Rust

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.

Iterators

A common pattern for implementing data structures in many languages (including Java, C++, and Rust) is to also implement an *iterator* object that can be used in a loop to access all elements of the collection. Iterators have three fundamental operations:

• *get current item*: return a reference to the element of the collection that the iterator currently points to.

• *go to next item*: mutate the iterator object to point to the next item in the collection.

• *check if done*: some mechanism to programmatically test if the entire collection has been iterated

Rust uses a standard-library Iterator trait which combines all three operations into a single next()method which returns either the current item or a sentinel value indicating the end of the collection has been reached, incrementing the iterator to point to the next item if the end of the collection has not been reached.

Iterators exercise the memory-management features of a language quite thoroughly, as their validity depends on the lifetime of the collection they are iterating, yet they may not maintain an explicit reference to the iterated collection or be capable of keeping the collection alive independent of other references.

Building a Binary Tree Iterator

We can build an iterator that will produce the elements of a BST in sorted order1. 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 not 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

1 The approach is taken from [this article,](https://medium.com/algorithm-problems/binary-search-tree-iterator-19615ec585a) which includes sample code in Java.

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; }

For Rust, you may find a recursive algorithm easier to get the borrow-checker to accept.

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 in sorted order. 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, the same as an in-order traversal:

• The nodes on the stack are precisely the current node and 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.

As an example, consider the binary tree **(a)**, the iterator **(b)** initialized to the smallest (leftmost)

value in **(a)**, and the iterator **(c)**, the result of incrementing **(b)** to the next value:



8 (8)

(2)



2 9

6



top



(8) (6) (5)

top



5 7

**(a)**



**(b)**



**(c)**



Question 1

Rewrite all the methods in the Rust starter file bst.rs that include the macro

unimplemented!(). You may find the Rust documentation for Option2 and Iterator3

2 <https://doc.rust-lang.org/std/option/index.html>

3 <https://doc.rust-lang.org/std/iter/index.html>

useful. You should not change the provided code in methods that are not unimplemented, or the type signatures of the unimplemented methods listed below:

• **[2]** fill\_left(&mut self, node: &'a Bst<E>)

o Inserts node (and its recursive left children) into the iterator stack, as described on page 2.

• **[1]** new(node: &'a Bst<E>) -> BstIter<'a,E>

o Initialize an iterator for the tree rooted at node.

• **[7]** next(&mut self) -> Option<Self::Item>

o 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.

• **[2]** new(value: E) -> Self

o Creates a new BST node with the given value and empty left and right subtrees.

• **[6]** insert(&mut self, val: E) -> bool

o Insert a new value into the tree. Searches down to find the proper location to insert, and creates a new node if necessary.

o Returns falseif the value was already in the tree, trueif it was added.

o *Tip:* You should solve this recursively, it’s incredibly painful to write the iterative version in Rust, handling the lifetimes gets very complex.

• **[4]** sum(&'a self) -> E

o Returns the sum of the elements of the tree.

o You should use the iter()method (possibly as part of a **for**-loop) to traverse the tree.

o *Tip:* the extra trait bounds on this implblock mean that you can convert integers (like 0) to the element type, and also use the +=operator with a reference to the element type (like the one returned by the iterator) as a right-hand-side.

Style

**[3]** Your 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 code by compiling and running test\_bst.rs:

rustc test\_bst.rs

./test\_bst

The output for the program should be this:

Testing insert

3 5 7

3 5

Testing iterator

3

5

7

Testing sum

15

Question 2

**a. [4]** Between this course and its pre-requisites, you now have written assignments in Java, C, C++, Haskell, and Rust. Pick one feature (e.g. memory management, trait implementation, language syntax) of Rust you used for this homework, and write a short (~1 paragraph) reflection comparing and contrasting that feature to a similar feature in one of the other languages. Was one easier to use? Did one make more powerful coding patterns available? Do you have any other thoughts comparing the two?

**b. [1]** How much time did the assignment take you? What was the most challenging function to write? What was the easiest?