COMP 348: Principles of Programming Languages

Section 16: Composite Data Structures

Introduction

- Clojure data structures
  - Maps
  - Vectors
  - Lists
  - Sets
- Creation and access methods
- Immutability
  - How it’s done in practice

Managing data

- In the last section, we looked at simple Clojure types
  - ints, floats, strings, and booleans
- But we can also work with powerful, composite data types.
- Clojure includes:
  - Maps
  - Vectors
  - Lists
  - Sets

Maps

- Clojure maps are the equivalent of Python dictionaries.
  - That is, they provide key/value lookup pairs.
- An empty map can be declared as { }
- To define a map with a collection of key/value pairs, we can do the following

```
; profession map
{
 “John” “professor”
 “Sue” “doctor”
 “Ahmed” “astronaut”
}
```

Maps…cont’d

- We can also do these things

```
; number map with int keys
{
 1 “one”
 2 “two”
 3 “three”
}
```

```
; function map
{
 “func1” (fn [] (println “func1”))
 “func2” (fn [] (println “func2”))
}
```

Keys in practice

- In practice, Clojure keys are often created using what are known as keywords.
- A keyword is a label prefixed with a “:”
- In a map definition, we might have:

```
; profession map
{
 :john “professor”
 :sue “doctor”
 :ahmed “astronaut”
}
```

- Keywords are essentially strings, but provide faster internal lookups
  - Keywords are case sensitive
  - You will see them a lot

Map lookups

- Of course, we need to be able to find things in a map.
- For this we use the `get` function
  - Note that in many cases, you will want to provide a name for your maps in order to perform the lookup

```
(def jobs {
 :john “professor”
 :sue “doctor”
 :ahmed “astronaut”
})

(get jobs :sue) ; => “doctor”
(get jobs :Sue) ; => nil
```

…and a little more

- Now let’s put a few of these things together

```
; function lookup table
(def funcs {
 :func1 (fn [] (println “func1”))
 :func2 (fn [] (println “func2”))
 :func3 (fn [] (println “func3”))
})

; execute the third function
((get funcs :func3)) ; => func3
```

- What is the return value?
  - nil (from the println)

Odds and Ends

- Maps can be nested
  - The `get-in` function can access the inner map(s)

```
(def nest {
 :a “eh”
 :b {:dog “fido” :cat “fluffy”}
})

(get nest :a) ; => “eh”
(get nest :b)
  ; => {:dog “fido”, :cat “fluffy”}

(get-in nest [:b :cat]) ; => “fluffy”
```

Odds and ends…cont'd

- You can provide default values for lookups
  - (get jobs :bill "not found")
  - Returns "not found" if :bill is not in the map
- You can also use keywords like a function invocation (instead of get)
  - (:john jobs)
  - (:john jobs "not found")
- There are other useful map functions, including, but not limited to
  - (keys map_name)
  - (vals map_name
  - (contains? map_key)

One last thing

- We have said that Clojure uses whitespace to separate elements of the language
- While this is true, it is worth pointing out that commas are considered to be whitespace by Clojure
  - In other words, it simply ignores them
- However, you can sometimes use commas to make your code slightly more readable.
  - This is sometimes done with maps

```
{:john “prof” :sure “doctor”}

{:john “prof”, :sure “doctor”}
```

Vectors

- A vector is essentially a 0–indexed sequence (array).
  - It is delimited by [ ] brackets
- Creating a vector is easy, using either of two techniques:

```
(def myVec [1 2 3])

(def myVec (vector 1 2 3))
```

- Again, commas could be used but this would not normally be done

Vectors…cont’d

- Again, we can use the `get` function to retrieve elements.
  - nil is returned if the index is out of bounds

```
(def things [
 “dog”
 2
 (fn [] “do something”)
])

(get things 0) ; => “dog”
(get things 1) : => 2
(get things 10) : => nil
```

- Note that we can put anything into a vector, including functions.

Lists

- As one would expect, Clojure provides a list data structure.
- Lists are delimited by the ( ) parentheses
- If you want to define a list using the ( ) notation, however, you have to prefix the list with a single quote character. Why?
  - If you don’t do this, Clojure will interpret the list as an expression and try to execute it (which will probably fail)

```
(1 2 3)  ; => error: 1 is not a function

`(1 2 3) ; => valid list

(+ 2 2)  ; => function call

`(+ 2 2) ; => valid list
```

List content

- Note that, like vectors, it is also possible to define a list using the (list *list_content*) form.
- In addition, one can add any type of data to a list, including lists, vectors and maps.

```
(def boo (list 1 “two” [1 2 3]))

(def foo ‘(2 {:a “eh”, :b “bee”} 3))
```

Lists…cont'd

- Lists have no indexing
- The `get` function does not work with lists (it always returns nil)
- Instead, use the `nth` function
  - An *out of bounds* error is generated for invalid indexes

```
(nth `(1 2 3) 1)   ; => 2

(nth `(1 2 3) 10)   ; error
```

Lists…cont’d

- How fast is `nth`?
- Recall that searches through a list take time proportional to the length of the list
  - On average, this is O(n)
- This isn’t a problem in itself, but you should be aware that `nth` on a list is slower than `get` on a vector
  - Often, this dictates which data structure you will use in practice.
  - There isn’t a lot of difference between the two so the choice depends on how you would like to access and update the data.

Sets

- Like Python, Clojure has a *set* data structure
- It’s purpose is to maintain a collection of unique values.
- The syntax use the #{ } notation
  - The # is necessary to distinguish a set from a map
- It is also possible to use the `(hash-set values)` form.

```
#{1 2 “dog” “cat”}

(hash-set 1 2 “dog” :c)
```

Sets…cont’d

- Sets will automatically discard duplicates

```
#{1 2 “dog” 2} ; => #{1 2 “dog”}
```

- We can also create sets from existing lists and vectors, eliminating duplicates in the process

```
(set ‘( 1 1 2 2)) ; => #{1 2}

(set [“dog” “cat” “dog”) ; => #{“dog” “cat”}
```

Searching the set

- There are in fact three ways to search for an element in a set
  - get
  - Using a keyword as a function
  - contains?

```
(def foo #{ 1 2 3 :a} )

(get foo 3)          ; => 3
(get foo 4)          ; => nil

(:a foo)             ; =>:a, returns itself

(contains? foo 2) ; => true
```

Immutability

- Clojure data structures are immutable.
  - That is, one does not actually change the values within a structure.
- Specifically, when we make “changes”, we are actually creating a new/second data structure.
  - Same as the first, except with the modification.
- You can see this simply by using the REPL interface to modify an existing structure, and then checking the structure again.

Adding to a sequence

- To show this, note that Clojure provides a function called `conj` (i.e., conjoin) to add a value to lists and vectors
- In practice, conj adds a value to the *beginning* of a list and the *end* of a vector. Why?

```
(def foo1 `(1 2 3))
(conj foo1 0)   ; => (0 1 2 3)
 foo1           ; => (1 2 3), no change

(def foo2 (conj foo1 0))
 foo1           ; => (1 2 3)
 foo2           ; => (0 1 2 3)
```

Efficiency

- What if I need to make thousands of changes to a large list
- I would now have thousands of versions of the large list
  - YUCK!!!
- This could be catastrophic in terms of:
  - memory consumption
  - processing performance (all of that copying)
- Is Clojure (and other functional languages) really that bad?

Efficiency…cont’d

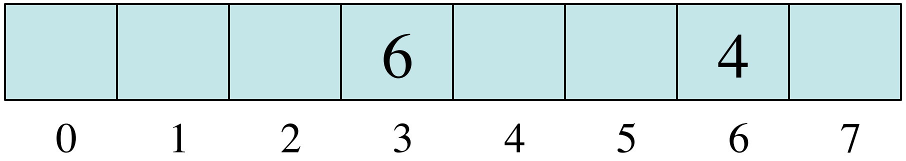
- It is true that Clojure does make copies of the data structures.
- But it does this in a very clever way.
- We will use the example of a vector, but the principle can be applied to other data structures
- Before looking at the model, we will note that the `assoc` function can be used to update a specific vector position
  - Again, the “update” does not actually change the original vector

```
(def foo [1 2 3])
 foo   ; => [1 2 3]

(assoc foo 1 “dog”)   ; => [1 “dog” 3]
 foo   ; => [1 2 3]
```

Modifications

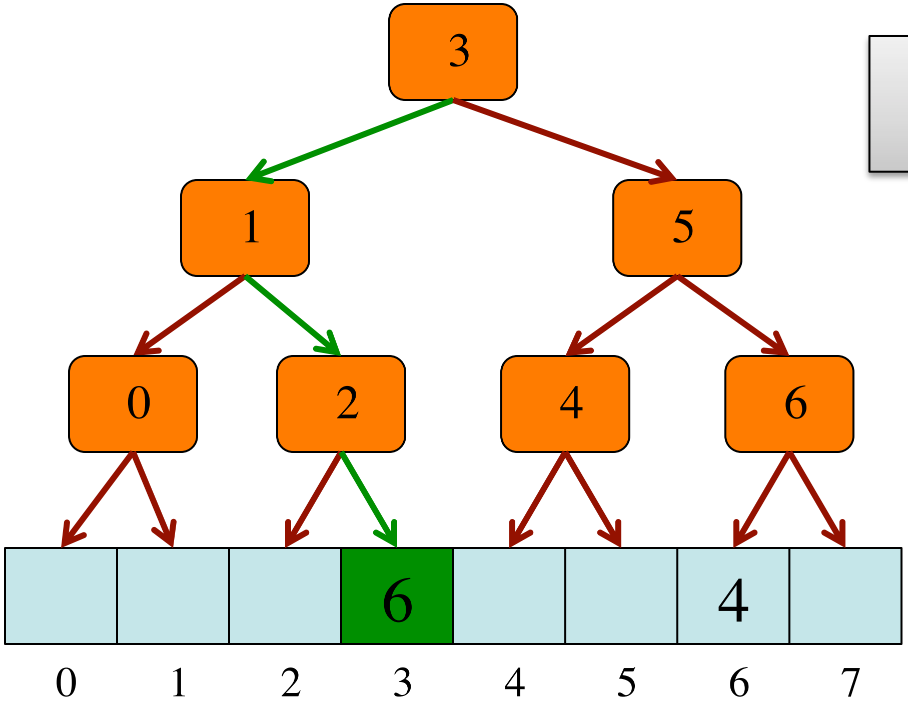
- Let's start by looking at a basic vector implementation
- Below, we have a simple array
- In this case, 2 of the 8 cells have a non-nil value
- Let's say that we want to change the 6 to a 9.
  - Clearly, we could modify the 3rd cell in place but this would destroy the immutability property
  - We could also create a new array with the updated value, but this would be very inefficient



			6			4	
0	1	2	3	4	5	6	7

Modification…cont’d

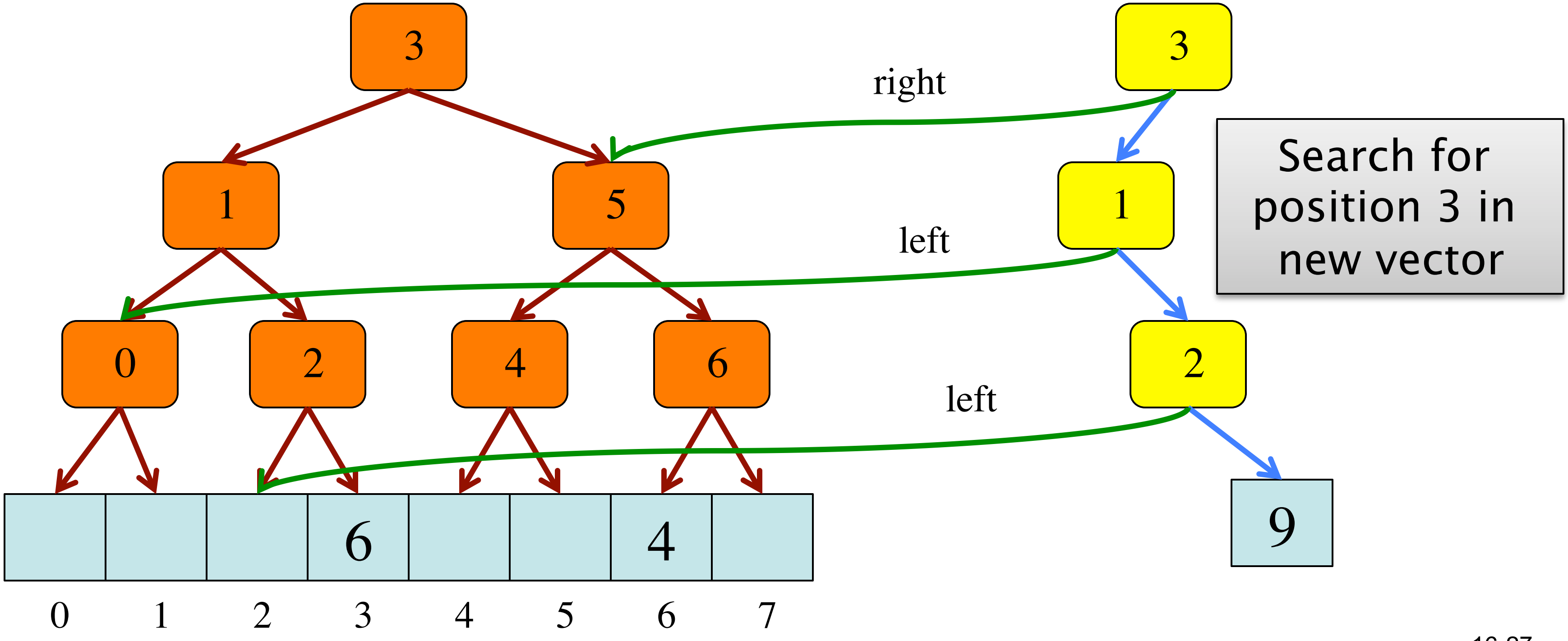
- Instead, let’s augment the basic data structure itself by including an indexing structure
- The index will be traversed by going left if the required position is <= the node value, and right otherwise



Search for position 3

Modification…cont’d

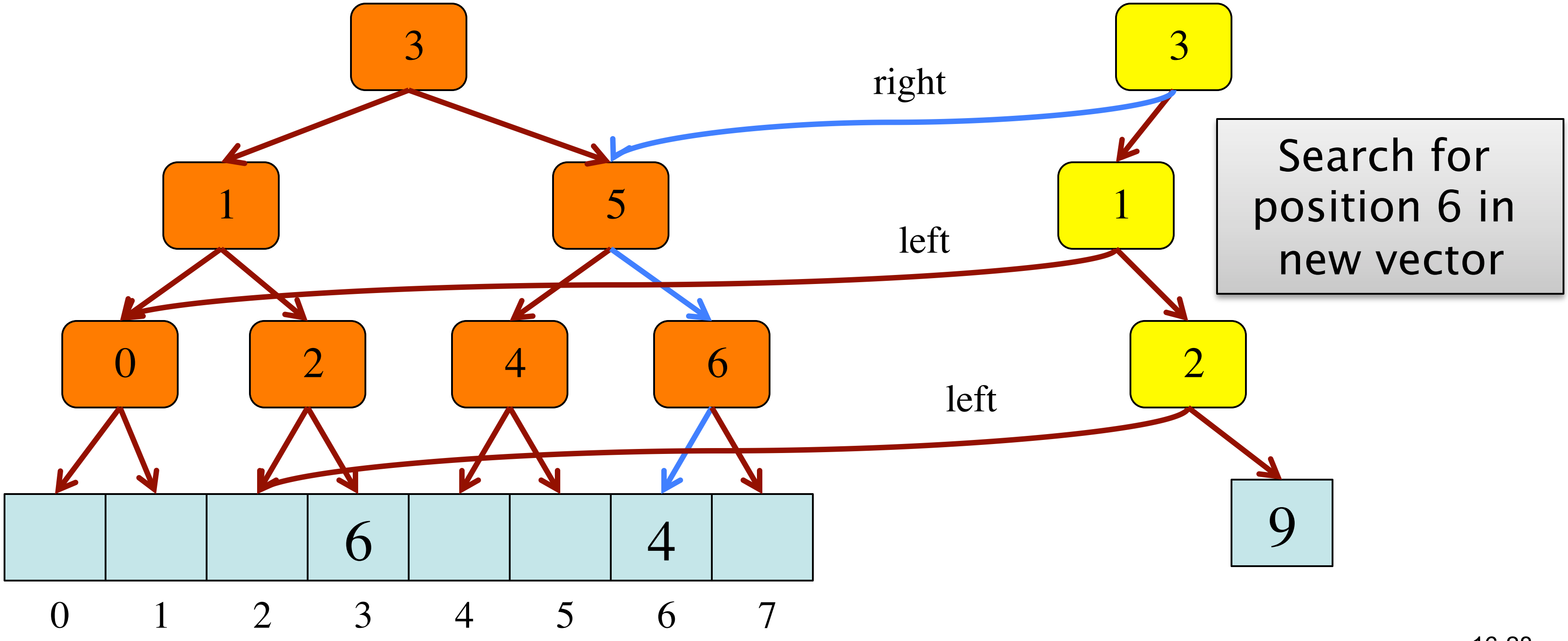
- Now let’s update 6 to 9
- To do this, we will create a new data structure that simply consists of a new path for the updated element
- We will then connect the remaining paths to the original index
- To find the updated element, we will use the new index (see path in blue)



Search for position 3 in new vector

Searching the new vector

- Now let’s search for position 6 in the new vector.
- In this case, we navigate back to the first data structure and follow the index to position 6 (see path in blue).



Search for position 6 in new vector

Modification…cont’d

- Now, we have two distinct data structures, with the second allowing access to the updated elements.
- However, for all other elements, the unmodified elements are shared.
- In effect, this creates a core *read-only* data structure that is modified as required.
- If we now modify the second data structure, a third data structure can point back to #2 or to #1.

Modifications…cont’d

- What does this cost?
- Without going into a great deal of theory, the indexes can be “balanced”, so that they are essentially shaped like a triangle.
- Such trees have logarithmic ($\log_2$) access time.
  - For example, an index with 1000 nodes has a height of just 10, and a tree with 1M nodes has a height of just 20.
- Since logarithms grow VERY slowly, access times are very good, though not of course quite as fast as the O(1) times of the original vector.