UIC COMPUTER SCIENCE

CS 251 : Data Structures
Spring 2020

Project #04: Threaded BST: bstt<KeyT, ValueT>

Complete By: **Bonus:** Saturday 2/29 @ 11:59pm (+10%)
On-time: Monday, 3/2 @ 11:59pm
Late: Tuesday, 3/3 @ 11:59pm (-10%)

Assignment: "bstt.h" file

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

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

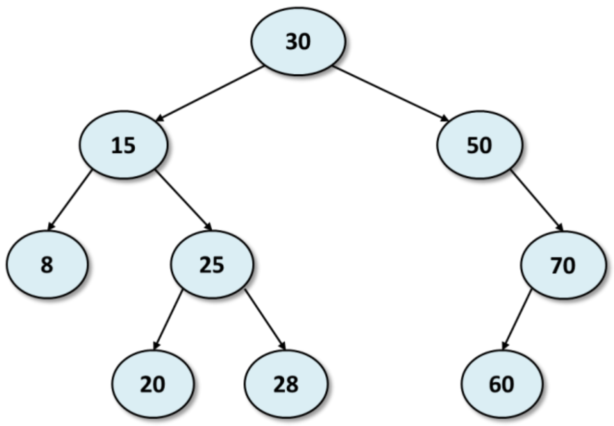
Background

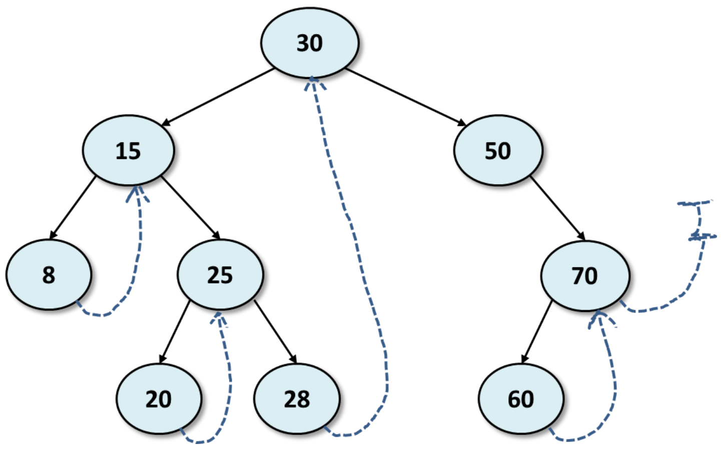
As you might recall, the **map** abstraction in C++ offers the ability to iterate through the (key, value) pairs in key order. For example, we used this in project #03 to output the contents of a symbol table scope:

```
for (auto& pair : scope.Symbols)
{
   output << pair.first << ": " << pair.second << endl;
}
```

Since map is based on the concept of a search tree --- where recursion is typically used --- how was this done?

A common answer is the idea of a **threaded binary search tree**. A threaded BST takes advantage of the observation that half the pointers in a tree are nullptr; that's a lot of wasted space. Instead, we re-use the **right** pointers as follows: if that pointer is nullptr, we re-use as a **thread** to the next inorder key; see the dashed links below. Threads make it possible to traverse a tree in key order without recursion or a stack.





Assignment

The assignment is to implement a **threaded binary search tree** class<KeyT, ValueT> in the file "bstt.h" capable of storing (key, value) pairs in a threaded manner. Since the goal is to save space, you are required to implement the tree such that the Right pointer is re-used as the thread when nullptr (vs. adding a third pointer field). How do you know when the Right pointer is being re-used? A boolean "**isThreaded**" field is added to every node:

```
template<typename KeyT, typename ValueT>
class bstt
{
private:
  struct NODE
  {
     KeyT    Key;
     ValueT  Value;
     NODE*   Left;
     NODE*   Right;
     bool    isThreaded;  // true => Right is a thread, false => non-threaded
  };
```

The "isThreaded" field will be set to true when the Right pointer is being re-used as a thread, and false otherwise.

One of the side-effects of a threaded tree is that traversing the tree is now different. For example, during search, in a non-threaded tree we traverse left or right in the typical manner:

```
if (key < cur->Key)
  cur = cur->Left;
else
  cur = cur->Right;
```

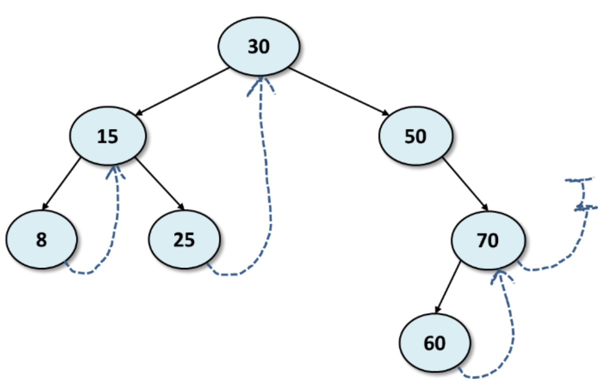
This no longer works, because the Right pointer could be threaded. In a threaded tree, traversal is now performed as follows:

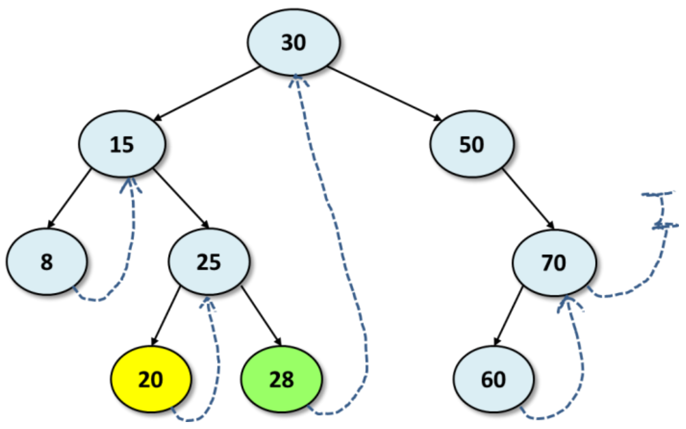
```
if (key < cur->Key)
  cur = cur->Left;
else
{
  if (cur->isThreaded) // there is no Right pointer in traditional traversal:
    cur = nullptr;
  else
    cur = cur->Right;
}
```

When traversing a tree in a normal top-down fashion, a threaded pointer is equivalent to nullptr.

How are threads added to the tree? Threads are added during insertion. The algorithm is simple: suppose a new node N is being inserted. If N is added to the left of its parent P, then N's thread points to P. If

N is added to the right of its parent P, then N inherits the thread of its parent. For example, consider the following threaded BST on the left. Focus on node 25, who's thread denotes 30:





When 20 is inserted to the left of 25, then 20's thread is set to 25 --- the next inorder key. When 28 is inserted to the right of 25, then 25's right pointer is no longer threaded --- it now denotes 28. Instead, 28 inherits 25's thread, so that 28's thread now denotes 30 --- the next inorder key.

When are the threads actually used? The threads are used when the tree is traversed in key order. In our case, 2 functions will be added to the bstt class to enable traversal: **begin()** and **next()**. These functions are used as follows:

```
bstt<int, int> tree;
int            key;

tree.insert(123, 456);
.
.
.

tree.begin();  // prepare for inorder traversal:

while (tree.next(key))  // for each key:
{
   cout << key << " ";
}
cout << endl;
```

More details in the next section.

bstt<KeyT, ValueT>

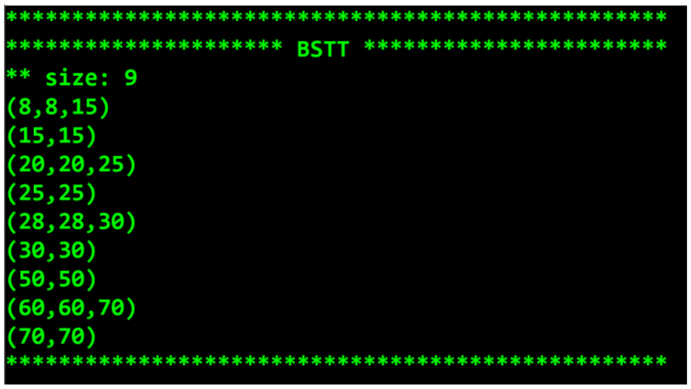
For completeness, here's the "bstt.h" file that you are required to implement. Electronic copies of this file are available via Codio, and the course dropbox. You are required to use the **NODE** struct as given.

```
/*bstt.h*/

//
// Threaded binary search tree
//

#pragma once

#include <iostream>

using namespace std;

template<typename KeyT, typename ValueT>
class bstt
{
private:
  struct NODE
  {
    KeyT   Key;
    ValueT Value;
    NODE*  Left;
    NODE*  Right;
    bool   isThreaded;
  };

  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.
  //
  bstt()
  {
    Root = nullptr;
    Size = 0;
  }

  //
  // copy constructor
  //
  bstt(const bstt& 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 ~bstt()
  {
    //
    // TODO
    //
  }

  //
  // operator=
  //
  // Clears "this" tree and then makes a copy of the "other" tree.
  //
  bstt& operator=(const bstt& 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;
  }

  //
  // 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) on average
  //
  bool search(KeyT key, ValueT& value) const
  {
    //
    // TODO
    //
  }

  //
  // insert
  //
  // Inserts the given key into the tree; if the key has already been insert then
  // the function returns without changing the tree.
  //
  // Time complexity:  O(lgN) on average
  //
  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) on average
  //
  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.
  //
```

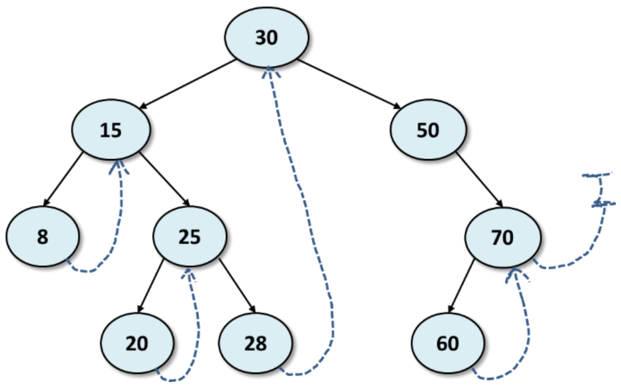
```
    // Time complexity:  O(lgN) on average
    //
    KeyT operator()(KeyT key) const
    {
      //
      // TODO
      //

      return KeyT{ };
    }

    //
    // 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) on average
    //
    // 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) on average
    //
    // 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.  Since dump is for debugging purposes, recursion is
  // used for the output so you check the correctness of threads.
  //
  void dump(ostream& output) const
  {
    output << "**************************************************" << endl;
    output << "********************* BSTT ***********************" << endl;

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

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



```
**************************************************
********************* BSTT ***********************
** size: 9
(8,8,15)
(15,15)
(20,20,25)
(25,25)
(28,28,30)
(30,30)
(50,50)
(60,60,70)
(70,70)
**************************************************
```



Requirements

1. As noted earlier, you must implement the threaded binary search tree class using the provided **NODE** struct. No additional data members are allowed as part of the NODE struct.
2. You are required to free all memory allocated by your class. Valgrind will be used to confirm that memory has been properly freed.
3. 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.

Grading, electronic submission, and Gradescope

Submission: "bstt.h".

Your score on this project is based on two factors: (1) correctness of "bstt.h" as determined by Gradescope, and (2) manual review of "bstt.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.

In this project, your "bstt.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 .