CS 352 HW 4: Memory Management

Due Friday, March 20, 10pm

This assignment is on memory management, and is built around another data structures problem. You will complete the same code in both C++ and Rust, making sure to avoid memory errors as you do.

This assignment involves building some key parts of a *binary search tree* (BST) and its *iterator*. As a quick refresher on BSTs, they are a data structure where each node contains a single data value and references to a left and right subtree. The left and right subtrees are also BSTs, with the constraint that all values in the left subtree are less than the value in the root node and all values in the right subtree are greater than the value in the root node. In the context of memory management, we generally need to ensure that the left and right subtrees are recursively destroyed when their root node is.

We can build an iterator that will produce the elements of a BST in sorted order[1]. The approach is to keep a stack of references to tree nodes to return to (unlike the references in the tree nodes, these references should <u>not</u> destroy the tree nodes when the iterator is destroyed). The stack depends on the idea of the *leftmost descendant* of a tree node, which is the node containing the smallest value in the tree rooted at node. The leftmost descendant can be found using the following pseudocode:

```
node* leftmost = crnt;
while leftmost->left exists { leftmost = leftmost->left; }
```

When a node is added to the iterator, both that node and all its left children along the path to its leftmost descendant are added to the node stack in the order encountered. This initializes the iterator to the smallest element in the tree, while storing all the parent nodes that will need to be traversed later as well.

The iterator acts like a pointer; in this case, the pointed-to value is the value in the tree node at the top of the stack. Also like a pointer, an iterator can be incremented to point to the next element. This update is performed by removing the top node of the iterator stack, and, if it has a right child, adding that right child and the path to its leftmost descendant to the iterator stack.

The iterator stack obeys the following invariants, then:

- The nodes on the stack are precisely those which have the current node in their left subtree.
- A node is only produced by the iterator after its entire left subtree has been traversed.
- After the iterator is incremented past a node, it will traverse its entire right subtree.

See the link in the footnote for sample iterator code in Java.

[1] The approach is taken from this article

Question 1

Rewrite all the methods in the C++ starter file `bst.hpp` that include the text `unimplemented`. This generally includes replacing the dummy return value. You should not change the provided code in methods that are unimplemented, as listed below:

- **[2]** `static std::unique_ptr<bst> clone(const std::unique_ptr<bst>& o)`
  - Should produce a deep copy of `o` if `o` is non-null, or a null pointer otherwise.
- **[2]** `bst(const bst& o)` (copy constructor)
  - Should produce a deep copy of `o`; `clone()` may be useful as an implementation detail.
- **[3]** `bst& operator= (const bst& o)` (copy assignment)
  - Should assign a deep copy of `o`, similar to the copy-constructor.
  - Remember to check for self-assignment.
- **[2]** `void fill_left(const bst* node)`
  - Inserts node (and its recursive left children) into the iterator stack.
- **[1]** `const_iterator(const bst& node)`
  - Initialize an iterator for the tree rooted at `node`
- **[1]** `explicit operator bool() const`
  - True if iterator has any nodes remaining (i.e. node stack is non-empty)
- **[4]** `const_iterator& operator++()`
  - Go to the next node using the increment algorithm described on the first page.
- **[2]** `const T& operator* () const`
  - Get the current node's value (current node is top of iterator stack).
- **[6]** `bool insert(const T& node)`
  - Insert a new value into the tree. Searches down to find the proper location to insert, and creates a new node if necessary.
  - Returns `false` if the value was already in the tree, `true` if it was added.

Question 2

Rewrite all the methods in the Rust starter file bst.rs that include the macro `unimplemented!()`. You should not change the provided code in methods that are unimplemented, as listed below:

- **[2]** `fill_left(&mut self, mut node: &'a Bst<E>)`
  - Inserts node (and its recursive left children) into the iterator stack.
- **[1]** `new(node: &'a Bst<E>) -> BstIter<'a,E>`
  - Initialize an iterator for the tree rooted at node.
- **[7]** `next(&mut self) -> Option<Self::Item>`
  - If the iterator has any nodes remaining, returns the current node (wrapped in `Some`) and advances the iterator to the next node; otherwise returns `None`.
  - Analogous to `operator bool`, `operator++`, and `operator*` in C++.
- **[2]** `new(value: E) -> Self`
  - Creates a new BST node with the given value and empty left and right subtrees.

- **[6]** `insert(&mut self, node: E) -> bool`
    - Insert a new value into the tree. Searches down to find the proper location to insert, and creates a new node if necessary.
    - Returns `false` if the value was already in the tree, `true` if it was added.

Style

[6] Your C++ and Rust code should not be significantly more complex than necessary, and should be structured in a way that facilitates understanding. Make use of whitespace and comments as appropriate to make your code clearly readable. Function and variable names should clearly represent their purpose or be sufficiently limited in scope to be clear from context.

Testing

You can test your C++ code by compiling and running `test_bst.cpp`:

```
g++ -std=c++14 -o bst_cpp test_bst.cpp
./bst_cpp
```

You can test your Rust code by compiling and running `test_bst.rs`:

```
rustc -o bst_rs test_bst.rs
./bst_rs
```

The output for both programs should be this:

```
3 5 7
3 5
3
5
7
```

Question 3

[7] Write a short (~1 paragraph) reflection on your experience of writing the same code in both C++ and Rust. Was the memory management easier in one language? Did one require less boilerplate? Was one easier to debug? Do you have any other opinion on the difference between the two?

Submission Instructions

Your code and testing files for Questions 1 & 2 should be put into a zip file; include all code that was provided in the `cs352_hw4.zip` starter file.

Your answer to Question 3 should be put in a PDF file. Consult the instructor if you have trouble generating a PDF from your word processing software.

Both the zip and PDF files should be submitted on Moodle by the assignment due date.