go-tree-iterator : AI took over!

I was diddling with the new **range over function iterator** scheme in Golang and thought of a test case that would help me understand it. So I did some implementation and testing of a version of an iterator for a binary search tree.

Once I started on this effort, it kind of morphed more into an exercise using AI to generate some or all of the code instead of the iterator itself.

The Data Structure

A brief description of my initial intention to create an iterator.

Hash Map vs Tree Map

Golang has a built in hash **map** that fits most use cases for a key/value store. Iterating over a Golang **map** does not guarantee any particular order.

On the other hand, in the first cut of the C++ STL (98), the **std::map** container was implemented as a binary search tree rather than a hash map. The **std::map** had a behavior that the Go hash map does not have : iterating over the keys/value in order (https://en.cppreference.com/w/cpp/container/map). The actual implementation is usually a red-black tree. Subsequent updates of the C++ STL (11) added the **std::unordered_map** which was a hash map.

Most of the time a hash map will have insert,lookup and delete operations that O(1). That's better than a generic search tree which would have average O(log N) insert, lookup and delete, but could degenerate to O(N) if values were inserted in a particular order. A red-black tree (https://en.wikipedia.org/wiki/Red%E2%80%93black_tree) avoids that situation by maintaining a mostly balanced tree structure. Insert, lookup and delete are worst case O(log N). AVL tree fans please cut me some slack.

In any case, there could be specific instances where iterating over a map in order is more important than insert/delete time. For example if the structure is only built once or added infrequently, but iterating in order is done more often.

Implementing a Red-Black Tree in Go

I figured if I was going to do this, I should bite the bullet and code up something better than a simple binary tree. I picked a red-black tree.

As an aside, I learned on and have always used the Sedgwick algorithms (https://algs4.cs.princeton.edu/home/) as a reference for implementation and analysis. I found the CLRS book (https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/) a bit too mathy for me, because I'm not smart enough.

The Princeton/Sedgewick website provides a free online course, book and Java implementation for all the algorithms covered in the book and course. Highly recommended if you are learning algorithms on your own. I find it very accessible.

Since it might take me a month or two to remember how to implement a red-black tree from scratch, if I could do it at all, I planned to do a port of the Sedgick Java implementation RBT to Go for this exercise (https://algs4.cs.princeton.edu/code/edu/princeton/cs/algs4/RedBlackBST.java.html).

I also decided to make the implementation generic for key/value types.

AI Took Over

I could have hand coded the port line by line, but since it's 2024, I used a bit (actually a lot) of AI to help me out. This turned out to be a better learning experience than implementing the iterator.

I started with Github Copilot.

Github Copilot - pkg/copilot

I have been using Github Copilot for a while and it has really accerated my development time. By a lot. I just followed the Java example function by function and let Copilot do the conversion.

At the top of the new file, I added a link to the online Java implementation in a comment. I also downloaded it to a local directory in case Copilot would use that as a model. I'm not sure if Copilot used the link or local file, but probably it had already been trained on the exact Java file because it knew the code from the start.

I created the generic Node and RBT structs first, following the Java example. After that I just started porting all the functions in order from the Java file. All I had to do was add a comment describing what I wanted and Copilot filled in code that matched almost exactly to the Java version. Occasionally I had to write the func header and it filled in the body.

One feature is that it added error returns for some functions, like 'Get' when the key is not found. More on that below.

I worked through the whole Java file and got it running. I only had to fix a few glitches. Copilot didn't capitalize the funcs that should be exported, and I had to massage a Node structure to be generic. It took about an hour to get it all done. The Java file had a lot of functionality besides just get and put. It is about 700 lines long.

I added a rudimentary set of tests and they all passed. I had to do very little manual labor to get this thing running.

Gemini Flash LLM - pkg/gemini

Once I had that working I had the bright idea to try another approach. Since the Java file is pretty big, I wanted to see if I could get a different AI to generate the entire Go code with one prompt.

To do this, I used the sgpt command line tool with the Google Gemini Flash model (https://deepmind.google/technologies/gemini/flash/). I had watched how to use it with sgpt from a video by AiCodeKing https://www.youtube.com/watch?v=BoihrNkJ9dY. Btw, his series of AI videos is really great. It covers how to use all the different models and associated software locally. At a low level, not just using an existing web interface. Recommended.

Here's the prompt I used: "$sgpt --code "using the file at https://algs4.cs.princeton.edu/code/edu/princeton/cs/algs4/RedBlackBST.java.html as a model, create a version of that code using the go language. The implementation should be generic with respect to key and value types."

It took about a minute to generate a result. The code it generated was almost perfect. I had a very few things I needed to fix.

The one thing that it differed from the Copilot version that instead of returning errors (for things like 'no such key'), it return a boolean, which I translated to 'ok' in the code. It was clear that Gemini either used the link I gave it or it had already trained on this code because it almost word for word matched the Java version.

Again I had to fix the function namesthat are marked public in the Java code so they are exported. I probably could have refined my prompt to have it do that in the generator by telling it to capitalize Java functionsl marked public.

I also had to add **constraints.Ordered** to the key key (K) type specs. This showed up as syntax errors in the places where the keys were being compared. Gemini tried to use 'comparable' (check for equality) instead of Ordered.

I was able to get this whole thing passing in about 5 minutes.

ChatGpt LLM - pkg/chatgpt

Well, since that worked pretty well, I decided to try OpenAI ChatGPT. I used the OpenAI ChatGPT online interface. I used the same prompt as I did for Gemini:

"$sgpt --code "using the file at https://algs4.cs.princeton.edu/code/edu/princeton/cs/algs4/RedBlackBST.java.html as a model, create a version of that code using the go language. The implementation should be generic with respect to key and value types"

It also took about a minute and I plugged in the result to a file. The code it generated was good but it left out a set of functions hat retrieved the list of keys, which I needed. - Like Gemini it used a bool 'ok' instead of throwing errors for things like key not found or empty tree.

I ran two versions. For the first one I uploaded the bst.java file and prompted it to use that. This attempt did not seem to use the file because it generated a very stripped down version. For the second attempt I prompted it to use the GitHub file instead of the local one and it did much better.

I had to fix the exports, add a Keys function, copied from the Gemini version, and again had to add **constraints.Ordered** to the key 'K' type specs. This showed up as syntax errors in the places where the keys were being compared.

After about 15 minutes of massaging this version passed the tests.

Extra Work for Compatibility

At this point I wanted to make the 3 versions as compatible with the tests as possible. I added an interface definition for a red-black-tree model in pkg/rbt.

```go
package rbt

import "golang.org/x/exp/constraints"

type KeyValuePair[K constraints.Ordered, V any] struct {
    Key K
    Val V
}

type RBT[K constraints.Ordered, V any] interface {
    Put(key K, val V)
    Get(key K) (V, bool)
    IsEmpty() bool
    GetAll() []KeyValuePair[K, V]
    Iterator() func(yield func(KeyValuePair[K, V]) bool)
}
```

I had to fix the Copilot version to use the bool returns instead of errors. That took a few minutes.

Implementing the Iterator

Ok, back to the original intent, implementing the range over function iterator.

Iterating over a binary search tree is pretty simple, just do an inorder traversal and emit the key/value pairs. Copilot created the iterator for me. I just had to ask it for one. It magically knew to use an inorder traversal, totally different from one in the Java code.

For the iterator to compile, I used go 1.22.5 with the GOEXPERIMENT=rangefunc env variable.

```go
func (t *RBT[K, V]) Iterator() func(yield func(K, V) bool) {
    return func(yield func(K, V) bool) {
        var inorder func(*Node[K, V])
        inorder = func(n *Node[K, V]) {
            if n == nil {
                return
            }
            inorder(n.left)
            if !yield(n.key, n.val) {
                return
            }
            inorder(n.right)
        }
        inorder(t.root)
    }
}

// using the iterator
for r := range rbt.Iterator() {
        fmt.Println(r.Key, r.Val)
```

```
  }
```

https://algs4.cs.princeton.edu/code/edu/princeton/cs/algs4/RedBlackBST.java.html
https://bitfieldconsulting.com/posts/iterators