

CS 696 Functional Programming and Design
Fall Semester, 2015
Doc 7 Destructuring, Battle Ship, Life, BST
Sep 15, 2015
Modified Sept 17

Copyright ©, All rights reserved. 2014 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (<http://www.opencontent.org/openpub/>) license defines the copyright on this document.

Destructuring - Positional

```
(let [[a b c] (range 5)]  
  (println "a b c are: " a b c))
```

a b c are: 0 1 2

```
(let [[a b c :as all] [1 2 3 4 5]]  
  (println "a b c are:" a b c)  
  (println "all is:" all))
```

a b c are: 1 2 3
all is: [1 2 3 4 5]

```
(let [[a b c & more :as all] (range 5)]  
  (println "a b c are:" a b c)  
  (println "more is:" more))
```

a b c are: 0 1 2
more is: (3 4)

```
(let [[a b c & more :as all] (range 5)]  
  (println "a b c are:" a b c)  
  (println "more is:" more)  
  (println "all is:" all))
```

a b c are: 0 1 2
more is: (3 4)
all is: (0 1 2 3 4)

Destructuring - Positional

```
(defn destructuring  
  [[a b c & more :as all] z]  
  (println "a b c are:" a b c)  
  (println "more is:" more)  
  (println "all is:" all)  
  (println "z is:" z))
```

```
(destructuring [1 2 3 4 5] "cat")
```

```
a b c are: 1 2 3  
more is: (4 5)  
all is: [1 2 3 4 5]  
z is: cat
```

Associative Destructuring

Index



```
(let [{first 0, third 2, last 4} [1 2 3 4 5]]  
  [first third last])
```

```
[1 3 5]
```

Destructuring - Maps

```
(def guys-name-map {:first-name "Guy" :middle-name "Lewis"  
                   :last-name "Steele"})
```

```
(let [{l-name :last-name, f-name :first-name} guys-name-map]  
    (str f-name " " l-name))
```

```
(let [{:keys [last-name first-name]} guys-name-map]  
    (str first-name " " last-name))
```

Destructuring - :keys, :strs, :syms

[{:keys [a b c]} map]

a, b, c get values at keys :a :b :c in map

[{:strs [a b c]} map]

a, b, c get values at keys "a" "b" "c" in map

[{:syms [a b c]} map]

a, b, c get values at keys 'a 'b 'c in map

Destructuring :as - The Entire map

```
(def guys-name-map {:first-name "Guy" :middle-name "Lewis"  
                   :last-name "Steele"})
```

```
(let [{l-name :last-name, f-name :first-name :as whole-name} guys-name-map]  
    (println f-name " " l-name)  
    whole-name)  
  
;; Guy Steele  
;;{:first-name "Guy", :middle-name "Lewis", :last-name "Steele"}
```

Destructuring :or - Default Values

```
(def guys-name-map {:first-name "Guy" :middle-name "Lewis"  
                   :last-name "Steele"})
```

```
(let [{l-name :last-name, title :title,  
      :or {title "Mr."} guys-name-map]  
    (str title " " f-name " " l-name))
```


Battleship Example

The Problem

Context - Writing a battleship game

Need a function that determines

- Is an enemy ship within range of our ships weapon

- But weapon has a blast area so cannot use weapon if

 - Enemy ship is too close to us or other friendly ships

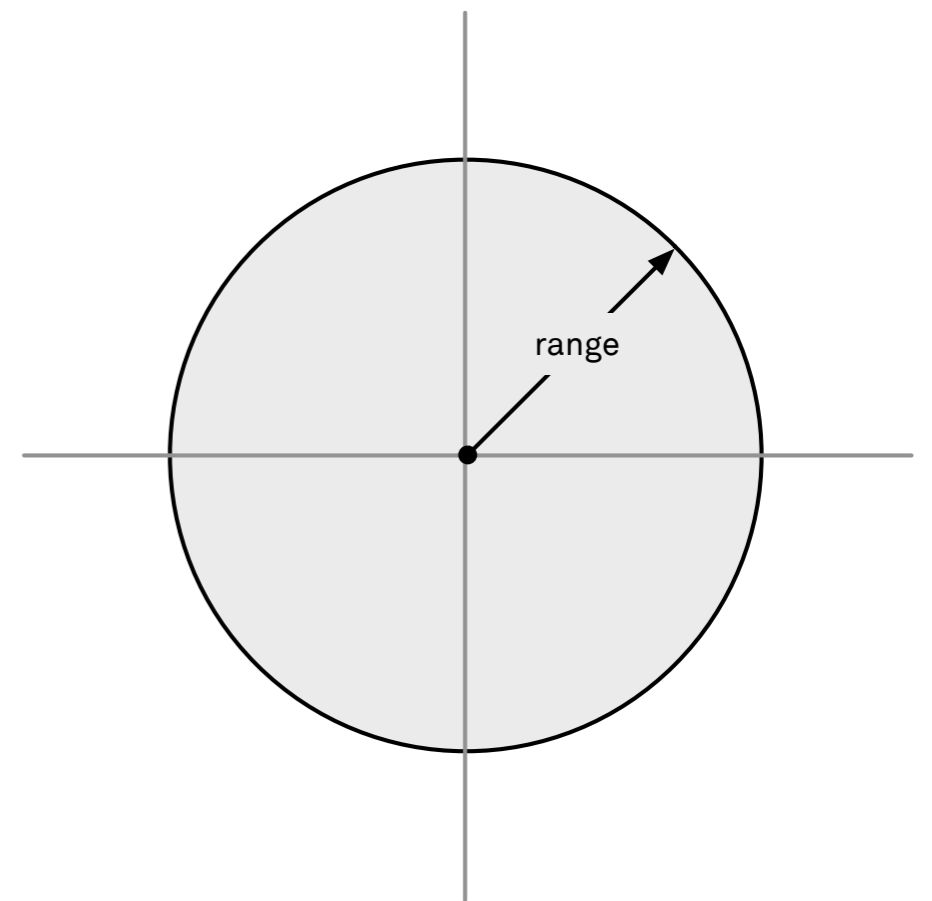
First Pass

Assume we are at origin

Point - [x y]

Given a point & range

Is point within range



```
(defn in-range-1
```

```
  [position range]
```

```
  (let [pos-x (first position)
```

```
        pos-y (last position)
```

```
        target-distance (Math/sqrt (+ (* pos-x pos-x) (* pos-y pos-y)))]
```

```
    (< target-distance range)))
```

```
(in-range-1 [1 1] 1)
```

false

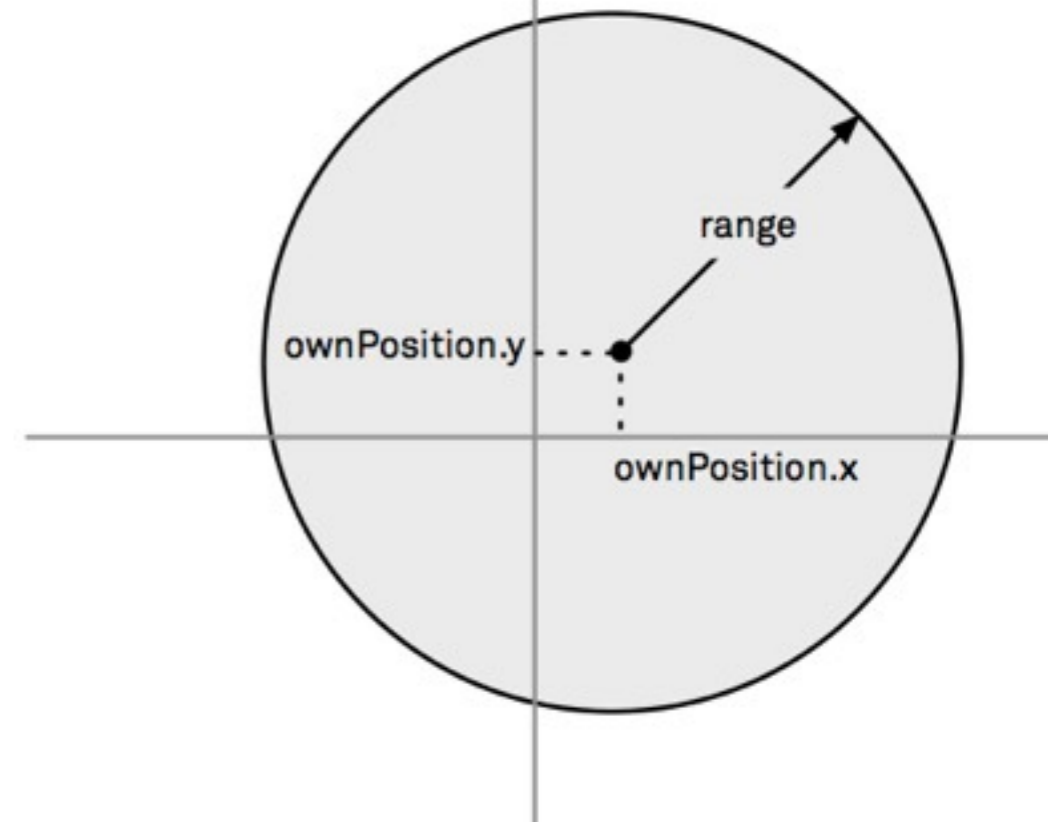
```
(in-range-1 [1 1] 2)
```

true

Second Pass

Let our position be any location

```
(defn in-range-2
  [position own-position range]
  (let [pos-x (first position)
        pos-y (last position)
        own-x (first own-position)
        own-y (last own-position)
        dx (- pos-x own-x)
        dy (- pos-y own-y)
        target-distance (Math/sqrt (+ (* dx dx) (* dy dy)))]
    (< target-distance range)))
```



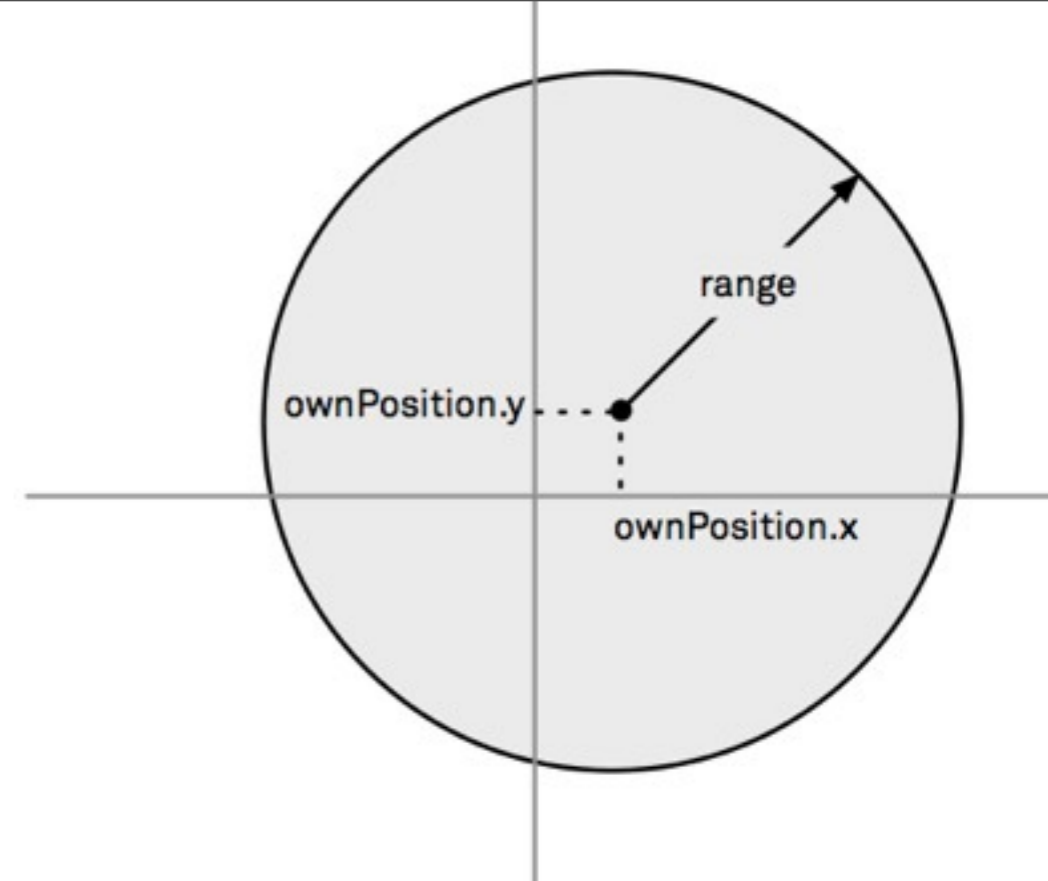
This is a Java program
using Clojure syntax

Second Pass - a

Using destructuring

What do we gain? lose?

```
(defn in-range-2a
  [[pos-x pos-y] [own-pos-x own-pos-y] range]
  (let [dx (- own-pos-x pos-x)
        dy (- own-pos-y pos-y)
        target-distance (Math/sqrt (+ (* dx dx) (* dy dy)))]
    (< target-distance range)))
```

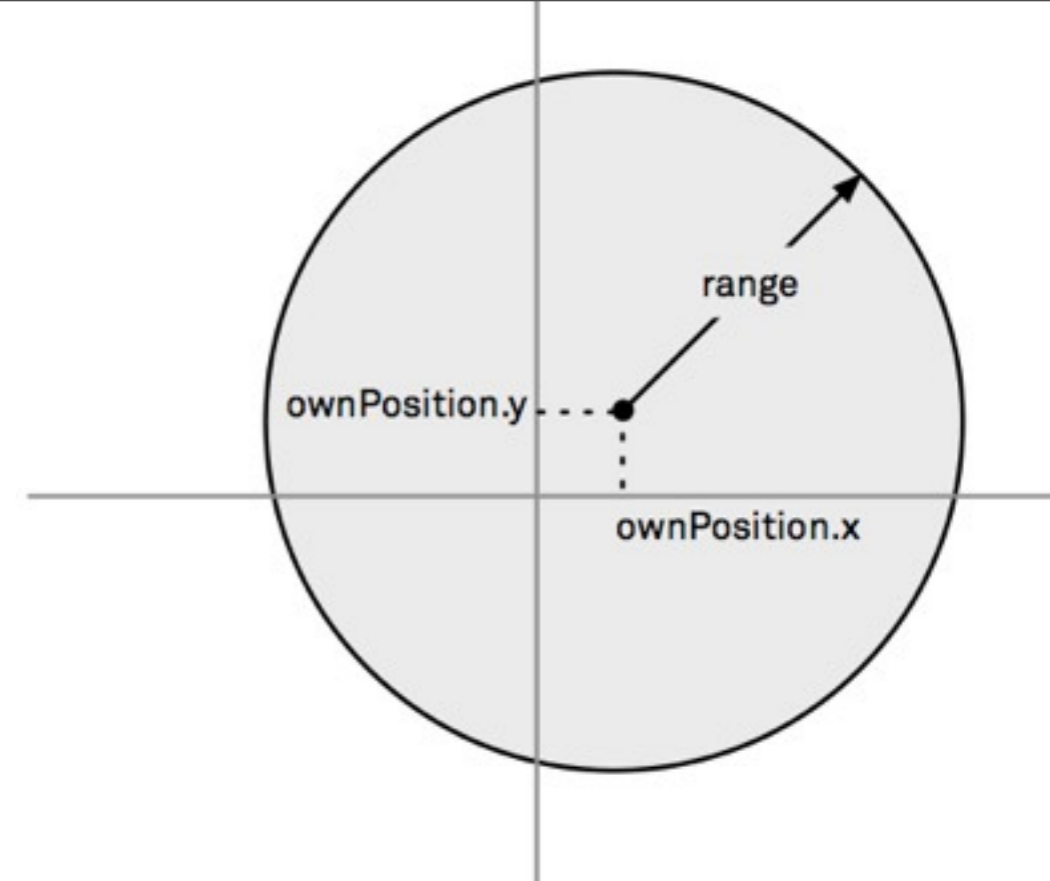


Second Pass - b

With map

What do we gain? lose?

```
(defn in-range-2b
  [position own-position range]
  (let [[dx dy] (map - position own-position)
        target-distance (Math/sqrt (+ (* dx dx) (* dy dy)))]
    (< target-distance range)))
```

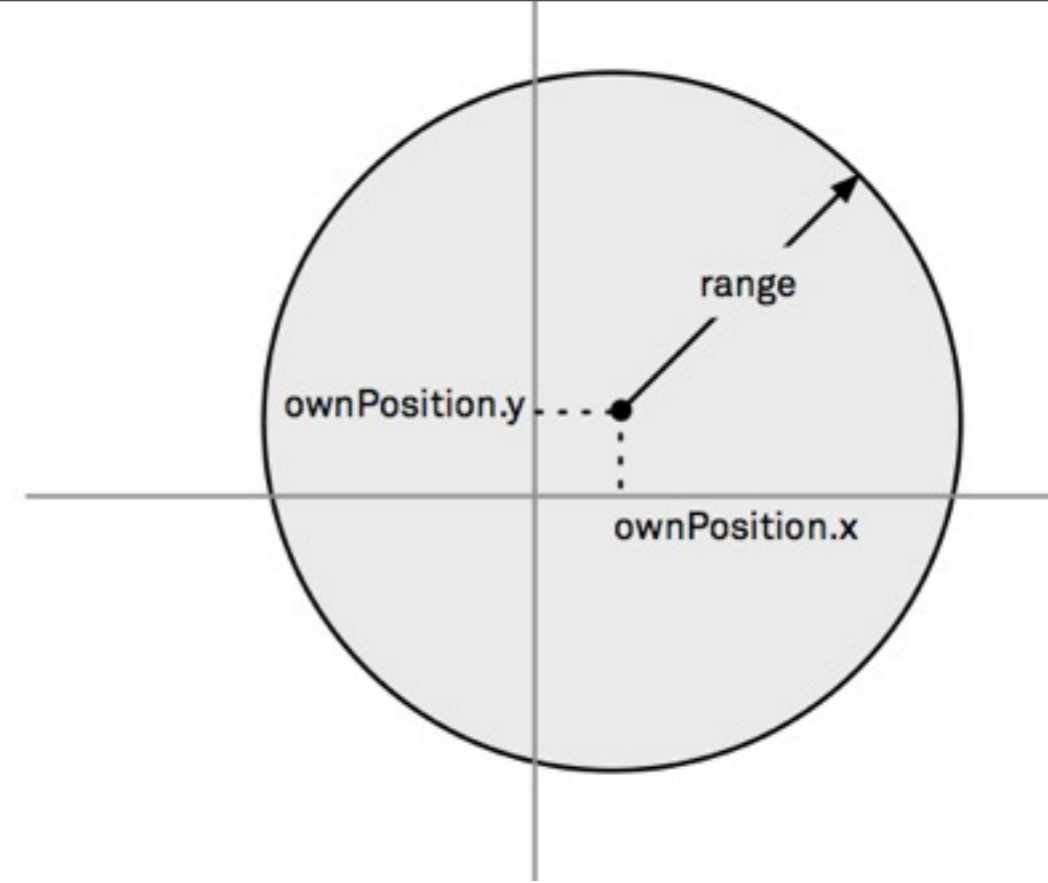


Second Pass - c

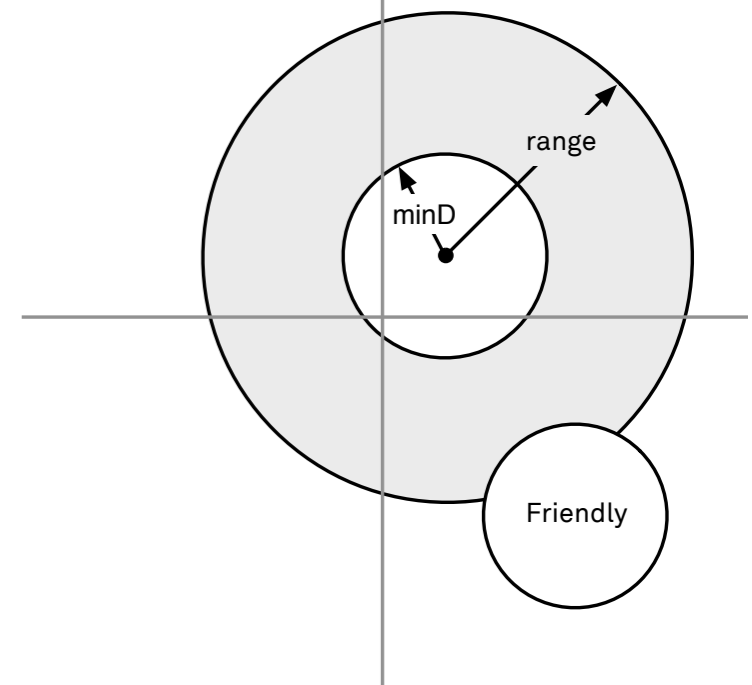
Using map & reduce

What do we gain? lose?

```
(defn in-range-2c
  [position own-position range]
  (let [delta (map - position own-position)
        target-distance (Math/sqrt (reduce + (map * delta delta)))]
    (< target-distance range)))
```



Third Pass



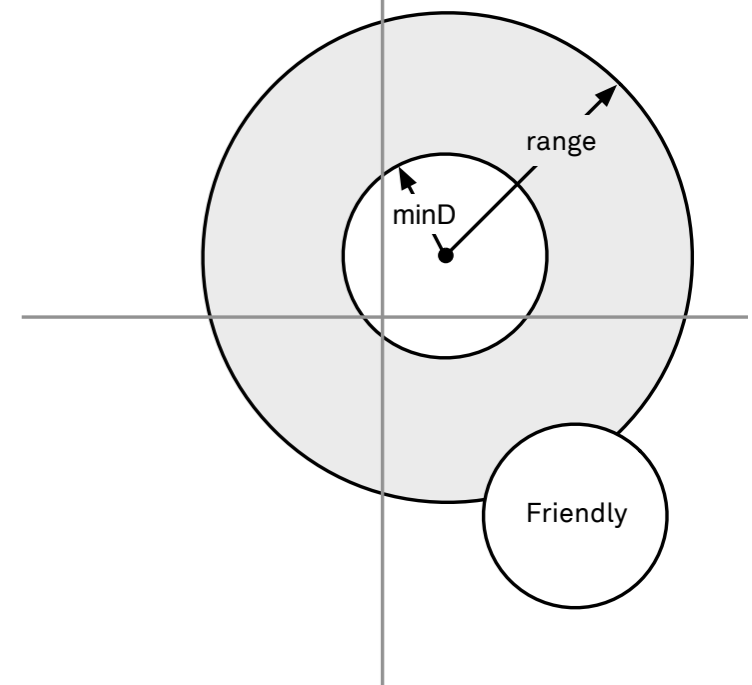
```
(defn in-range-3
  [safe-distance range own-position position friend-position]
  (let [delta (map - position own-position)
        target-distance (Math/sqrt (reduce + (map * delta delta)))
        friend-delta (map - position friend-position)
        target->friend (Math/sqrt (reduce + (map * friend-delta friend-delta)))]
    (and
      (< safe-distance target->friend)
      (< safe-distance target-distance range))))
```


Third Pass

```
(defn distance-between  
  [a b]  
  (let [delta (map - a b)]  
    (Math/sqrt (reduce + (map * delta delta))))))
```

```
(defn in-range-3a  
  [safe-distance range self target friend]  
  (and  
    (< safe-distance (distance-between friend target))  
    (< safe-distance (distance-between self target) range)))
```

```
(def in-torpedo-range (partial in-range-3a 1.5 20))  
(def in-cannon-range (partial in-range-3a 3 500))
```



What is the Abstraction?

What are we doing?

Dealing with circles

shapes

Union

Intersection

Complement

Is a point in a shape

circle - returns a function

```
(defn circle
  ([radius]
   (circle [0 0] radius))
  ([center radius]
   (fn
     [point]
     (<= (distance-between center point) radius))))
```

```
(def small-circle (circle 1))
```

```
(small-circle [0.5 0])      true
(small-circle [1 2])       false
```

outside

```
(defn outside  
  [shape]  
  (complement shape))
```

```
(def small-circle (circle 1))
```

```
((outside small-circle) [0.5 0])    false  
((outside small-circle) [1 2])     true
```

union

```
(defn union
  ([shape]
   shape)

  ([shape-a shape-b]
   (fn [point]
     (or (shape-a point) (shape-b point)))))

  ([shape-a shape-b & shapes]
   (fn [point]
     (let [all-shapes (conj shapes shape-a shape-b)]
       (reduce #(or %1 (%2 point)) false all-shapes)))))
```

Higher Level in range

```
(defn in-range-4
  [safe-distance range self target friend]
  (let [self-safe-zone (outside (circle self safe-distance))
        friend-safe-zone (outside (circle friend safe-distance))
        weapon-area (circle self range)
        target-zone (intersection weapon-area friend-safe-zone self-safe-zone)]
    (target-zone target)))
```

Game of Life

Conway's Game of Life







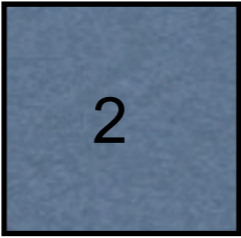

Any live cell with fewer than two live neighbours dies, as if caused by under-population




Any live cell with two or three live neighbours lives on to the next generation

Any live cell with more than three live neighbours dies, as if by overcrowding

Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction

	1	1	1		
	2	 1	2		
	3	 2	3		
	2	 1	2		
	1	1	1		

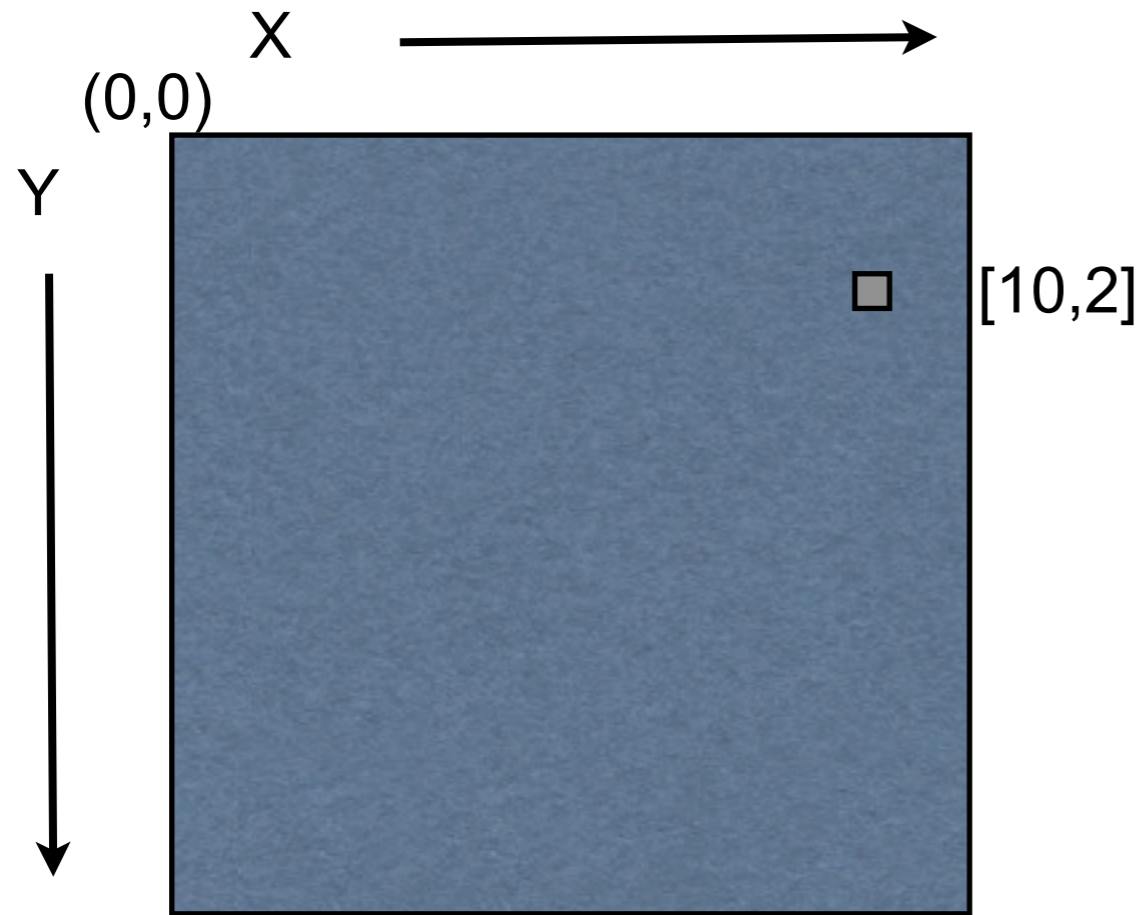
					

Basic Algorithm

Count the number of live cells neighboring each cell

Apply rules to compute next generation

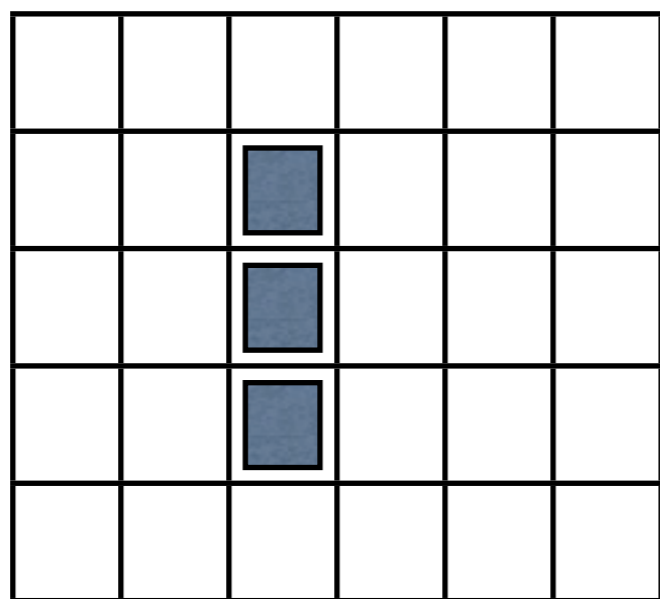
Representing the Data



Each live cell represented
In Clojure by a vector

[x, y]
[10,2]

All live cells are sequence
of vectors



[[2 1] [2 2] [2 3]]

Neighbors of a Cell

[4 4] [5 4] [6 4]

[4 5] [5 5] [6 5]

[4 6] [5 6] [6 6]

[x y]



[x±1 y±1] without [x y]

General Rule

Write function to process one element

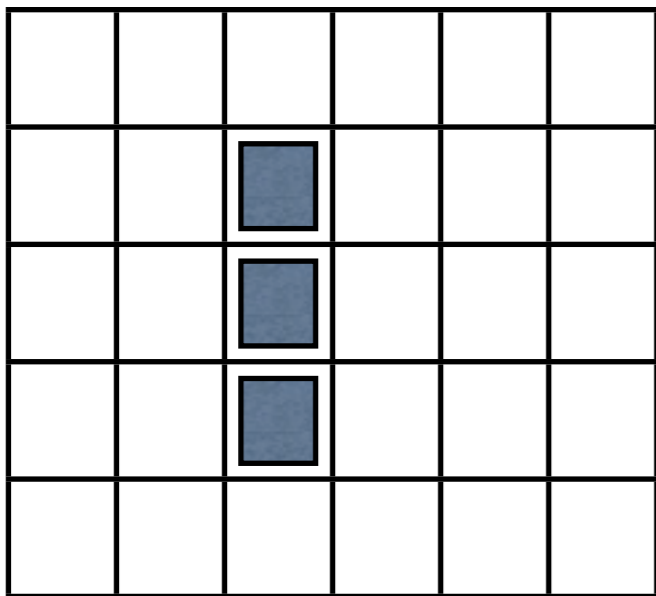
Use higher order function (map, filter, ...) to process collection of element

Finding all the neighbors of a point

```
(defn neighbors
  "Determines all the neighbors of a given coordinate"
  [[x y]]
  (for [dx [-1 0 1]
        dy [-1 0 1]
        :when (not= 0 dx dy)]
    [(+ dx x) (+ dy y)]))
```

```
(neighbors [1 1])      ([0 0] [0 1] [0 2] [1 0] [1 2] [2 0] [2 1] [2 2])
```

```
(neighbors [0 0])      ([-1 -1] [-1 0] [-1 1] [0 -1] [0 1] [1 -1] [1 0] [1 1])
```

[[2 1] [2 2] [2 3]]

(map neighbors [[2 1] [2 2] [2 3]])

((([1 0] [1 1] [1 2] [2 0] [2 2] [3 0] [3 1] [3 2])
([1 1] [1 2] [1 3] [2 1] [2 3] [3 1] [3 2] [3 3])
([1 2] [1 3] [1 4] [2 2] [2 4] [3 2] [3 3] [3 4])))

But neighbors returns sequence for each element

(mapcat neighbors [[2 1] [2 2] [2 3]])

(([1 0] [1 1] [1 2] [2 0] [2 2] [3 0] [3 1] [3 2]
[1 1] [1 2] [1 3] [2 1] [2 3] [3 1] [3 2] [3 3]
[1 2] [1 3] [1 4] [2 2] [2 4] [3 2] [3 3] [3 4]))

Need to count each time cell is in the list

(frequencies [2 1 1 2 3 2])

{2 3, 1 2, 3 1}

(->> [[2 1] [2 2] [2 3]]
(mapcat neighbors)
frequencies)

{[2 1] 1, [3 2] 3, [1 0] 1, [2 2] 2, [3 3] 2,
[1 1] 2, [2 3] 1, [3 4] 1, [1 2] 3, [2 4] 1,
[1 3] 2, [1 4] 1, [3 0] 1, [3 1] 2, [2 0] 1}

(let [neighbor-cells (mapcat neighbors [[2 1] [2 2] [2 3]])]
frequencies neighbor-cells))

(frequencies (mapcat neighbors [[2 1] [2 2] [2 3])))

Store live cells in a set
Insure no duplicates
Can use as function

```
(defn survive?  
  [neighbor-count]  
  (#{3 2} neighbor-count))
```

```
(defn birth?  
  [neighbor-count]  
  (#{3} neighbor-count))
```

```
(defn should-be-live?  
  [live-cells-set cell neighbor-count]  
  (if (live-cells-set cell)  
      (survive? neighbor-count)  
      (birth? neighbor-count)))
```

```
(defn next-generation
  [live-cells-set]
  (let [cell-counts (frequencies (mapcat neighbors live-cells-set))
        next-generation (for [[cell neighbor-count] cell-counts
                              :when (should-be-live? live-cells-set cell neighbor-count)]
                          cell)]
    (set next-generation)))
```

```
(next-generation (set [[2 1] [2 2] [2 3]]))           #{[3 2] [2 2] [1 2]}
```

```
(-> #{[2 1] [2 2] [2 3]}
    next-generation
    next-generation)                                #{[2 1] [2 2] [2 3]}
```

Some Fun

```
(def all-moves (iterate next-generation #{[2 1] [2 2] [2 3]}))
```

```
(defn next-move  
  []  
  (let [next (first all-moves)]  
    (alter-var-root (var all-moves) rest)  
    next))
```

```
(next-move)           #{[2 1] [2 2] [2 3]}
```

```
(next-move)           #{[3 2] [2 2] [1 2]}
```

```
(next-move)           #{[2 1] [2 2] [2 3]}
```

But that is not the version of the game found on-line

Stepper

Input

neighbors - function that computes neighbors of cells

birth? - function that determines if cell should be filled

survive? -function that determines if cell should remain filled

Output

function that produces next generation

```
(defn stepper
  [neighbors birth? survive?]
  (fn [cells]
    (set (for [[loc n] (frequencies (mapcat neighbors cells))
              :when (if (cells loc)
                        (survive? n)
                        (birth? n))]
             loc))))
```

```
(defn stepper
  [neighbors birth? survive?]
  (fn [cells]
    (set (for [[loc n] (frequencies (mapcat neighbors cells))]
              :when (if (cells loc)
                        (survive? n)
                        (birth? n)))
          loc))))
```

```
(def conway-stepper (stepper neighbors #{3} #{2 3}))
```

Selects existing live cell if 2 or 3 neighbors are live

Select dead cell if 3 neighbors are live

Cheap IO

```
(defn create-world
  "Creates rectangular world with the specified width and height.
  Optionally takes coordinates of living cells."
  [w h & living-cells]
  (vec (for [y (range w)]
           (vec (for [x (range h)]
                    (if (contains? (first living-cells) [y x]) "X" " "))))))
```

```
(create-world 4 4)
```

```
[[ " " " " " " " " " " ]
 [ " " " " " " " " " " ]
 [ " " " " " " " " " " ]
 [ " " " " " " " " " " ]]
```

```
(create-world 4 4 #{[0 0] [1 1] [2 2]})
```

```
[[ "X" " " " " " " " " " " ]
 [ " " "X" " " " " " " " " ]
 [ " " " " "X" " " " " " " ]
 [ " " " " " " " " " " " " ]]
```

Running the Game

```
(defn conway
```

```
"Generates world of given size with initial pattern in specified generation"
```

```
[[w h] pattern iterations]
```

```
(->> (iterate conway-stepper pattern)
```

```
      (drop iterations)
```

```
      first
```

```
      (create-world w h)
```

```
      (map println)))
```

Example

(conway [5 15] glider 0)

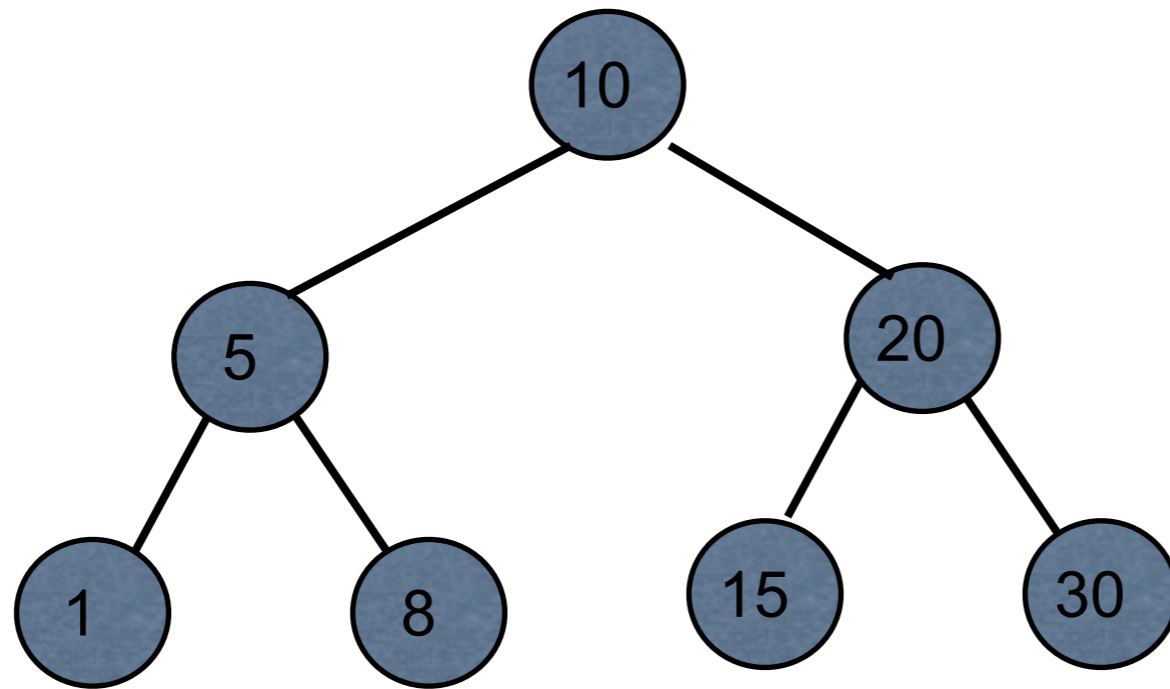
```
([ X      ]  
[  X      ]  
[X X X    ]  
[          ]  
[          ]  
nil nil nil nil nil)
```

(conway [5 15] glider 1)

```
([          ]  
[X X      ]  
[  X X    ]  
[  X      ]  
[          ]  
nil nil nil nil nil)
```

BST

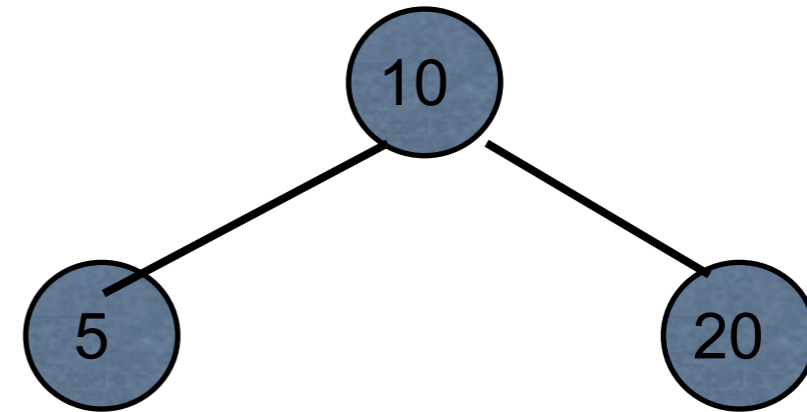
Binary Search Tree



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]]

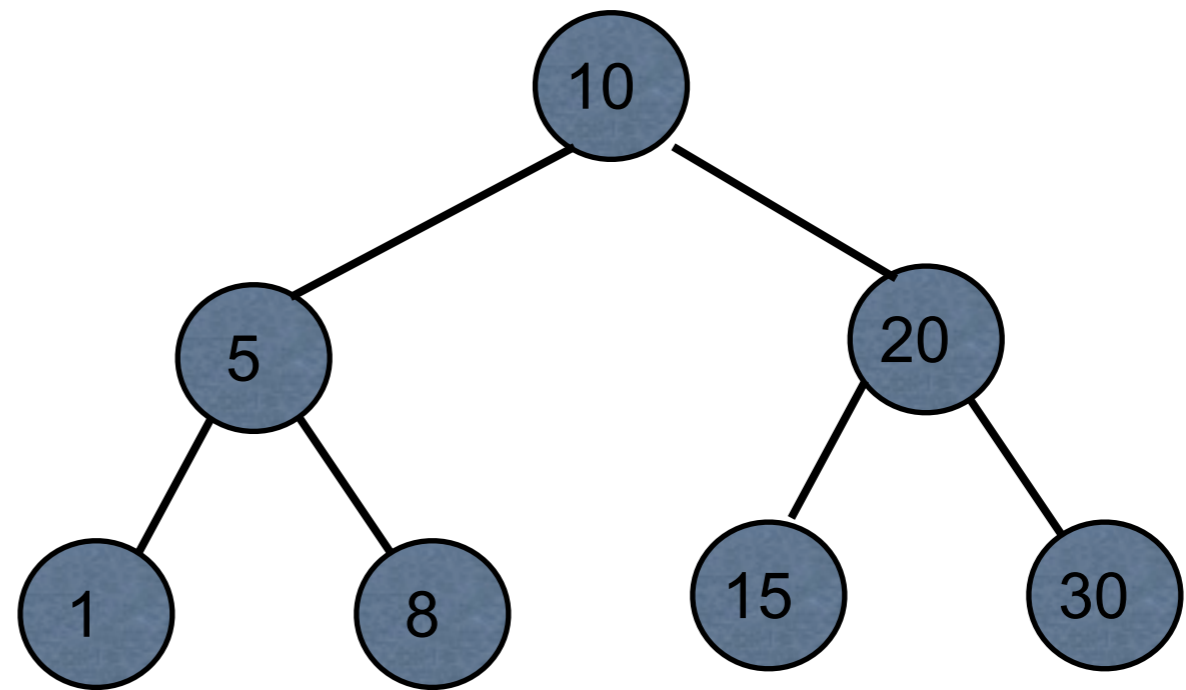
{:key 10, :left {:key 5 }, :right {:key 20}}

{: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))
```

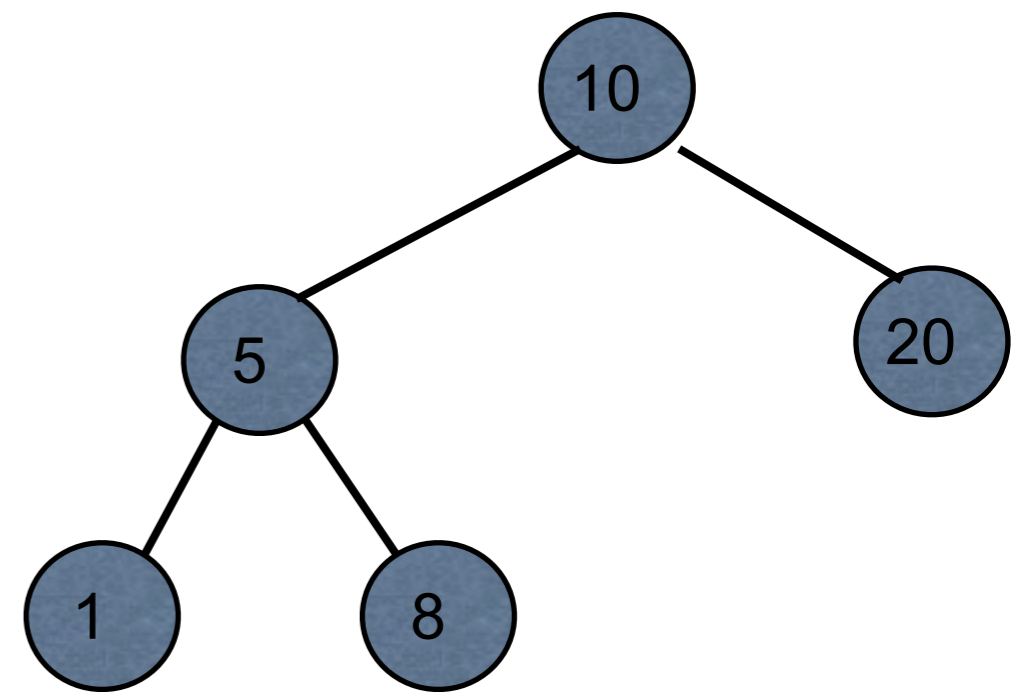
```
(defn value  
  [node]  
  (node 0))
```


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)
```

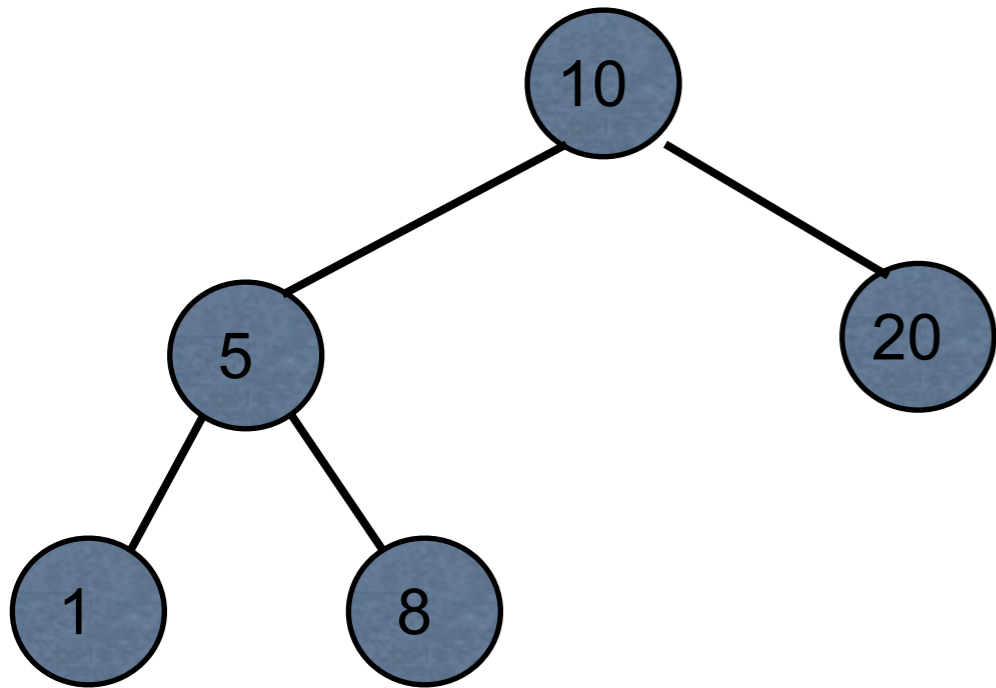


Standard Search

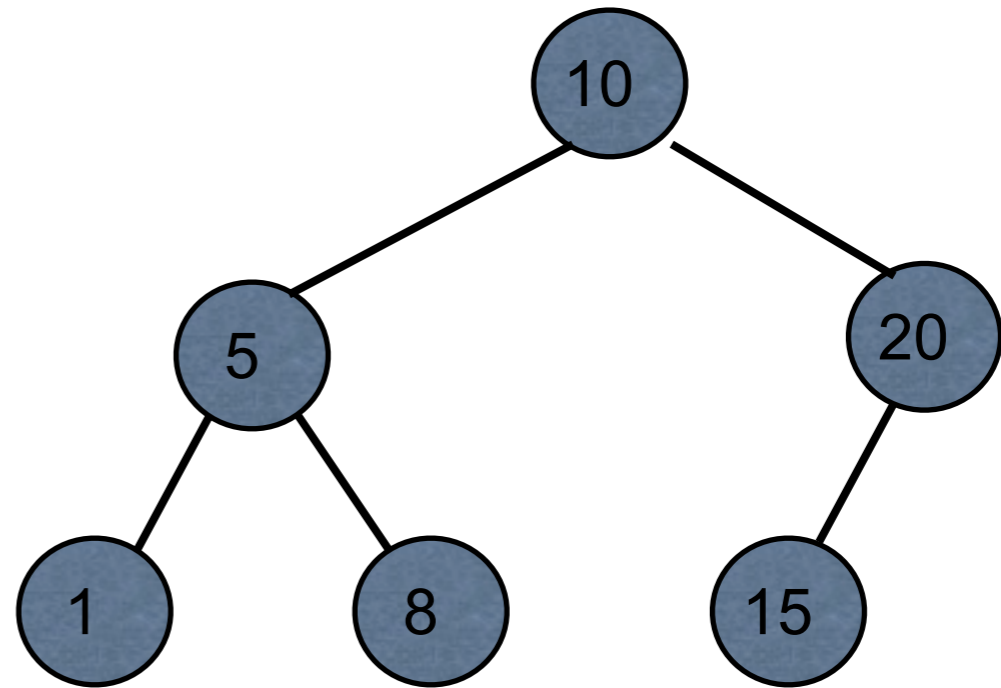
```
(defn find-key
  [tree k]
  (let [left (left-child tree)
        right (right-child tree)
        value (value tree)]
    (cond
      (= k value) k
      (and left (< k value)) (find-key left k)
      (and right (> k value)) (find-key right k)
      :default nil)))
```

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

Inserting



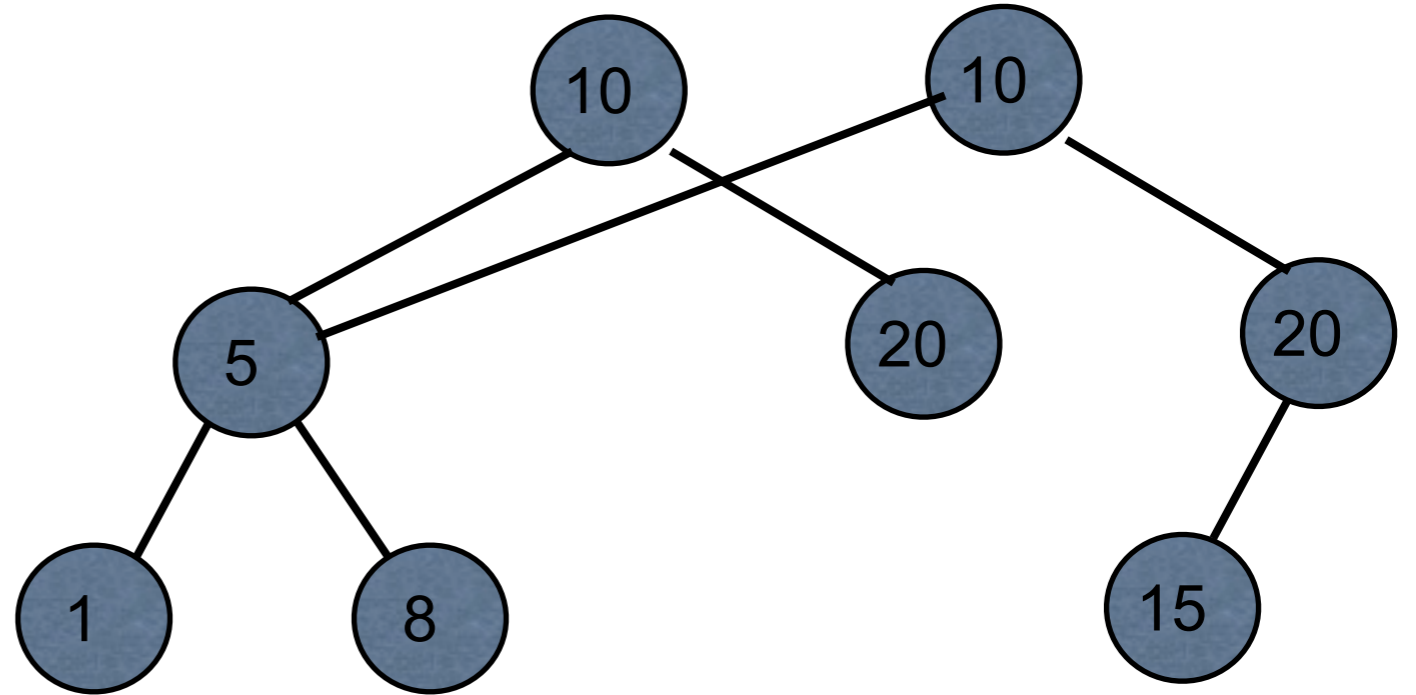
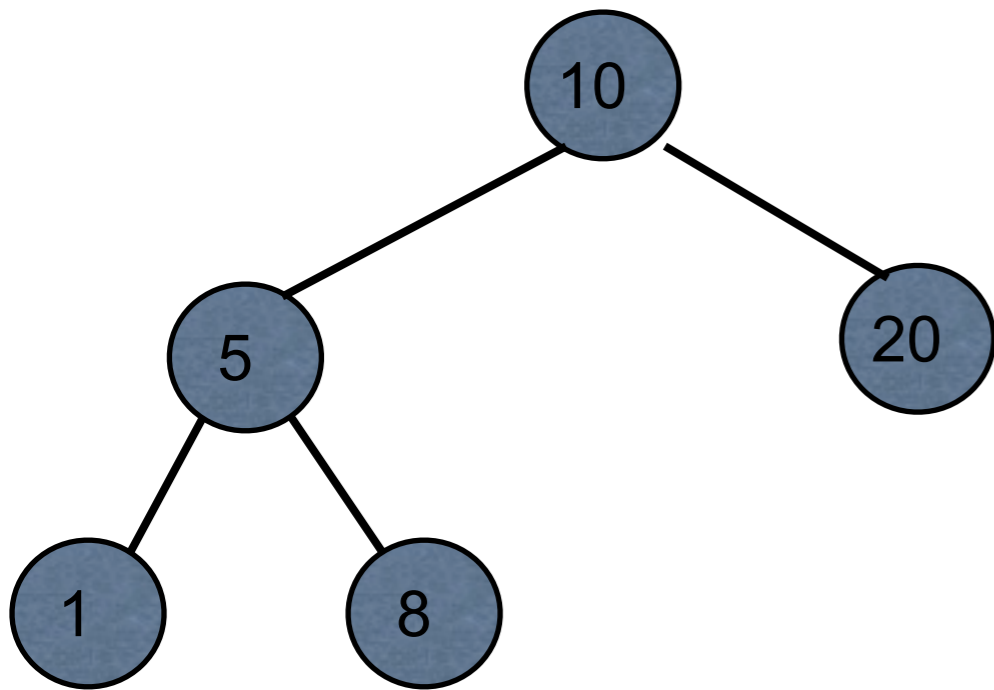
Add 15



But we have persistence & immutability

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}]
```

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

(defn position-of	(position-of tree 10)	nil
"Return path to k in tree"		
[tree k]	(position-of tree 5)	(1)
(let [left (left-child tree)	(position-of tree 1)	(1 1)
right (right-child tree)		
value (value tree)]	(position-of tree 8)	(1 2)
(cond		
(= k value) nil	(position-of tree 20)	(2)
(and left (< k value)) (cons 1 (position-of left k))		
(< k value) [1]	(position-of tree 15)	(2 1)
(and right (> k value)) (cons 2 (position-of right k))		
(> k value) [2])))	(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

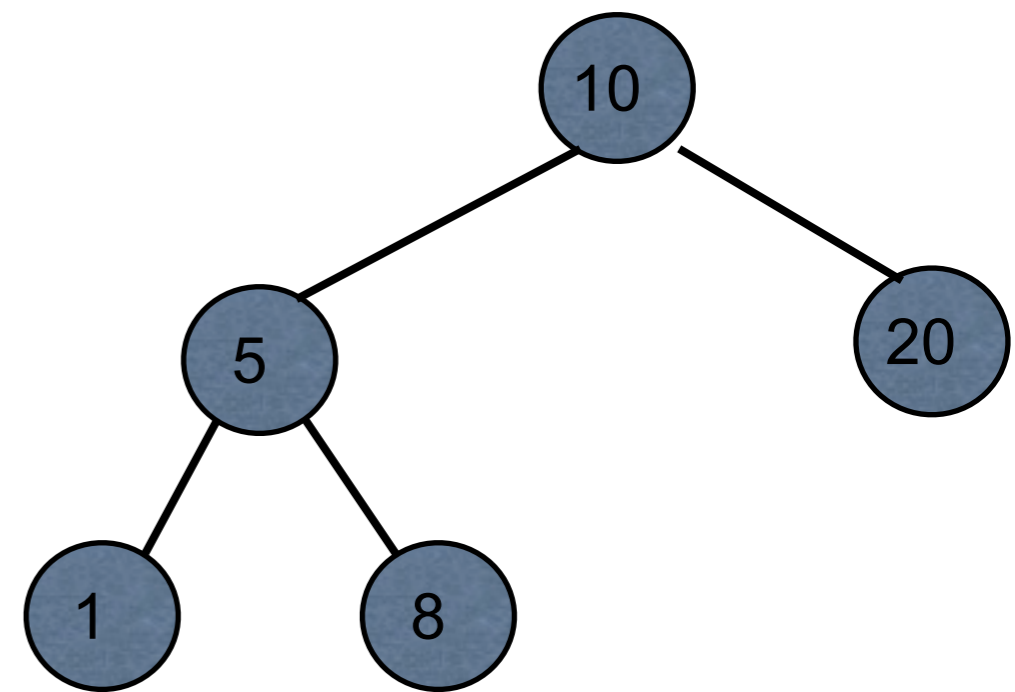
- node, root

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]]
```

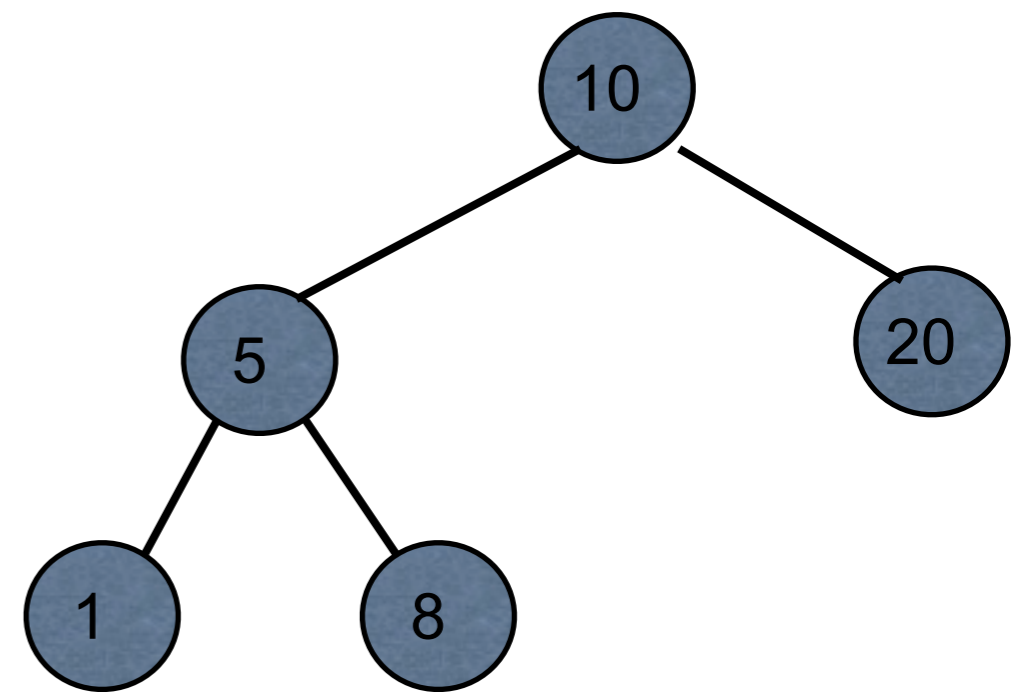
```
(-> large-tree
  zip/vector-zip      10
  zip/down
  zip/node)
```

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

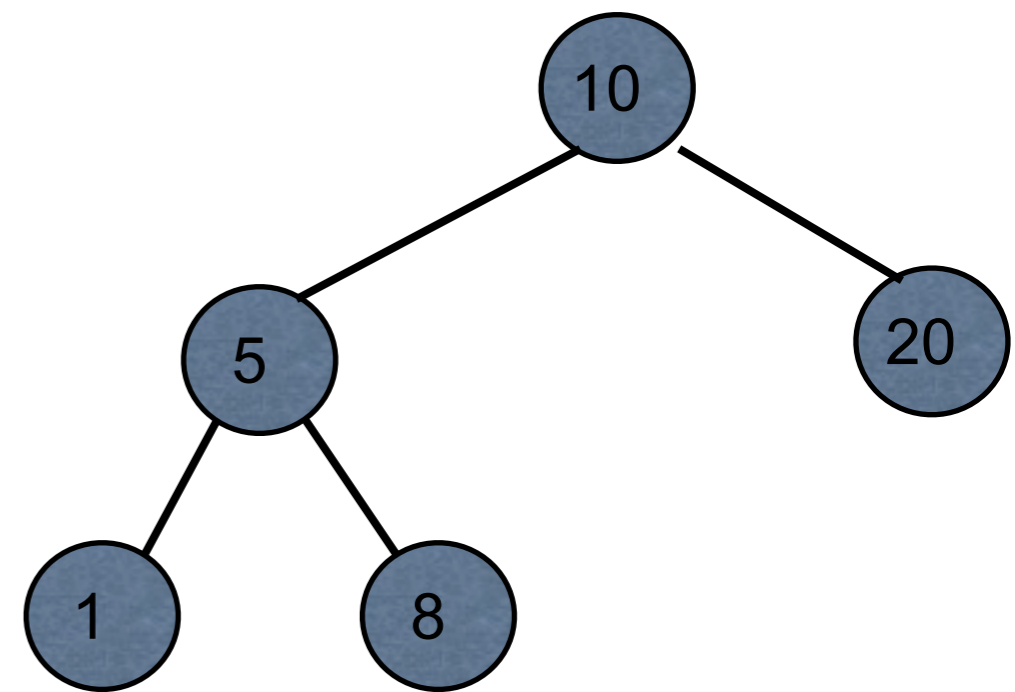
```
(-> large-tree  
  zip/vector-zip  
  zip/down  
  zip/right           5  
  zip/down  
  zip/node)
```

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
```

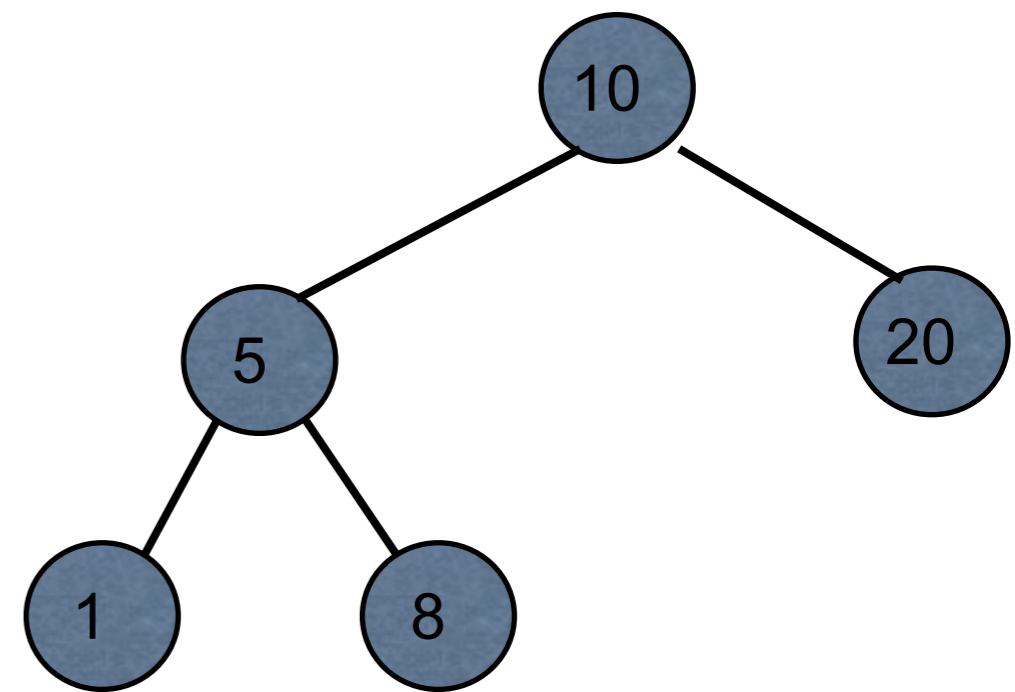
```
  zip/down
```

```
  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]])
```

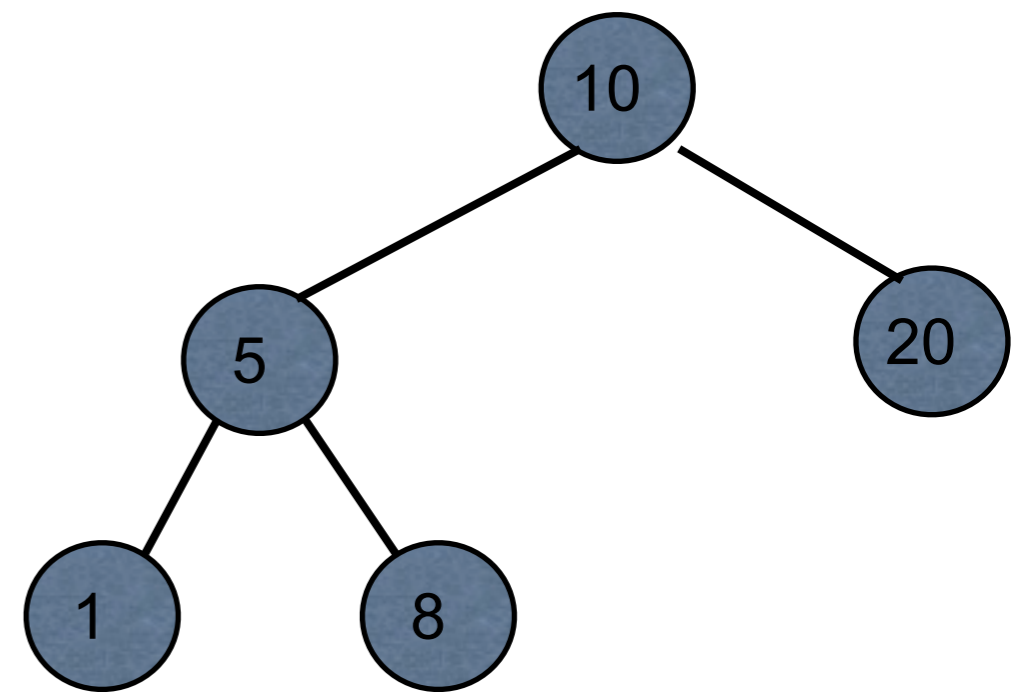


```
(-> large-tree
  zip/vector-zip
  zip/down
  zip/right
  zip/right
  (zip/replace [50 nil nil])
  zip/root)
```

```
[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

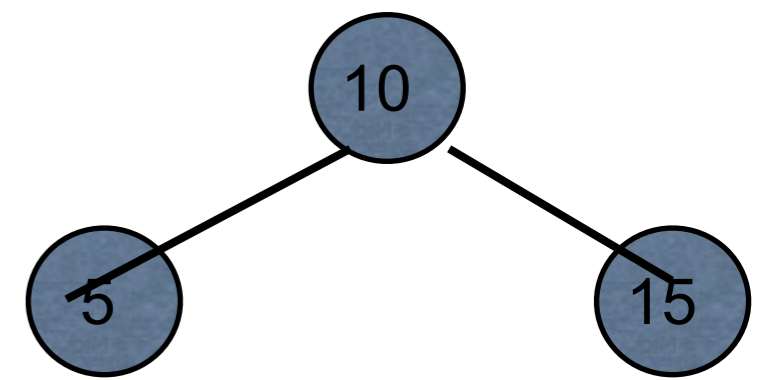
```
(defn tree->children  
  [map]  
  [(:value map) (:left map) (:right map)])
```

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-map))
```


Shorter Way - partial

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