CS 211
Fundamentals of Computer Programming II
Winter 2024

Northwestern University

Project #08 (v1.4)

Assignment: set using threaded binary search trees
Submission: via Gradescope (4 submissions per 24-hr period)
Policy: individual work only, late work is accepted
Complete By: Friday March 8th @ 11:59pm CST
Late submissions: see syllabus for late policy… No submissions accepted after Sunday 03/10 @ 11:59pm

Pre-requisites: Lectures 16 and 17 (Canvas has lecture slides / recordings)

Overview

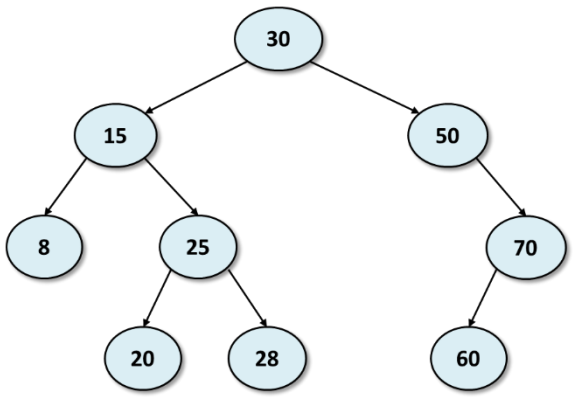
In class we have been discussing how **set** (and **map**) are implemented using search trees. The implementation of set is a good example of how (1) C and C++ work together to provide an efficient implementation, and (2) common C++ abstractions such as *[ ]* and *iterators* are implemented. Here in project 08 we are going to continue the implementation of set, building a "threaded" binary search tree so that *foreach*-based iteration can be supported in O(1) space and O(lgN) time.

Recall that C++ provides a "foreach" style loop for iterating through containers such as vector, map, and set. For example, the code below inserts 9 elements into the set, building the search tree shown on the right. The "foreach" loop then outputs the elements **in order**:

```
set<int> S;

S.insert(30);
S.insert(15);
S.insert(50);
S.insert(8);
S.insert(25);
S.insert(70);
S.insert(20);
S.insert(28);
S.insert(60);

cout << "set: ";
for (int x : S)  // foreach x in S:
  cout << x << " ";
cout << endl;
```




```
set: 8 15 20 25 28 30 50 60 70
```

How is this iteration done, especially since the tree is built with pointers going downward, and not in sorted order? Step 1 is to realize that C++ converts the foreach-style loop into an iterator-based while loop. The earlier example would be converted into the following:

```
for (auto x : S)  // foreach x in S:
  cout << x << " ";

auto iter = S.begin();

while (iter != S.end()) {
  int x = *iter;
  cout << x << " ";
  iter++;
}
```

```
set: 8 15 20 25 28 30 50 60 70
```

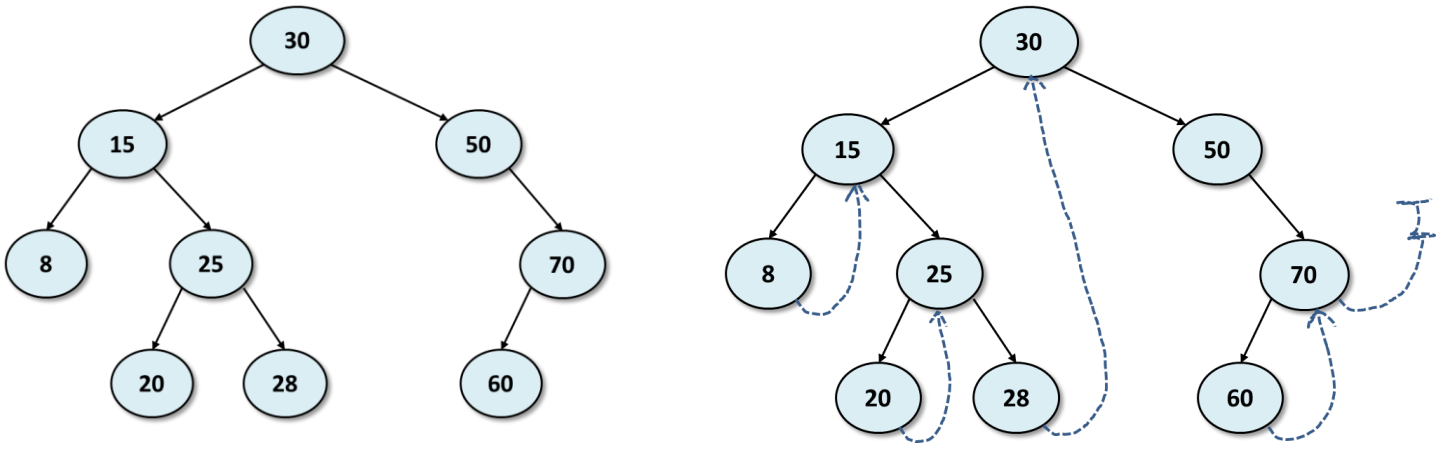
Iterators are used to safely access each element of the set, and the iterator overloads the **++** operator to enable moving to the next **inorder** element of the set.

As discussed in class (lecture 17), an iterator is an object that provides "safe" pointer access by overloading the ***** operator. But how does the iterator enable in order iteration through the tree? This is step 2: figuring out how to overload the **++** operator to provide in order traversal through the tree.

There are a variety of approaches. For example, the iterator object could store a private copy of the set in a vector, and then move an index through the vector. This would require O(N) additional space for the vector, and O(N) time to fill the vector. We can do better.

Another approach is to store "parent" pointers in the tree, i.e. every node stores a pointer back to its parent (see an diagram here). This allows traversals both down and up the tree. This approach requires O(N) additional space to store all the parent pointers. We can do better.

Our solution is to build a **threaded binary search tree**. A threaded BST takes advantage of the observation that half the pointers in a tree are nullptr (all the leaves at the bottom) --- that's O(N) wasted space. Instead, a threaded BST uses the **right** pointers as follows: if that pointer is nullptr, we re-use as a **thread** to the next inorder key. For example, on the left below is the original BST, and to the right is the equivalent threaded BST with the threads shown as dashed lines. Threads make it possible to traverse a tree in order using O(1) space and O(lgN) time:



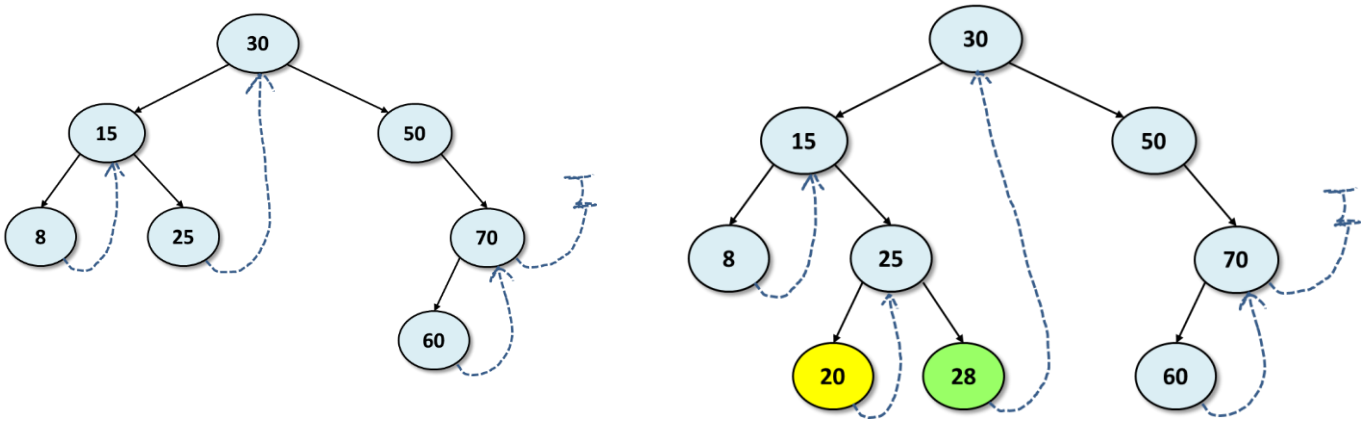
Building a threaded BST

How are threads added to the tree? First, the definition of a NODE is extended to include an additional **isThreaded** bit. This boolean value is set to **false** if the *Right* pointer is not a thread, and **true** if the *Right* pointer is being used as a thread:

```
//
// A node in the search tree:
//
class NODE {
private:
  TKey  Key;
  bool  isThreaded : 1; // 1 bit
  NODE* Left;
  NODE* Right;
```

Interestingly, this additional field typically takes no additional space based on how structures / classes are allocated in memory[1].

Threads are added during insertion. 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 P (and P is no longer threaded). 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 --- while 28 inherits 25's thread. As a result 28's thread now denotes 30, the next inorder key.

Traversing a threaded BST

One of the side-effects of a threaded tree is that traversing the tree is slightly different. When searching a non-threaded tree, we traverse left or right by simply accessing the Left or Right pointers:

[1] You can test this: output sizeof(NODE) with and without the additional field. It's the same size in the case of set<int>.

```
//
// traversing a non-threaded tree:
//
if (key < cur->Key)  // key is smaller, go left:
  cur = cur->Left;
else if (cur->Key < key)  // key is bigger, go right:
  cur = cur->Right;
else // key matches:
  ...
```

The above no longer works, because the **Right** pointer could be threaded. In a threaded tree, traversal must be performed as follows:

```
if (key < cur->Key)  // key is smaller, go left:
  cur = cur->Left;
else if (cur->Key < key)  // key is bigger, go right:
{
  if (cur->isThreaded)  // Right was originally nullptr before threading:
    cur = nullptr;
  else
    cur = cur->Right;
}
```

When traversing a tree in a normal top-down fashion, a threaded pointer is equivalent to nullptr. Since it's hard to remember this, we can use good object-oriented design to help us. In the NODE class, the data members will be private and access provided via traditional *getter* and *setter* methods. The **get_Right( )** getter is then be programmed to check the *isThreaded* value and return appropriately:

```
//
// A node in the search tree:
//
class NODE {
private:
  TKey  Key;
  bool  isThreaded : 1; // 1 bit
  NODE* Left;
  NODE* Right;

public:
  // constructor:
  NODE(TKey key)
    : Key(key), isThreaded(false), Left(nullptr), Right(nullptr) { }

  // getters:
  TKey  get_Key()        { return this->Key; }
  bool  get_isThreaded() { return this->isThreaded; }
  NODE* get_Left()       { return this->Left; }

  // NOTE: this ignores the thread, call to perform "normal" traversals
  NODE* get_Right() {
    if (this->isThreaded)
      return nullptr;
    else
      return this->Right;
  }
```

Now we can safely traverse the tree using the getter methods, and essentially forget about the threads:

```
//
// safely traverse a threaded tree:
//
if (key < cur->get_Key())  // key is smaller, go left:
  cur = cur->get_Left();
else if (cur->get_Key() < key)  // key is bigger, go right:
  cur = cur->get_Right();
else  // key matches:
  ...
```

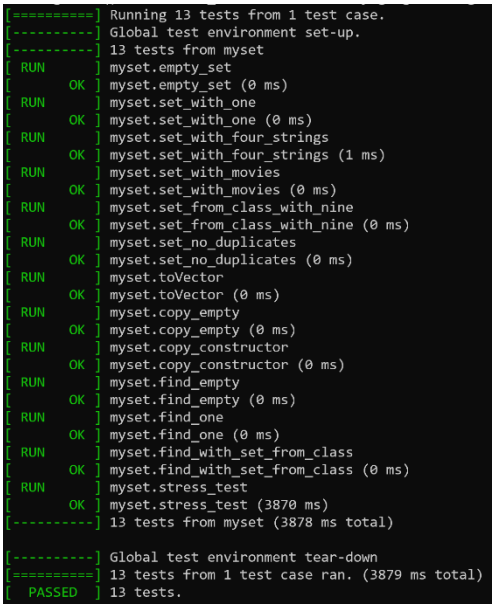
We strongly encourage you to add additional getters / setters as needed.

Getting Started

We will continue working on the EECS computers. To get started, setup your project08 folder and copy the provided files (these files can also be found on dropbox). The provided code is very similar to the "set.h" developed in class:

1. Open a terminal window in VS Code. Or, if you are working on a Mac or Linux, open a terminal; if you are working on Windows, you can open Git Bash.
2. Login to moore: **`ssh moore`** or **`ssh YOUR_NETID@moore.wot.eecs.northwestern.edu`**
3. Make a directory for project 08 **`mkdir project08`**
4. Make this directory private **`chmod 700 project08`**
5. Move ("change") into this directory **`cd project08`**
6. Copy the files needed for project 08: **`cp -r /home/cs211/w2024/project08/release .`**
7. Change into release directory: **`cd release`**
8. List the contents of the directory **`ls`**
9. Build the program **`make`**
10. Run **`./a.out`**

Since **set** has a well-defined API, we are using unit tests and the *googletest* framework for testing. The provided code has 13 tests to help you get started.



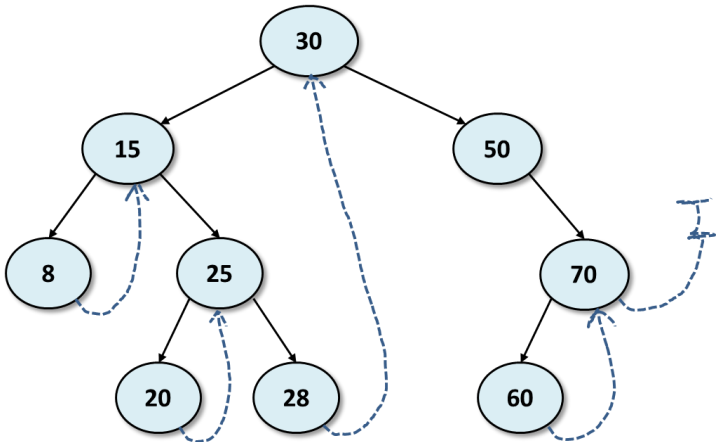
Assignment details

The provided "set.h" implements most of the functionality of a set. What's missing is the ability to foreach through the set in order, which requires a threaded tree. So that's where we start:

1. Modify the set's *insert( )* method to add threads.

2. How do we test to make sure the threads were added correctly? Since the threads are essentially a hidden implementation detail, we need a way to surface them. The set class contains a *toVector( )* method that returns the elements of a set in order. Study how this method is implemented…

   Add a similar *toPairs( )* method that returns a vector of pairs: the first element in the pair is the set element, and the second element is the one obtained by following the thread (if threaded). For example, suppose we have this threaded tree:



   A call to **toPairs(-1)** returns the following pairs (the -1 is used when a node is not threaded and no second element is available):

```
(8,15), (15,-1), (20,25), (25,-1), (28,30), (30,-1), (50,-1), (60,70), (70,-1)
```

   Since nodes 15, 25, 30 and 50 are not threaded, the second element in their pair is returned as -1. Here's the declaration of the toPairs( ) method:

```
//
// toPairs
//
// Returns pairs of elements: <element, threaded element>.
// If a node is not threaded: <element, no_element value>.
//
std::vector<std::pair<TKey, TKey>> toPairs(TKey no_element)
{
   // TODO: see toVector()
}
```

3. Add unit tests to test toPairs( ). Test with an empty set, test with a set containing 1 element, test with a set containing multiple elements, test with a set containing non-numeric values.

4. Add foreach-based iteration support. As discussed earlier, recall that a foreach loop is converted to an iterator-based while loop. This implies you need to add the following to the set class, while adhering to the requirement of O(1) additional space (this was discussed on page 1):

   a. Add a **begin( )** method to set, returning an iterator denoting the first inorder element in the set. If the set is empty, begin( ) should return the same result as end( ). Your solution must be O(lgN) time and O(1) space (e.g. you cannot call toVector nor toPairs).

   b. Overload the **!=** operator in the iterator class:

```
//
// !=
//
// Returns true if the given iterator is not equal to
// this iterator.
//
bool operator!=(iterator other)
{ … }
```

   c. Overload the **++** operator in the iterator class, which advances to the next inorder element in the set (or nullptr at the end of the iteration). Take advantage of the threads in the tree to enable this, but you'll need to think about what to do when the current node is not threaded --- which node contains the next inorder key? [Your solution must be O(lgN) time, and O(1) space, e.g. you cannot call toVector nor toPairs.]

```
//
// ++
//
// Advances the iterator to the next ordered element of
// the set; if the iterator cannot be advanced, ++ has
// no effect.
//
void operator++()
{ … }
```

5. Add unit tests with foreach loops. Test with an empty set, test with a set containing 1 element, test with a set containing multiple elements, test with a set containing non-numeric values. Stress test with a large set.

6. Uncomment the destructor, and run valgrind to test for memory errors / leaks. Use "make valgrind".

7. That's it, congratulations! Not just in completing this project, but working through all 8 projects this quarter. Well done.

Grading and Electronic Submission

You will submit your set.h and tests.cpp files to Gradescope for evaluation. To submit from the EECS computers, run the following command (you must run this command from inside your **project08/release** directory):

```
/home/cs211/w2024/tools/project08  submit  set.h  tests.cpp
```

or just type

```
make submit
```

Gradescope submissions will open 2-3 days before the due date; you'll have 4 submissions per 24-hr period.

By default, Gradescope records the score of your **LAST SUBMISSION** (not your highest submission score). If you want us to grade a different submission, activate a different submission via your "Submission History". This must be done before the due date.

The autograding portion of the project will be worth 80/100 points. The remaining 20 points will be determined manually by confirming that you have written 10 or more reasonable test cases to test your threading and foreach-based iteration support. We provided 13 test cases, do not modify those tests. Write at least 10 additional test cases that test the features you have added: toPairs(), foreach, etc. You will earn 2 points for each reasonable test case you supply. This is week 10, the staff will not comment on the quality of your test cases --- you should know the difference between a reasonable test case and one that is just a copy-paste of a previous one with little change. When in doubt, build sets that have 10 or more elements, and build at least 1 set with 1,000,000 or more elements as a stress test. Test building a set with a non-numeric data type, e.g. std::string. Test your features against an empty set. Test against a set with one element. Use loops, vectors, even the built-in std::set to test your work.

Note that you MUST build a threaded binary search tree as described in this handout, and use the threads in the tree to compute toPairs( ) and support foreach-style iteration. Any other solution will be considered invalid and earn a score of 0. The TAs will manually check this.

Academic Conduct Policy

Northwestern publishes a basic guide to academic integrity, which can be found here. In summary, here are NU's eight cardinal rules of academic integrity:

1. *Know your rights*
2. *Acknowledge your sources*
3. *Protect your work*
4. *Avoid suspicion*
5. *Do you own work*
6. *Never falsify a record or permit another person to do so*
7. *Never fabricate data, citations, or experimental results*
8. *Always tell the truth when discussing your work with your instructor*

School policies and more information can be found on NU's academic integrity website. With regards to CS 211, unless stated otherwise, all work submitted for grading *must* be done individually. While we encourage you to talk and learn from the course staff, peers, and others, 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 use of AI (ChatGPT, Co-pilot, etc.) is currently forbidden.

Examples of what is not allowed? Downloading work from a github repository and submitting it as your own, whether partial or complete. Downloading answers from StackOverflow and submitting them as your own. Emailing your work to another student, or receiving work from another student. Sharing your screen with another student so they can see your work (and potentially submit it as their own). Participating in a screen share and taking screenshots or photos of someone else's work, and then using that as a guide to submit your own work. Copying answers posted to Piazza, making a few simple changes, and then submitting as your own work. Allowing someone else to write / type the answer for you. Using AI (ChatGPT, Co-pilot, etc.) to generate code for you which you then submit as your own.

Okay, so what is allowed? Talking to the instructor or course staff, and getting insights --- but not direct answers --- to help you solve the assignment. Talking to other students about the assignment, using diagrams / natural language / pseudo-code to convey ideas. Searching the internet for guidance, and using that guidance to help you form your own solution. If you do receive help / guidance, it's always best to cite your source by name or URL just in case there is a question as to where the work came from.