Лабораторная работа #2

Дисциплина: "Функциональное программирование"

Дата: 2021/10

Выполнил: Федоров Сергей, P34113

Название: "Реализация структуры данных. Property-based testing"

Цель работы: освоиться с построением пользовательских типов данных, полиморфизмом, рекурсивными алгоритмами и средствами тестирования (unit testing, property-based testing).

Вариант:

ЯП	Clojure
Структура	Open addressing Hash-Map

Требования:

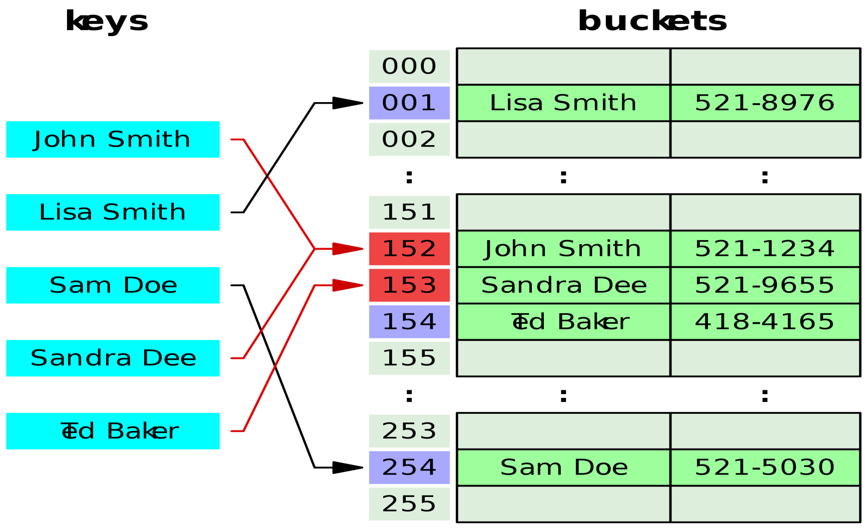
1. Функции:
    - добавление и удаление элементов;
    - фильтрация;
    - отображение (map);
    - свертки (левая и правая);
    - структура должна быть моноидом.
2. Структуры данных должны быть неизменяемыми. Если язык допускает изменение данных – необходимо это протестировать.
3. Реализованные функции должны быть встроены/совместимы со стандартными интерфейсами/библиотекой.
4. Библиотека должна быть протестирована в рамках unit testing.
5. Библиотека должна быть протестирована в рамках property-based тестирования (как минимум 3 свойства).
6. Структура должна быть полиморфной.
7. Требуется использовать идиоматичный для технологии стиль программирования.

Выполнение

Ссылка на репозиторий

Структура данных

Hash-Map с открытой адресацией. Отличие от *"стандартной"* реализации, заключается в механизме разрешения коллизий. Вместо выстраивания цепочек после перехода по хешу, мы продолжаем делать *хопы* помощью "смежных" хэш функций, пока не найдем свободное место.



Программная реализация

Структура проекта:

```
.
├── project.clj
├── src
│     └── functional_programming_itmo_2021
│         └── lab_2
│             ├── main.clj
│             └── hash-map.clj
│             └── src
└── test
      └── functional_programming_itmo_2021
          └── lab_2
              ├── map-unit-testing.clj
              └── map-property-testing.clj
```

Настройка Clojure-окружения с помощью Leiningen:

`project.clj`

```
(defproject functional-programming-itmo-2021 "0.0.1"
            :description "FIXME: write description"
            :url "http://example.com/FIXME"
            :dependencies [[org.clojure/clojure "1.10.1"]
                           [nrepl/lein-nrepl "0.3.2"]]
            :profiles {
                       :lab_1 {
                               :repl-options {
                                              :init-ns functional-programming-itmo-2021.lab-1.main
                                              :package functional-programming-itmo-2021.lab-1
                                              }
                               :main functional-programming-itmo-2021.lab-1.main
                               }
                       :lab_2 {
                               :repl-options {
                                              :init-ns functional-programming-itmo-2021.lab-2.main
                                              :package functional-programming-itmo-2021.lab-2
                                              }
                               :main functional-programming-itmo-2021.lab-2.main
                               }
                       }
            )
```

Имплементация Hash-Map:

`hash-map.clj`

```
(ns functional-programming-itmo-2021.lab-2.hash-map
   (:import (clojure.lang IPersistentMap Associative Util ILookup IMapEntry Seqable IPersistentCollection IMeta MapEquivalenc
      (java.util Map)))

(defrecord OpenMapEntry [key tombstone val]
   IMapEntry
   (getKey [_] key)
   (getValue [_] val))

(defn exact-entry? [key candidate]
   (and (not (:tombstone candidate)) (.equals key (:key candidate))))

(defn find-entry [arr key]
   (loop [computed (-> key hash int Math/abs)
          times-left (count arr)]
      (let [pos (rem computed (count arr))]
         (if-let [entry (nth arr pos)]
            (if (exact-entry? key entry)
               [entry pos]
               (if (> times-left 0)
                  (recur (inc computed) (dec times-left)) [nil nil]))
            [nil pos]))
      ))
      ))

(defn insert-entries
   [amount arr new-entries]
   (if (or (empty? new-entries) (<= amount 0))
      [arr new-entries]
      (if-let [pos (last (find-entry arr (:key (first new-entries))))]
         (recur (dec amount) (assoc arr pos (first new-entries)) (rest new-entries))
         (println "Reached nil: " amount arr new-entries)
         ))
   )

(defn inc-load-by [load arr amount] (+ load (/ (double amount) (count arr))))

(defn non-empty-cells [arr] (filter #(not (or (nil? %) (:tombstone %))) arr))

(defn rebalance
   ([arr] (rebalance arr 2))
   ([arr coef]
    (let [current-size (count arr)
          filtered (non-empty-cells arr)]
       (->> filtered
            (insert-entries (count filtered) (vec (repeat (* coef (max current-size 1)) nil)))
            first

            )
       )))

(defn insert [load arr new]
   (let [next-load (inc-load-by load arr 1)]
      (if (>= next-load 0.8)
         (let [rebalanced (rebalance arr)
               balanced-load (/ (-> rebalanced non-empty-cells count double) (count rebalanced))]
            (insert balanced-load rebalanced new))
         [(first (insert-entries 1 arr [new])) next-load]
         )))

(defn delete [arr key]
   (if-let [pos (last (find-entry arr key))]
      (assoc arr pos (->OpenMapEntry key true nil))
      arr
      ))

(def compute-meta
   (memoize (fn [contents] {:size (count (non-empty-cells contents))} )))

(declare ->OpenAddressesMap)
(deftype OpenAddressesMap [contents load]
   IMeta
   (meta [_] (compute-meta contents))

   ILookup
   (valAt [_ k not-found]
      (if-let [[attempt _] (find-entry contents k)]
         (:val attempt)
         not-found))

   (valAt [m k] (.valAt m k nil))

   Iterable
   (iterator [m] (.iterator (seq m)))

   Seqable
   (seq [_] (non-empty-cells contents))

   IPersistentMap
   (assoc [_ k v] (apply ->OpenAddressesMap (insert load contents (->OpenMapEntry k false v))))
   (assocEx [m k v] (if (.containsKey m k)
                       (.runtimeException Util "Key already present")
                       (.assoc m k v)))
   (without [_ k] (->OpenAddressesMap (delete contents k) load))

   MapEquivalence

   IPersistentCollection
   (count [m] (:size (.meta m)))
   (cons [m new] (cond
                    (and (instance? IPersistentVector new) (>= (count new) 2)) (assoc m (first new) (nth new 2))
                    (instance? IMapEntry new) (assoc m (key new) (val new))
                    (instance? Seqable new) (reduce #(assoc %1 (key %2) (val %2)) m (seq new))
```

```
                      (instance? Seqable new) (reduce #(assoc %1 (key %2) (val %2)) m (seq new))
                      ))
     (empty [_] (->OpenAddressesMap [] 1.0))
     (equiv [m o]
        (if (or
               (not (or (instance? Map o) (instance? IPersistentMap o)))
               (and
                  (instance? IPersistentMap o)
                  (->> o (instance? MapEquivalence) not))
               (not= (count o) (count m)))
           false
           (loop [elems (seq m)]
              (let [cur-elem (first elems)]
                 (if-not (empty? elems)
                    (if (or
                           (not (contains? o (.getKey cur-elem)))
                           (not (= (.getValue cur-elem) (get o (.getKey cur-elem)))))
                       false
                       (recur (rest elems)))
                    true))))
        )

     Associative
     (containsKey [_ k] (let [[attempt _] (find-entry contents k)]

                           (if-not (or (nil? attempt) (:tombstone attempt))
                              true
                              false
                              )))
     (entryAt [_ k] (first (find-entry contents k)))

     )

  (defn open-address-map
     ([]  (->OpenAddressesMap [nil nil nil nil] 0.0))
     ([src-map] (open-address-map src-map 2))
     ([src-map coef] (->OpenAddressesMap (->>
                                            src-map
                                            (map #(->OpenMapEntry (first %) false (last %)))
                                            (#(rebalance % coef))
                                            ) (/ 1.0 coef))))

  (def example (open-address-map {1 2 3 4 5 6 7 8}))
```

Демонстрация работы коллекции

main.clj

```
  (ns functional-programming-itmo-2021.lab-2.main
     (:require [functional-programming-itmo-2021.lab-2.hash-map :refer :all])
     )

  (defn -main []
     (let [hashmap (open-address-map {9 9 10 10})
           merged-with-example (merge hashmap example)
           dissoced (dissoc merged-with-example 1 2 9)
           updated (assoc dissoced 10 "OTHER VALUE")
           retrieved-value (get updated 10)]
        (do
           (println "Example of working with Open-Addressing Hash Map")
           (println "Example hash map: " example)
           (println "Other hash map: " hashmap)
           (println "Merged hash map: " merged-with-example)
           (println "Dissoced hash map: " dissoced)
           (println "Updated hash map: " updated)
           (println "Retrieved value by key 10: " retrieved-value)
           )))
```

stdout

```
   Example of working with Open-Addressing Hash Map
   Example hash map:  {1 2, 3 4, 7 8, 5 6}
   Other hash map:  {10 10, 9 9}
   Merged hash map:  {7 8, 5 6, 10 10, 1 2, 3 4, 9 9}
   Dissoced hash map:  {7 8, 5 6, 10 10, 3 4}
```

```
Updated hash map:  {10 OTHER VALUE, 5 6, 7 8, 3 4}
Retrieved value by key 10:  OTHER VALUE
```

Тестирование коллекции:

1. Unit-тестирование с базовыми проверками

`map-unit-testing.clj`

```
(ns functional-programming-itmo-2021.lab-2.map-unit-testing
   (:require [clojure.test :refer :all]
      [functional-programming-itmo-2021.lab-2.hash-map :refer :all]))

(def full-map (open-address-map (reduce #(assoc %1 %2 %2) {} (range 9))))
(def mixed-map (open-address-map (reduce #(assoc %1 %2 %2) {} (range 5))))
(def empty-map (open-address-map {}))

(deftest full-get-test
         (is (= (range 9) (map #(get full-map %) (range 9)))))

(deftest mixed-get-test
         (is (= (concat (range 5) (repeat 4 nil)) (map #(get mixed-map %) (range 9)))))

(deftest empty-get-test
         (is (= (repeat 9 nil) (map #(get empty-map %) (range 9)))))

(deftest full-insert-test
         (is (= (concat (range 13) [nil]) (map #(get (merge full-map {9 9 10 10 11 11 12 12}) %) (range 14)))))

(deftest mixed-insert-test
         (is (= (concat (range 5) (repeat 4 nil) (range 9 13) [nil]) (map #(get (merge mixed-map {9 9 10 10 11 11 12 12}) %)

(deftest empty-insert-test
         (is (= (concat (repeat 9 nil) (range 9 13) [nil]) (map #(get (merge empty-map {9 9 10 10 11 11 12 12}) %) (range 14)

(deftest full-delete-test
         (is (= (concat (repeat 4 nil) (range 4 9) (repeat 5 nil)) (map #(get (dissoc full-map 0 1 2 3) %) (range 14)))))

(deftest mixed-delete-test
         (is (= (concat (repeat 4 nil) [4] (repeat 9 nil)) (map #(get (dissoc mixed-map 0 1 2 3) %) (range 14)))))

(deftest full-delete-test
         (is (= (repeat 14 nil) (map #(get (dissoc empty-map 0 1 2 3) %) (range 14)))))

(deftest full-count-test
         (is (= 9 (count full-map))))

(deftest mixed-count-test
         (is (= 5 (count mixed-map))))

(deftest empty-count-test
         (is (= 0 (count empty-map))))

(deftest full-map-equiv-test
         (is (.equiv full-map (reduce #(assoc %1 %2 %2) {} (range 9)))))

(deftest mixed-map-equiv-test
         (is (.equiv mixed-map (reduce #(assoc %1 %2 %2) {} (range 5)))))

(deftest empty-map-equiv-test
         (is (.equiv empty-map {})))
```

2. Property-based тестирование с тремя правилами

`map-property-testing.clj`

```
(ns functional-programming-itmo-2021.lab-2.map-property-testing
   (:require [clojure.test :refer :all]
      [functional-programming-itmo-2021.lab-2.hash-map :refer :all]))

(defn run-test [test-fn times] (reduce #(and %1 %2) (repeatedly times test-fn)))

; Three properties
; 1. If Map was build using keyA, it contains keyA
; 2. If Map was disassociated by keyA, it no longer contains it
; 3. If Map was merged with another one, it contains all subset of keys

(defn generate-vec [size] (repeatedly size #(rand-int 10E+6)))

; First property (map <- ... keyA ...) contains keyA

(defn contains-key-prop []
   (let [limit 1000
         data (generate-vec limit)
         generated (open-address-map (reduce #(assoc %1 %2 %2) {} data))
         rnd-idx (rand-int limit)]
      (contains? generated (nth data rnd-idx))
      ))

(deftest first-property
         (is (run-test contains-key-prop 100)))

; Second property (dissoc map keyA) not contains keyA

(defn dissoc-key-prop []
   (let [limit 1000
         data (generate-vec limit)
         generated (open-address-map (reduce #(assoc %1 %2 %2) {} data))
         rnd-key (nth data (rand-int limit))
         stripped (dissoc generated rnd-key)]
      (not (contains? stripped rnd-key))))

(deftest second-property
         (is (run-test dissoc-key-prop 100)))

; Third property (merge map1 map2) contains all keys

(defn merge-key-prop []
   (let [limit 1000
         data-1 (generate-vec limit)
         data-2 (generate-vec limit)
         generated-1 (open-address-map (reduce #(assoc %1 %2 %2) {} data-1))
         generated-2 (open-address-map (reduce #(assoc %1 %2 %2) {} data-2))
         all-keys (reduce #(conj %1 %2) #{} (concat data-1 data-2))
         all-merged (merge generated-1 generated-2)]
      (reduce #(and %1 %2) (map #(contains? all-merged %) all-keys))
      ))

(deftest third-property
         (is (run-test merge-key-prop 100)))
```