CS 596 Functional Programming and Design
Fall Semester, 2014
Doc 9 Some Higher Order Functions, Examples
Oct 2, 2014

Common Operations on Collections

Combine elements into one result
    sum all elements,
    min

Transform each element
    add 10 to each element

Pass each element as argument to function
    Print each element to standard out

Select all elements that meet a condition
    all elements greater than 10

Select one elements that meet a condition
    First element greater than 10

Group elements by some criteria
    group strings by size

Map, Reduce, Filter

Higher order functions

Very important

- Map
  - Apply a function to each element of a collection, return resulting collection
  - Ruby - collect, map
  - Smalltalk - collect

- Filter
  - Returns elements of collection that make

- Reduce
  - Applies function

Reduce

```
(reduce + [1 2 3 4])                    10
(reductions + [1 2 3 4])                (1 3 6 10)
(reduce small-add [1 2 3 4 5 6])        6
```

```
(defn small-add
  [subresult x]
  (if (< x 4)
    (+ subresult x)
    (reduced subresult)))
```

Map

Map - the noun

`{:a 1 :c 10}`

Map - the verb

`(map inc [1 2 3])`   `(2 3 4)`

Map - the Verb

(map f coll)
(map f c1 c2)
(map f c1 c2 c3)
(map f c1 c2 c3 & colls)

(map inc [1 2 3])	(2 3 4)
(map + [1 2 3] [4 5 6])	(5 7 9)
(map + [1 2 3 4 5] [4 5 6])	(5 7 9)
(map inc #{1 2 3})	(2 4 3)
(map + [1 2 3] #{4 5 6})	(5 8 8)

map	Returns lazy sequence
mapv	Returns vector
pmap	Done in parallel, semi-lazy
map-indexed	f gets index & element

map-indexed

(map-indexed vector [:a :b :c])

([0 :a] [1 :b] [2 :c])

pmap

Distributes work among cores, not separate processors/machines

Operation needs to be computationally intense

```
(time (doall (map inc (range 10000))))          "Elapsed time: 4.73 msecs"

(time (doall (pmap inc (range 10000))))         "Elapsed time: 529.905 msecs"
```

Parallel Example

```
(defn long-running-job [n]
   (Thread/sleep 3000) ; wait for 3 seconds
   (+ n 10))


(time (doall (map long-running-job (range 4))))            12.005 secs

(time (doall (map long-running-job (range 8))))            24.005 secs


(time (doall (pmap long-running-job (range 4))))             3.01 secs

(time (doall (pmap long-running-job (range 8))))             3.01 secs

(time (doall (pmap long-running-job (range 64))))            6.01 secs
```

Since the job is not doing any real work pmap performs very well. It can use multiple threads on one processor and the threads can all perform at the same time

Slightly More Realistic Example

```
(defn long-running-job
  [n]
  (reduce + (take 10000000 (iterate #(Math/sin %) n))))
```

```
(time (doall (map long-running-job (range N))))
(time (doall (pmap long-running-job (range N))))
```

N	map time secs	pmap time secs
2	7.5	4.8
4	15.3	10.1

2.13 GHz Intel Core 2 Duo

Partition Size

One can control the size of data send to each thread

partition-all

filter

`(filter even? [1 2 3 4 5 6 7])`	`(2 4 6)`
`(first (filter even? [1 2 3 4 5 6 7]))`	`2`
`(filter #{3 5 9 12} [1 2 3 4 5 6 7])`	`(3 5)`

Specialized filter functions

```
(take-while neg? [-2 -1 0 1 2 3])                    (-2 -1)

(take-while neg? [-2 -1 0 -1 -2 3])                  (-2 -1)



(drop-while neg? [-1 -2 -6 -7 1 2 3 4 -5 -6 0 1])    (1 2 3 4 -5 -6 0 1)


(split-with #(< % 3) [1 2 3 4 5 1])                  [(1 2) (3 4 5 1)]


(split-with pred coll)                [(take-while pred coll) (drop-while pred coll)]
```

Sample Problem

Given a list of numbers
- Square each number
- Sum all the squares

```
double[] numbers = read the values
double sum = 0;

for (int k = 0; k < numbers.length; k++) {
     double item = numbers[k];
     sum =+ item*item
}


for (number in numbers)
     sum =+ number * number
```

```
(def numbers [1 2 3 4 5])


(reduce + (map #(%*%) numbers))
```

How

What

Map-Reduce Google

Inspired by functional programming map & reduce

Distributes data randomly across clusters

Map - filters & sorts

Reduce - summary operation

Google no longer uses Map-Reduce framework

Hadoop - open source implementation

Pig-Pen

Map-Reduce in Clojure

Developed and used at Netflix

Write map-reduce queries as programs

Process massive amounts of data on clusters of machines

Article

http://tinyurl.com/l7l9dgt

When Processing Collections Consider Using

map
reduce
filter
for
some
repeatedly
sort-by
keep
take-while
drop-while

Common Operations on Collections

Combine elements into one result	reduce
Transform each element	map
Pass each element as argument to function	for, doseq
Select all elements that meet a condition	filter, take-while, drop-while
Select one elements that meet a condition	(first (filter condition xs))
Group elements by some criteria	group-by, partition-by partition

Evaluating Lazy Sequences

(map println [1 2 3])	No output
(dorun (map println [1 2 3]))	Output, evaluates one at a time Returns nil
(doall (map println [1 2 3]))	Output, evaluates one at a time Returns head, All elements are in memory at once

Evaluating Lazy Sequences

```
(for [x [1 2 3]]
  (println x))
```

no output

```
(doseq [x [1 2 3]]
  (println x))
```

Output

Examples

Conway's Game of Life

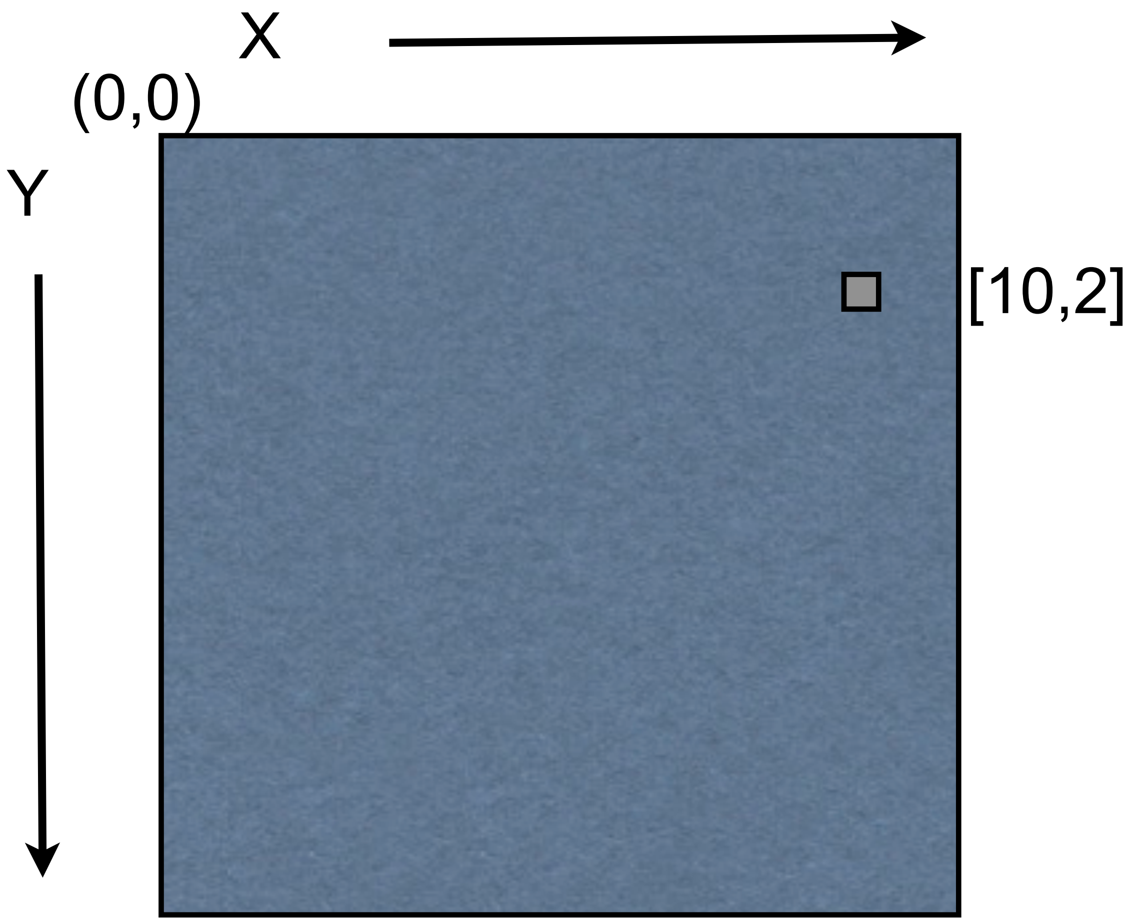
Any live cell with fewer than two live neighbours dies, as if caused by under-population

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

Representing the Data



Each live cell represented
In Clojure by a vector

[x, y]
[10,2]

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])
```

Source http://programmablelife.blogspot.com/2012/08/conways-game-of-life-in-clojure.html

Stepper

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

How stepper Works

```
[[2 3] [2 2]]
```

```
(mapcat neighbors cells)
```

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

```
(frequencies (mapcat neighbors cells))
```

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

```
(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

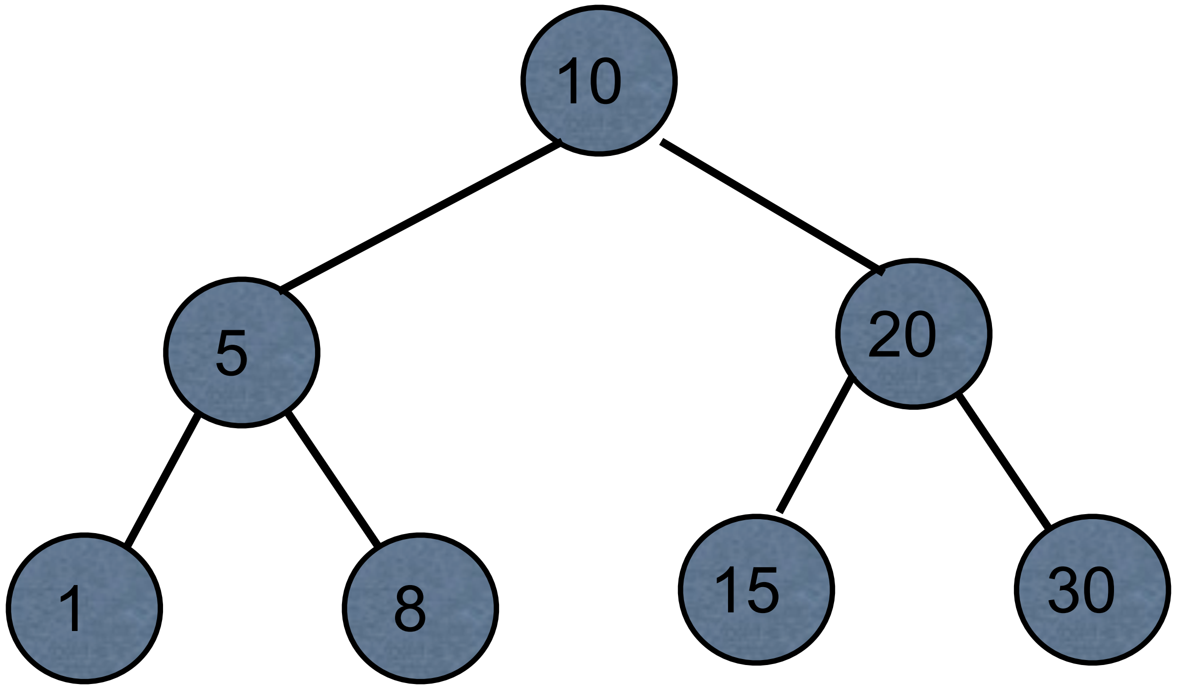
```
(conway [5 15] glider 0)

([  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)
```

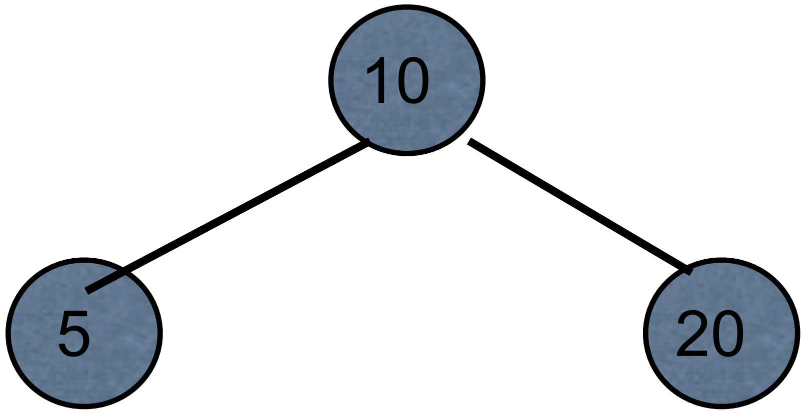
Binary Search Tree



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]]

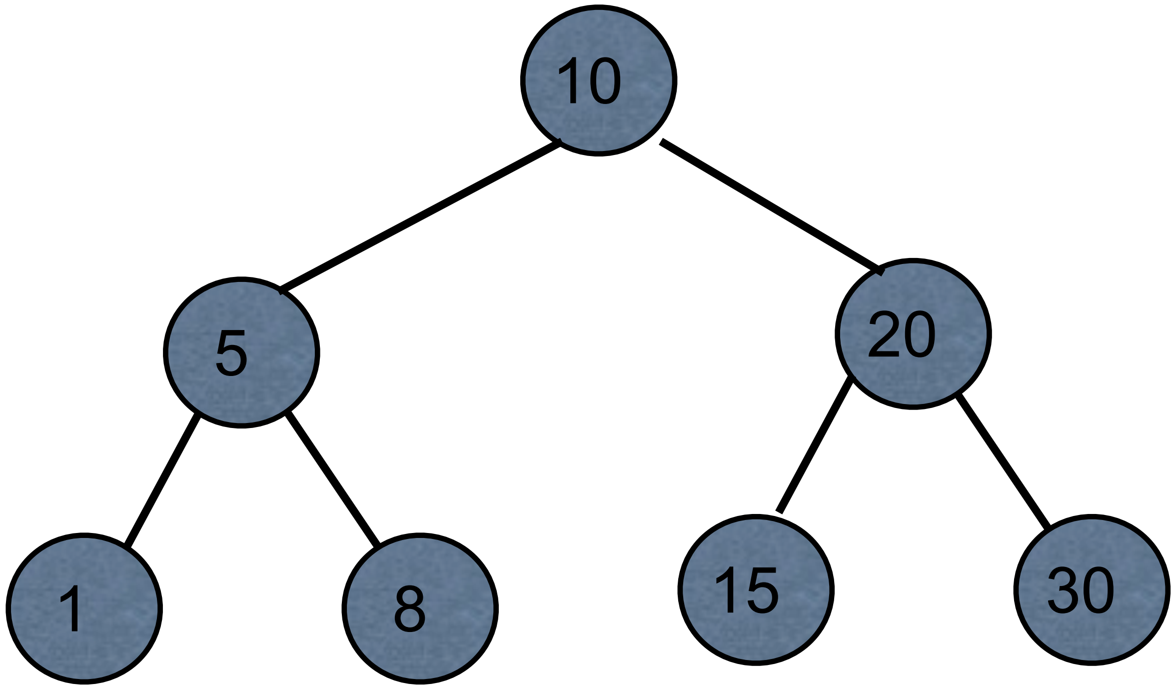
[[5 nil nil] 10 [20 nil nil]]

{: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]]])
```

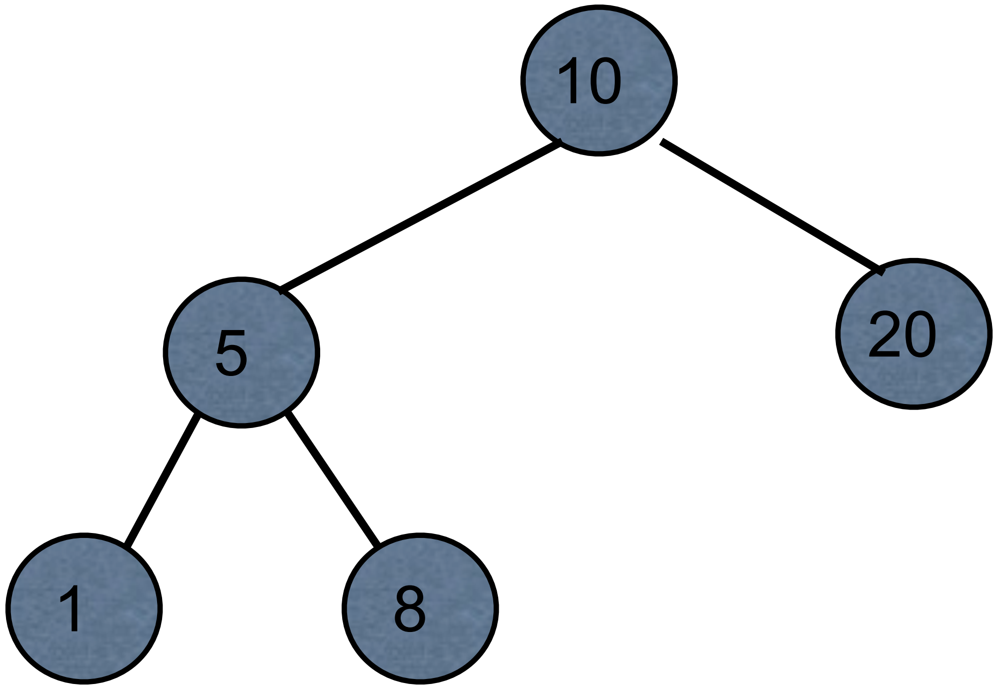
Hiding the Structure of Node

```
(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

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
  "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])))
```

```
(position-of tree 10)        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

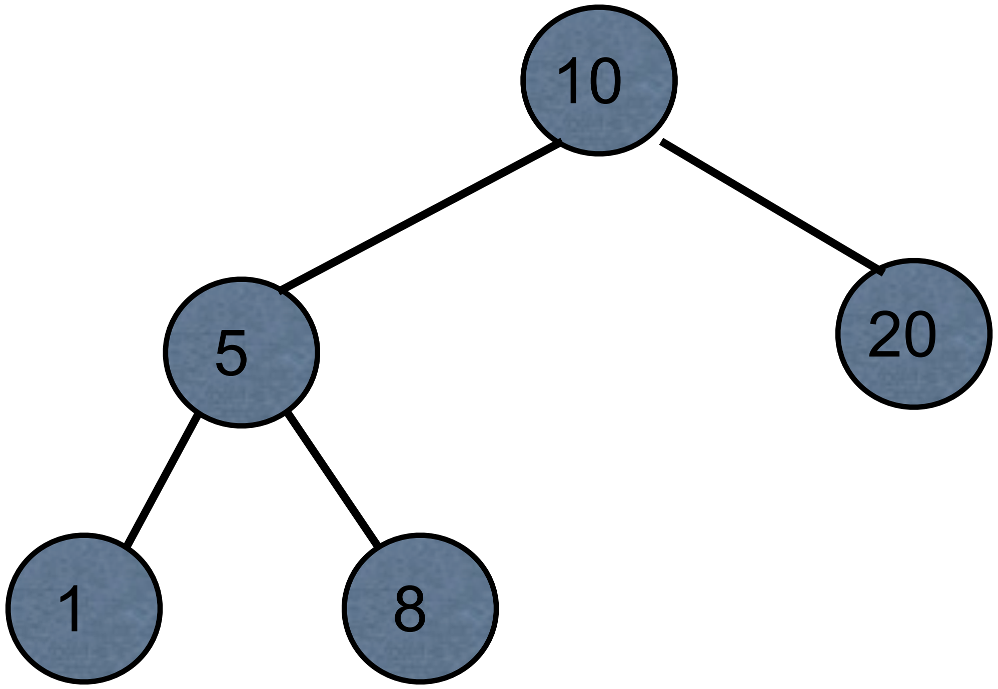
- seq-zip
- vector-zip
- xml-zip

Keeps track of where you are

Can go

up, down, left, right, next, prev

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]]
```

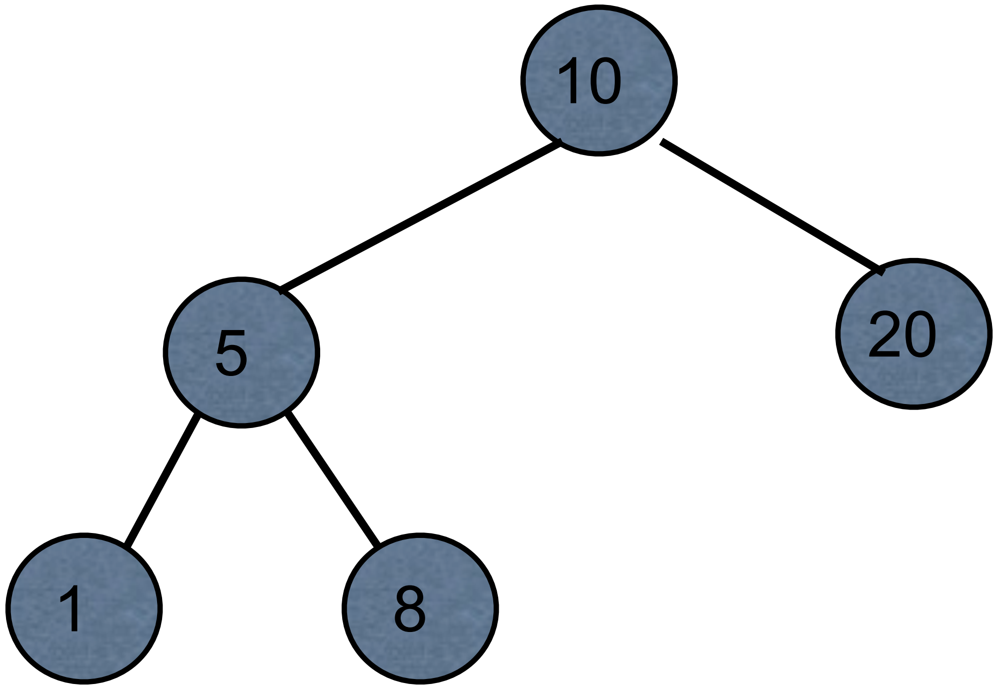
```
(-> large-tree
   zip/vector-zip
   zip/down
   zip/node)
```

```
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]])
```

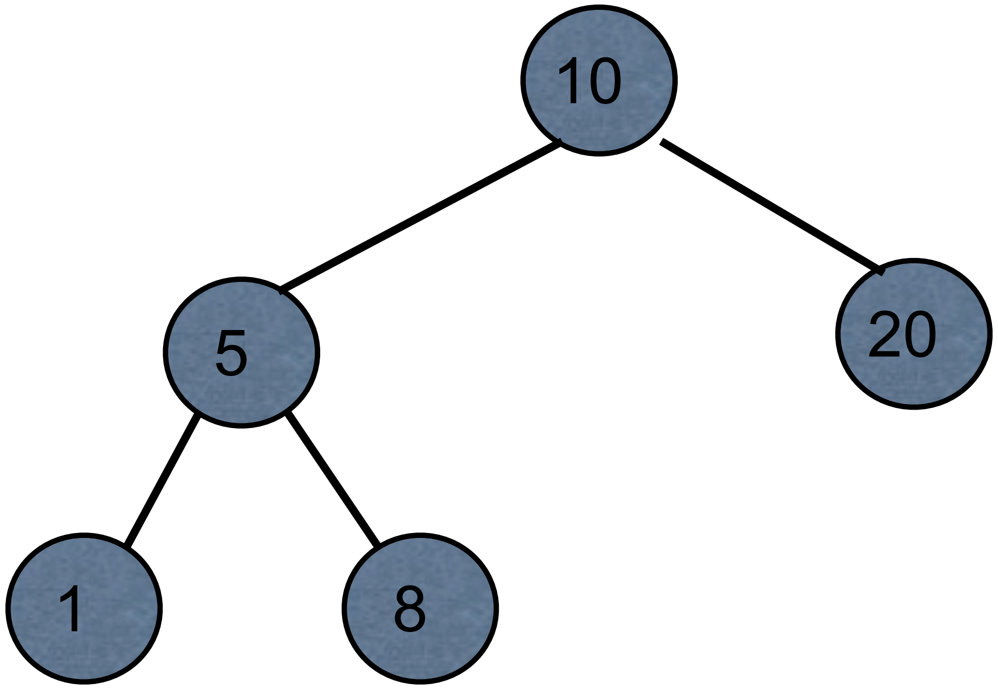
```
(-> large-tree
   zip/vector-zip
   zip/down
   zip/right
   zip/right
   zip/node)
```

[20 nil nil]

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

5

Zipper Examples

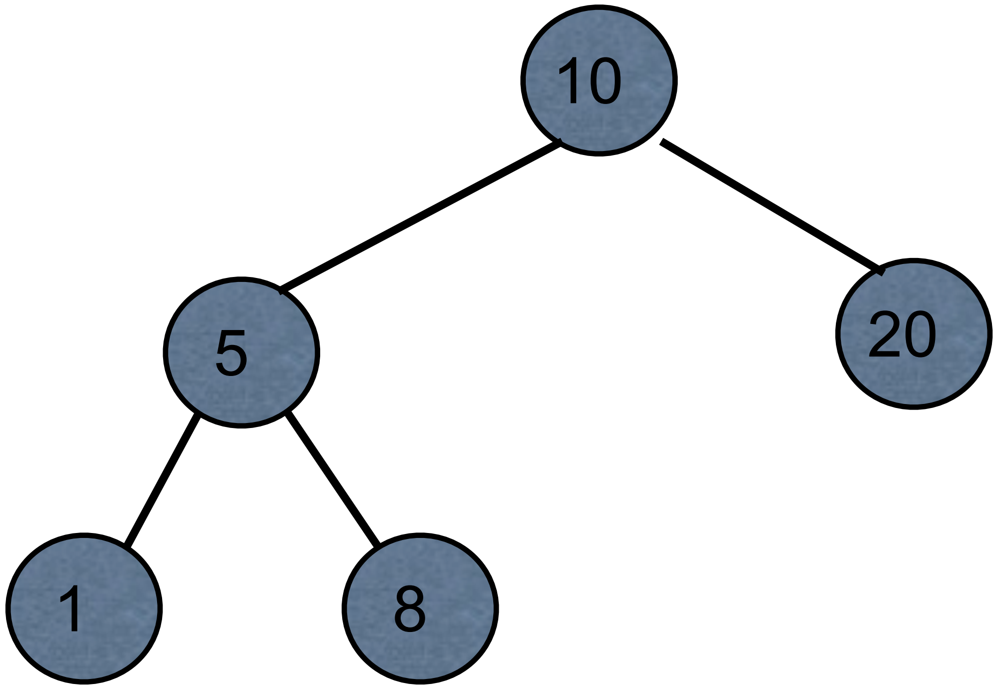


```
(ns basiclectures.basic-language.zip
  (:require [clojure.zip :as zip] ))
(def large-tree [10 [5 [1 nil nil] [8 nil nil]] [20 nil nil]])
```

```
(-> 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

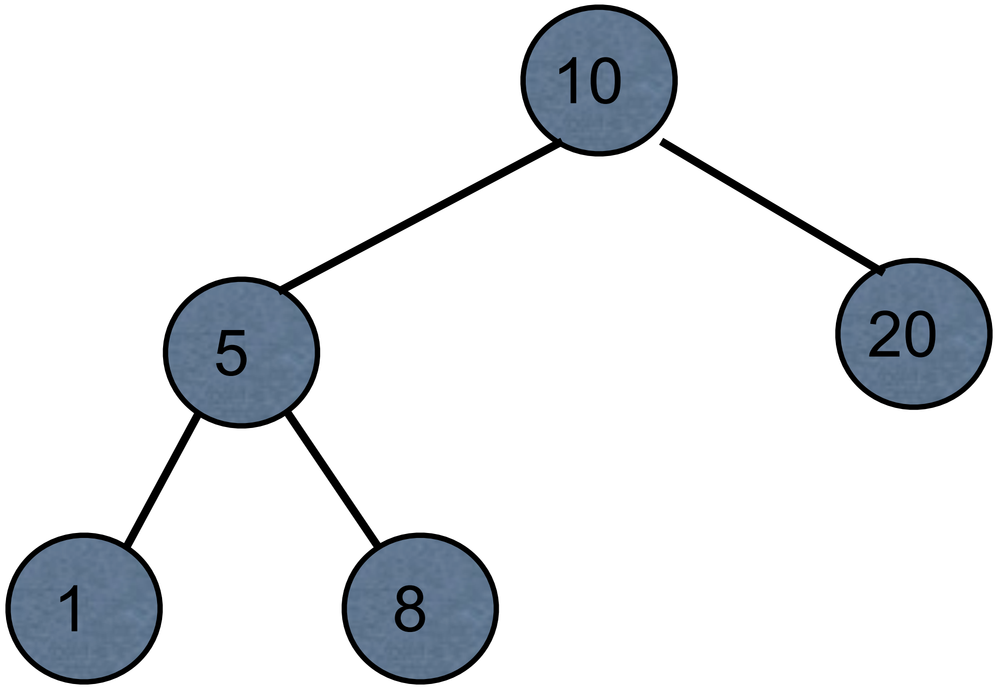


```
(ns basiclectures.basic-language.zip
  (:require [clojure.zip :as zip] ))
(def large-tree [10 [5 [1 nil nil] [8 nil nil]] [20 nil nil]])
```

```
(-> 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]]
```

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 "Smith", :first "Alan"}
 {:last "Zen", :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`

sdsu-nth

Given n can we produce

(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
```

Example from Joy of Clojure, Second Edition

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)
```