Лабораторная работа №2

Словарь на бинарном дереве

Выполнил: Куприянов А.А., P34113
Санкт-Петербург, 2021

> В попытке справится со скобками я поставил Rainbow Brackets, но теперь я начал путать цвета

Метод создания ноды:

```
(defn make-node
  ([left right key value]   ; Изменены названия less - left, greater - right, key, value
   {:left left :right right :key key :value value})
  ([key value]
   (make-node nil nil key value)))
```

В каждой ноде будет записано значение вида:

```
{:left left :right right :key key :value value}
```

```
(defn nodekey-wrapper
  [node]
  (if (instance? Number (:key node))
    (:key node)
    (hash (:key node)))
  )

(defn comparator-over-nodes
  "возвращает функцию компаратор между нодами"
  ([func]
   (fn [node1 node2]
     (func (nodekey-wrapper node1) (nodekey-wrapper node2))))
  ([func & args]
   (comparator-over-nodes #(func args %))))

(def right-than
  (comparator-over-nodes >))

(def equals
  (comparator-over-nodes =))
```

```
(defn height
  "Глубина бинарного дерева"
  ([root count]
   (if root
     (max (height (:left root) (inc count))
          (height (:right root) (inc count)))
     count))
  ([root]
   (height root 0)))
```

```
(defn insert-node
  "Вставляет новую ноду в бинарное дерево"
  [root new-node]
  (if (nil? root)
    new-node
    (if (right-than new-node root)
      (assoc root :right (insert-node (:right root) new-node))

      (if (equals new-node root)
        (assoc root :key (:key new-node))
        (assoc root :left (insert-node (:left root) new-node))))))
```

Здесь следует обратить внимание на этот фрагмент

```
(if (equals new-node root)
    (assoc root :key (:key new-node))
; если нода с идентичным ключом уже существует, то мы заменяем её значение
```

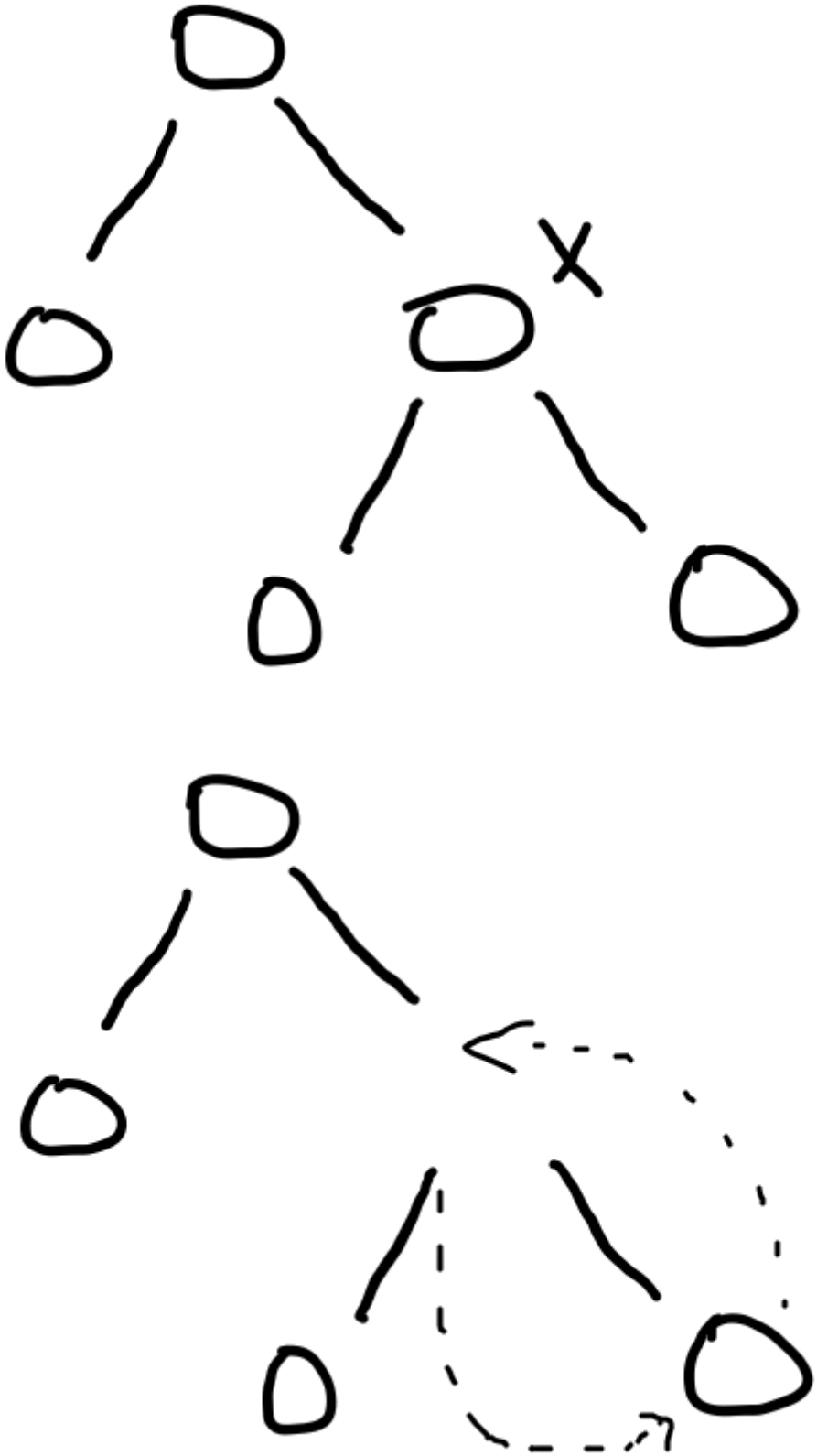
Далее, разумеется, нужна будет функция поиска

```
(defn find-node
  "Поиск по дереву"
  [root key]
  (if (nil? (:key root))
    nil
    (if (= key (:key root))
      root
      (if (> key (:key root))
        (find-node (:right root) key)
        (find-node (:left root) key)))))
```

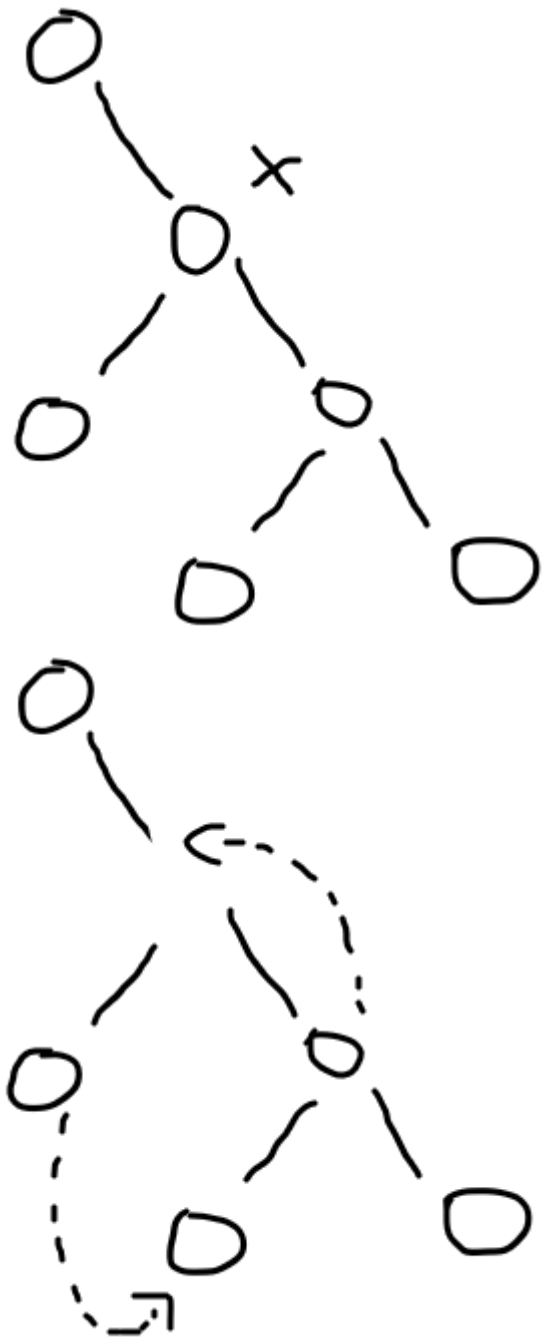
Далее, нужно реализовать удаление значения из списка, что является самой сложной задачей, так как при изменении структуры дерева, её нужно восстановить, чтобы правила оставались теми же.

При удалении какого-то объекта из дерева необходимо заменить её объектом (изменить ссылку у родителя), который находится справа, а далее левый объект присоединить к самому левому объекту, который был у правого (так как он заведомо меньше неё).

Пример простой замены



Пример замены при присутствии дочерних нод у заменяющего



Пример замены в левой части дерева

```
(defn find-smallest-node
  [root]
  (if (nil? (:left root))
    root
    (find-smallest-node (:left root))))

(defn find-node-parent
  "returns {:parent parent :child root}"
  ([root key parent]
   (if (nil? (:key root))
     nil
     (if (= key (:key root))
       {:parent parent :child root}
       (if (> key (:key root))
         (find-node-parent (:right root) key root)
         (find-node-parent (:left root) key root)))))
  ([root key]
   (find-node-parent root key nil)))

(defn add-left-node
  [left-node root]
  (if (nil? root)
    left-node
    (assoc (find-smallest-node root) :left left-node)))

(defn remove-child
  [e] ; {:parent parent :child root} (object ?) struct
  (if (nil? (:parent e))
    (if (nil? (:right (:child e)))
      (:left (:child e))
      (add-left-node (:left (:child e)) (:right (:child e))))
    (if (right-than (:child e) (:parent e))
      (assoc (:parent e) :right (add-left-node (:left (:child e)) (:right (:child e))))
      (assoc (:parent e) :left (add-left-node (:left (:child e)) (:right (:child e)))))))


(defn remove-node
  [root remove-key]
  (remove-child
   (find-node-parent root remove-key)))
```

Мы почти готовы к запуску

```
(defn add-to-dict
  ([dict entry]
   (insert-node dict (make-node (:key entry) (:value entry))))

  ([entry]
   (insert-node nil (make-node (:key entry) (:value entry))))
  )

(defn value-from-dict
  [dict key]
  (:payload (find-node dict key))
  )

(defn create-dict
  [& args]
```

```
  (apply add-to-dict args)
  )
```

И собственно сам запуск

```
(defn main
  [& args]
  (println
   (remove-node
    (add-to-dict (create-dict {:key 666 :value 100}) {:key 111 :value 1500}) 666)))
```

PDD — Panic Driven Development

Осталось написать тесты для проверки адекватности бинарного дерева и словаря (конечно же тесты должны писаться в конце)

Используем стандартную библиотеку тестирования:

```
(use
  'clojure.test)
```

Вывод которого может вызвать мысль в голове "Ну, да, это же кложур!"

```
(is (= 10 (+ 5 4))))
```

```
FAIL in (sample-test) (Dictionary.clj:158)
expected: (= 10 (+ 5 4))
  actual: (not (= 10 9))
```

```
; TEST ------------------------------------

(deftest create-dict-test
  (is
   (=
    {:left nil, :right nil, :key 666, :value 777}
    (create-dict {:key 666, :value 777}))))

(deftest create-nil-dict-test
  (is
   (=
    {:left nil, :right nil, :key nil, :value nil}
    (create-dict nil))))

(deftest add-to-dict-test
  (is
   (=
    {:left
     {:left nil, :right nil,
      :key  111, :value 1500},
     :right nil,
     :key   666,
     :value 100}
```

```
    (add-to-dict (create-dict {:key 666 :value 100}) {:key 111 :value 1500})))))

(deftest add-to-nil-test
  (is
   (=
    {:left nil, :right nil,
     :key  666, :value 100}
    (add-to-dict nil {:key 666 :value 100}))))

(deftest add-nil-to-dict-test
  (is
   (=
    {:left nil, :right nil,
     :key  555, :value 0}
    (add-to-dict (create-dict {:key 555 :value 0}) nil))))

(deftest add-existing-value-test
  (is
   (=
    {:left
     {:left nil, :right nil,
      :key  111, :value 1500},
     :right nil,
     :key   666,
     :value -20}
    (add-to-dict (add-to-dict (create-dict {:key 666 :value 100}) {:key 111 :value 1500}) {:key
666 :value -20})
     )))

(deftest get-value-test
  (is
   (=
    1500
    (value-from-dict (add-to-dict (create-dict {:key 666 :value 100}) {:key 111 :value 1500})
111)
     )))


(deftest get-non-existing-value-test
  (is
   (=
    nil
    (value-from-dict (add-to-dict (create-dict {:key 666 :value 100}) {:key 111 :value 1500}) 0)
    )))

(deftest simple-height-test
  (is
   (=
    2
    (height (add-to-dict (create-dict {:key 666 :value 100}) {:key 111 :value 1500}))
    )))

(deftest bidirectional-tree-height-test
  (is
   (=
    2
    (height (add-to-dict (add-to-dict (create-dict {:key 666 :value 100}) {:key 111 :value
1500}) {:key 1000 :value 1000}))
    )))

(deftest bidirectional-tree-height-3-test
```

```
(is
 (=
  3
  (height (add-to-dict
           (add-to-dict
            (add-to-dict
             (create-dict {:key 666 :value 100})
             {:key 111 :value 1500})
            {:key 1000 :value 1000})
           {:key 2000 :value -1})))
```

> Программирование — это искусство. Искусство имеет внутреннюю инстинктивную составляющую. Доверься своей интуиции. Напишите код. Разверните его. Только смелым улыбается удача

Вывод

В ходе лабораторной работы было реализовано примитивное бинарное дерево, которое использовалось для имплементации словаря на языке Clojure. Язык Clojure имеет динамическую типизацию, что может усложнить задачу по обработке различных ситуаций, например, при передаче неверной структуры.

Также реализованное бинарное дерево имеет достаточно большой минус в том, что при сравнении ключей, если значение не имеет подкласс Number берется его hash, как показано в функции nodekey-wrapper

```
(defn nodekey-wrapper
  [node]
  (if (instance? Number (:key node))
    (:key node)
    (hash (:key node)))
  )
```

В таком случае могут возникать коллизии хэшей, что приведет к потере значения одного из элементов в словаре, что может критически сказаться при выполнении работы программы. Поэтому ни в коем случае не использовать в продакшене!

По самому Clojure, все еще трудно отвыкать от императивного стиля и писать в функциональном стиле. С одной стороны хочется разбить все функции на мелкие, с другой стороны не понятно как это все вообще должно выглядеть.

Тем не менее, удалось разбить программу на несколько основных функций, которые должны соблюдать контракт между собой