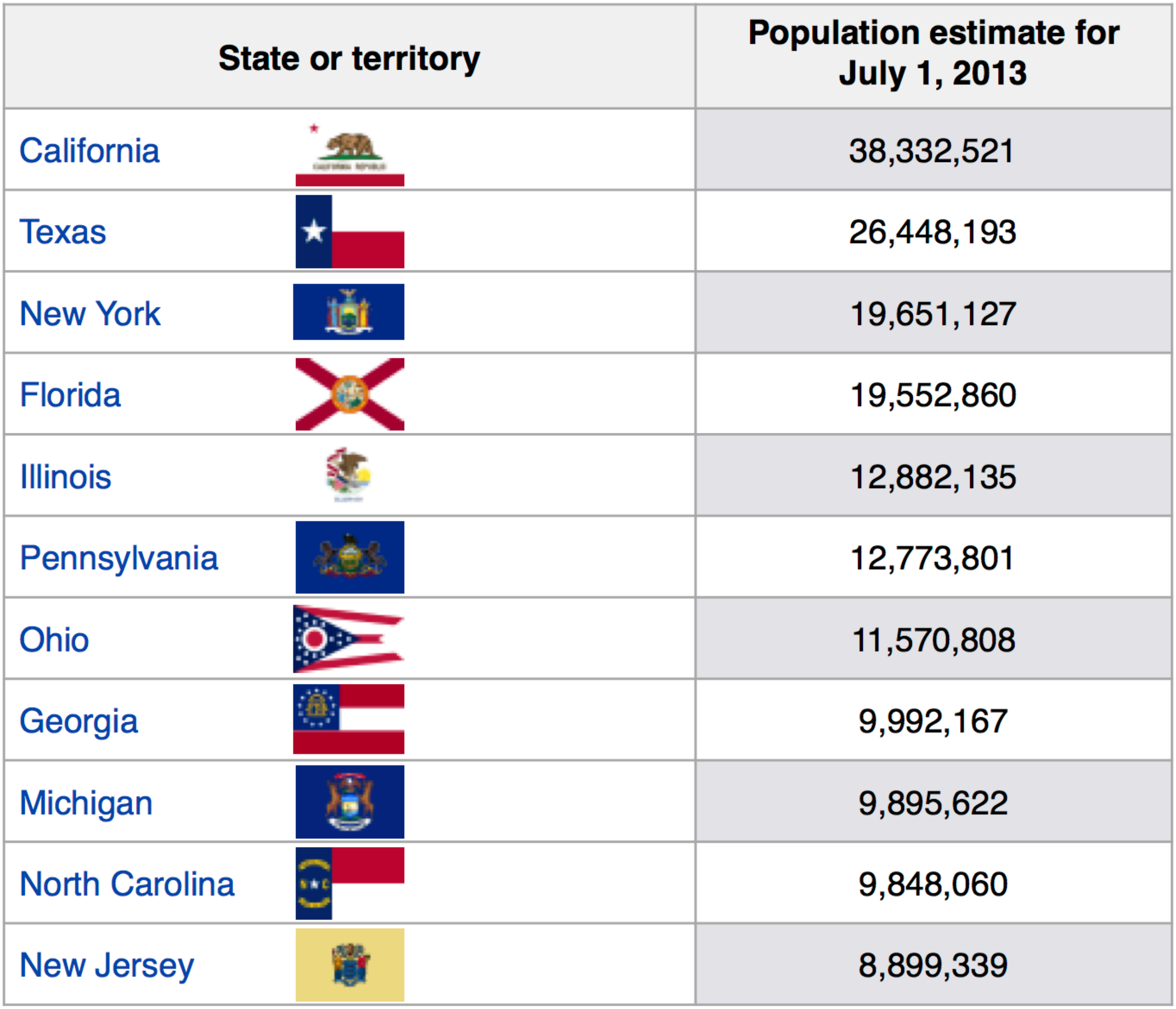
Carl Kingsford, 02-201, Fall 2015

Lecture 11: Maps

What if you want to store populations of US states?



State or territory	Population estimate for July 1, 2013
California	38,332,521
Texas	26,448,193
New York	19,651,127
Florida	19,552,860
Illinois	12,882,135
Pennsylvania	12,773,801
Ohio	11,570,808
Georgia	9,992,167
Michigan	9,895,622
North Carolina	9,848,060
New Jersey	8,899,339

Creating `map` variables

maps let you do this. Maps are a built-in data type in Go. You can declare them using the following syntax:

```
map[KEYTYPE]VALUETYPE
```

Maps associate a key with a value.

For example:

```
var grades map[string]int           // strings to ints
var rules map[string]string     // strings to strings
var multi map[string][]string       // strings to string slices
var pop map[string]float64          // strings to floats
var ssn map[int]string           // ints to strings
var families map[string]map[string]int
```

As with lists, you have to `make` a map after you declare it.

```
grades = make(map[string]int)
rules = make(map[string]string)
multi = make(map[string][]string)
pop = make(map[string]float64)
ssn = make(map[int]string)
families = make(map[string]map[string]int)
```

Note that you don't have to give a "size" for the map to `make`: the map will grow and shrink automatically as needed.

Using maps

Maps act a lot like lists, but you can access elements using keys instead of integer indices:

```
var statePop map[string]int = make(map[string]int)

statePop["PA"] = 12773801
statePop["CA"] = 38332521

fmt.Println("The population of PA is", statePop["PA"])
```

Some more examples:

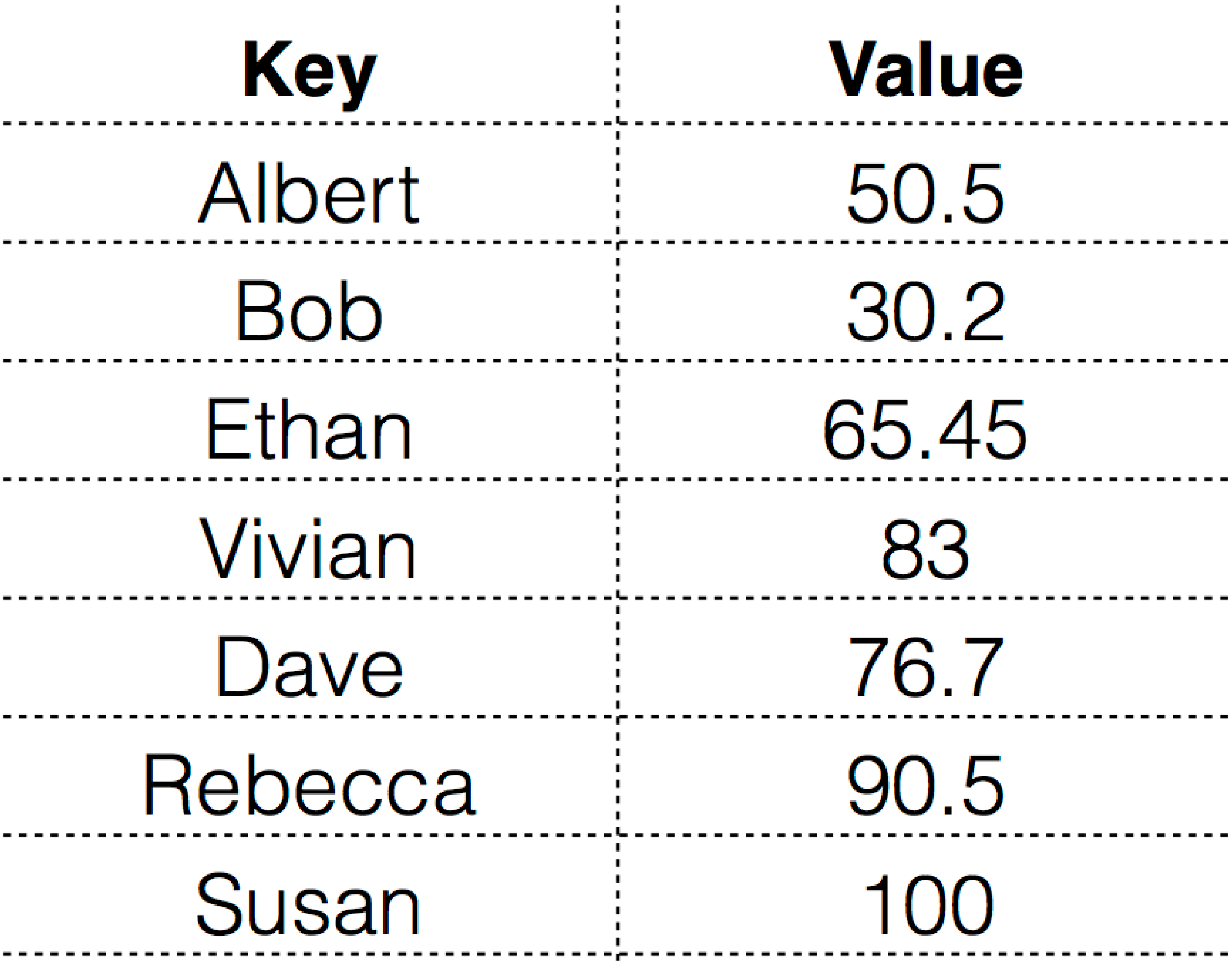
```
grades[“Carl”] = "A+++"
fmt.Println("Rule for", x, "is", rules[x])
ssn[627729183] = "Dave"
paPop = pop["PA"]
```

> **Test yourself!** What types are the maps in the above example?

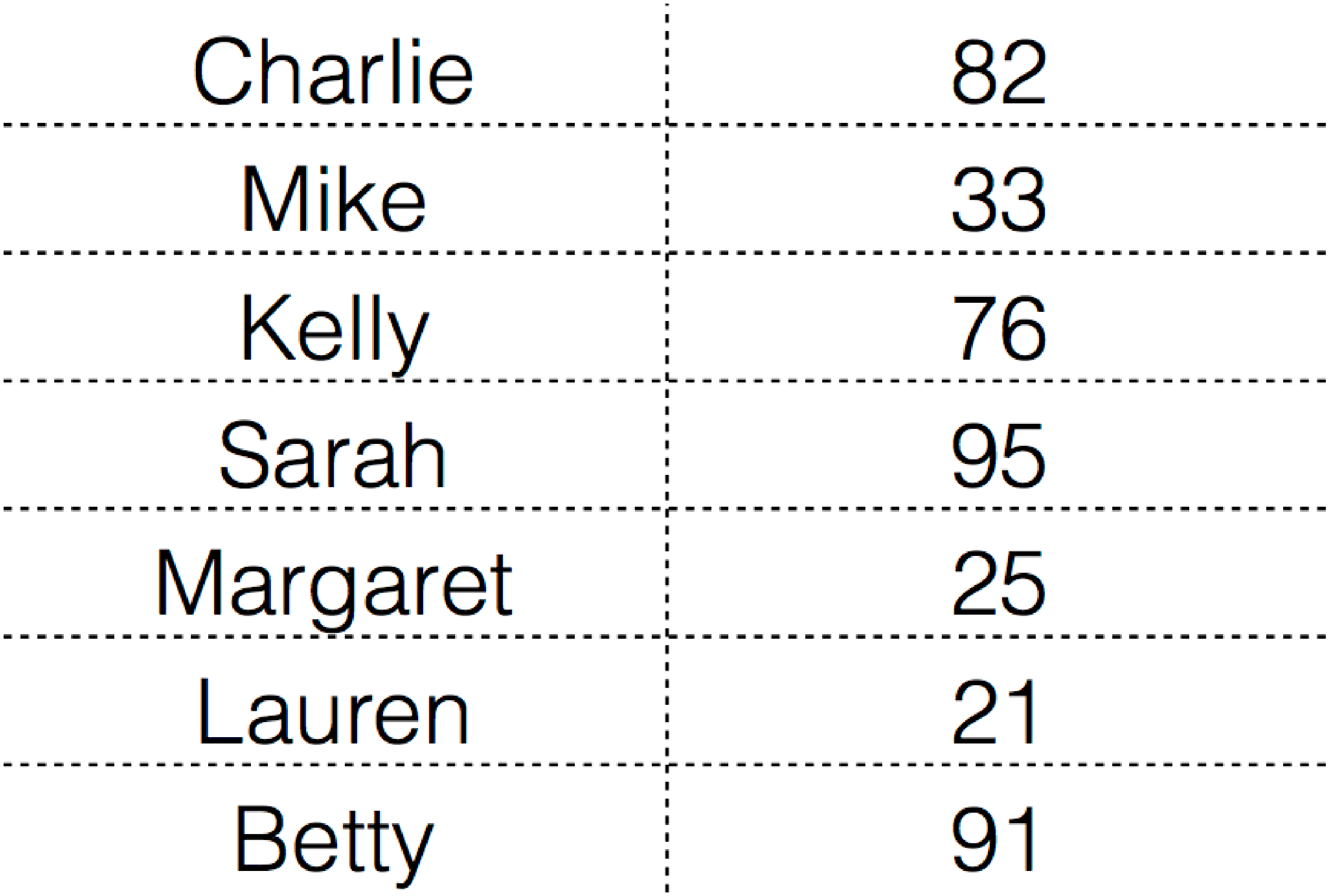
After `make` , each element of the map contains its "0" value:

```
grades := make(map[string]string)
fmt.Println(grades["Chuck"]) // will print ""
```

Mental image of a map



Key	**Value**
Albert	50.5
Bob	30.2
Ethan	65.45
Vivian	83
Dave	76.7
Rebecca	90.5
Susan	100



Charlie	82
Mike	33
Kelly	76
Sarah	95
Margaret	25
Lauren	21
Betty	91

Checking if an element has ever been set in a map

You can access an element of a map using the syntax `paPop = pop[“PA”]`. In this case, if key "PA" has been given a value in the map, you will get the value, otherwise, you will get a "0" value.

You can explictly check whether an element has been set by using a double-assignment:

```
paPop, exists := pop[“PA”]
```

`paPop` will be set as above, but `exists` will be `false` if nothing was stored previously for key `"PA"` in the map.

(You can use any variable name where `exists` occurs above --- `exists` is a `bool` variable.) This is useful to check whether a key has ever been given a value:

```go
paPop, exists := pop[“PA”]
if !exists {
    fmt.Println(“Never set PA pop!”)
}
```

Getting the number of elements in a map

Use the `len()` function to get the number of things that have been added to a map: `len(pop)` if `pop` is a map. For example,

```go
m := make(map[int]int)
m[1] = 0
m[7] = 10
m[8] = 0
fmt.Println(len(m)) // will print "3"
```

Deleting an element from a map

You can remove an item from a map (so it looks like you never set it to a non-zero value):

```go
delete(pop, “PA”)
delete(rules, “A”)
delete(ssn, x)
```

Here `pop`, `rules` and `ssn` are maps, and the second parameter is the key to delete.

Map literals

Just as with lists, you can explicitly list what you want to be in a map at the start:

```go
rules := map[string]string{
    “A”: “B-A-B”,
    “B”: “A+B+A”,
}
```

(Note that if you put the `key:value` items each on their own line, the last pair must have a "," following it just like all the rest.)

Looping through the items in a map

Just as with lists, we can loop over the elements in a map using `for ... range`:

```
for k, v := range pop {
    fmt.Println(“The population of”, k, “is”, v)
}
```

The difference is that we provide 2 variables in the `for` statement (e.g. `k` and `v` above). These will loop through all the key and value pairs.

Note: there is **no guarantee** about which order the elements of the map will be accessed in a `for...range` statement.

Example use of maps

Recall that we wrote something similar to this in the Lindenmayer example:

```
// gets the Rhs for a given Lhs for a rule
func getRhsFor(char string, lhs, rhs []string) (string, bool) {
    for i, l := range lhs {
        if l == char {
            return rhs[i], true
        }
    }
    return "", false
}
```

This assumed we had rules encoded like this:

```
lhs := []string{"A", "B"}
rhs := []string{"B-A-B", "A+B+A"}
```

But the rules are more logically and easily encoded as a `map` from a string (lhs) to another string (rhs):

```go
rules := make(map[string]string)
rules["A"] = "B-A-B"
rules["B"] = "A+B+A"
```

Now we can write getRhsFor() much more easily:

```go
// gets the Rhs for a given Lhs for a rule
func getRhsFor(char string, rules map[string]string) (string, bool) {
    rhs, exists := rules[char]
    return rhs, exists
}
```

Maps of maps

Just like lists of lists (and lists of lists of lists, etc.) maps of maps are allowed (as are maps of maps of maps of ...). You declare a map of maps using the syntax like:

```go
var mom map[int]map[string]int
```

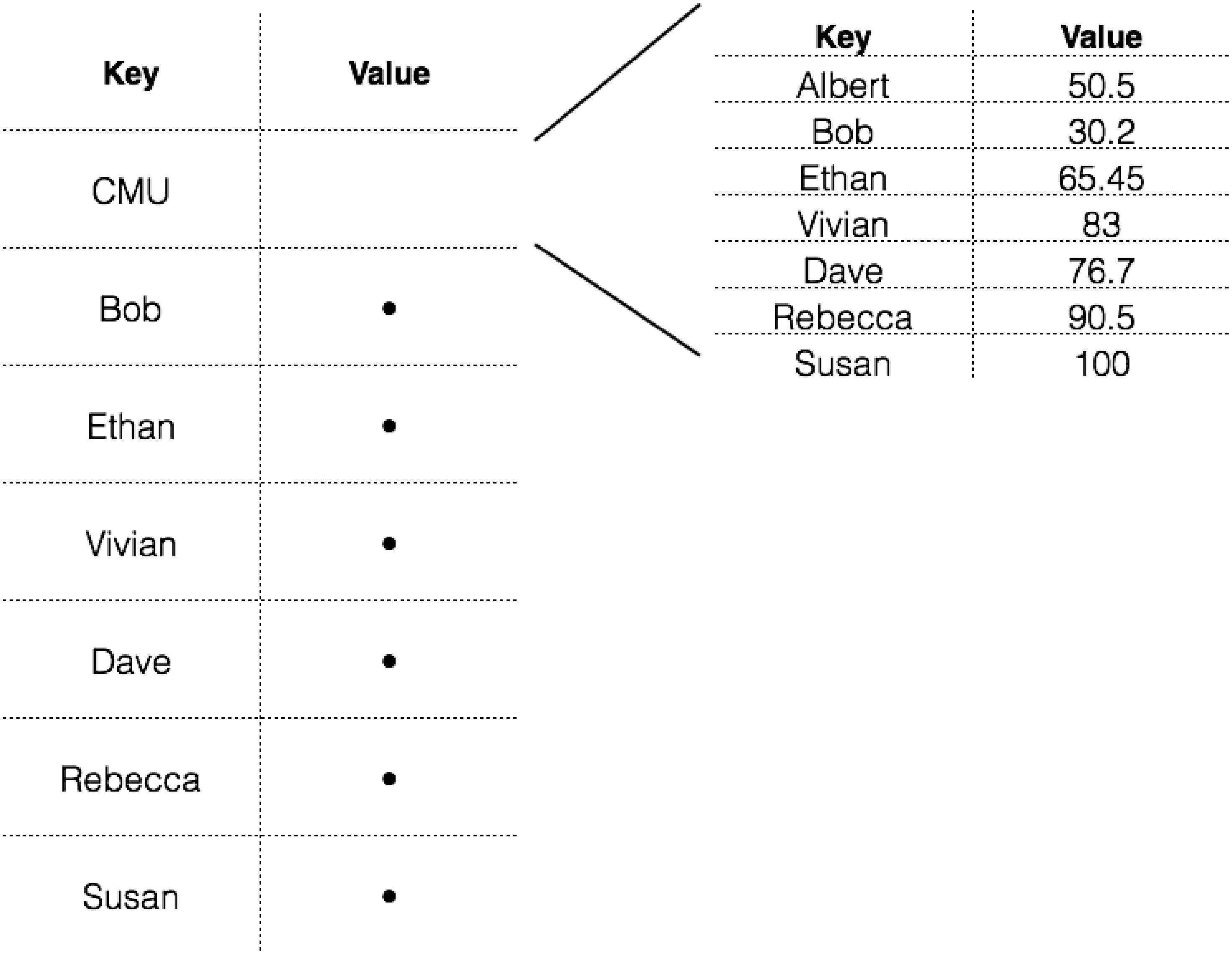
This is a map of integers to maps of strings to integers. In other words, `mom[10]` is a variable of type `map[string]int`. It's also true that `mom[10]["hi"] is an integer.

Just like with 2-D lists, you have to explictly create the "inner" maps:

```go
mom = make(map[int]map[string]int)
mom[10]["hi"] = 3 // error at this point

mom[10] = make(map[string]int)
mom[10]["hi"] = 3 // ok now
```

Mental image of a map of maps:



Key	Value
CMU	
Bob	•
Ethan	•
Vivian	•
Dave	•
Rebecca	•
Susan	•

Key	Value
Albert	50.5
Bob	30.2
Ethan	65.45
Vivian	83
Dave	76.7
Rebecca	90.5
Susan	100

Why use lists instead of maps?

You can think of list `[]float64` as a `map[int]float64` where only keys between 0 and the length of the list are allowed. Maps can handle much of what lists can do (and some programming languages like AWK only include maps and not lists). So why would you ever use lists?

- Lists can use less memory than maps if you really need to store an element for every index in the list.
- Lists can be appended to.
- The *order* of the elements of a list is specified, while for maps there is no fixed ordering (although you can fake it).

In summary, often you could use either and which one you use is a matter of style and clarity (is what you doing more like keeping a list of items? or more like associating items with integers?)

When you *can* use a list, you probably should. Otherwise, use a `map`.

Summary

- Maps store associations between a key and a value.
- Keys must be unique within a map.
- You can use them like lists, but with more general keys.
- Maps are extremely useful, often more useful than lists.

Glossary

- key: a value used to index items in a map
- value: a value retrieved for a given key in a map
- key/value pair: a pair of keys and values that are associated in a map
- map: stores associations between keys and values