Solved Exercise: Step by Step Guide

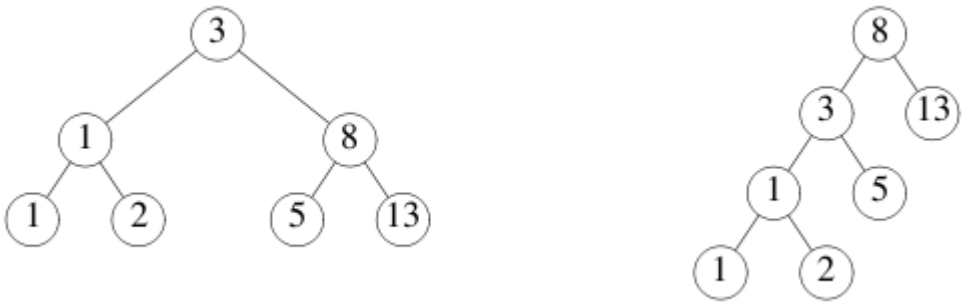
This section has a few questions for you try out and test your understanding of concurrency

WE'LL COVER THE FOLLOWING

Assignment

Online Assignment

There can be many different binary trees with the same sequence of values stored at the leaves. For example, here are two binary trees storing the sequence 1, 1, 2, 3, 5, 8, 13.



A function to check whether two binary trees store the same sequence is quite complex in most languages. We'll use Go's concurrency and channels to write a simple solution.

This example uses the `tree` package, which defines the type:

```
type Tree struct {
    Left  *Tree
    Value int
    Right *Tree
}
```

1. Implement the Walk function.
2. Test the Walk function.

The function `tree.New(k)` constructs a randomly-structured binary tree holding the values `k`, `2k`, `3k`, ..., `10k`.

Create a new channel `ch` and kick off the walker:

```
go Walk(tree.New(1), ch)
```

Then read and print 10 values from the channel. It should be the numbers `1`, `2`, `3`, ..., `10`.

3. Implement the Same function using `Walk` to determine whether `t1` and `t2` store the same values.

4. Test the `Same` function.

`Same(tree.New(1), tree.New(1))` should return `true`, and `Same(tree.New(1), tree.New(2))` should return `false`.

Solution

If you print `tree.New(1)` you will see the following tree:

```
((((1 (2)) 3 (4)) 5 ((6) 7 ((8) 9))) 10)
```

To implement the `Walk` function, we need two things:

- walk each side of the tree and print the values
- close the channel so the `range` call isn't stuck.

We need to set a recursive call and for that, we are defining a non-exported `recWalk` function, the function walks the left side first, then pushes the value to the channel and then walks the right side. This allows our range to get the values in the right order. Once all branches have been walked, we can close the channel to indicate to the range that the walking is over.

Environment Variables

Key:	Value:
GOPATH	/go

```
package main

import (
  "git://github.com/golang/tour"
        "fmt"
)

// Walk walks the tree t sending all values
// from the tree to the channel ch.
func Walk(t *tree.Tree, ch chan int) {
        recWalk(t, ch)
        // closing the channel so range can finish
        close(ch)
}

// recWalk walks recursively through the tree and push values to the channel
// at each recursion
func recWalk(t *tree.Tree, ch chan int) {
        if t != nil {
                // send the left part of the tree to be iterated over first
                recWalk(t.Left, ch)
                // push the value to the channel
                ch <- t.Value
                // send the right part of the tree to be iterated over last
                recWalk(t.Right, ch)
        }
}

// Same determines whether the trees
// t1 and t2 contain the same values.
func Same(t1, t2 *tree.Tree) bool {
        ch1 := make(chan int)
        ch2 := make(chan int)
        go Walk(t1, ch1)
        go Walk(t2, ch2)

        for {
                x1, ok1 := <-ch1
                x2, ok2 := <-ch2
                switch {
                case ok1 != ok2:
                        // not the same size
                        return false
                case !ok1:
                        // both channels are empty
                        return true
                case x1 != x2:
                        // elements are different
                        return false
                default:
                        // keep iterating
                }
        }
}

func main() {
        ch := make(chan int)
```

```
        go Walk(tree.New(1), ch)
        for v := range ch {
                fmt.Println(v)

        }
        fmt.Println(Same(tree.New(1), tree.New(1)))
        fmt.Println(Same(tree.New(1), tree.New(2)))
}

// Walk walks the tree t sending all values
// from the tree to the channel ch.
func Walk(t *tree.Tree, ch chan int) {
        recWalk(t, ch)
        // closing the channel so range can finish
        close(ch)
}

// recWalk walks recursively through the tree and push values to the channel
// at each recursion
func recWalk(t *tree.Tree, ch chan int) {
        if t != nil {
                // send the left part of the tree to be iterated over first
                recWalk(t.Left, ch)
                // push the value to the channel
                ch <- t.Value
                // send the right part of the tree to be iterated over last
                recWalk(t.Right, ch)
        }
}

// Same determines whether the trees
// t1 and t2 contain the same values.
func Same(t1, t2 *tree.Tree) bool {
        ch1 := make(chan int)
        ch2 := make(chan int)
        go Walk(t1, ch1)
        go Walk(t2, ch2)

        for {
                x1, ok1 := <-ch1
                x2, ok2 := <-ch2
                switch {
                case ok1 != ok2:
                        // not the same size
                        return false
                case !ok1:
                        // both channels are empty
                        return true
                case x1 != x2:
                        // elements are different
                        return false
                default:
                        // keep iterating
                }
        }
}

func main() {
        ch := make(chan int)
        go Walk(tree.New(1), ch)
        for v := range ch {
                fmt.Println(v)
        }
```

```
        fmt.Println(Same(tree.New(1), tree.New(1)))
        fmt.Println(Same(tree.New(1), tree.New(2)))
}
```

The comparison of the two trees is trivial once we know how to extract the values of each tree. We just need to loop through the first tree (via the channel), read the value, get the value from the second channel (walking the second tree) and compare the two values.