " << this->height() << endl;

  //
```

```
    // inorder traversal, with one output per line: either
    // (key,value,height) or (key,value,height,THREAD)
    //
    // (key,value,height) if the node is not threaded OR thread==nullptr
    // (key,value,height,THREAD) if the node is threaded and THREAD denotes the next
inorder key
    //

    //
    // TODO
    //

    output << "************************************************" << endl;
  }

};
```

Here's an example of the output from dump, for the following tree (assume key and value are the same):

```
************************************************
******************** AVLT **********************
** size: 9
** height: 3
(8,8,0,15)
(15,15,2)
(20,20,0,25)
(25,25,1)
(28,28,0,30)
(30,30,3)
(50,50,0,60)
(60,60,1)
(70,70,0)
************************************************
```



Requirements

1. The AVL class is implemented following the definition of an AVL tree. Do not balance the tree using another approach; you are required to follow exactly the definition of AVL trees and their balancing properties as discussed in class.
2. You must implement the threaded AVL tree class using the provided **NODE** struct. No additional data members are allowed as part of the NODE struct.
3. You are required to free all memory allocated by your class. Valgrind will be used to confirm that memory has been properly freed.
4. The public functions have time and space complexity requirements, e.g. O(1) or O(lgN). Failure to meet these requirements will score the function as a 0, even if it passes the test cases.
5. You are free to copy-paste your solution to project #04 as the foundation for project #05; you are also free to use our provided solution. You may not use the work of another student.

Grading, electronic submission, and Gradescope

Submission: "avlt.h".

Your score on this project is based on two factors: (1) correctness of avlt.h" as determined by Gradescope, and (2) manual review of "avlt.h" for commenting, style, and approach (e.g. adherence to time and space complexity requirements). The entire project is worth 150 points: 100 points for correctness, and 50 points for commenting, style, and approach. In this class we expect all submissions to compile, run, and pass at least some of the test cases; do not expect partial credit with regards to correctness. Example: if you submit to Gradescope and your score is a reported as a 0, then that's your correctness score. The only way to raise your correctness score is to re-submit. [**Update**: 20/50 points are now based on the test cases submitted as part of lab week 08. See lab handout for more details.]

In this project, your "avlt.h" will be allowed **12 free submissions**. After 12 submissions, each additional submission will cost 1 point. Example: suppose you score 100 after 15 submissions, and you activate this submission for grading. Your autograder score will be 97: 100 – 3 extra submissions. Note that you cannot use another student's account to test your work; this is considered academic misconduct because you have given your code to another student for submission on their account.

Note that the TAs will also review for adherence to requirements; breaking a requirement can result in a final score of 0 out of 150. We take all requirements seriously.

By default, we grade your **last** submission. Gradescope keeps a complete submission history, so you can **activate** an earlier submission if you want us to grade a different one; this must be done before the due date. We assume *every* submission on your Gradescope account is your own work; do not submit someone else's work for any reason, otherwise it will be considered academic misconduct.

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 .