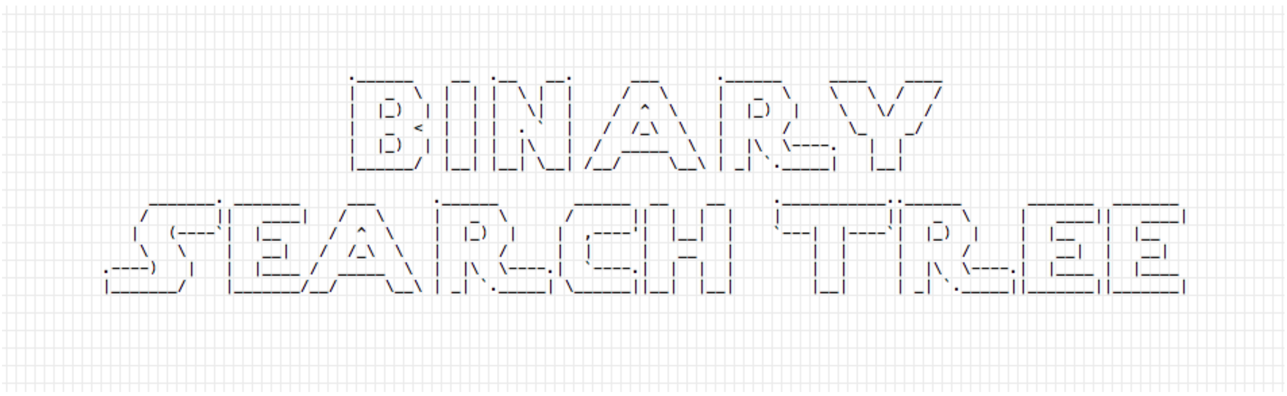
THE GO DATA STRUCTURES BOOK

This ebook aims to cover all the main classic Data Structures implemented in Go.

Data structures covered, in alphabetical order:

- Binary Search Tree
- Dictionary
- Graph
- Hash Table
- Linked List
- Queue
- Set
- Stack



A **tree** is a representation of a hierarchical structure. It’s easy to imagine a tree by thinking about a family genealogy tree.

Like a hash table or a graph, is a non-sequential data structure.

A **binary tree** is a tree where every node has max 2 children.

A **binary search tree** has the property of the left node having a value less than the value on the right node.

This is what we’ll build in this article. It’s a very useful data structure for efficient storing and indexing of data, and data retrieval.

Terminology

Root: the level 0 of the tree

Child: each node of the leaf that’s not the root

Internal node: each node with at least a child

Leaf: each node that as no childs

Subtree: the set of node with a certain node as root

Preliminary info

A binary search tree data structure will expose those methods:

- `Insert(t)` inserts the Item t in the tree
- `Search(t)` returns true if the Item t exists in the tree
- `InOrderTraverse()` visits all nodes with in-order traversing
- `PreOrderTraverse()` visits all nodes with pre-order traversing
- `PostOrderTraverse()` visits all nodes with post-order traversing
- `Min()` returns the Item with min value stored in the tree
- `Max()` returns the Item with max value stored in the tree
- `Remove(t)` removes the Item t from the tree
- `String()` prints a CLI readable rendering of the tree

I'll create an `ItemBinarySearchTree` generic type, concurrency safe, that can generate trees containing any type by using `genny`, to create a type-specific tree implementation, encapsulating the actual value-specific data structure containing the data.

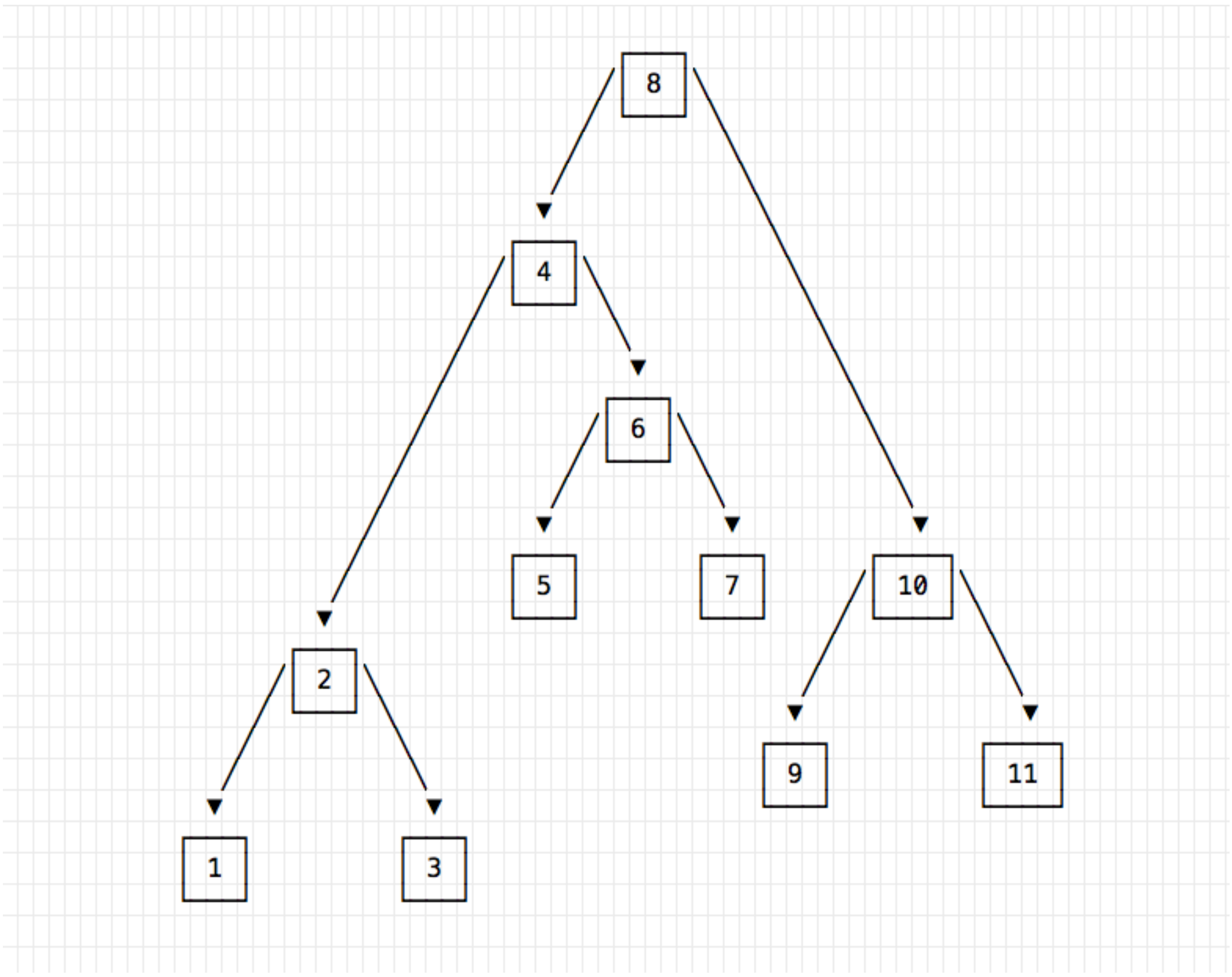
I define each note as

```
// Node a single node that composes the tree
type Node struct {
    key   int
    value Item
    left  *Node //left
    right *Node //right
}
```

The key value allows to use any kind of Item type, and use a separate value for calculating the correct place. I implemented it as an integer, but it could take any type that can be compared.

Inserting an item into a tree requires the use of recursion, since we need to find the correct place for it. The rule is, if the key of the node is < than the current node tree, we put it as the left child, if there is no left child. Otherwise, we recalculate the position by using the left child as the base node. Same goes for the right child.

Traversing is the process of navigating the tree, and we implement 3 ways to do it, since there are 3 different approaches. Taken this binary search tree:



this is how we could traverse it:

- **in-order**: we visit all nodes by following the smallest link until we find the left-most leaf, then processes the leaf and moves to other nodes by going into the next smallest key value linked to the current one. In the above picture: `1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11`
- **pre-order**: before visiting the children, this approach visits the node first. In the above picture: `8 -> 4 -> 2 -> 1 -> 3 -> 6 -> 5 -> 7 -> 10 -> 9 -> 11`
- **post-order**: we find the smallest leaf (the left-most one), then processes the sibling and then the parent node, then goes to the

next subtree, and navigates up the parent. In the above picture: `1 -> 3 -> 2 -> 5 -> 7 -> 6 -> 4 -> 9 -> 11 -> 10 -> 8`

The `String` method, used in the tests to have a visual feedback on the methods, will print the above tree as

```
------------------------------------------------
                     ---[ 1
              ---[ 2
                     ---[ 3
       ---[ 4
                     ---[ 5
              ---[ 6
                     ---[ 7
---[ 8
              ---[ 9
       ---[ 10
              ---[ 11
------------------------------------------------
```

Implementation

```go
// Package binarysearchtree creates a ItemBinarySearchTree data structure for the Item type
package binarysearchtree

import (
    "fmt"
    "sync"

    "github.com/cheekybits/genny/generic"
)

// Item the type of the binary search tree
type Item generic.Type

// Node a single node that composes the tree
type Node struct {
    key   int
    value Item
    left  *Node //left
    right *Node //right
}

// ItemBinarySearchTree the binary search tree of Items
type ItemBinarySearchTree struct {
    root *Node
    lock sync.RWMutex
}

// Insert inserts the Item t in the tree
func (bst *ItemBinarySearchTree) Insert(key int, value Item) {
    bst.lock.Lock()
    defer bst.lock.Unlock()
    n := &Node{key, value, nil, nil}
    if bst.root == nil {
        bst.root = n
    } else {
        insertNode(bst.root, n)
    }
}

// internal function to find the correct place for a node in a tree
func insertNode(node, newNode *Node) {
    if newNode.key < node.key {
        if node.left == nil {
            node.left = newNode
        } else {
            insertNode(node.left, newNode)
        }
    } else {
        if node.right == nil {
            node.right = newNode
        } else {
            insertNode(node.right, newNode)
        }
    }
}

// InOrderTraverse visits all nodes with in-order traversing
func (bst *ItemBinarySearchTree) InOrderTraverse(f func(Item)) {
    bst.lock.RLock()
    defer bst.lock.RUnlock()
    inOrderTraverse(bst.root, f)
}

// internal recursive function to traverse in order
func inOrderTraverse(n *Node, f func(Item)) {
    if n != nil {
        inOrderTraverse(n.left, f)
        f(n.value)
        inOrderTraverse(n.right, f)
    }
}

// PreOrderTraverse visits all nodes with pre-order traversing
func (bst *ItemBinarySearchTree) PreOrderTraverse(f func(Item)) {
    bst.lock.Lock()
    defer bst.lock.Unlock()
    preOrderTraverse(bst.root, f)
}

// internal recursive function to traverse pre order
func preOrderTraverse(n *Node, f func(Item)) {
    if n != nil {
        f(n.value)
        preOrderTraverse(n.left, f)
        preOrderTraverse(n.right, f)
    }
}

// PostOrderTraverse visits all nodes with post-order traversing
func (bst *ItemBinarySearchTree) PostOrderTraverse(f func(Item)) {
    bst.lock.Lock()
    defer bst.lock.Unlock()
    postOrderTraverse(bst.root, f)
}
```

```go
// internal recursive function to traverse post order
func postOrderTraverse(n *Node, f func(Item)) {
    if n != nil {
        postOrderTraverse(n.left, f)
        postOrderTraverse(n.right, f)
        f(n.value)
    }
}

// Min returns the Item with min value stored in the tree
func (bst *ItemBinarySearchTree) Min() *Item {
    bst.lock.RLock()
    defer bst.lock.RUnlock()
    n := bst.root
    if n == nil {
        return nil
    }
    for {
        if n.left == nil {
            return &n.value
        }
        n = n.left
    }
}

// Max returns the Item with max value stored in the tree
func (bst *ItemBinarySearchTree) Max() *Item {
    bst.lock.RLock()
    defer bst.lock.RUnlock()
    n := bst.root
    if n == nil {
        return nil
    }
    for {
        if n.right == nil {
            return &n.value
        }
        n = n.right
    }
}

// Search returns true if the Item t exists in the tree
func (bst *ItemBinarySearchTree) Search(key int) bool {
    bst.lock.RLock()
    defer bst.lock.RUnlock()
    return search(bst.root, key)
}

// internal recursive function to search an item in the tree
func search(n *Node, key int) bool {
    if n == nil {
        return false
    }
    if key < n.key {
        return search(n.left, key)
    }
    if key > n.key {
        return search(n.right, key)
    }
    return true
}

// Remove removes the Item with key `key` from the tree
func (bst *ItemBinarySearchTree) Remove(key int) {
    bst.lock.Lock()
    defer bst.lock.Unlock()
    remove(bst.root, key)
}

// internal recursive function to remove an item
func remove(node *Node, key int) *Node {
    if node == nil {
        return nil
    }
    if key < node.key {
        node.left = remove(node.left, key)
        return node
    }
    if key > node.key {
        node.right = remove(node.right, key)
        return node
    }
    // key == node.key
    if node.left == nil && node.right == nil {
        node = nil
        return nil
    }
    if node.left == nil {
        node = node.right
        return node
    }
    if node.right == nil {
        node = node.left
        return node
    }
    leftmostrightside := node.right
    for {
        //find smallest value on the right side
        if leftmostrightside != nil && leftmostrightside.left != nil {
            leftmostrightside = leftmostrightside.left
        } else {
            break
        }
    }
    node.key, node.value = leftmostrightside.key, leftmostrightside.value
    node.right = remove(node.right, node.key)
    return node
}

// String prints a visual representation of the tree
func (bst *ItemBinarySearchTree) String() {
    bst.lock.Lock()
    defer bst.lock.Unlock()
    fmt.Println("------------------------------------------------")
    stringify(bst.root, 0)
    fmt.Println("------------------------------------------------")
}

// internal recursive function to print a tree
func stringify(n *Node, level int) {
    if n != nil {
        format := ""
        for i := 0; i < level; i++ {
            format += "       "
        }
        format += "---[ "
        level++
        stringify(n.left, level)
        fmt.Printf(format+"%d\n", n.key)
        stringify(n.right, level)
    }
}
```

Tests

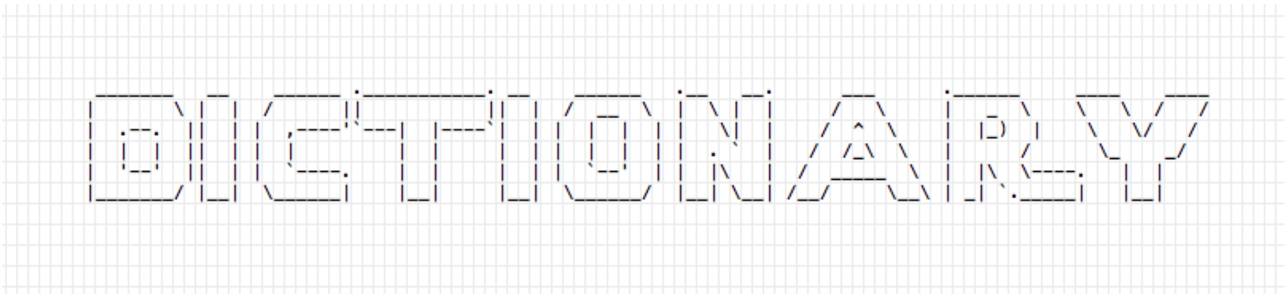
The tests describe the usage of the above implementation.

```go
package binarysearchtree

import (
    "fmt"
    "testing"
)

var bst ItemBinarySearchTree

func fillTree(bst *ItemBinarySearchTree) {
    bst.Insert(8, "8")
    bst.Insert(4, "4")
    bst.Insert(10, "10")
    bst.Insert(2, "2")
    bst.Insert(6, "6")
    bst.Insert(1, "1")
    bst.Insert(3, "3")
    bst.Insert(5, "5")
    bst.Insert(7, "7")
    bst.Insert(9, "9")
}

func TestInsert(t *testing.T) {
    fillTree(&bst)
    bst.String()

    bst.Insert(11, "11")
    bst.String()
}

// isSameSlice returns true if the 2 slices are identical
func isSameSlice(a, b []string) bool {
    if a == nil && b == nil {
        return true
    }
    if a == nil || b == nil {
        return false
    }
    if len(a) != len(b) {
        return false
    }
    for i := range a {
        if a[i] != b[i] {
            return false
        }
    }
    return true
}

func TestInOrderTraverse(t *testing.T) {
    var result []string
    bst.InOrderTraverse(func(i Item) {
        result = append(result, fmt.Sprintf("%s", i))
    })
    if !isSameSlice(result, []string{"1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11"}) {
        t.Errorf("Traversal order incorrect, got %v", result)
    }
}

func TestPreOrderTraverse(t *testing.T) {
    var result []string
    bst.PreOrderTraverse(func(i Item) {
        result = append(result, fmt.Sprintf("%s", i))
    })
    if !isSameSlice(result, []string{"8", "4", "2", "1", "3", "6", "5", "7", "10", "9", "11"}) {
        t.Errorf("Traversal order incorrect, got %v instead of %v", result, []string{"8", "4", "
    }
}

func TestPostOrderTraverse(t *testing.T) {
    var result []string
    bst.PostOrderTraverse(func(i Item) {
        result = append(result, fmt.Sprintf("%s", i))
    })
    if !isSameSlice(result, []string{"1", "3", "2", "5", "7", "6", "4", "9", "11", "10", "8"}) {
        t.Errorf("Traversal order incorrect, got %v instead of %v", result, []string{"1", "3", "
    }
}

func TestMin(t *testing.T) {
    if fmt.Sprintf("%s", *bst.Min()) != "1" {
        t.Errorf("min should be 1")
    }
}

func TestMax(t *testing.T) {
    if fmt.Sprintf("%s", *bst.Max()) != "11" {
        t.Errorf("max should be 11")
    }
}

func TestSearch(t *testing.T) {
    if !bst.Search(1) || !bst.Search(8) || !bst.Search(11) {
        t.Errorf("search not working")
    }
}

func TestRemove(t *testing.T) {
    bst.Remove(1)
    if fmt.Sprintf("%s", *bst.Min()) != "2" {
        t.Errorf("min should be 2")
    }
}
```

Creating a concrete tree data structure

You can use this generic implemenation to generate type-specific trees, using

```
//generate a `IntBinarySearchTree` binary search tree of `int` values
genny -in binarysearchtree.go -out binarysearchtree-int.go gen "Item=int"

//generate a `StringBinarySearchTree` binary search tree of `string` values
genny -in binarysearchtree.go -out binarysearchtree-string.go gen "Item=string"
```



A dictionary stores `[key, value]` pairs. Go provides a very convenient implementation of a dictionary by its built-in `map` type.

In this article I’ll enrich the `map` built-in type with some convenient operations to get information out of it, and to change its content.

I’ll create a `ItemDictionary` generic type, concurrency safe, that can generate dictionaries for any type. By running `genny` the code will create a type-specific Dictionary implementation, encapsulating the actual `map` containing the data.

Goal

The dictionary is created using `dict := ValueDictionary{}` and provides this set of exported methods:

- `Set()`
- `Delete()`
- `Has()`
- `Get()`
- `Clear()`
- `Size()`
- `Keys()`
- `Values()`

Implementation

```go
// Package dictionary creates a ValueDictionary data structure for the Item type
package dictionary

import (
    "sync"

    "github.com/cheekybits/genny/generic"
)

// Key the key of the dictionary
type Key generic.Type

// Value the content of the dictionary
type Value generic.Type

// ValueDictionary the set of Items
type ValueDictionary struct {
    items map[Key]Value
    lock  sync.RWMutex
}

// Set adds a new item to the dictionary
func (d *ValueDictionary) Set(k Key, v Value) {
    d.lock.Lock()
    defer d.lock.Unlock()
    if d.items == nil {
        d.items = make(map[Key]Value)
    }
    d.items[k] = v
}

// Delete removes a value from the dictionary, given its key
func (d *ValueDictionary) Delete(k Key) bool {
    d.lock.Lock()
    defer d.lock.Unlock()
    _, ok := d.items[k]
    if ok {
        delete(d.items, k)
    }
    return ok
}
```

```go
// Has returns true if the key exists in the dictionary
func (d *ValueDictionary) Has(k Key) bool {
    d.lock.RLock()
    defer d.lock.RUnlock()
    _, ok := d.items[k]
    return ok
}

// Get returns the value associated with the key
func (d *ValueDictionary) Get(k Key) Value {
    d.lock.RLock()
    defer d.lock.RUnlock()
    return d.items[k]
}

// Clear removes all the items from the dictionary
func (d *ValueDictionary) Clear() {
    d.lock.Lock()
    defer d.lock.Unlock()
    d.items = make(map[Key]Value)
}

// Size returns the amount of elements in the dictionary
func (d *ValueDictionary) Size() int {
    d.lock.RLock()
    defer d.lock.RUnlock()
    return len(d.items)
}

// Keys returns a slice of all the keys present
func (d *ValueDictionary) Keys() []Key {
    d.lock.RLock()
    defer d.lock.RUnlock()
    keys := []Key{}
    for i := range d.items {
        keys = append(keys, i)
    }
    return keys
}

// Values returns a slice of all the values present
func (d *ValueDictionary) Values() []Value {
    d.lock.RLock()
    defer d.lock.RUnlock()
    values := []Value{}
    for i := range d.items {
        values = append(values, d.items[i])
    }
    return values
}
```

Tests

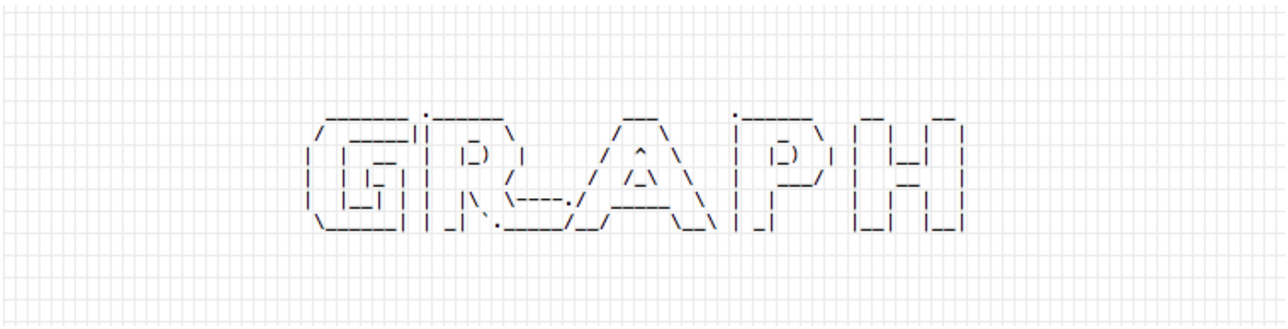
The tests describe the usage of the above implementation. Notice that we never interact with the underlying `map` type, which might as well be implemented in another way if only Go didn’t provide us a map type already.

```go
package dictionary

import (
    "fmt"
    "testing"
)

func populateDictionary(count int, start int) *ValueDictionary {
    dict := ValueDictionary{}
    for i := start; i < (start + count); i++ {
        dict.Set(fmt.Sprintf("key%d", i), fmt.Sprintf("value%d", i))
    }
    return &dict
}

func TestSet(t *testing.T) {
    dict := populateDictionary(3, 0)
    if size := dict.Size(); size != 3 {
        t.Errorf("wrong count, expected 3 and got %d", size)
    }
    dict.Set("key1", "value1") //should not add a new one, just change the existing one
    if size := dict.Size(); size != 3 {
        t.Errorf("wrong count, expected 3 and got %d", size)
    }
    dict.Set("key4", "value4") //should add it
    if size := dict.Size(); size != 4 {
        t.Errorf("wrong count, expected 4 and got %d", size)
    }
}

func TestDelete(t *testing.T) {
    dict := populateDictionary(3, 0)
    dict.Delete("key2")
    if size := dict.Size(); size != 2 {
        t.Errorf("wrong count, expected 2 and got %d", size)
    }
}

func TestClear(t *testing.T) {
    dict := populateDictionary(3, 0)
    dict.Clear()
    if size := dict.Size(); size != 0 {
        t.Errorf("wrong count, expected 0 and got %d", size)
    }
}

func TestHas(t *testing.T) {
    dict := populateDictionary(3, 0)
    has := dict.Has("key2")
    if !has {
        t.Errorf("expected key2 to be there")
    }
    dict.Delete("key2")
    has = dict.Has("key2")
    if has {
        t.Errorf("expected key2 to be removed")
    }
    dict.Delete("key1")
    has = dict.Has("key1")
    if has {
        t.Errorf("expected key1 to be removed")
    }
}

func TestKeys(t *testing.T) {
    dict := populateDictionary(3, 0)
    items := dict.Keys()
    if len(items) != 3 {
        t.Errorf("wrong count, expected 3 and got %d", len(items))
    }
    dict = populateDictionary(520, 0)
    items = dict.Keys()
    if len(items) != 520 {
        t.Errorf("wrong count, expected 520 and got %d", len(items))
    }
}

func TestValues(t *testing.T) {
    dict := populateDictionary(3, 0)
    items := dict.Values()
    if len(items) != 3 {
        t.Errorf("wrong count, expected 3 and got %d", len(items))
    }
    dict = populateDictionary(520, 0)
    items = dict.Values()
    if len(items) != 520 {
        t.Errorf("wrong count, expected 520 and got %d", len(items))
    }
}

func TestSize(t *testing.T) {
    dict := populateDictionary(3, 0)
    items := dict.Values()
    if len(items) != dict.Size() {
        t.Errorf("wrong count, expected %d and got %d", dict.Size(), len(items))
    }
    dict = populateDictionary(0, 0)
    items = dict.Values()
    if len(items) != dict.Size() {
        t.Errorf("wrong count, expected %d and got %d", dict.Size(), len(items))
    }
    dict = populateDictionary(10000, 0)
    items = dict.Values()
    if len(items) != dict.Size() {
        t.Errorf("wrong count, expected %d and got %d", dict.Size(), len(items))
    }
}
```

Creating a concrete dictionary data structure

You can use this generic implemenation to generate type-specific dictionaries, using

```
//generate a `IntDictionary` dictionary of `string` keys associated to `int` values
genny -in dictionary.go -out dictionary-string-int.go gen "Key=string Value=int"

//generate a `StringDictionary` dictionary of `string` keys associated to `string` values
genny -in dictionary.go -out dictionary-string-string.go gen "Key=string Value=string"
```



A **graph** is a representation of a network structure. There are tons of graph real world examples, the Internet and the social graph being the classic ones.

It's basically a set of **nodes** connected by **edges**.

I'll skip the mathematical concepts since you can find them everywhere and jump directly to a Go implementation of a graph.

Implementation

A graph data structure will expose those methods:

- `AddNode()` inserts a node
- `AddEdge()` adds an edge between two nodes

and `String()` for inspection purposes.

A Graph is defined as

```go
type ItemGraph struct {
    nodes []*Node
    edges map[Node][]*Node
    lock  sync.RWMutex
}
```

with Node being

```go
type Node struct {
    value Item
}
```

I'll implement an undirected graph, which means that adding an edge from A to B also adds an edge from B to A.

Implementation

```go
// Package graph creates a ItemGraph data structure for the Item type
package graph

import (
    "fmt"
    "sync"

    "github.com/cheekybits/genny/generic"
)

// Item the type of the binary search tree
type Item generic.Type

// Node a single node that composes the tree
type Node struct {
    value Item
}

func (n *Node) String() string {
    return fmt.Sprintf("%v", n.value)
}

// ItemGraph the Items graph
type ItemGraph struct {
    nodes []*Node
    edges map[Node][]*Node
    lock  sync.RWMutex
}

// AddNode adds a node to the graph
func (g *ItemGraph) AddNode(n *Node) {
    g.lock.Lock()
    g.nodes = append(g.nodes, n)
    g.lock.Unlock()
}

// AddEdge adds an edge to the graph
```

```
func (g *ItemGraph) AddEdge(n1, n2 *Node) {
    g.lock.Lock()
    if g.edges == nil {
        g.edges = make(map[Node][]*Node)
    }
    g.edges[*n1] = append(g.edges[*n1], n2)
    g.edges[*n2] = append(g.edges[*n2], n1)
    g.lock.Unlock()
}

// AddEdge adds an edge to the graph
func (g *ItemGraph) String() {
    g.lock.RLock()
    s := ""
    for i := 0; i < len(g.nodes); i++ {
        s += g.nodes[i].String() + " -> "
        near := g.edges[*g.nodes[i]]
        for j := 0; j < len(near); j++ {
            s += near[j].String() + " "
        }
        s += "\n"
    }
    fmt.Println(s)
    g.lock.RUnlock()
}
```

Quite straightforward. Here’s a test that when run, will populate the graph and print

```
$ go test
A -> B C D
B -> A E
C -> A E
D -> A
E -> B C F
F -> E

PASS
```

```
package graph

import (
    "fmt"
    "testing"
)

var g ItemGraph

func fillGraph() {
    nA := Node{"A"}
    nB := Node{"B"}
    nC := Node{"C"}
    nD := Node{"D"}
    nE := Node{"E"}
    nF := Node{"F"}
    g.AddNode(&nA)
    g.AddNode(&nB)
    g.AddNode(&nC)
    g.AddNode(&nD)
    g.AddNode(&nE)
    g.AddNode(&nF)

    g.AddEdge(&nA, &nB)
    g.AddEdge(&nA, &nC)
    g.AddEdge(&nB, &nE)
    g.AddEdge(&nC, &nE)
    g.AddEdge(&nE, &nF)
    g.AddEdge(&nD, &nA)
}

func TestAdd(t *testing.T) {
    fillGraph()
    g.String()
}
```

Traversing the graph: BFS

BFS (Breadth-First Search) is one of the most widely known algorithm to traverse a graph. Starting from a node, it first traverses all its directly linked nodes, then processes the nodes linked to those, and so on.

It's implemented using a queue, which is generated from my Go implementation of a queue data structure with code generation for the node type:

```
// This file was automatically generated by genny.
// Any changes will be lost if this file is regenerated.
// see https://github.com/cheekybits/genny
package graph

import "sync"

// NodeQueue the queue of Nodes
type NodeQueue struct {
    items []Node
    lock  sync.RWMutex
}

// New creates a new NodeQueue
func (s *NodeQueue) New() *NodeQueue {
    s.lock.Lock()
    s.items = []Node{}
    s.lock.Unlock()
    return s
}

// Enqueue adds an Node to the end of the queue
func (s *NodeQueue) Enqueue(t Node) {
    s.lock.Lock()
    s.items = append(s.items, t)
    s.lock.Unlock()
}

// Dequeue removes an Node from the start of the queue
func (s *NodeQueue) Dequeue() *Node {
    s.lock.Lock()
    item := s.items[0]
    s.items = s.items[1:len(s.items)]
    s.lock.Unlock()
    return &item
}

// Front returns the item next in the queue, without removing it
func (s *NodeQueue) Front() *Node {
    s.lock.RLock()
    item := s.items[0]
    s.lock.RUnlock()
    return &item
}

// IsEmpty returns true if the queue is empty
func (s *NodeQueue) IsEmpty() bool {
    s.lock.RLock()
    defer s.lock.RUnlock()
    return len(s.items) == 0
}

// Size returns the number of Nodes in the queue
func (s *NodeQueue) Size() int {
    s.lock.RLock()
    defer s.lock.RUnlock()
    return len(s.items)
}
```

Traverse method

Here’s the BFS implementation:

```
// Traverse implements the BFS traversing algorithm
func (g *ItemGraph) Traverse(f func(*Node)) {
    g.lock.RLock()
    q := NodeQueue{}
    q.New()
    n := g.nodes[0]
    q.Enqueue(*n)
    visited := make(map[*Node]bool)
    for {
        if q.IsEmpty() {
            break
        }
        node := q.Dequeue()
        visited[node] = true
        near := g.edges[*node]

        for i := 0; i < len(near); i++ {
            j := near[i]
            if !visited[j] {
                q.Enqueue(*j)
                visited[j] = true
            }
        }
        if f != nil {
            f(node)
        }
    }
    g.lock.RUnlock()
}
```

which can be tested with

```
func TestTraverse(t *testing.T) {
    g.Traverse(func(n *Node) {
        fmt.Printf("%v\n", n)
    })
}
```

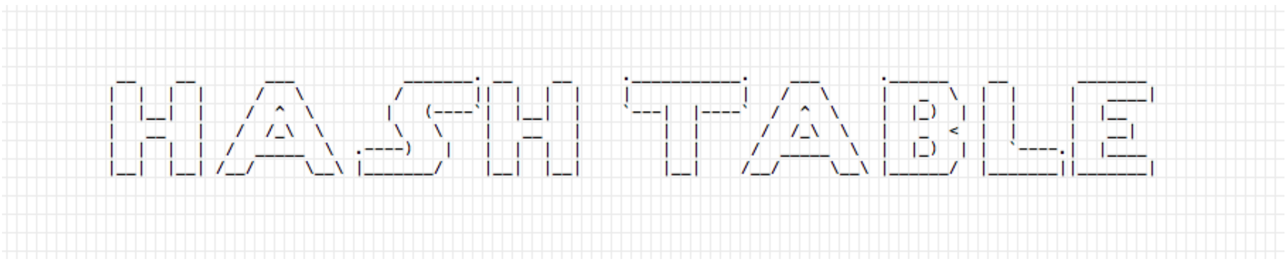
which after being added to our tests, will print the road it took to traverse all our nodes:

```
A
B
C
D
A
E
F
```

Creating a concrete graph data structure

You can use this generic implemenation to generate type-specific graphs, using

```
//generate a `IntGraph` graph of `int` values
genny -in graph.go -out graph-int.go gen "Item=int"

//generate a `StringGraph` graph of `string` values
genny -in graph.go -out graph-string.go gen "Item=string"
```



HASH TABLE

An Hash Table is a hash implementation of a hash table data structure. Instead of using a custom key to store the value in a map, the data structure performs a hashing function on the key to return the exact index of an item in the array.

Implementation

Internally the hash table will be represented with a `map` type, and I'll expose the

- `Put()`
- `Remove()`
- `Get()`
- `Size()`

methods.

I'll create a `ValueHashtable` generic type, concurrency safe, that can generate hash tables containing any type. Keys can be any type as long as it implements the `Stringer` interface. By running `go generate` the code will create a type-specific Hash Table implementation, encapsulating the actual value-specific data structure containing the data.

```
// Package hashtable creates a ValueHashtable data structure for the Item type
package hashtable
```

```go
import (
    "fmt"
    "sync"

    "github.com/cheekybits/genny/generic"
)

// Key the key of the dictionary
type Key generic.Type

// Value the content of the dictionary
type Value generic.Type

// ValueHashtable the set of Items
type ValueHashtable struct {
    items map[int]Value
    lock  sync.RWMutex
}

// the hash() private function uses the famous Horner's method
// to generate a hash of a string with O(n) complexity
func hash(k Key) int {
    key := fmt.Sprintf("%s", k)
    h := 0
    for i := 0; i < len(key); i++ {
        h = 31*h + int(key[i])
    }
    return h
}

// Put item with value v and key k into the hashtable
func (ht *ValueHashtable) Put(k Key, v Value) {
    ht.lock.Lock()
    defer ht.lock.Unlock()
    i := hash(k)
    if ht.items == nil {
        ht.items = make(map[int]Value)
    }
    ht.items[i] = v
}

// Remove item with key k from hashtable
func (ht *ValueHashtable) Remove(k Key) {
    ht.lock.Lock()
    defer ht.lock.Unlock()
    i := hash(k)
    delete(ht.items, i)
}

// Get item with key k from the hashtable
func (ht *ValueHashtable) Get(k Key) Value {
    ht.lock.RLock()
    defer ht.lock.RUnlock()
    i := hash(k)
    return ht.items[i]
}

// Size returns the number of the hashtable elements
func (ht *ValueHashtable) Size() int {
    ht.lock.RLock()
    defer ht.lock.RUnlock()
    return len(ht.items)
}
```

Tests

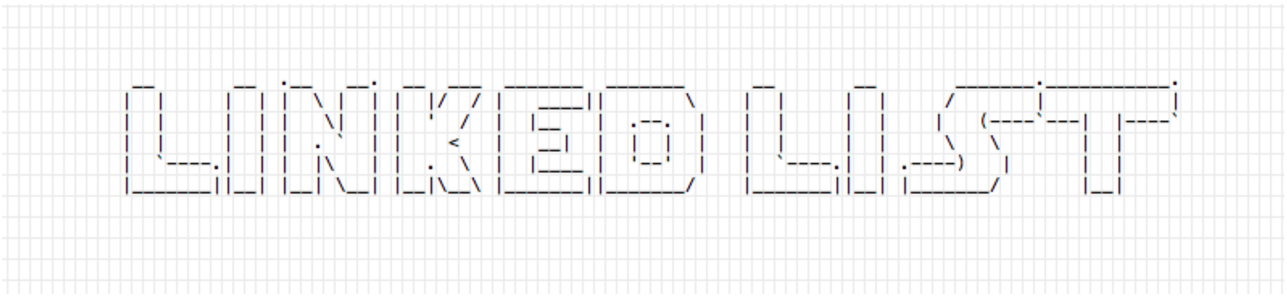
The tests describe the usage of the above implementation. Notice that we never interact with the underlying `map` type, which might as well be implemented in another way if only Go didn't provide us a map type already.

```go
package hashtable

import (
    "fmt"
    "testing"
)

func populateHashtable(count int, start int) *ValueHashtable {
    dict := ValueHashtable{}
    for i := start; i < (start + count); i++ {
        dict.Put(fmt.Sprintf("key%d", i), fmt.Sprintf("value%d", i))
    }
    return &dict
}

func TestPut(t *testing.T) {
    dict := populateHashtable(3, 0)
    if size := dict.Size(); size != 3 {
        t.Errorf("wrong count, expected 3 and got %d", size)
    }
    dict.Put("key1", "value1") //should not add a new one, just change the existing one
    if size := dict.Size(); size != 3 {
        t.Errorf("wrong count, expected 3 and got %d", size)
    }
    dict.Put("key4", "value4") //should add it
    if size := dict.Size(); size != 4 {
        t.Errorf("wrong count, expected 4 and got %d", size)
    }
}

func TestRemove(t *testing.T) {
    dict := populateHashtable(3, 0)
    dict.Remove("key2")
    if size := dict.Size(); size != 2 {
        t.Errorf("wrong count, expected 2 and got %d", size)
    }
}
```

Creating a concrete hash table data structure

You can use this generic implemenation to generate type-specific hash tables, using

```
//generate a `IntHashtable` hash table of `int` values
genny -in hashtable.go -out hashtable-int.go gen "Value=int"

//generate a `StringHashtable` hash table of `string` values
genny -in hashtable.go -out hashtable-string.go gen "Value=string"
```



A linked list provides a data structure similar to an array, but with the big advantage that inserting an element in the middle of the list is very cheap, compared to doing so in an array, where we need to shift all elements after the current position.

While arrays keep all the elements in the same block of memory, next to each other, linked lists can contain elements scattered around memory, by storing a pointer to the next element.

A disadvantage over arrays on the other hand is that if we want to pick an element in the middle of the list, we don't know its address, so we need to start at the beginning of the list, and iterate on the list until we find it.

Implementation

A linked list will provide these methods:

- `Append(t)` adds an Item t to the end of the linked list
- `Insert(i, t)` adds an Item t at position i
- `RemoveAt(i)` removes a node at position i
- `IndexOf()` returns the position of the Item t
- `IsEmpty()` returns true if the list is empty
- `Size()` returns the linked list size
- `String()` returns a string representation of the list
- `Head()` returns the first node, so we can iterate on it

I'll create an `ItemLinkedList` generic type, concurrency safe, that can generate linked lists containing any type by using `genny`, to create a type-specific linked list implementation, encapsulating the actual value-specific data structure containing the data.

```go
// Package linkedlist creates a ItemLinkedList data structure for the Item type
package linkedlist

import (
    "fmt"
    "sync"

    "github.com/cheekybits/genny/generic"
)

// Item the type of the linked list
type Item generic.Type

// Node a single node that composes the list
type Node struct {
    content Item
    next    *Node
}

// ItemLinkedList the linked list of Items
type ItemLinkedList struct {
    head *Node
    size int
    lock sync.RWMutex
}

// Append adds an Item to the end of the linked list
func (ll *ItemLinkedList) Append(t Item) {
    ll.lock.Lock()
    node := Node{t, nil}
    if ll.head == nil {
        ll.head = &node
    } else {
        last := ll.head
        for {
            if last.next == nil {
                break
            }
            last = last.next
        }
        last.next = &node
    }
    ll.size++
    ll.lock.Unlock()
}

// Insert adds an Item at position i
func (ll *ItemLinkedList) Insert(i int, t Item) error {
    ll.lock.Lock()
    defer ll.lock.Unlock()
    if i < 0 || i > ll.size {
        return fmt.Errorf("Index out of bounds")
    }
    addNode := Node{t, nil}
    if i == 0 {
        addNode.next = ll.head
        ll.head = &addNode
        return nil
    }
    node := ll.head
    j := 0
    for j < i-2 {
        j++
        node = node.next
    }
    addNode.next = node.next
    node.next = &addNode
    ll.size++
    return nil
}

// RemoveAt removes a node at position i
func (ll *ItemLinkedList) RemoveAt(i int) (*Item, error) {
    ll.lock.Lock()
    defer ll.lock.Unlock()
    if i < 0 || i > ll.size {
        return nil, fmt.Errorf("Index out of bounds")
    }
    node := ll.head
    j := 0
    for j < i-1 {
        j++
        node = node.next
    }
    remove := node.next
    node.next = remove.next
    ll.size--
    return &remove.content, nil
}

// IndexOf returns the position of the Item t
func (ll *ItemLinkedList) IndexOf(t Item) int {
    ll.lock.RLock()
    defer ll.lock.RUnlock()
    node := ll.head
    j := 0
    for {
        if node.content == t {
            return j
        }
        if node.next == nil {
            return -1
        }
        node = node.next
        j++
    }
}

// IsEmpty returns true if the list is empty
func (ll *ItemLinkedList) IsEmpty() bool {
    ll.lock.RLock()
    defer ll.lock.RUnlock()
    if ll.head == nil {
        return true
    }
    return false
}

// Size returns the linked list size
func (ll *ItemLinkedList) Size() int {
    ll.lock.RLock()
    defer ll.lock.RUnlock()
    size := 1
    last := ll.head
    for {
        if last == nil || last.next == nil {
            break
        }
        last = last.next
        size++
    }
    return size
}

// Insert adds an Item at position i
func (ll *ItemLinkedList) String() {
    ll.lock.RLock()
    defer ll.lock.RUnlock()
    node := ll.head
    j := 0
    for {
        if node == nil {
            break
        }
        j++
        fmt.Print(node.content)
        fmt.Print(" ")
        node = node.next
    }
    fmt.Println()
}

// Head returns a pointer to the first node of the list
func (ll *ItemLinkedList) Head() *Node {
    ll.lock.RLock()
    defer ll.lock.RUnlock()
    return ll.head
}
```

Tests

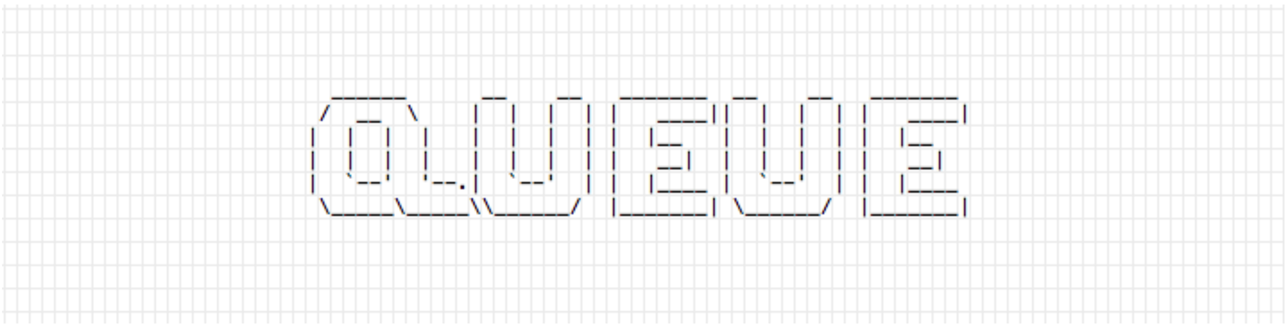
The tests describe the usage of the above implementation.

```
package linkedlist

import (
    "fmt"
    "testing"
)

var ll ItemLinkedList

func TestAppend(t *testing.T) {
    if !ll.IsEmpty() {
        t.Errorf("list should be empty")
    }

    ll.Append("first")
    if ll.IsEmpty() {
        t.Errorf("list should not be empty")
    }

    if size := ll.Size(); size != 1 {
        t.Errorf("wrong count, expected 1 and got %d", size)
    }

    ll.Append("second")
    ll.Append("third")

    if size := ll.Size(); size != 3 {
        t.Errorf("wrong count, expected 3 and got %d", size)
    }
}

func TestRemoveAt(t *testing.T) {
    _, err := ll.RemoveAt(1)
    if err != nil {
        t.Errorf("unexpected error %s", err)
    }

    if size := ll.Size(); size != 2 {
        t.Errorf("wrong count, expected 2 and got %d", size)
    }
}

func TestInsert(t *testing.T) {
    //test inserting in the middle
    err := ll.Insert(2, "second2")
    if err != nil {
        t.Errorf("unexpected error %s", err)
    }
    if size := ll.Size(); size != 3 {
        t.Errorf("wrong count, expected 3 and got %d", size)
    }

    //test inserting at head position
    err = ll.Insert(0, "zero")
    if err != nil {
        t.Errorf("unexpected error %s", err)
    }
}

func TestIndexOf(t *testing.T) {
    if i := ll.IndexOf("zero"); i != 0 {
        t.Errorf("expected position 0 but got %d", i)
    }
    if i := ll.IndexOf("first"); i != 1 {
        t.Errorf("expected position 1 but got %d", i)
    }
    if i := ll.IndexOf("second2"); i != 2 {
        t.Errorf("expected position 2 but got %d", i)
    }
    if i := ll.IndexOf("third"); i != 3 {
        t.Errorf("expected position 3 but got %d", i)
    }
}

func TestHead(t *testing.T) {
    h := ll.Head()
    if "zero" != fmt.Sprint(h.content) {
        t.Errorf("Expected `zero` but got %s", fmt.Sprint(h.content))
    }
}
```

Creating a concrete linked list data structure

You can use this generic implemenation to generate type-specific linked lists, using

```
//generate a `IntLinkedList` linked list of `int` values
genny -in linkedlist.go -out linkedlist-int.go gen "Item=int"

//generate a `StringLinkedList` linked list of `string` values
genny -in linkedlist.go -out linkedlist-string.go gen "Item=string"
```



A queue is a an ordered collection of items following the First-In-First-Out (FIFO) principle. We add items from one end of the queue, and we retrieve items from the other end.

Queue applications have all sorts of uses, which can be easily inferred from real-world queues. It's not hard to imagine a queue representation of a supermarket line or the queue to get into a concert hall.

Implementation

Internally the queue will be represented with a `slice` type, and I'll expose the

- `New()`
- `Enqueue()`
- `Dequeue()`
- `Front()`
- `IsEmpty()`
- `Size()`

methods.

`New()` serves as the constructor, which initializes the internal slice when we start using it.

I'll create an `ItemQueue` generic type, concurrency safe, that can generate queues containing any type by using `genny`, to create a type-specific queue implementation, encapsulating the actual value-specific data structure containing the data.

```go
// Package queue creates a ItemQueue data structure for the Item type
package queue

import (
    "sync"

    "github.com/cheekybits/genny/generic"
)

// Item the type of the queue
type Item generic.Type

// ItemQueue the queue of Items
type ItemQueue struct {
    items []Item
    lock  sync.RWMutex
}

// New creates a new ItemQueue
func (s *ItemQueue) New() *ItemQueue {
    s.items = []Item{}
    return s
}

// Enqueue adds an Item to the end of the queue
func (s *ItemQueue) Enqueue(t Item) {
    s.lock.Lock()
    s.items = append(s.items, t)
    s.lock.Unlock()
}

// Dequeue removes an Item from the start of the queue
func (s *ItemQueue) Dequeue() *Item {
    s.lock.Lock()
    item := s.items[0]
    s.items = s.items[1:len(s.items)]
    s.lock.Unlock()
    return &item
}

// Front returns the item next in the queue, without removing it
func (s *ItemQueue) Front() *Item {
    s.lock.RLock()
    item := s.items[0]
    s.lock.RUnlock()
    return &item
}

// IsEmpty returns true if the queue is empty
func (s *ItemQueue) IsEmpty() bool {
    return len(s.items) == 0
}

// Size returns the number of Items in the queue
func (s *ItemQueue) Size() int {
    return len(s.items)
}
```

Tests

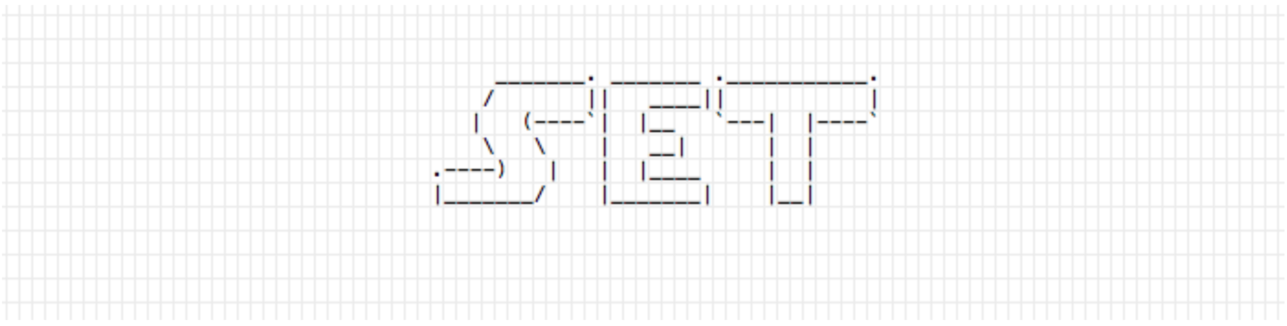
The tests describe the usage of the above implementation.

```go
package queue

import (
    "testing"
)

var s ItemQueue

func initQueue() *ItemQueue {
    if s.items == nil {
        s = ItemQueue{}
        s.New()
    }
    return &s
}

func TestEnqueue(t *testing.T) {
    s := initQueue()
    s.Enqueue(1)
    s.Enqueue(2)
    s.Enqueue(3)

    if size := s.Size(); size != 3 {
        t.Errorf("wrong count, expected 3 and got %d", size)
    }
}

func TestDequeue(t *testing.T) {
    s.Dequeue()
    if size := len(s.items); size != 2 {
        t.Errorf("wrong count, expected 2 and got %d", size)
    }

    s.Dequeue()
    s.Dequeue()
    if size := len(s.items); size != 0 {
        t.Errorf("wrong count, expected 0 and got %d", size)
    }

    if !s.IsEmpty() {
        t.Errorf("IsEmpty should return true")
    }
}
```

Creating a concrete queue data structure

You can use this generic implemenation to generate type-specific queues, using

```
//generate a `IntQueue` queue of `int` values
genny -in queue.go -out queue-int.go gen "Item=int"

//generate a `StringQueue` queue of `string` values
genny -in queue.go -out queue-string.go gen "Item=string"
```



A Set is a collection of values. You can iterate over those values, add new values, remove values and clear the set, get the set size, and check if the set contains an item. A value in the set might only be stored once, duplicates are not possible.

First implementation

Here is a simple implementation of the set, not yet concurrency safe, without locking resources for the benefit of simplicity and understanding. I'll add locking later in the article.

Notice the second line, which allows us to use the Set for any type we want, by generating a specific implementation of this generic data structure.

set.go

```go
// Package set creates a ItemSet data structure for the Item type
package set

import "github.com/cheekybits/genny/generic"

// Item the type of the Set
type Item generic.Type

// ItemSet the set of Items
type ItemSet struct {
    items map[Item]bool
}

// Add adds a new element to the Set. Returns a pointer to the Set.
func (s *ItemSet) Add(t Item) *ItemSet {
    if s.items == nil {
        s.items = make(map[Item]bool)
    }
    _, ok := s.items[t]
    if !ok {
        s.items[t] = true
    }
    return s
}

// Clear removes all elements from the Set
func (s *ItemSet) Clear() {
    s.items = make(map[Item]bool)
}

// Delete removes the Item from the Set and returns Has(Item)
func (s *ItemSet) Delete(item Item) bool {
    _, ok := s.items[item]
    if ok {
        delete(s.items, item)
    }
    return ok
}

// Has returns true if the Set contains the Item
func (s *ItemSet) Has(item Item) bool {
    _, ok := s.items[item]
    return ok
}

// Items returns the Item(s) stored
func (s *ItemSet) Items() []Item {
    items := []Item{}
    for i := range s.items {
        items = append(items, i)
    }
    return items
}

// Size returns the size of the set
func (s *ItemSet) Size() int {
    return len(s.items)
}
```

Testing the implementation

Here is the test suite for the above code, which explains how to use it in details, and the expected results for any operation:

> *set_test.go*

```go
package set

import (
    "fmt"
    "testing"
)

func populateSet(count int, start int) *ItemSet {
    set := ItemSet{}
    for i := start; i < (start + count); i++ {
        set.Add(fmt.Sprintf("item%d", i))
    }
    return &set
}

func TestAdd(t *testing.T) {
    set := populateSet(3, 0)
    if size := set.Size(); size != 3 {
        t.Errorf("wrong count, expected 3 and got %d", size)
    }
    set.Add("item1") //should not add it, already there
    if size := set.Size(); size != 3 {
        t.Errorf("wrong count, expected 3 and got %d", size)
    }
    set.Add("item4") //should not add it, already there
    if size := set.Size(); size != 4 {
        t.Errorf("wrong count, expected 4 and got %d", size)
    }
}

func TestClear(t *testing.T) {
    set := populateSet(3, 0)
    set.Clear()
    if size := set.Size(); size != 0 {
        t.Errorf("wrong count, expected 0 and got %d", size)
    }
}

func TestDelete(t *testing.T) {
    set := populateSet(3, 0)
    set.Delete("item2")
    if size := set.Size(); size != 2 {
        t.Errorf("wrong count, expected 2 and got %d", size)
    }
}
```

```go
func TestHas(t *testing.T) {
    set := populateSet(3, 0)
    has := set.Has("item2")
    if !has {
        t.Errorf("expected item2 to be there")
    }
    set.Delete("item2")
    has = set.Has("item2")
    if has {
        t.Errorf("expected item2 to be removed")
    }
    set.Delete("item1")
    has = set.Has("item1")
    if has {
        t.Errorf("expected item1 to be removed")
    }
}

func TestItems(t *testing.T) {
    set := populateSet(3, 0)
    items := set.Items()
    if len(items) != 3 {
        t.Errorf("wrong count, expected 3 and got %d", len(items))
    }
    set = populateSet(520, 0)
    items = set.Items()
    if len(items) != 520 {
        t.Errorf("wrong count, expected 520 and got %d", len(items))
    }
}

func TestSize(t *testing.T) {
    set := populateSet(3, 0)
    items := set.Items()
    if len(items) != set.Size() {
        t.Errorf("wrong count, expected %d and got %d", set.Size(), len(items))
    }
    set = populateSet(0, 0)
    items = set.Items()
    if len(items) != set.Size() {
        t.Errorf("wrong count, expected %d and got %d", set.Size(), len(items))
    }
    set = populateSet(10000, 0)
    items = set.Items()
    if len(items) != set.Size() {
        t.Errorf("wrong count, expected %d and got %d", set.Size(), len(items))
    }
}
```

Concurrency safe version

The first version is not concurrency safe because a routine might add an item to the set while another routine is getting the list of items, or the size.

The following code adds a `sync.RWMutex` to the data structure, making it concurrency safe. The above tests are running fine without any modification to this implementation as well.

The implementation is very simple and we're good with simply adding a lock and unlocking with a `defer`. Since we'll generate the code for specific implementations which might be more than one in the same file, we need to either add the lock inside the struct, or add the generic type into the lock name (`Itemlock`). I chose the first option:

> *set.go*

```go
// Package set creates a ItemSet data structure for the Item type
package set

import (
	"sync"

	"github.com/cheekybits/genny/generic"
)

// Item the type of the Set
type Item generic.Type

// ItemSet the set of Items
type ItemSet struct {
	items map[Item]bool
	lock  sync.RWMutex
}

// Add adds a new element to the Set. Returns a pointer to the Set.
func (s *ItemSet) Add(t Item) *ItemSet {
	s.lock.Lock()
	defer s.lock.Unlock()
	if s.items == nil {
		s.items = make(map[Item]bool)
	}
	_, ok := s.items[t]
	if !ok {
		s.items[t] = true
	}
	return s
}

// Clear removes all elements from the Set
func (s *ItemSet) Clear() {
	s.lock.Lock()
```

```
    defer s.lock.Unlock()
    s.items = make(map[Item]bool)
}

// Delete removes the Item from the Set and returns Has(Item)
func (s *ItemSet) Delete(item Item) bool {
    s.lock.Lock()
    defer s.lock.Unlock()
    _, ok := s.items[item]
    if ok {
        delete(s.items, item)
    }
    return ok
}

// Has returns true if the Set contains the Item
func (s *ItemSet) Has(item Item) bool {
    s.lock.RLock()
    defer s.lock.RUnlock()
    _, ok := s.items[item]
    return ok
}

// Items returns the Item(s) stored
func (s *ItemSet) Items() []Item {
    s.lock.RLock()
    defer s.lock.RUnlock()
    items := []Item{}
    for i := range s.items {
        items = append(items, i)
    }
    return items
}

// Size returns the size of the set
func (s *ItemSet) Size() int {
    s.lock.RLock()
    defer s.lock.RUnlock()
    return len(s.items)
}
```

Creating a concrete set data structure

Install `genny` if you don't have it already.

Run

```
//generate a `StringSet` set of `string`s
genny -in set.go -out set-string.go gen "Iten=string Value=int"

//generate a `IntSet` set of `int`s
genny -in set.go -out set-int.go gen "Item=int"
```

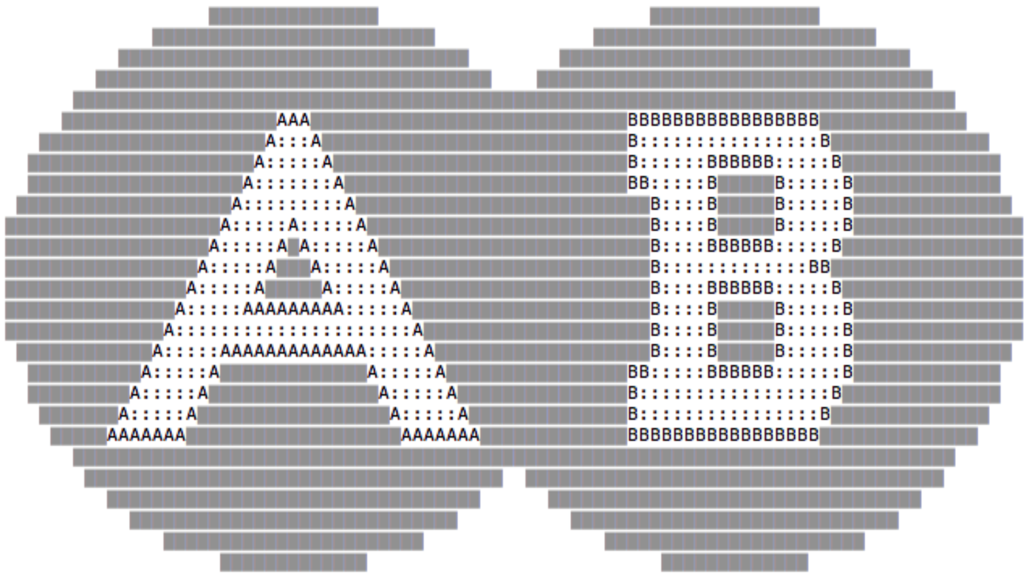
Here's an example of the `set-string.go` generated file:

```go
// This file was automatically generated by genny.
// Any changes will be lost if this file is regenerated.
// see https://github.com/cheekybits/genny

// Package set creates a StringSet data structure for the string type
package set

import "sync"

// StringSet the set of Strings
type StringSet struct {
    items map[string]bool
    lock  sync.RWMutex
}

// Add adds a new element to the Set. Returns a pointer to the Set.
func (s *StringSet) Add(t string) *StringSet {
    s.lock.Lock()
    defer s.lock.Unlock()
    if s.items == nil {
        s.items = make(map[string]bool)
    }
    _, ok := s.items[t]
    if !ok {
        s.items[t] = true
    }
    return s
}

// Clear removes all elements from the Set
func (s *StringSet) Clear() {
    s.lock.Lock()
    defer s.lock.Unlock()
    s.items = make(map[string]bool)
}

// Delete removes the string from the Set and returns Has(string)
func (s *StringSet) Delete(item string) bool {
    s.lock.Lock()
    defer s.lock.Unlock()
    _, ok := s.items[item]
    if ok {
        delete(s.items, item)
    }
    return ok
}

// Has returns true if the Set contains the string
func (s *StringSet) Has(item string) bool {
    s.lock.RLock()
    defer s.lock.RUnlock()
    _, ok := s.items[item]
    return ok
}

// Strings returns the string(s) stored
func (s *StringSet) Strings() []string {
    s.lock.RLock()
    defer s.lock.RUnlock()
    items := []string{}
    for i := range s.items {
        items = append(items, i)
    }
    return items
}

// Size returns the size of the set
func (s *StringSet) Size() int {
    s.lock.RLock()
    defer s.lock.RUnlock()
    return len(s.items)
}

// Package set creates a IntSet data structure for the int type

// IntSet the set of Ints
type IntSet struct {
    items map[int]bool
    lock  sync.RWMutex
}

// Add adds a new element to the Set. Returns a pointer to the Set.
func (s *IntSet) Add(t int) *IntSet {
    s.lock.Lock()
    defer s.lock.Unlock()
    if s.items == nil {
        s.items = make(map[int]bool)
    }
    _, ok := s.items[t]
    if !ok {
        s.items[t] = true
    }
    return s
}

// Clear removes all elements from the Set
func (s *IntSet) Clear() {
    s.lock.Lock()
    defer s.lock.Unlock()
    s.items = make(map[int]bool)
}

// Delete removes the int from the Set and returns Has(int)
func (s *IntSet) Delete(item int) bool {
    s.lock.Lock()
    defer s.lock.Unlock()
    _, ok := s.items[item]
    if ok {
        delete(s.items, item)
    }
    return ok
}

// Has returns true if the Set contains the int
func (s *IntSet) Has(item int) bool {
    s.lock.RLock()
    defer s.lock.RUnlock()
    _, ok := s.items[item]
    return ok
}

// Ints returns the int(s) stored
func (s *IntSet) Ints() []int {
    s.lock.RLock()
    defer s.lock.RUnlock()
    items := []int{}
    for i := range s.items {
        items = append(items, i)
    }
    return items
}

// Size returns the size of the set
func (s *IntSet) Size() int {
    s.lock.RLock()
    defer s.lock.RUnlock()
    return len(s.items)
}
```

Add more Set operations

Our Set is an interesting data structure at this point, but it can be improved a lot more by implementing some common mathematical set operations: `union`, `intersection`, `difference` and `subset`.

Union

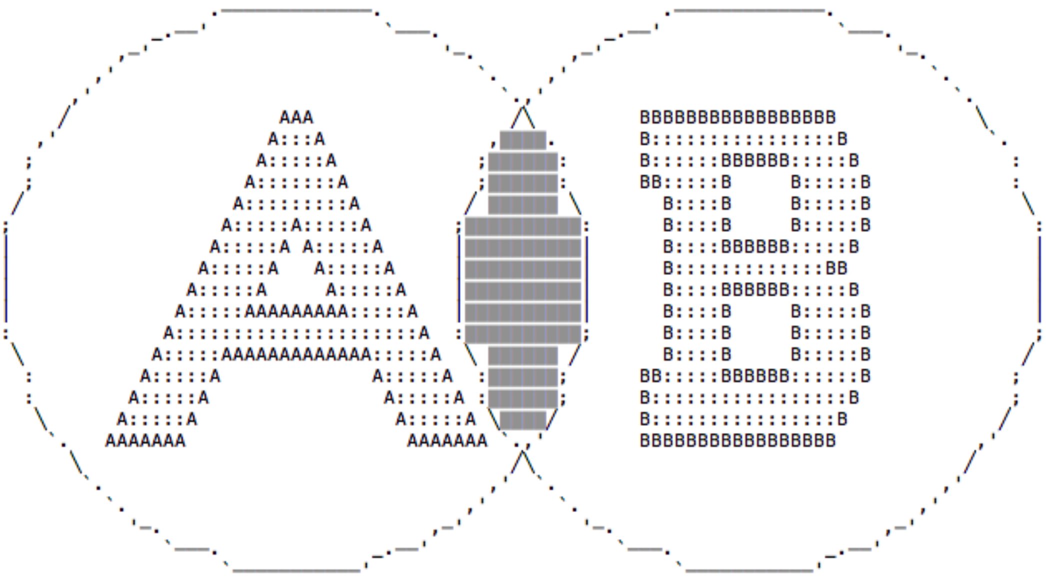


```
// Union returns a new set with elements from both
// the given sets
func (s *ItemSet) Union(s2 *ItemSet) *ItemSet {
    s3 := ItemSet{}
    s3.items = make(map[Item]bool)
    s.lock.RLock()
    for i := range s.items {
        s3.items[i] = true
    }
    s.lock.RUnlock()
    s2.lock.RLock()
    for i := range s2.items {
        _, ok := s3.items[i]
        if !ok {
            s3.items[i] = true
        }
    }
    s2.lock.RUnlock()
    return &s3
}
```

Test

```
func TestUnion(t *testing.T) {
    set1 := populateSet(3, 0)
    set2 := populateSet(2, 3)

    set3 := set1.Union(set2)

    if len(set3.Items()) != 5 {
        t.Errorf("wrong count, expected 5 and got %d", set3.Size())
    }
    //don't edit original sets
    if len(set1.Items()) != 3 {
        t.Errorf("wrong count, expected 3 and got %d", set1.Size())
    }
    if len(set2.Items()) != 2 {
        t.Errorf("wrong count, expected 2 and got %d", set2.Size())
    }
}
```

Intersection

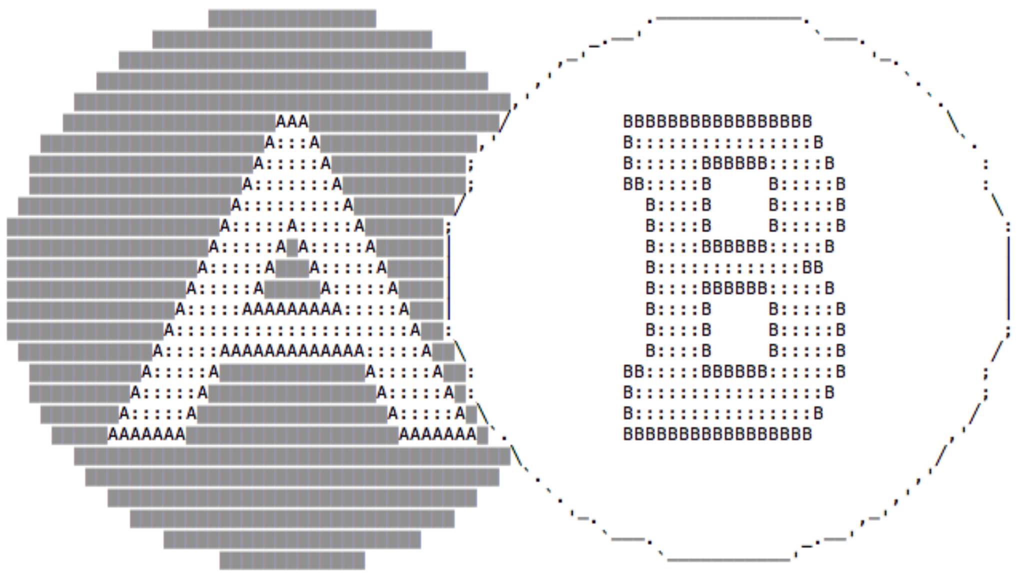


```
// Intersection returns a new set with elements that exist in
// both sets
func (s *ItemSet) Intersection(s2 *ItemSet) *ItemSet {
    s3 := ItemSet{}
    s3.items = make(map[Item]bool)
    s.lock.RLock()
    s2.lock.RLock()
    defer s.lock.RUnlock()
    defer s2.lock.RUnlock()
    for i := range s2.items {
        _, ok := s.items[i]
        if ok {
            s3.items[i] = true
        }
    }
    return &s3
}
```

Test

```
func TestIntersection(t *testing.T) {
    set1 := populateSet(3, 0)
    set2 := populateSet(2, 0)

    set3 := set1.Intersection(set2)

    if len(set3.Items()) != 2 {
        t.Errorf("wrong count, expected 2 and got %d", set3.Size())
    }
    //don't edit original sets
    if len(set1.Items()) != 3 {
        t.Errorf("wrong count, expected 3 and got %d", set1.Size())
    }
    if len(set2.Items()) != 2 {
        t.Errorf("wrong count, expected 2 and got %d", set2.Size())
    }
}
```

Difference

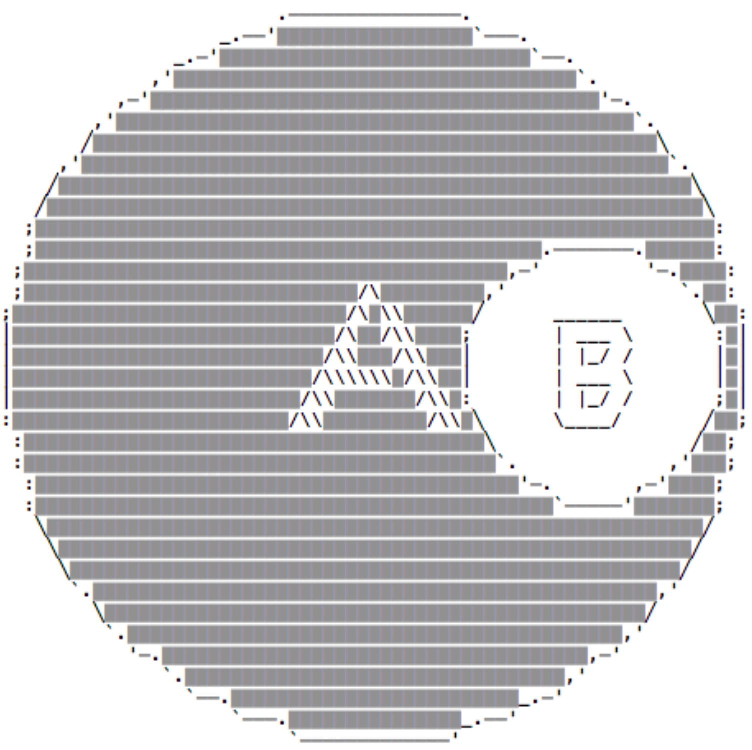


```
// Difference returns a new set with all the elements that
// exist in the first set and don't exist in the second set
func (s *ItemSet) Difference(s2 *ItemSet) *ItemSet {
    s3 := ItemSet{}
    s3.items = make(map[Item]bool)
    s.lock.RLock()
    s2.lock.RLock()
    defer s.lock.RUnlock()
    defer s2.lock.RUnlock()
    for i := range s.items {
        _, ok := s2.items[i]
        if !ok {
            s3.items[i] = true
        }
    }
    return &s3
}
```

Test

```
func TestDifference(t *testing.T) {
    set1 := populateSet(3, 0)
    set2 := populateSet(2, 0)

    set3 := set1.Difference(set2)

    if len(set3.Items()) != 1 {
        t.Errorf("wrong count, expected 2 and got %d", set3.Size())
    }
    //don't edit original sets
    if len(set1.Items()) != 3 {
        t.Errorf("wrong count, expected 3 and got %d", set1.Size())
    }
    if len(set2.Items()) != 2 {
        t.Errorf("wrong count, expected 2 and got %d", set2.Size())
    }
}
```

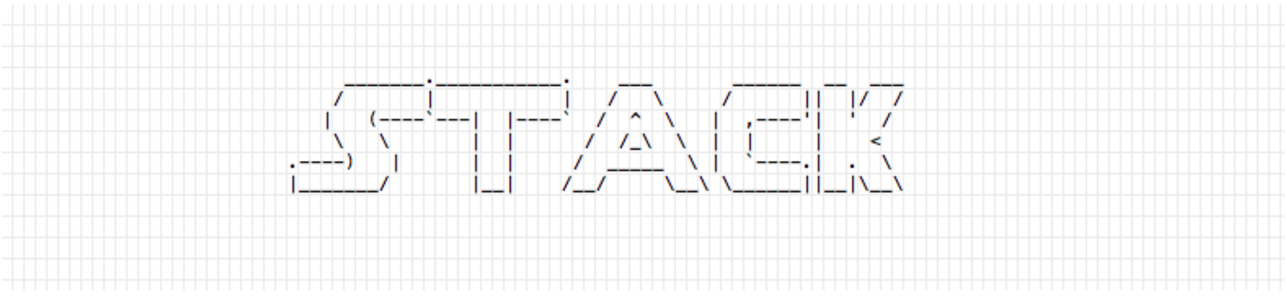
Subset



```
// Subset returns true if s is a subset of s2
func (s *ItemSet) Subset(s2 *ItemSet) bool {
    s.lock.RLock()
    s2.lock.RLock()
    defer s.lock.RUnlock()
    defer s2.lock.RUnlock()
    for i := range s.items {
        _, ok := s2.items[i]
        if !ok {
            return false
        }
    }
    return true
}
```

Test

```go
func TestSubset(t *testing.T) {
    set1 := populateSet(3, 0)
    set2 := populateSet(2, 0)

    if set1.Subset(set2) {
        t.Errorf("expected false and got true")
    }

    //don't edit original sets
    if len(set1.Items()) != 3 {
        t.Errorf("wrong count, expected 3 and got %d", set1.Size())
    }
    if len(set2.Items()) != 2 {
        t.Errorf("wrong count, expected 2 and got %d", set2.Size())
    }

    //try real subsets
    set1 = populateSet(2, 0)
    if !set1.Subset(set2) {
        t.Errorf("expected true and got false")
    }

    set1 = populateSet(1, 0)
    if !set1.Subset(set2) {
        t.Errorf("expected true and got false")
    }
}
```



A stack is an ordered collection of items following the Last-In-First-Out (LIFO) principle. We add and remove items from the same part of the pile, called the top. We cannot remove items from the base. Just like a pile of books.

Stacks have tons of uses, from creating a history of pages visited to commands entered and to store actions that can be undone.

Implementation

Internally the stack will be represented with a `slice` type, and I'll expose the

- `Push()`
- `Pull()`
- `New()`

methods.

`New()` serves as the constructor, which initializes the internal slice when we start using it.

I'll create an `ItemStack` generic type, concurrency safe, that can generate stacks containing any type by using `genny`, to create a type-specific stack implementation, encapsulating the actual value-specific data structure containing the data.

```go
// Package stack creates a ItemStack data structure for the Item type
package stack

import (
    "sync"

    "github.com/cheekybits/genny/generic"
)

// Item the type of the stack
type Item generic.Type

// ItemStack the stack of Items
type ItemStack struct {
    items []Item
    lock  sync.RWMutex
}

// New creates a new ItemStack
func (s *ItemStack) New() *ItemStack {
    s.items = []Item{}
    return s
}
```

```
// Push adds an Item to the top of the stack
func (s *ItemStack) Push(t Item) {
    s.lock.Lock()
    s.items = append(s.items, t)
    s.lock.Unlock()
}

// Pop removes an Item from the top of the stack
func (s *ItemStack) Pop() *Item {
    s.lock.Lock()
    item := s.items[len(s.items)-1]
    s.items = s.items[0 : len(s.items)-1]
    s.lock.Unlock()
    return &item
}
```

Tests

The tests describe the usage of the above implementation.

```
package stack

import (
    "testing"
)

var s ItemStack

func initStack() *ItemStack {
    if s.items == nil {
        s = ItemStack{}
        s.New()
    }
    return &s
}

func TestPush(t *testing.T) {
    s := initStack()
    s.Push(1)
    s.Push(2)
    s.Push(3)
    if size := len(s.items); size != 3 {
        t.Errorf("wrong count, expected 3 and got %d", size)
    }
}

func TestPop(t *testing.T) {
    s.Pop()
    if size := len(s.items); size != 2 {
        t.Errorf("wrong count, expected 2 and got %d", size)
    }

    s.Pop()
    s.Pop()
    if size := len(s.items); size != 0 {
        t.Errorf("wrong count, expected 0 and got %d", size)
    }
}
```

Creating a concrete stack data structure

You can use this generic implemenation to generate type-specific stacks, using

```
//generate a `IntStack` stack of `int` values
genny -in stack.go -out stack-int.go gen "Item=int"

//generate a `StringStack` stack of `string` values
genny -in stack.go -out stack-string.go gen "Item=string"
```