CS 596 Functional Programming and Design
Fall Semester, 2014
Doc 15 Assignment 4, Records, Protocols
Oct 28, 2014

The Internet

registration required.

Implementing min heap in clojure

I am trying to implement a min heap using either a vector or a map to represent it.To add a new value to the heap add it is added to the top of the heap. If the heap is empty the new value becomes the root.

If heap is not empty then the smaller of the two values (current value in heap and the new value) is kept in the root. The larger of the two values is added the sub-heap with the smallest height. If both sub-heaps have the same height then the larger of the two values is added to the left heap. The process is repeated until a value is added to a new node at the bottom of the heap.

I wrote the following code:

```
(= value member) tree

(< value member) (make-tree (insert(:left tree) member) value (:right tree))

:else (height tree value member))

(make-tree nil value nil)))

;; to find where to insert next element

(defn height[tree value m]

  (let [a(:left tree)

        b(:right tree) ]

    (cond

     (= a 0) (make-tree (insert (:left tree) value) m (:right tree)))

    :else (make-tree (:left tree) m (insert (:right tree) value))))


;;making min heap

(def heap
  (make-tree nil 30 nil)
  )

(-> heap
```

Min Heap In Clojure?

Implement a **min heap** using either a vector or a map. In the **min-heap** each node has a value and 0, 1 or 2 children heaps.To add a new value to the **heap** it is added to the top of the **heap**. If the **heap** is empty the new value becomes the root.If **heap** is not empty then the smaller of the two values (current value in **heap** and the new value) is kept in the root. The larger of the two values is added the sub-**heap** with the smallest height. If both sub-heaps have the same height then the larger of the two values is added to the left **heap**. The process is repeated until a value is added to a new node at the bottom of the
heap. I understand the logic, just not so much implementing it in **Clojure**.

```
;;make a tree

(defn make-tree[left value right]

{:left left :val value :right right})

;;to insert a value in the tree

(defn insert[tree value]

(if-let [member (:val tree)]

(cond

(= value member) tree

(< value member) (make-tree (insert(:left tree) member) value
(:right tree))

:else (height tree value member))

(make-tree nil value nil)))
```

Min Heap Implementation in Clojure with inserting elements from top (root)

0

Bids	Avg Bid (USD)	Project Budget (USD)
6	$88	$30 - $250

5 days, 11 hours left

OPEN

Project Description:

A min-heap is a heap in which the root has the smallest value in the heap. Normally one would implement a heap using a single array with the indexing indicating. In this case we want to use either a vector or a map to represent. In our min-heap each node has a value and 0, 1 or 2 children heaps. The root node of a min-heap contains the smallest value in the heap. If you pick any node in the heap it has a value that is equal to or less than all the nodes in either the left or right sub-heaps of the node. To add a new value to the heap add it to the top of the heap. If the heap is empty the new value becomes the root.
If heap is not empty then the smaller of the two values (current value in heap and the new value) is kept in the root. The larger of the two values is added the sub-heap with the smallest height. If both sub-heaps have the same height then the larger of the two values is added to the left heap. The process is repeated until a value is added to a new node at the bottom of the heap. (There are better ways of adding to a min-heap, but this is the way we are to use in this case.)

Code is to have the following functionality:
1. Add elements to the min-heap. You can assume that the programmer will not mix different types in a single heap. There is no need to implement delete.
2. Print out the values in the heap in pre order (root, the left sub-heap, right sub-heap).
3. Print out the strings that end in "ing" in the heap in pre order. Heap should work for both numbers and string but won't be a mix of both.

Do not flatten your heap to traverse it.

This needs to be implemented in CLOJURE programming language using Light Table IDE.
I need this done by this Sunday Oct 19th 2014.

About the employer:

(0 Reviews)

VERIFIED

Assignment 4

Decisions to Make

How to represent the heap

- Map
- Vector

Height

- Compute Each time
- Store in each node
- memoize

Insertion

- assoc-in
- Zipper
- Recursion

Edge Cases

- Isolate

Decisions

Height - Store in each node

Computing each time - O(N*N)

memoize - no way to flush the cache

Decisions

Heap representation

{:key 5 :height 1 :left nil :right nil}

Insertion

Use recursion

Values can change in nodes from root to leave

Height of tree is O(lg(N))

Height

```
(defn height
  "Returns height of tree.
  0 if tree is nil or empty"
  [tree]
  (get tree :height 0))
```

```
(defn height
  "Returns height of tree.
  0 if tree is nil or empty"
  [tree]
  (if (seq tree)
     (tree :height)
     0)
```

```
(defn make-tree
  [key left-tree right-tree]
  (let [left-height (height left-tree)
        right-height (height right-tree)
        max-child-height (max left-height right-height)]
    {:key key :height (inc max-child-height) :left left-tree :right right-tree}))
```

```
(defn make-tree
  [key left-tree right-tree]
  (let [[left-height right-height] (map height [left-tree right-tree])
        max-child-height (max left-height right-height)]
    {:key key :height (inc max-child-height) :left left-tree :right right-tree}))
```

Using Map-reduce

```
(defn make-tree
  [key left-tree right-tree]
  (let [heights (map height [left-tree right-tree])
        max-height (reduce max heights)]
    {:key key :height (inc max-height) :left left-tree :right right-tree}))
```

Helper Function

```
(defn left-taller?
  [left-tree right-tree]
  (> (height left-tree) (height right-tree)))
```

```
(defmulti insert (fn [tree key & keys](when-not (seq tree) :empty)))

(defmethod insert :empty
  insert-empty-tree

  [_ key]
  {:key key :height 1})

(defmethod insert :default
  insert-has-tree

  ([{:keys [key left right]} insert-key]
  (let [[small large] (sort [key insert-key])]
    (if (left-taller? left right)
      (make-tree small left (insert right large))
      (make-tree small (insert left large) right))))

  ([tree key & keys]
   (let [all-keys (cons key  keys)]
     (reduce insert tree all-keys))))
```

```
(insert {} 1)
(insert nil 1)
(insert {:key 2, :height 1} 1)
(insert {:key 5, :height 1} 1 2 3 4)
```

But multiple arguments inconvenient

Supporting (insert tree collection)

```
(defn insert-selecter
  [tree key]
  (cond
   (coll? key) :multi-insert
   (not-empty tree) :default
   :else :empty-tree))

(defmulti insert insert-selecter)

(defmethod insert :empty-tree
  insert-empty-tree
  [_ key]
  {:key key :height 1})
```

Insert Methods

```
(defmethod insert :multi-insert
  multi-insert
  [tree keys]
  (reduce insert tree keys))

(defmethod insert :default
  insert-has-tree
  ([{:keys [key left right]} insert-key]
   (let [[small large] (sort [key insert-key])]
     (if (left-taller? left right)
       (make-tree small left (insert right large))
       (make-tree small (insert left large) right)))))
```

Handling Preorder

```
(defn pre-order
  [tree]
  (when (seq tree)
    (let [{:keys [key left right]} tree]
      (lazy-cat [key] (pre-order left) (pre-order right)))))
```

```
(pre-order (insert {} [1 2 3 4]))
```

```
(filter #(.contains % "ing") (pre-order (insert {} ["cat" "running" "rat" "fooling"])))
```

Dynamic Dispatch

Selecting which implementation of an operation to call at run time

In Java

foo.bar()

Which bar method to run is determined at at run time

Single Dispatch

Which operation is called is determined by one value

In Java

foo.bar()

Which bar method to run is determined at at run time based only on receiver

Multiple Dispatch

Which operation is called is determined by multiple values

```
(defn insert-selecter
  [tree key]
  (cond
   (coll? key) :multi-insert
   (not-empty tree) :default
   :else :empty-tree))

(defmulti insert insert-selecter)
```

Clojure supports multiple dispatch

Visitor Pattern

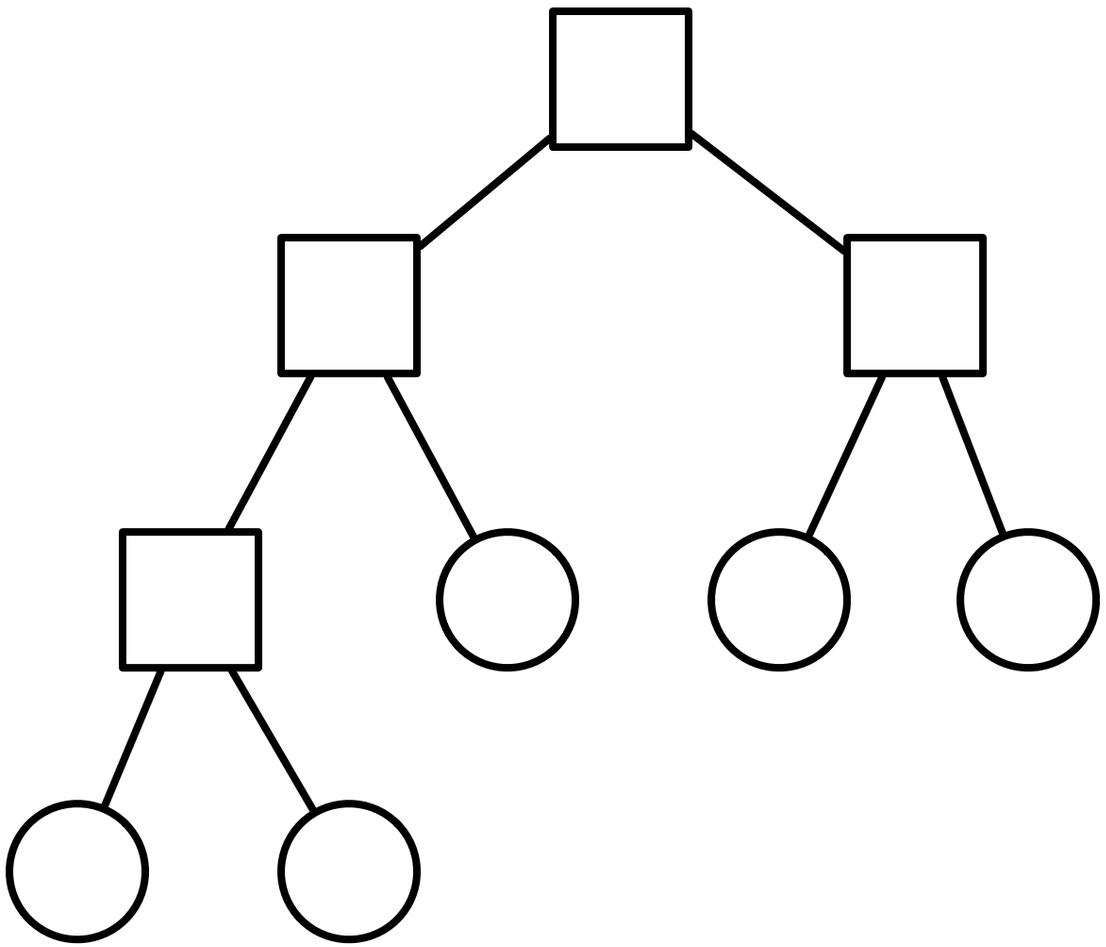
Intent

Represent an operation to be performed on the elements of an object structure

Visitor Pattern

Define a new operation without changing the classes of the elements on which it operates

Tree Example



```
class Node { ... }

class BinaryTreeNode extends Node {...}

class BinaryTreeLeaf extends Node {...}

class Tree { ... }
```

Tree Printing

HTML Print

PDF Print

TeX Print

RTF Print

Others likely in future

Operations are complex

Do different things on different types of nodes

Need to traverse tree

Not part of BST abstraction

Need Multiple Dispatch

Needs to select dynamically operation based on two values

- Type of Structure
- Type of printing

class BinaryTreeNode

class BinaryTreeLeaf

class Tree

HTML Print

- htmlPrintBinaryTreeNode(...)
- htmlPrintBinaryTreeLeaf(...)
- htmlPrintBinaryTree(...)

PDF Print

- pdfPrintBinaryTreeNode(...)
- pdfPrintBinaryTreeLeaf(...)
- pdfPrintBinaryTree(...)

TeX

- etc

Visitor Pattern

How to implement double dispatch in languages than only have single disptach

Visitor Pattern & Clojure

Lisp has multiple dispatch via multi-methods

Visitor pattern

- Not needed
- Just used to deal with language defect

Back to Assignment 4

Solution was longish

Several helper methods

Multi method use is questionable

Without multimethods

```
(defn make-tree
  [key left-tree right-tree]
  (let [[left-height right-height] (map #(get % :height 0) [left-tree right-tree])
        max-height (max left-height right-height)]
    {:key key :height (inc max-height) :left left-tree :right right-tree}))

(defn insert
  [{:keys [key left right]} insert-key]
  (if (nil? key)
    {:key insert-key :height 1}
    (let [[small large] (sort [key insert-key])]
      (if (> (get left :height 0) (get right :height 0))
        (make-tree small left (insert right large))
        (make-tree small (insert left large) right)))))
```

The Rest of the Assignment

```
(defn pre-order
  [tree]
  (when (seq tree)
    (let [{:keys [key left right]} tree]
      (lazy-cat [key] (pre-order left) (pre-order right)))))

(defn insert-all
  ([values]
    (insert-all nil values))
  ([tree values]
    (reduce insert tree values)))
```

Records

Defining Clojure Types

(defrecord Point [x y])

(deftype Point [x y])

deftype - lower level construct

Both

Compile to Java class with final fields

Accessing & updating fields faster than Clojure maps

Use Java naming convention

Creating & Accessing

```
(defrecord Point [x y])

(def a (Point. 2 3))

(.x a)          2

(:x a)          2

(:z a 0)        0
```

Creating with Types

```
(defrecord NamedPoint [^String name ^long x ^long y])

(def b (NamedPoint. "Small" 2 4))

(:x b)

(NamedPoint/getBasis)                    [name x y]
```

This avoid the autoboxing of the values

Records

Support value sematics

Act like maps

Metadata support

Reader support

Value Semantics

Immutable

If fields of two records are equal than Records are equal

(= (Point. 1 2) (Point. 1 2))	true
(= 3 3N)	true
(= (Point. 1 2) (Point. 1N 2N))	true

Records are like Maps

```
(let [{:keys [x y]} (Point. 2 3)]
  x)                                  2

(assoc (Point. 1 2) :z 5)             #user.Point{:x 1, :y 2, :z 5}

(dissoc (Point. 1 2) :x)              {:y 2}

(seq (Point. 1 2))                    ([:x 1] [:y 2])

(into {} (Point. 3 4))                {:x 3, :y 4}
```

assoc returns a Point
dissoc returns a map

But Records are not Maps

(= (Point. 1 2) {:x 1 :y 2})	false
({:x 1 :y 2} :x)	1
((Point. 1 2) :x)	Exception

Records are not Defined in Namespaces

Records are Java Classes

Not included when import/require Clojure namespace

Have to require the Record

Namespace record is declared in is part of the full name of the Record

Auxiliary Constructor

(Point. 1 2 {:foo :bar} {:z 3})

metadata

More fields

Constructors & Factory Functions

Text recomends you provide functions to create records

Functions can be used by higher order functions

Makes it easier to change record definition

Built in Factory Methods

->RecordType	positional
map->RecordType	from a map

```
(->Point 2 3)

(map->Point {:y 2 :x 1})
```

Using defrecord Assignment 4

```
(defrecord HeapNode [key height left right])

(defn height
 [tree]
 (get tree :height 0))

(defn make-tree
  [key left-tree right-tree]
  (let [[left-height right-height] (map height [left-tree right-tree])
      max-height (max left-height right-height)]
   (HeapNode. key (inc max-height) left-tree right-tree)))
```

```
(defn insert
  [{:keys [key left right]} insert-key]
  (if (nil? key)
    (HeapNode. insert-key 1 nil nil)
    (let [[small large] (sort [key insert-key])]
      (if (> (height left) (height right))
        (make-tree small left (insert right large))
        (make-tree small (insert left large) right)))))

(defn pre-order
  [tree]
  (when (seq tree)
    (let [{:keys [key left right]} tree]
      (lazy-cat [key] (pre-order left) (pre-order right)))))

(defn insert-all
  ([values]
    (insert-all nil values))
  ([tree values]
    (reduce insert tree values)))
```

Timings

N	Map-Multimethod Time	defrecord Multi-M Time	Map No Multi-method	defrecord No Multi
1000	27.7	44.4	28.5	25.4
2000	44.2	59.5	54.3	38.1
4000	103.2	124.4	99.0	82.0
8000	221.0	230.0	232.0	211.2
16000	505.1	542.6	463.8	422.6

Inserting N items into heap

Time in milliseconds

Protocols

Like Java interfaces

Contains one or more methods

Each method can have multiple arities

Each method has at least one argument

Single dispatch on first argument

```
(defprotocol ProtocolName
    "documentation"
    (a-method [this arg1 arg2] "method docstring")
    (another-method [x] [x arg] "docstring"))
```

Protocols

```
(defprotocol Shape
  (area [s] )
  (perimeter [s]))

(defrecord Rectangle [length width]
  Shape
  (area [this] (* length width))
  (perimeter [this] (+ (* 2 length)
                       (* 2 width))))

(defrecord Circle [radius]
  Shape
  (area [this] (* (Math/PI) radius radius))
  (perimeter [this] (* 2 (Math/PI) radius)))
```

```
(area (Circle. 2))
(area (Rectangle. 2 3))
```

Extending Existing Types

```
(defprotocol FIFO
  (fifo-push [fifo value])
  (fifo-pop [fifo])
  (fifo-peek [fifo]))

(extend-type clojure.lang.IPersistentVector
  FIFO
  (fifo-push [vector value]
    (conj vector value))
  (fifo-pop [vector]
    (pop vector))
  (fifo-peek [vector]
    (last vector)))
```

```
(fifo-pop [1 2 3 4])
(fifo-peek [1 2 3])
```

Extending Existing Types

```
(extend-type clojure.lang.PersistentList
  FIFO
  (fifo-push [seq value]
    (conj seq value))
  (fifo-pop [seq]
    (pop seq))
  (fifo-peek [seq]
    (first seq)))
```

```
(fifo-push '(1 2 3) 4)
```

Protocols & Assignment 4

```
(declare make-tree)

(defprotocol Heap
  (get-height [this])
  (insert [this value])
  (insert-all [this coll])
  (pre-order [this]))
```

Extending nil - Edge cases

```
(extend-type nil
  Heap
  (get-height [this]
    0)

  (pre-order [this]
    nil)

  (insert [this value]
    (HeapNode. value 1 nil nil))

  (insert-all [this coll]
    (reduce insert this coll)))
```

HeapNode

```
(defrecord HeapNode [key height left right]
  Heap
  (get-height [this]
    height)

  (pre-order [this]
    (lazy-cat [key] (pre-order left) (pre-order right)))

  (insert   [this value]
    (let [[small large] (sort [key value])]
      (if (> (get-height left) (get-height right))
       (make-tree small left (insert right large))
       (make-tree small (insert left large) right))))

  (insert-all [this coll]
    (reduce insert this coll)))
```

make-tree again

```
(defn make-tree
  [key left-tree right-tree]
  (let [[left-height right-height] (map get-height [left-tree right-tree])
        max-height (max left-height right-height)]
    (HeapNode. key (inc max-height) left-tree right-tree)))
```

Timings

N	Map-Multimethod Time	defrecord Multi Time	Map No Multi	defrecord No Multi	defrecord protocol
1000	27.7	44.4	28.5	25.4	32.6
2000	44.2	59.5	54.3	38.1	33.9
4000	103.2	124.4	99.0	82.0	91.6
8000	221.0	230.0	232.0	211.2	184.5
16000	505.1	542.6	463.8	422.6	371.0

Inserting N items into heap

Time in milliseconds