Go Containers

January 23, 2014

John Graham-Cumming

Six interesting containers

- From `pkg/container`
  - `container/list`
  - `container/ring`
  - `container/heap`
- Built in
  - `map`
  - slice
- Channels as queues

container/list

- Doubly-linked list implementation
- Uses `interface{}` for values

```
l  := list.New()
e0  := l.PushBack(42)
e1  := l.PushFront(13)
e2  := l.PushBack(7)
l.InsertBefore(3, e0)
l.InsertAfter(196, e1)
l.InsertAfter(1729, e2)

for e := l.Front(); e != nil; e = e.Next() {
  fmt.Printf("%d ", e.Value.(int))
}
fmt.Printf("\n")
```

Pity there's no 'map' function

`e.Value` to get the stored value

```
13 196 3 42 7 1729
```

container/list

All work on elements not values

```
l.MoveToFront(e2)
l.MoveToBack(e1)
l.Remove(e0)
```

Returns the value removed

```
for e := l.Front(); e != nil; e = e.Next() {
  fmt.Printf("%d ", e.Value.(int))
}
fmt.Printf("\n")
```

```
7 196 3 1729 13
```

container/ring

- A circular ‘list’

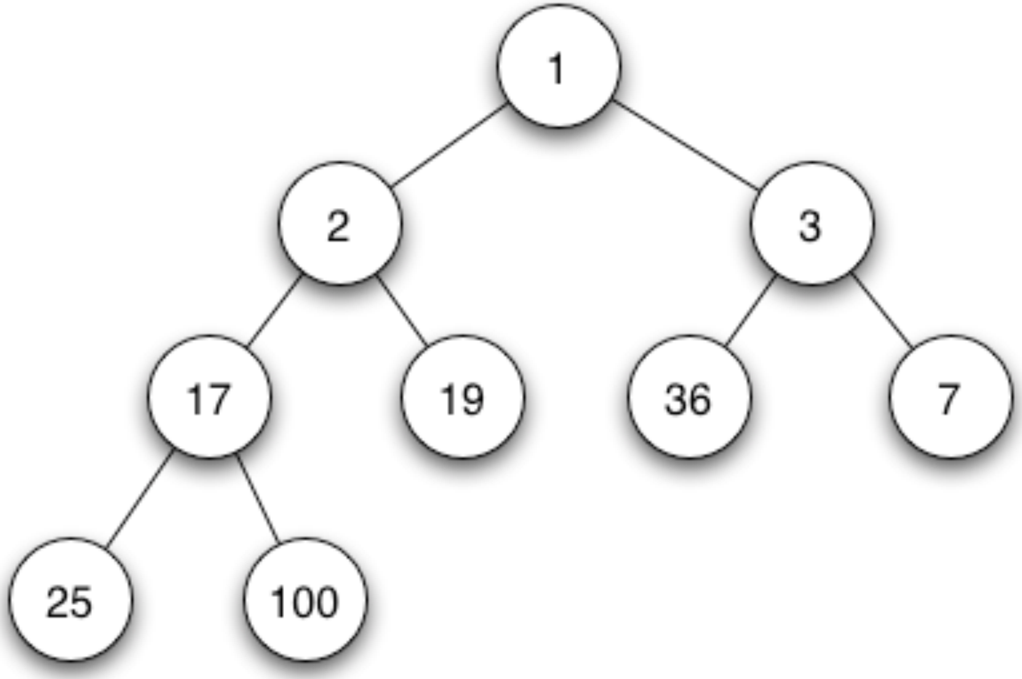
```
parus := []string{”major", “holsti”, “carpi”}

r := ring.New(len(parus))
for i := 0; i < r.Len(); i++ {
  r.Value = parus[i]
  r = r.Next()
}

r.Do(func(x interface{}) {
  fmt.Printf(“Parus %s\n”, x.(string))
})
```

- Move `n` elements through ring

```
r.Move(n)
```

container/heap

- Implements a “min-heap” (i.e. tree where each node is the “minimum” element in its subtree)



- Needs a notion of “Less” and a way to “Swap”

container/heap

- The single most confusing definition in all of Go

```
type Interface interface {
  sort.Interface
  Push(x interface{}) // add x as element Len()
  Pop() interface{}   // remove/return element Len()-1
}

// Note that Push and Pop in this interface are for
// package heap's implementation to call. To add and
// remove things from the heap, use heap.Push and
// heap.Pop.
```

container/heap

- Simple example

```
type OrderedInts []int

func (h OrderedInts) Len() int { return len(h) }
func (h OrderedInts) Less(i, j int) bool {
    return h[i] < h[j]
}
func (h OrderedInts) Swap(i,j int) {h[i],h[j]=h[j],h[i]}
func (h *OrderedInts) Push(x interface{}) {
       *h = append(*h, x.(int))
}
func (h *OrderedInts) Pop() interface{} {
       old := *h
       n := len(old)-1
       x := old[n]
       *h = old[:n]
       return x
}
```

container/heap

- Using a heap

```
h := &OrderedInts{33,76,55,24,48,63,86,83,83,12}

heap.Init(h)

fmt.Printf("min: %d\n", (*h)[0])

for h.Len() > 0 {
  fmt.Printf("%d ", heap.Pop(h))
}

fmt.Printf(“\n”)
```

container/heap

- Heaps are useful for...
  - Make a priority queue
  - Sorting
  - Graph algorithms

MAP

map

- Maps are typed

```
dictionary := make(map[string]string)
dictionary := map[string]string{}
```

- They are not concurrency safe
  - Use a lock or channel for concurrent read/write access

```
counts := struct{
    sync.RWMutex
    m map[string]int
}{m: make(map[string]int)}

counts.RLock()
fmt.Printf(“foo count”, counts.m[“foo”]
counts.RUnlock()

counts.Lock()
counts.m[“foo”] += num_foos
counts.Unlock()
```

Multiple readers, one writer

map iteration

```
m := map[string]int{
   "bar”: 54,
   "foo”: 42,
   "baz”: -1,
}

for k := range m {
   // k is foo, bar, baz
}

for _, v := range m {
   // v is 54, 42, -1 in some order
}

for k, v := range m {
   // k and v are as above
}
```

Order of iteration is undefined

Common `map` operations

- Remove an element

```
delete(dictionary, “twerking”)
```

- Test presence of an element

```
definition, present := dictionary[“hoopy”]

_, present := dictionary[“sigil”]
```

- Missing element gives a “zero” value

```
fmt.Printf(“[%s]\n”, dictionary[“ewyfgwyegfweygf”])

[]
```

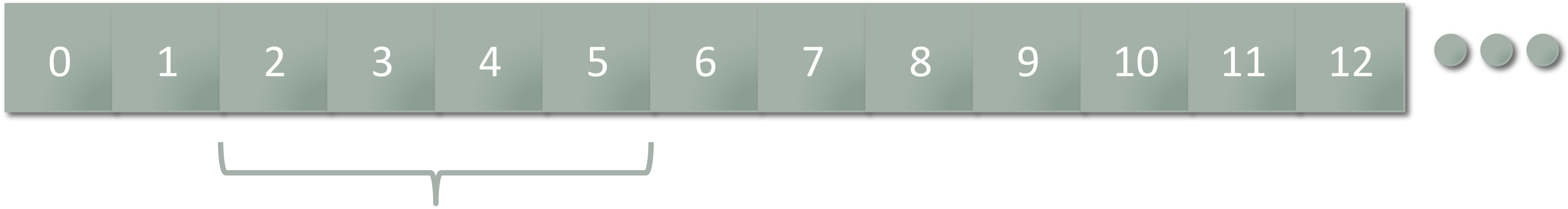
SLICE

Slices

- A slice is part of an array

```
var arrayOfInts [256]int

var part []int = arrayOfInts[2:6]
```

- `arrayOfInts` is 256 `ints` contiguously in memory



- `part` consists of a pointer (to `arrayOfInts[2]`) and a length (4)

Slice passing

- A slice is passed (like everything else) by copy

```
var arrayOfInts [256]int

var part []int = arrayOfInts[2:6]

func fill(s []int) {
    for i, _ := range s {
        s[i] = i*2
    }

    s = s[1:]
}

fill(part)
fmt.Printf(“%#v”, part)
```

Contents of s can be modified

Changes contents of underlying array

Does nothing to `part`

```
% ./slice
[]int{0, 2, 4, 6}
```

Slice passing, part 2

- Can pass a pointer to a slice to modify the slice

```
var arrayOfInts [256]int

var part intSlice = arrayOfInts[2:6]

type intSlice []int
func (s *intSlice) fill() {
    for i, _ := range *s {
        (*s)[i] = i*2
    }
    *s = (*s)[1:]
}

part.fill()
fmt.Printf("%#v\n", part)
```

Contents of s can be modified and s can be changed

Changes part

```
% ./slice
[]int{2, 4, 6}
```

Slice iteration

```
prime := []int{2, 3, 5, 7, 11}

for i := range prime {
  // i is 0, 1, 2, 3, 4
}

for _, e := range prime{
  // e is 2, 3, 5, 7, 11
}

for i, e := range prime {
  // i and e as above
}
```

Copying slices

- copy builtin

```
morePrimes := make([]int, len(primes), 2*cap(primes))

copy(morePrimes, primes)
```

- copy allows source and destination to overlap

```
primes := [10]int{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
odds := primes[1:7]

odds = odds[0:len(odds)+1]
copy(odds[4:], odds[3:])
odds[3] = 9
fmt.Printf("%#v\n", odds)
```

```
[]int{3, 5, 7, 9, 11, 13, 17}
```

Appending slices

```
s := []int{1, 3, 6, 10}
t := []int{36, 45, 55, 66, 78}

s = append(s, 15)
s = append(s, 21, 28)

s = append(s, t...)

nu := append([]int(nil), s...)

s = append(s, s...)

fmt.Printf(“%#v\n”, s)
```

Adding individual elements

Adding an entire slice

Copying a slice (use copy instead)

```
[]int{1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 1, 3,
6, 10, 15, 21, 28, 36, 45, 55, 66, 78}
```

CHANNELS AS QUEUES

A buffered channel is a FIFO queue

- A typed queue of up to 10 `Things`

```
queue := make(chan Thing, 10)
```

- Get the next element from the queue if there is one

```
select {
case t := <-queue: // got one
default:           // queue is empty
}
```

- Add to queue if there’s room

```
select {
case queue <- t: // added to queue
default:         // queue is full
}
```

GENERICS

Perhaps heretical

- But... I wish Go had some generics
  - `interface{}` is like `void *;` Type assertions similar to casts

```
l := list.New()
l.PushFront("Hello, World!")
v := l.Front()
i := v.Value.(int)
```

```
% go build l.go
% ./l
panic: interface conversion: interface is
string, not int

goroutine 1 [running]:
runtime.panic(0x49bdc0, 0xc210041000)
        /extra/go/src/pkg/runtime/panic.c:266
+0xb6
main.main()
        /extra/src/mc/generic.go:12 +0xaa
```

Sources etc.

- Slides and code samples for this talk:

  https://github.com/cloudflare/jgc-talks/tree/master/Go_London_User_Group/Go_Containers

- All my talks (with data/code) on the CloudFlare Github

  https://github.com/cloudflare/jgc-talks

- All my talks on the CloudFlare SlideShare

  http://www.slideshare.net/cloudflare