Leetcode in Rust

Contents

1 Rust in a Nutshell

1.1 Why Rust?

1.2 Cargo

1.3 Cargo Doc

1.4 Crates

1.5 Basic Data Structures

1.5.1 Sequences

1.5.1.1 Vec

Rust's growable array type is called `Vec`, short for `vector` (which comes from C++). In python any many functional languages it's called a list, and in other languages, an array.

To create a vector, one can use this syntax:

```
let mut v = Vec::new();
```

The main operation of a vector is `push`, which appends one element to the end of the vector.

```
let mut v = Vec::new();
v.push(5); // The vector now looks like this: [5].
```

Pushing has an amortized time complexity of O(1). Pushing to a vector has a worst case time complexity of O(n), because vectors dynamically grow.

If a vector is full, and you push back to a vector, the vector must do the following:

1. Allocate a buffer that is twice the size of its previous buffer
2. Copy over its current items to the new buffer
3. Add the new element to the buffer.
4. Free the previous buffer.

The first, third, and fourth step all take constant time, but copying over every item from the previous to the new buffer takes linear (O(n)) time. That being said, as long as you avoid this case as much as possible (which you can mitigate by doubling the buffer size every time) the time complexity for pushing to a vector is constant time on average.

If you want to check the contents of a vector at any given time, you can print it:

```rust
let mut v = Vec::new();
v.push(5);
println!("{:?}", v);
```

1.5.1.2 VecDeque

1.5.1.3 LinkedList

1.5.2 Maps

1.5.2.1 HashMap

1.5.2.2 BTreeMap

1.5.3 Sets

1.5.3.1 HashSet

1.5.3.2 BTreeSet

1.5.4 Other

1.5.4.1 BinaryHeap

1.6 Basic Algorithms

1.7 Other Useful things

1.8 Regex

1.9 Derive Macros

1.10 Counting in O(1) space with slices

2 Macros for Rust

2.1 A macro for testing

Unlike C and C++, a testing framework is built into rust. We can create our own tests by creating a `mod` block and letting cargo know that we want to test it.

Let's say we create this function:

```
fn add(a: i32, b: i32) -> i32 {
  a + b
}
```

We can test it at the bottom of the file:

```
#[cfg(test)]
mod test {
  use super::*;

  #[test]
  fn add_one_and_one() {
    assert_eq!(add(1, 1), 2);
  }

  #[test]
  fn add_one_and_two() {
    assert_eq!(add(1, 2), 3);
  }
}
```

Macros let us reduce most of the boilerplate:

```
#[macro_export]
macro_rules! test {
    ($($name:ident: $left:expr, $right:expr,)*) => {
        #[cfg(test)]
        mod test {
        use super::*;
            $(
                #[test]
                fn $name() {
                    assert_eq!($left, $right);
```

```
                }
            )*
        }
    }
}
```

Our tests can then be rewritten like so:

```
test! {
  add_one_to_one: add(1, 1), 2,
  add_one_to_two: add(1, 2), 3,
}
```

And running them gives us this result:

```
$ cargo test
running 2 tests
test test::add_one_and_one ... ok
test test::add_one_and_two ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
↪ in 0.01s
```

3 Trees

3.1 Validate Binary Search Tree

```
use crate::*;

pub fn is_valid_bst(root: BSTNode) -> bool {
    fn helper(node: &BSTNode, possible_min: i64, possible_max: i64) -> bool {
        if let Some(n) = node {
            let borrowed = n.borrow();
            let left = &borrowed.left;
            let right = &borrowed.right;
            let val: i64 = borrowed.val.into();
            if val >= possible_min && val <= possible_max {
                helper(&left, possible_min, val) && helper(&right, val, possible_max)
            } else {
                false
            }
        } else {
            true
        }
    }
    helper(&root, i64::MIN, i64::MAX)
}

test! {
    test_1: is_valid_bst(btree![2, 1, 3]), true,
    test_2: is_valid_bst(btree![5, 1, 3]), false,
}
```

3.2 Same Tree

3.2.1 Problem

> Given the roots of two binary trees p and q, write a function to check if they are the same or not. Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.

3.2.2 Intuition

This question tests your knowledge of recursion. To do so, start off with the base case:

- What happens when left is None and right has a value? Return false.
- What happens when left has a value and right is None? Return false.
- What happens when both left and right are None? Return true.
- What happens when left and right have different values? Return false.
- What happens when left and right have the same values? Test their left and right nodes for equality as well.

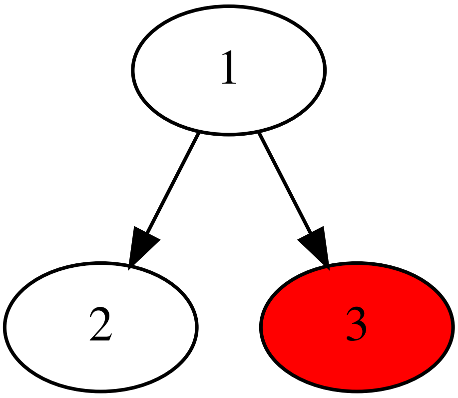
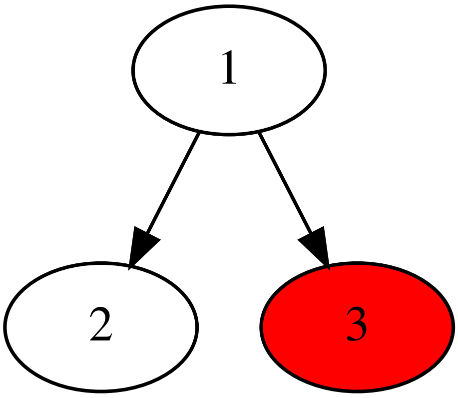
3.2.3 Test Cases

```
test! {
    test_1: is_same_tree(btree![], btree![]), true,
    test_2: is_same_tree(btree![1], btree![]), false,
    test_3: is_same_tree(btree![1,2,3], btree![1,2,3]), true,
    test_4: is_same_tree(btree![1,2,3,4], btree![1,2,3]), false,
}
```

3.2.4 Answer

```
/// Calculates if two binary search trees have the same values.
/// In this question, there are four possible cases:
/// 1. Both left and right point to a `None` node. In this case, return true.
/// 2. Both left and right point to nodes with the same value. Continue recursing
↪   through both
///    trees left and right subtrees.
/// 3. For any other case, return false.
pub fn is_same_tree(p: BSTNode, q: BSTNode) -> bool {
    fn same(p: &BSTNode, q: &BSTNode) -> bool {
        match (p, q) {
            (Some(left), Some(right)) => {
                let left = left.borrow();
                let right = right.borrow();
                left.val == right.val
                    && same(&left.left, &right.left)
                    && same(&left.right, &right.right)
            }
            (None, None) => true,
            _ => false,
```

```
        }
    }
    same(&p, &q)
}
```

Left	Right
	

3.3 Maximum Path through a Binary Tree

```rust
use crate::*;
use std::cmp::max;

/// Finds the maximum path sum through a binary tree.
pub fn max_path_sum(root: BSTNode) -> i32 {
  let mut max_so_far = i32::MIN;
  fn helper(node: &BSTNode, max_so_far: &mut i32) -> i32 {
    match node {
      Some(n) => {
        let val = n.borrow().val;
        let l = max(0, helper(&n.borrow().left, max_so_far));
        let r = max(0, helper(&n.borrow().right, max_so_far));
        *max_so_far = max(*max_so_far, val + l + r);
        val + max(l, r)
      }
      None => 0,
    }
  }
  helper(&root, &mut max_so_far);
```

```
  max_so_far
}

test! {
    test_1: max_path_sum(btree![1,2,3]), 6,
    test_2: max_path_sum(btree![-10, 9, 20, null, null, 15, 7]), 42,
}
```