Binary Search Tree

Data structure books only show keys at each node

But each node has a key and a value
Representing a Tree

[10 [5 nil nil] [20 nil nil]]

[[5 nil nil] 10 [20 nil nil]]

{:key 10, :left {:key 5}, :right {:key 20}}

{:key 10 :value foo
 :left {:key 5 :value bar}
 :right {:key 20 :value foo-bar}}

We will see other ways to represent a tree
Representing Tree

(key left right)

(def tree [10 [5 [1 nil nil] [8 nil nil]] [20 [15 nil nil] [30 nil nil]]])
Hiding the Structure of Node

(defn left-child
 [node]
 (node 1))

(defn right-child
 [node]
 (node 2))

(defn value
 [node]
 (node 0))
Navigating the Tree

(def large-tree [10 [5 [1 nil nil] [8 nil nil]] [20 nil nil]])

(right-child (left-child large-tree))

(-> large-tree
   left-child
   right-child)
Standard Search

(defn find-key
  [tree k]
  (let [left (left-child tree)
         right (right-child tree)
         value (value tree)]
    (cond
      (= k value) k
      (and left (< k value)) (find-key left k)
      (and right (> k value)) (find-key right k)
      :default nil)))

This is where you really want a key & value at each node of the tree
Inserting

But we have persistence & immutability

Add 15
Inserting

Add 15
Inserting - Three Ways

Build the tree as you traverse the tree

Find path to node and use assoc-in

Use a zipper
Build Tree as Traverse

Tree node {left left-child :value value :right right-child}

(defn make-tree
  [left value right]
  {left left :val value :right right})

(defn insert [tree value]
  (if-let [member (:value tree)]
    (cond
      (< value member) (make-tree (insert (:left tree) value) member (:right tree))
      (> value member) (make-tree (:left tree) member (insert (:right tree) value))
      :else tree)
    (make-tree nil value nil)))
assoc-in

Associates a value in a nested structure

(def users [{:name "James" :age 26} {:name "John" :age 43}])

(assoc-in users [1 :age] 44)

[{:name "James", :age 26} {:name "John", :age 44}]

(assoc-in users [1 :password] "nhoJ")

[{:name "James", :age 26} {:password "nhoJ", :name "John", :age 43}]
(defn position-of
  "Return path to k in tree"
  [tree k]
  (let [left (left-child tree)
         right (right-child tree)
         value (value tree)]
    (cond
      (= k value) nil
      (and left (< k value)) (cons 1 (position-of left k))
      (< k value) [1]
      (and right (> k value)) (cons 2 (position-of right k))
      (> k value) [2]))))

(def tree [10 [5 [1 nil nil] [8 nil nil]] [20 [15 nil nil] [30 nil nil]]])

(position-of tree 5) (1)
(position-of tree 1) (1 1)
(position-of tree 8) (1 2)
(position-of tree 20) (2)
(position-of tree 15) (2 1)
(position-of tree -1) (1 1 1)
Insert

(defn bst-insert
  [tree value]
  (assoc-in tree (position-of tree value) [value nil nil]))

(def small-tree [10 nil nil])

(bst-insert small-tree 5) ; [10 [5 nil nil] nil]

-> small-tree
  (bst-insert 5)
  (bst-insert 20)
  (bst-insert 1)) ; [10 [5 [1 nil nil] nil] [20 nil nil]]
Zippers

Allow you to navigate & change structures
  seq-zip
  vector-zip
  xml-zip

Keeps pointer to current location in structure

Moving
  up, down, left, right, next, prev, leftmost, rightmost

Accessing structure
  node, root

Editing
  remove, replace, edit, insert-child
Zipper Examples

(ns basiclectures.basic-language.zip
 (:require [clojure.zip :as zip] ))
(def large-tree [10 [5 [1 nil nil] [8 nil nil]] [20 nil nil]])

(-> large-tree
    zip/vector-zip
    zip/node) [10 [5 [1 nil nil] [8 nil nil]] [20 nil nil]]

(-> large-tree
    zip/vector-zip
    zip/down
    zip/node) 10

(-> large-tree
    zip/vector-zip
    zip/down
    zip/right
    zip/node) [5 [1 nil nil] [8 nil nil]]
Zipper Examples

(ns basiclectures.basic-language.zip
 (:require [clojure.zip :as zip] ))
(def large-tree [10 [5 [1 nil nil] [8 nil nil]] [20 nil nil]])

(-> large-tree
    zip/vector-zip
    zip/down
    zip/right
    zip/right
    zip/node)

(-> large-tree
    zip/vector-zip
    zip/down
    zip/right
    [20 nil nil]
    zip/right
    5
    zip/down
    zip/node)
Zipper Examples

(ns basiclectures.basic-language.zip
 (:require [clojure.zip :as zip] ))
(def large-tree [10 [5 [1 nil nil] [8 nil nil]] [20 nil nil]])

(-> large-tree
  zip/vector-zip
  zip/down
  zip/right)

[[5 [1 nil nil] [8 nil nil]] {:l [10], :pnodes [[10 [5 [1 nil nil] [8 nil nil]] [20 nil nil]]], :ppath nil, :r ([20 nil nil])}]
Zipper Examples

(ns basiclectures.basic-language.zip
 (:require [clojure.zip :as zip] ))

(def large-tree [10 [5 [1 nil nil] [8 nil nil]] [20 nil nil]])

(-> large-tree
    zip/vector-zip
    zip/down
    zip/right
    zip/right
    (zip/replace [50 nil nil])
    zip/root)

[10 [5 [1 nil nil] [8 nil nil]] [50 nil nil]]
Zipper Examples

(ns basiclectures.basic-language.zip
  (:require [clojure.zip :as zip] )
(def large-tree [10 [5 [1 nil nil] [8 nil nil]] [20 nil nil]])

(-> large-tree
    zip/vector-zip
    zip/down
    (zip/replace 11)
    zip/root)
[11 [5 [1 nil nil] [8 nil nil]] [20 nil nil]]
BST Insert with Zipper

[key left right]  Tree representation

Accessing

(defn zipper->left-child
  [zipper]
  (-> zipper zip/down zip/right))

(defn zipper->right-child
  [zipper]
  (-> zipper zip/down zip/rightmost))

(defn zipper->value
  [zipper]
  (if (zip/node zipper)
      (-> zipper zip/down zip/node)
      nil))
Replacing/Testing

(defn replace-node
  [zipper replacement]
  (let [location (zip/node zipper)
         node (zip/make-node zipper location [replacement nil nil])]
    (-> zipper (zip/replace node) zip/root)))

(defn tree-empty?
  [zipper]
  (not (zip/node zipper)))
The Insert

(defn bst-zipper-insert
  [zipper x]
  (let [value (zipper->value zipper)]
    (cond
      (tree-empty? zipper) (replace-node zipper x)
      (= x value) (zip/root zipper)
      (< x value) (recur (zipper->left-child zipper) x)
      (> x value) (recur (zipper->right-child zipper) x))))

(defn bst-insert
  [tree x]
  (bst-zipper-insert (zip/vector-zip tree) x))
BST as Maps & Zippers

Zippers are defined for
  XML
  vectors
  seq

What about other structures?

{"left{"value 5 :left nil :right nil} :value 10 :right{"value 15 :left nil :right nil}}

Can define zippers on other types
Making New Zippers

(zipper branch? children make-node root)

branch?
  One argument - node
  Returns true if node can have children

children
  One argument - node
  Returns sequence of the node's children

make-node
  Two arguments - Existing node, seq of children
  Returns new node from the children

Root
  Root of the structure
Zipper for BST as a map

{:left {value 5 :left nil :right nil} :value 10 :right nil}

branch?
  map?

children
  (defn tree->children
    [map]
    [(:value map) (:left map) (:right map)])

make-node
  (defn children->tree
    [sequence]
    {:value (first sequence)
     :left (second sequence)
     :right (last sequence)})

Order has to match that in tree->children
Using the Zipper

(def map-tree {:left {:value 5 :left nil :right nil} :value 10 :right {:value 15 :left nil :right nil}})

(def map-zipper (zip/zipper map? tree->children children->tree map-tree))

(-> map-zipper
    zip/down
    zip/right
    zip/node)  
{:value 5, :left nil, :right nil}
Doing insert in BST as map

(defn bst-map-insert
  [tree x]
  (bst-zipper-insert
    (zip/zipper map? tree->children children->tree tree)
    x))
Notice the repeat

(zip/zipper map? tree->children children->tree tree)

Once we figure out the needed functions would like to forget about it.

(defn bst-map-zipper
  [tree-map]
  (zip/zipper map? tree->children children->tree tree tree-map))
Shorter Way - partial

(defn bst-map-zipper (partial zip/zipper map? tree->children children children->tree)

(partial f arg1 arg2 ... argk)

f - function with n > k arguments
arg1 arg2 ... argk - first k arguments of f
Return function that needs n - k arguments
Examples

```
(def hundred-times (partial * 100))
(hundred-times 5)  500

(hundred-times 5 4)  2000

(reduce + (take-while (partial > 1000) (iterate inc 0)))  499500

(def to/english (partial clojure.pprint/cl-format nil "~@[~@
[~R~]~^ ~A.~)"))

(to/english 123456)

"One hundred twenty-three thousand, four hundred fifty-six"
```
Currying

Currying
Multi-argument function -> chain of single-argument functions

adder(a, b c) {a + b + c;}

addA = adder.curry();
addB = addA(2);
addC = addB(3);
answer = addC(4);
Manipulating Functions
**juxt**

Combines a set of functions
Returns vector applying each function to input

```
(def basic-math (juxt + - * /))
(basic-math 2 5)                      [7 -3 10 2/5]
```

```
(def split-collection (juxt take drop))
(split-collection 4 (range 9))       [(0 1 2 3) (4 5 6 7 8)]
```
juxt

((juxt :last :first) {:last "Adams" :first "Zak"})  

[ "Adams" "Zak"]

(sort-by (juxt :last :first) [{:last "Adams" :first "Zak"}
  {:last "Zen" :first "Alan"}
  {:last "Smith" :first "Alan"}])

({:last "Adams", :first "Zak"}
 {:last "Zen", :first "Alan"}
 {:last "Smith", :first "Alan"})

(sort-by (juxt :first :last) [{:last "Adams" :first "Zak"}
  {:last "Zen" :first "Alan"}
  {:last "Smith" :first "Alan"}])

({:last "Smith", :first "Alan"}
 {:last "Zen", :first "Alan"}
 {:last "Adams", :first "Zak"})
comp

Takes a sequence of functions
Composes the functions

((comp str +) 8 8 8)  "24"

(def fourth (comp first rest rest rest))
(fourth [:a :b :c :d :e])  :d
Given $n$ can we produce

$$(\text{comp first rest rest rest ... rest})$$

where we have $n-1$ rest's?
Yes We Can!

(defn fnth
 [n]
 (apply comp
   (cons first
     (take (dec n) (repeat rest))))

((fnth 1) [:a :b :c :d :e]) :a
((fnth 3) [:a :b :c :d :e]) :c
How does this work?

(repeat rest)  infinite lazy sequence of rest

(take (dec n) (repeat rest))  '(rest rest … rest) ; n-1 rest's

(cons first  
  (take (dec n) (repeat rest)))  '(first rest rest … rest)

(apply comp  
  (cons first  
    (take (dec n) (repeat rest))))  (comp first rest rest … rest)
memoize

(memoize f)

Caches results of function f
Uses cached value next time f is called with same arguments

(defn adder
  [x]
  (println "adder" x)
  (inc x))

(def adder-memoized (memoize adder))

(adder-memoized 1)  prints 1, returns 2
(adder-memoized 1)  returns 2
(adder-memoized 2)  prints 2, returns 3
(adder-memoized 1)  returns 2
memoize - Cache Size

Cache is a map

Contains return values for each different set of input arguments
Delay

Suspends execution of code until delay is dereferenced
Caches result
Second time dereferenced returns cached result
Thread safe

(def wait (delay (println "do it now") (+ 1 2)))

@wait prints "do it now", returns 3
@wait returns 3
realized?

Returns true if a value has been produced for a promise, delay, future or lazy sequence.

(def wait (delay (println "do it now") (+ 1 2)))

(realized? wait) false
@wait prints "do it now", returns 3
(realized? wait) true
@wait returns 3
Example - Proxy for Expensive Operation

(defun fetch-page
  [url]
  {:url url
   :contents (delay (slurp url))})

(def result (fetch-page "http://www.eli.sdsu.edu/index.html"))

(:contents result) ;<Delay@2fcc470c: :pending>
(realized? (:contents result)) false

@(:contents result) "<!DOCTYPE html>
<html lang="en">
…"
@ and deref

@(:contents result)

(deref (:contents result))
Future

Computes body on another thread

Use @ or deref to get answer

@, deref blocks until computation is done

(def long-calculation (future (apply + (range 1e8))))
@long-calculation
Future & Delay in ending program

When you end your program there will be a 1 minute delay if you used future

End your program with (shutdown-agents)

(def long-calculation (future (apply + (range 1e8))))

@long-calculation

(shutdown-agents)
(shutdown-agents) & REPL

(shutdown-agents) shuts down your REPL
deref with Timeout

(deref (future (Thread/sleep 5000) :done!)
  1000
  :impatient!)
;= :impatient!
Future

(println "[Main] Start")

(def what-is-the-answer-to-life (future
   (println "[Future] started computation")
   (Thread/sleep 3000) ;;
   (println "[Future] completed computation")
   42))

(println "[Main] created future")

(Thread/sleep 1000)
(println "[Main] do other things")
(println "[Main] the result" @what-is-the-answer-to-life)