**МИНОБРНАУКИ РОССИИ**
**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**
**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**
**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**
**КАФЕДРА САПР**

# ОТЧЕТ

**по лабораторной работе №1**

**по дисциплине «Алгоритмы и структуры данных»**

**Тема: Ассоциативный массив**

Студентка гр. 9301 _____ Синицкая В. А.

Преподаватель _____ Тутуева А. В.

Санкт-Петербург
2021

# Оглавление

## 1 Постановка задачи

Реализовать шаблонный ассоциативный массив (map) на основе красно-черного дерева. Наличие unit-тестов ко всем реализуемым методам является обязательным требованием.

Список методов:

insert(ключ, значение) // добавление элемента с ключом и значением

remove(ключ) // удаление элемента дерева по ключу

find(ключ) // поиск элемента по ключу

clear // очищение ассоциативного массива

get_keys // возвращает список ключей

get_values // возвращает список значений

print // вывод в консоль

## 2 Реализуемые классы и методы

В проекте содержатся классы Node, Element, Iterator, Queue, Stack, RB_Tree, написанные шаблонно. Все классы, кроме RB_Tree были написаны и протестированы ранее, здесь они были лишь переписаны под формат шаблона.

Класс Node представляет собой структуру узла, в нем содержатся ключ, значение, цвет — красный или черный, указатели на родительский узел и на левого и правого сына. Класс имеет конструкторы — по умолчанию, от заданных полей, часть из которых может быть опущена, от другого узла. Класс содержит методы set и get для всех полей.

Классы Stack, Queue представляют собой реализации стека и очереди для элементов типа Element, который создан для оперирования в пределах данных классов с использованием данных в виде узлов Node.

Класс Iterator переписывается в классе RB_Tree, реализуясь в виде обхода бинарного (в данном случае, красно-черного) дерева в ширину.

Класс RB_Tree содержит поля root, Nil, size. Поле size — одно на объект типа RB_Tree, обозначает количество узлов, содержащихся в дереве. Поле Nil представляет собой элемент, на который ссылаются все листовые узлы, всегда имеет черный цвет, в данной реализации является родителем root. Root — это корневой узел дерева, когда в

дереве нет ни одного элемента, равен Nil. Nil и size не имеют методов set, Nil получает свое место в памяти единожды — при вызове конструктора, size изменяется автоматически при добавлении или удалении узла.

Класс RB_Tree содержит методы find, left_turn, right_turn, insert, remove, restoring_propertires_after_insert, restoring_propertires_after_delete, create_breadth_first_traverse_iterator (а так же переписанный класс Iterator), перегруженный метод print, get_keys, get_values, clear.

Метод find находит в дереве элемент по ключу и возвращает указатель на этот узел. В случае поиска в пустом дереве или по несуществующему ключу, вызывает исключение с соответствующим сообщением.

Методы right_turn и left_turn используются для реализации методов restoring_propertires_after_insert и restoring_propertires_after_delete, которые, в свою очередь, используются методами insert и remove. Первые четыре из указанных имеют модификатор доступа private. Методы right_turn и left_turn меняют местами два узла, один из которых был родителем, а другой — его сыном. Сын меняет направление — из левого становится правым, из правого — левым, потом, узел, бывший родителем, становится на место ребенка и наоборот. Связи, которые были у этих узлов с другими узлами дерева, меняются таким образом, чтобы свойства дерева как бинарного сохранялись. Методы принимают элемент-отца и ничего не возвращают.

Метод insert принимает значение ключа и значение данных, которые будут под ним храниться. Изначально метод находит место для нового элемента как искал бы его в обычном бинарном дереве, новый узел всегда добавляется красным, могут быть нарушены некоторые свойства красно-черного дерева: краснота корня, черная высота, чернота детей красного элемента. Чтобы восстановить эти свойства с помощью перекрашиваний и перестроений дерева, вызывается метод restoring_propertires_after_insert.

Метод remove принимает значение ключа, далее, используя метод find, находится сам удаляемый узел. Далее от удаляемого листа происходит переход в левое поддерево, в котором ищется самый правый элемент, имеющий ноль или одного ребенка, этот элемент является заменой: значения, хранимые в этом элементе, помещается в тот узел,

который нужно удалить, в итоге удаляется узел-замена. При таком удалении проще восстановить свойства красно-черного дерева, которые нарушаются лишь в том случае, если лист-замена оказался черным ребенком без детей. В других случаях, красный просто удаляется, и не может иметь 2 детей, так как тогда бы было нарушено свойство черной высоты. Черный с одним ребенком просто замещается своим единственным сыном. Итак, если узел-замена имеет черный цвет и не имеет детей, вызывается restoring_propertires_after_delete, так как, очевидно, при удалении черного узла из одной ветки, черная высота этой ветви изменяется, а соседней остается прежней. Нарушение черной высоты нужно исправить.

Методы-восстановления исполняют разные действия в зависимости от вида дерева в данный момент. Чтобы избавиться от так называемой «проблемы двух красных» и восстановить свойство черной высоты, используются перекрашивания, и, если они не помогают или нарушают средство черной высоты, выполняются повороты, перекрашивания, и снова повороты, до тех пор, пока алгоритмы не дойдут до корня или нельзя будет точно сказать на промежуточном этапе, что свойства дерева больше не нарушены, так как проблема разрешилась и нет необходимости переносить ее на более высокие узлы дерева.

Методы get_keys и get_values возвращают массивы ключей и значений соответственно. Используя поле size и итератор обхода в ширину, создается и возвращается пользователю требуемый массив.

Метод print выводит в консоль или, если методу передано имя файла, то в файл, пары вида ключ-значение. Метод использует итератор обхода в ширину.

Метод clear очищает все элементы дерева, возвращая его в состояние, в котором оно находилось до добавления первого элемента. Метод вызывает рекурсивно сам себя для ветвей элемента, и удаляет каждый элемент. Такой алгоритм проще, чем использование удаления из дерева, так как в данном случае нет необходимости следить за соблюдением свойств  красно-черного дерева.

## 3 Оценка временной сложности каждого метода

1) find — O(logN), так как благодаря существованию черной высоты, одна ветвь не может содержать больше, чем в два раза узлов, чем другая. Таким

образом, опуская константы, можно считать, что каждый переход на уровень ниже уменьшает количество будущих переходов вдвое.

2) left_turn, right_turn — O(1)
3) insert — O(logN) — как find, но не элемента, а места для него.
4) remove — O(logN) — как find для узла-замены.
5) restoring_propertires_after_insert — O(logN) — опуская константы, находясь в месте для вставки, мы уже выбрали одну из ветвей всего дерева, поднимаясь наверх, часть узлов тоже заведомо не будет рассмотрена, а поднявшись до корня, ниже уже не спустимся.
6) restoring_propertires_after_delete — O(logN) — аналогично 5), здесь подъем к корню происходит от узла-замены, но мы никогда не спускаемся вниз, хотя рассматривая конкретный узел внимание иногда обращается к более нижним узлам, но не далее правнуков, что списывается при оценке сложности как константа.
7) print — так как метод next() итератора имеет сложность O(N), имеет сложность O(N^2)
8) get_keys, get_values — O(N^2) по аналогии с 7) пунктом
9) clear — O(N), удаление происходит для каждого элемента.

## 4 Описание реализованных Unit test

Была проверена работа конструкторов, были проверены insert и delete, используемые ими методы были проверены внутри этих проверок. При проверке были проверены крайние случаи — один элемент, два, три, элементы по одну сторону, много элементов. Были проверены на правильность работы все вышеописанные методы.

## 5 Пример работы программы

Примером работы послужит реализация функции print() для дерева, полученного последовательной вставкой элементов с ключами, равными значению: 1, 2, 3, 4, 5, 6. Имя файла было указано, как "../out.txt", вследствие этого файл с названием out.txt появился в папке UnitTest_RB_Tree данного проекта. Результат вывода в файл показан на Рис. 5.1:

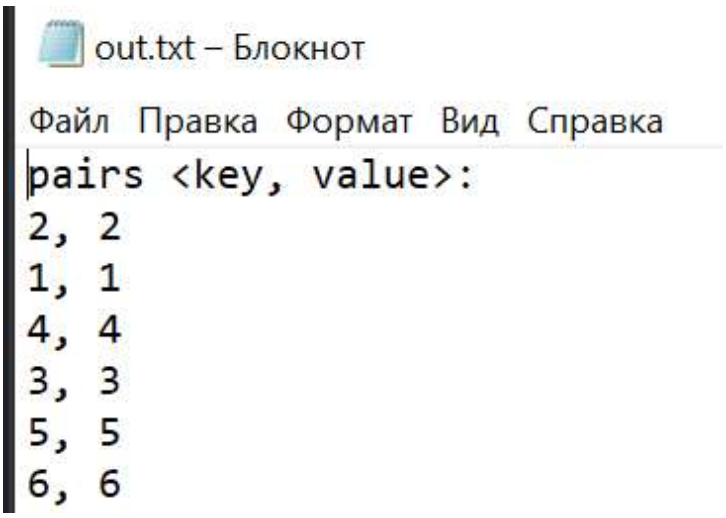


Рис. 5.1 — пример работы программы.

# 6 Листинг

## 6.1 Element.h

```
1  #pragma once
2  #include "Node.h"
3  template <class Key_Type, class Value_Type>
4  class Element
5  {
6    Element<Key_Type, Value_Type>* next;
7    Node<Key_Type, Value_Type>* data;
8    void set_next(Element<Key_Type, Value_Type>* new_element) { next = new_element; }
9    void set_data(Node<Key_Type, Value_Type>* new_data) { data = new_data; }
10
11 public:
12   Element() { data = nullptr; next = nullptr; }
13   Element<Key_Type, Value_Type>* get_next() { return next; }
14   Node<Key_Type, Value_Type>* get_data() { return data; }
15   ~Element() {}
16   template <class Key_Type, class Value_Type> friend class Queue;
17   template <class Key_Type, class Value_Type> friend class Stack;
18 };
```

## 6.2 Node.h

```
1  #pragma once
2  #include <stdexcept>
3  #include <iostream> // is needed?
4
5  using namespace std;
6  enum class Color { red, black };
7
8  template <class Key_Type, class Value_Type>
9  class Node {
10 private:
11   Key_Type key;
12   Value_Type value;
13   Color color;
14   Node<Key_Type, Value_Type>* left = nullptr; // left child of current node
15   Node<Key_Type, Value_Type>* right = nullptr; // right child of current node
16   Node<Key_Type, Value_Type>* parent = nullptr; // parent of current node
17   void set_key(Key_Type new_key) { key = new_key; }
```

```
18   void set_value(Value_Type new_value) { value = new_value; }
19   void set_color(Color new_color) { color = new_color; }
20   void set_left(Node<Key_Type, Value_Type>* new_left) { left = new_left; }
21   void set_right(Node<Key_Type, Value_Type>* new_right) { right = new_right; }
22   void set_parent(Node<Key_Type, Value_Type>* new_parent) { parent = new_parent; }
23
24 public:
25   Node(void) {}
26   Node(Key_Type new_key, Value_Type new_value, Color new_color = Color::red, Node<Key_Typ
     e, Value_Type>* new_left = nullptr, Node<Key_Type, Value_Type>* new_right = nullptr, Nod
     e<Key_Type, Value_Type>* new_parent = nullptr)
27   {
28            key = new_key;
29            value = new_value;
30            color = new_color;
31            left = new_left;
32            right = new_right;
33            parent = new_parent;
34   }
35   Node(Node<Key_Type, Value_Type>* node)
36   {
37            key = node->get_key();
38            value = node->get_value();
39            color = node->get_color();
40            left = node->get_left();
41            right = node->get_right();
42            parent = node->get_parent();
43   }
44   //Node& operator=(const Node<Key_Type, Value_Type>& right)
45   //{
46   //       if (this == &right) // if made like a = a
47   //                return (*this);
48   //       key = right->get_key();
49   //       value = right->get_value();
50   //       color = right->get_color();
51   //       left = right->get_left();
52   //       right = right->get_right();
53   //       parent = right->get_parent();
54   //       return *this;
55   //}
56   Key_Type get_key() { return key; }
57   Value_Type get_value() { return value; }
58   Color get_color() { return color; }
59   Node<Key_Type, Value_Type>* get_left() { return left; }
60   Node<Key_Type, Value_Type>* get_right() { return right; }
61   Node<Key_Type, Value_Type>* get_parent() { return parent; }
62   template <class Key_Type, class Value_Type> friend class RB_Tree;
63   template <class Key_Type, class Value_Type> friend class Element;
64   template <class Key_Type, class Value_Type> friend class Queue;
65   template <class Key_Type, class Value_Type> friend class Stack;
66   ~Node(void) {}
67 };
```

### 6.3 Iterator.h

```
1  #pragma once
2  #include "Node.h"
3  template <class Key_Type, class Value_Type>
4  class Iterator
5  {
```

```
6   public:
7     virtual Node<Key_Type, Value_Type>* next() = 0; // return current element and goes to t
    he next
8     virtual bool has_next() = 0; // return true if next exists
9   };
```

### 6.4 Queue.h

```
1   #pragma once
2   #include "Element.h"
3   template <class Key_Type, class Value_Type>
4   class Queue
5   {
6   private:
7     Element<Key_Type, Value_Type>* head; // the first in queue, leaves first
8     Element<Key_Type, Value_Type>* tail; // the last, new in queue
9     void set_head(Element<Key_Type, Value_Type>* key) { head = key; }
10    void set_tail(Element<Key_Type, Value_Type>* key) { tail = key; }
11  public:
12    Queue()
13    {
14            head = nullptr;
15            tail = nullptr;
16    }
17    bool is_empty() // returns true if queue is empty
18    {
19            if (tail == nullptr)
20                    return true;
21            return false;
22    }
23    int size() // return a number of elements in queue
24    {
25            if (is_empty())
26                    return 0;
27            int size = 1;
28            for (Element<Key_Type, Value_Type>* now = tail; now-
    >get_next() != nullptr; now = now->get_next())
29                    size++;
30            return size;
31    }
32    Element<Key_Type, Value_Type>* top() // to know who will leave first
33    { return head; }
34
35    Element<Key_Type, Value_Type>* push(Node<Key_Type, Value_Type>* data_key) // to add in
    the end of the queue the element with key-data and return it to user
36    {
37
      Element<Key_Type, Value_Type>* new_element = new Element<Key_Type, Value_Type>;
38            new_element->set_data(data_key);
39            if (is_empty())
40                    head = tail = new_element;
41            else
42            {
43                    new_element->set_next(tail);
44                    tail = new_element;
45            }
46            return new_element;
47    }
48
49    Element<Key_Type, Value_Type>* pop() // to delete first in queue and return it to user
```

```
50   {
51           Element<Key_Type, Value_Type>* to_delete;
52           if (is_empty())
53                   throw out_of_range("the queue is empty");
54           else if (size() == 1)
55           {
56                   to_delete = head;
57                   head = tail = nullptr;
58           }
59           else
60           {
61                   to_delete = head;
62                   for (Element<Key_Type, Value_Type>* now = tail; now-
   >get_next() != nullptr; now = now->get_next())
63                           head = now;
64                   head->next = nullptr;
65           }
66           return to_delete;
67   }
68
69   ~Queue()
70   {
71           while (!is_empty())
72                   pop();
73   }
74 };
```

### 6.5 Stack.h

```
1  #pragma once
2  #include "Element.h"
3  template <class Key_Type, class Value_Type>
4  class Stack
5  {
6  private:
7    Element<Key_Type, Value_Type>* top;
8    void set_top(Element<Key_Type, Value_Type>* top_element) { top = top_element; }
9  public:
10   Stack() { top = nullptr; }
11   bool is_empty()
12   {
13           if (top == nullptr)
14                   return true;
15           return false;
16   }
17   int size()
18   {
19           int size = 1;
20           if (is_empty())
21                   return 0;
22           for (Element<Key_Type, Value_Type>* now = top; now-
   >get_next() != nullptr; now = now->get_next())
23                   size++;
24           return size;
25   }
26   Element<Key_Type, Value_Type>* peek() { return top; } // to show who's on the top
27   Element<Key_Type, Value_Type>* pop() // to delte top and show it
28   {
29           Element<Key_Type, Value_Type>* to_delete = top;
30           if (is_empty())
```

```
31                  throw out_of_range("the stack is empty");
32          else
33                  top = top->get_next();
34          return to_delete;
35  }
36  Element<Key_Type, Value_Type>* push(Node<Key_Type, Value_Type>* data) // to push on top
    and show it
37  {
38          Element<Key_Type, Value_Type>* new_element = new Element;
39          new_element->set_data(data);
40          if (is_empty())
41                  top = new_element;
42          else
43          {
44                  new_element->set_next(top);
45                  top = new_element;
46          }
47          return new_element;
48  }
49  ~Stack()
50  {
51          while (!is_empty())
52                  pop();
53  }
54 };
```

### 6.6 RB_Tree.h

```
1  #pragma once
2  #include "Queue.h"
3  #include "Stack.h"
4  #include "Iterator.h"
5  #include <fstream>
6
7  template <class Key_Type, class Value_Type>
8  class RB_Tree {
9  private:
10
11   Node<Key_Type, Value_Type>* root = nullptr;
12   Node<Key_Type, Value_Type>* Nil = nullptr; // node after all lists, one big common chil
     d that is a parent for root
13   int size = 0; // amount of nodes in the tree
14   void set_root(Node<Key_Type, Value_Type>* new_root) // used when root already exists
15   { root = new_root; }
16
17   // change sides and order of two elements by rules of the left rotate
18   void left_turn(Node<Key_Type, Value_Type>* turn_node)
19   {
20           if (turn_node->get_right() == get_Nil())
21                   throw out_of_range("error-call of left-rotate with Nil-child");
22           Node<Key_Type, Value_Type>* right_of_turn_node = turn_node->get_right();
23           turn_node->set_right(right_of_turn_node->get_left());
24           if (right_of_turn_node->get_left() != get_Nil())
25                   right_of_turn_node->get_left()->set_parent(turn_node);
26           right_of_turn_node->set_parent(turn_node->get_parent());
27           if (turn_node->get_parent() == get_Nil())
28                   set_root(right_of_turn_node);
29           else if (turn_node == turn_node->get_parent()->get_left())
30                   turn_node->get_parent()->set_left(right_of_turn_node);
31           else
```

```
32                  turn_node->get_parent()->set_right(right_of_turn_node);
33          right_of_turn_node->set_left(turn_node);
34          turn_node->set_parent(right_of_turn_node);
35  }
36  // change sides and order of two elements by rules of the right rotate
37  void right_turn(Node<Key_Type, Value_Type>* turn_node)
38  {
39          if (turn_node->get_left() == get_Nil())
40                  throw out_of_range("error-call of left-rotate with Nil-child");
41          Node<Key_Type, Value_Type>* left_of_turn_node = turn_node->get_left();
42          turn_node->set_left(left_of_turn_node->get_right());
43          if (left_of_turn_node->get_right() != get_Nil())
44                  left_of_turn_node->get_right()->set_parent(turn_node);
45          left_of_turn_node->set_parent(turn_node->get_parent());
46          if (turn_node->get_parent() == get_Nil())
47                  set_root(left_of_turn_node);
48          else if (turn_node == turn_node->get_parent()->get_left())
49                  turn_node->get_parent()->set_left(left_of_turn_node);
50          else
51                  turn_node->get_parent()->set_right(left_of_turn_node);
52          left_of_turn_node->set_right(turn_node);
53          turn_node->set_parent(left_of_turn_node);
54  }
55
56  void restoring_propertires_after_insert(Node<Key_Type, Value_Type>* new_node)
57  {
58          while (new_node->get_parent()->get_color() == Color::red)
59          {
60                  Node<Key_Type, Value_Type>* P = new_node-
    >get_parent(); // father of new_node
61                  Node<Key_Type, Value_Type>* G = P-
    >get_parent(); //grand father of new_node
62                  Node<Key_Type, Value_Type>* U = G->get_right(); // uncle of new_node
63                  if (U == P)
64                          U = G->get_left();
65                  //case 1: red uncle
66                  if (U->get_color() == Color::red)
67                  {
68                          G->set_color(Color::red);
69                          P->set_color(Color::black);
70                          U->set_color(Color::black);
71                          restoring_propertires_after_insert(G);
72                          return;
73                  }
74                  else
75                  {
76                          //case 2: black uncle, other-sided G-son and P-son are
77                          if (((P == G->get_left()) && (new_node == P-
    >get_right())) || ((P == G->get_right()) && (new_node == P->get_left())))
78                          {
79                                  if (new_node == P->get_right())
80                                          left_turn(P);
81                                  else
82                                          right_turn(P);
83                                  //going to case 3
84                          }
85                          // case 3: black uncle, one-sided G-son and P-son
86                          if (P == G->get_left())
87                                  right_turn(G);
88                          else
89                                  left_turn(G);
```

```
90                              P->set_color(Color::black);
91                              G->set_color(Color::red);
92                      }
93              }
94              root->set_color(Color::black); // to fix root's color
95      }
96
97      void restoring_propertires_after_delete(Node<Key_Type, Value_Type>* D)
98      { // D - is the node to be deleted, we saved it only for restoring everything right
99              if (D == D->get_parent()->get_right())
100             {
101                     Node<Key_Type, Value_Type>* P = D-
    >get_parent(); // father of the deleting node
102                     Node<Key_Type, Value_Type>* S = P-
    >get_left(); // will be the only son after deleting D
103                     Node<Key_Type, Value_Type>* LgS = S->get_left(); // left grandson of P
104                     Node<Key_Type, Value_Type>* RgS = S-
    >get_right(); // right grandson of P
105                     Node<Key_Type, Value_Type>* L_LgS_S = LgS-
    >get_left(); // left son of LgS
106                     Node<Key_Type, Value_Type>* R_LgS_S = LgS-
    >get_right(); // right son of LgS
107                     Node<Key_Type, Value_Type>* L_RgS_S = RgS-
    >get_left(); // left son of RgS
108                     Node<Key_Type, Value_Type>* R_RgS_S = RgS-
    >get_right(); // right son of RgS
109                     if ((P->get_color() == Color::red) && (S-
    >get_color() == Color::black))
110                     {
111                             //case 1: red parent, black left child with black grandsons
112                             if ((LgS->get_color() == Color::black) && (RgS-
    >get_color() == Color::black))
113                             {
114                                     P->set_color(Color::black);
115                                     S->set_color(Color::red);
116                                     return;
117                             }
118                             //case 2: red parent, black left child with left red son
119                             if (LgS->get_color() == Color::red)
120                             {
121                                     right_turn(P);
122                                     P->set_color(Color::black);
123                                     S->set_color(Color::red);
124                                     LgS->set_color(Color::black);
125                                     return;
126                             }
127                             else if (RgS->get_color() == Color::red)
128                                     P->set_color(Color::black); // to get into the case 5
129                     }
130
131                     if (P->get_color() == Color::black)
132                     {
133                             if (S->get_color() == Color::red)
134                             {
135
    //case 3: black parent, red left son, RgS has black grandsons
136                                     if ((L_RgS_S-
    >get_color() == Color::black) && (R_RgS_S->get_color() == Color::black))
137                                     {
138                                             right_turn(P);
139                                             S->set_color(Color::black);
```

```
140                                             RgS-
    >set_color(Color::red); // he was black like son of the red one
141                                             return;
142                                     }
143
      //case 4: black parent, red left son, RgS has left red son
144                                     else if (L_RgS_S->get_color() == Color::red)
145                                     {
146                                             left_turn(S);
147                                             right_turn(P);
148                                             L_RgS_S->set_color(Color::black);
149                                             return;
150                                     }
151                             }
152                             else
153                             {
154
      //case 5: black parent, black left son with right red son
155                                     if (RgS->get_color() == Color::red)
156                                     {
157                                             left_turn(S);
158                                             right_turn(P);
159                                             RgS->set_color(Color::black);
160                                             return;
161                                     }
162
      //case 6: black parent, black left son with black sons
163                                     else if (LgS->get_color() == Color::black)
164                                     {
165                                             S->set_color(Color::red);
166                                             restoring_propertires_after_delete(P);
167                                             return;
168                                     }
169                             }
170                     }
171             }
172             else // simmetric cases
173             {
174                     Node<Key_Type, Value_Type>* P = D-
    >get_parent(); // father of the deleting node
175                     Node<Key_Type, Value_Type>* S = P-
    >get_right(); // will be the only son after deleting D
176                     Node<Key_Type, Value_Type>* LgS = S->get_left(); // left grandson of P
177                     Node<Key_Type, Value_Type>* RgS = S-
    >get_right(); // right grandson of P
178                     Node<Key_Type, Value_Type>* L_LgS_S = LgS-
    >get_left(); // left son of LgS
179                     Node<Key_Type, Value_Type>* R_LgS_S = LgS-
    >get_right(); // right son of LgS
180                     Node<Key_Type, Value_Type>* L_RgS_S = RgS-
    >get_left(); // left son of RgS
181                     Node<Key_Type, Value_Type>* R_RgS_S = RgS-
    >get_right(); // right son of RgS
182                     if ((P->get_color() == Color::red) && (S-
    >get_color() == Color::black))
183                     {
184                             //case 1: red parent, black right child with black grandsons
185                             if ((LgS->get_color() == Color::black) && (RgS-
    >get_color() == Color::black))
186                             {
187                                     P->set_color(Color::black);
```

```
188                                     S->set_color(Color::red);
189                                     return;
190                             }
191                             //case 2: red parent, black right child with right red son
192                             if (RgS->get_color() == Color::red)
193                             {
194                                     left_turn(P);
195                                     P->set_color(Color::black);
196                                     S->set_color(Color::red);
197                                     RgS->set_color(Color::black);
198                                     return;
199                             }
200                             else if (LgS->get_color() == Color::red)
201                                     P->set_color(Color::black); // to get into the case 5
202                     }
203                     if (P->get_color() == Color::black)
204                     {
205                             if (S->get_color() == Color::red)
206                             {
207
    //case 3: black parent, red left son, LgS has black grandsons
208                                     if ((L_LgS_S-
   >get_color() == Color::black) && (R_LgS_S->get_color() == Color::black))
209                                     {
210                                             left_turn(P);
211                                             S->set_color(Color::black);
212                                             LgS-
   >set_color(Color::red); // he was black like son of the red one
213                                             return;
214                                     }
215
    //case 4: black parent, red left son, LgS has right red son
216                                     else if (R_LgS_S->get_color() == Color::red)
217                                     {
218                                             right_turn(S);
219                                             left_turn(P);
220                                             R_LgS_S->set_color(Color::black);
221                                             return;
222                                     }
223                             }
224                             else
225                             {
226
    //case 5: black parent, black right son with left red son
227                                     if (LgS->get_color() == Color::red)
228                                     {
229                                             right_turn(S);
230                                             left_turn(P);
231                                             LgS->set_color(Color::black);
232                                             return;
233                                     }
234
    //case 6: black parent, black right son with black sons
235                                     else if (RgS->get_color() == Color::black)
236                                     {
237                                             S->set_color(Color::red);
238                                             restoring_propertires_after_delete(P);
239                                             return;
240                                     }
241                             }
242                     }
```

```
243             }
244     }
245
246 public:
247
248     RB_Tree()
249     {
250             Nil = new Node<Key_Type, Value_Type>;
251             Nil->set_color(Color::black);
252             set_root(get_Nil());
253     }
254
255     int get_size() { return size; }
256
257     Node<Key_Type, Value_Type>* get_Nil() { return Nil; }
258
259     Node<Key_Type, Value_Type>* get_root() { return root; }
260
261     //finds node by it's key
262     Node<Key_Type, Value_Type>* find(Key_Type required_key)
263     {
264             if (get_root() == get_Nil())
265                     throw out_of_range("search in the empty tree");
266             Node<Key_Type, Value_Type>* current = root;
267             while (current->get_key() != required_key)
268             {
269                     if (required_key < current->get_key())
270                             current = current->get_left();
271                     else
272                             current = current->get_right();
273                     if (current == get_Nil())
274                             throw out_of_range("search by non-existent key");
275             }
276             return current;
277     }
278
279     //inserting node by it's key and value
280     void insert(Key_Type new_key, Value_Type new_value)
281     {
282             size++;
283
    Node<Key_Type, Value_Type>* new_node = new Node<Key_Type, Value_Type>(new_key, new_valu
   e, Color::red, get_Nil(), get_Nil(), get_Nil());
284             Node<Key_Type, Value_Type>* current = get_root();
285             Node<Key_Type, Value_Type>* current_parent = get_Nil();
286             while (current != get_Nil()) // search for parent of new node
287             {
288                     current_parent = current;
289                     if (new_node->get_key() < current->get_key()) // choose branch
290                             current = current->get_left();
291                     else
292                             current = current->get_right();
293             }
294             new_node->set_parent(current_parent);
295             if (current_parent == get_Nil()) // set new node like a son for his parent
296
    root = new_node; // root is red now and it's not right, but it will be fixed in restori
   ng
297             else if (new_node->get_key() < current_parent-
   >get_key()) //choose for parent ehat son to set
298                     current_parent->set_left(new_node);
```

```
299             else
300                     current_parent->set_right(new_node);
301
    restoring_propertires_after_insert(new_node); // fixes root's color, problem of two red
  s and of black height
302     }
303
304     //removing element by it's key
305     void remove(Key_Type to_delete_key)
306     {
307             if (get_root() == get_Nil())
308                     throw out_of_range("remove from empty tree");
309             Node<Key_Type, Value_Type>* D = find(to_delete_key); // node to delete = D
310             size--;
311             if ((D->get_left() != get_Nil()) && (D-
  >get_right() != get_Nil())) // if D has two sons
312             { // search for max node in left branch for node to delete
313                     Node<Key_Type, Value_Type>* replace_D = D->get_left();
314                     while (replace_D->get_right() != get_Nil())
315                             replace_D = replace_D->get_right();
316                     D->set_key(replace_D->get_key()); // D won't be really deleted
317                     D->set_value(replace_D->get_value());
318
    D = replace_D; // replace_D will be really deleted, we've just changed data of nodes
319             }
320             // now D is a node that has 1 or 0 sons
321             if (D->get_color() == Color::black) // deleting node is black
322             {
323                     if (((D->get_right() != get_Nil()) || (D->get_left() != get_Nil())))
324
    {//case 1: black node with one son (son is red because he is the only one)
325                             Node<Key_Type, Value_Type>* replace_D = D->get_right();
326                             D->set_right(get_Nil());
327                             if (replace_D == get_Nil())
328                             {
329                                     replace_D = D->get_left();
330                                     D->set_left(get_Nil());
331                             }
332                             D->set_key(replace_D->get_key()); // D won't be really deleted
333                             D->set_value(replace_D-
  >get_value()); // exchange values and keys and delete node without sons
334                             delete replace_D;
335                             return;
336                     }
337                     else //case 2: deleting node is black without sons
338                     {
339                             if (D == get_root())
340                             {
341                                     delete D;
342                                     set_root(get_Nil());
343                             }
344                             else
345                             {
346
    // after deleting D the rule of the black height is broken
347                                     restoring_propertires_after_delete(D);
348                                     if (D->get_parent()->get_left() == D)
349                                             D->get_parent()->set_left(get_Nil());
350                                     else
351                                             D->get_parent()->set_right(get_Nil());
352                                     delete D;
```

```
353                             }
354                     }
355             }
356             else // deleting node is red, red node can't have 1 son
357             { // in this place after all we did, red node can have only 0 sons
358                     if (D->get_parent()->get_left() == D)
359                             D->get_parent()->set_left(get_Nil());
360                     else
361                             D->get_parent()->set_right(get_Nil());
362                     delete D;
363             }
364     }
365
366     Iterator<Key_Type, Value_Type>* create_breadth_first_traverse_iterator() // to realise
    this method of passing through elements
367     { return new BreadthFirstTraverse_Iterator(get_root()); }
368
369     class BreadthFirstTraverse_Iterator : public Iterator<Key_Type, Value_Type>
370     {
371     private:
372             Node<Key_Type, Value_Type>* current;
373             Queue<Key_Type, Value_Type> queue;
374             Node<Key_Type, Value_Type>* Nil;
375     public:
376             BreadthFirstTraverse_Iterator(Node<Key_Type, Value_Type>* start)
377             {
378                     Nil = start->get_parent(); // = root->parent = Nil of the tree
379                     current = start;
380                     queue.push(current);
381             }
382             bool has_next() override
383             { return (!queue.is_empty()); }
384             Node<Key_Type, Value_Type>* next() override
385             {
386                     if (queue.is_empty())
387                             return current = Nil;
388                     current = queue.top()->get_data();
389                     if (current->get_left() != Nil)
390                             queue.push(current->get_left());
391                     if (current->get_right() != Nil)
392                             queue.push(current->get_right());
393                     return queue.pop()->get_data();
394             }
395     };
396
397     void print(string filename) //print pairs <key, value> in file with data filename
398     {
399
    Iterator<Key_Type, Value_Type>* iterator = create_breadth_first_traverse_iterator();
400             ofstream out(filename);
401             out << "pairs <key, value>:" << endl;
402             while (iterator->has_next())
403             {
404
    Node<Key_Type, Value_Type>* now = new Node<Key_Type, Value_Type>(iterator->next());
405                     out << now->get_key() << ", " << now->get_value() << endl;
406             }
407             out.close();
408     }
409
410     void print() // print pairs <key, value> in console
```

```
411  {
412
     Iterator<Key_Type, Value_Type>* iterator = create_breadth_first_traverse_iterator();
413          cout << "pairs <key, value>:" << endl;
414          while (iterator->has_next())
415          {
416
     Node<Key_Type, Value_Type>* now = new Node<Key_Type, Value_Type>(iterator->next());
417                  cout << now->get_key() << ", " << now->get_value() << endl;
418          }
419  }
420
421  //returning an array of keys in order of bft iterator
422  Key_Type* get_keys()
423  {
424
     Iterator<Key_Type, Value_Type>* iterator = create_breadth_first_traverse_iterator();
425          Key_Type* keys = new Key_Type[get_size()];
426          int i = 0;
427          while (iterator->has_next())
428          {
429                  keys[i] = iterator->next()->get_key();
430                  i++;
431          }
432          return keys;
433  }
434
435  //returning an array of values in order of bft iterator
436  Value_Type* get_values()
437  {
438
     Iterator<Key_Type, Value_Type>* iterator = create_breadth_first_traverse_iterator();
439          Value_Type* values = new Value_Type[get_size()];
440          int i = 0;
441          while (iterator->has_next())
442          {
443                  values[i] = iterator->next()->get_value();
444                  i++;
445          }
446          return values;
447  }
448
449  // removing all elements of the tree
450  void clear(Node<Key_Type, Value_Type>* now = nullptr)
451  {
452          if (now == nullptr)
453                  now = get_root();
454          if (get_root() == get_Nil())
455                  throw out_of_range("clearing empry tree");
456          size--;
457          if (now->get_left() != Nil)
458                  clear(now->get_left());
459          if (now->get_right() != Nil)
460                  clear(now->get_right());
461          delete now;
462          if (size == 0)
463                  set_root(get_Nil());
464  }
465};
```

### 6.7 UnitTest_RB_Tree.cpp

```
1  #include "pch.h"
2  #include "CppUnitTest.h"
3  #include "../RB_Tree/RB_Tree.h"
4
5  using namespace Microsoft::VisualStudio::CppUnitTestFramework;
6
7  namespace UnitTestRBTree
8  {
9    TEST_CLASS(UnitTestRBTree)
10   {
11   public:
12
13           TEST_METHOD(TestMethod1)
14           {
15                   Assert::AreEqual(1, 1); // check assembly of the project
16           }
17           TEST_METHOD(TestNodeConstructorKey)
18           {
19                   Node<int, int> node(10, 12, Color::red);
20                   Assert::IsTrue(node.get_key() == 10);
21           }
22           TEST_METHOD(TestNodeConstructorRed)
23           {
24                   Node<int, int> node(10, 12);
25                   Assert::IsTrue(node.get_color() == Color::red);
26                   Assert::AreEqual(node.get_key(), 10);
27           }
28
29           TEST_METHOD(TestNodeConstructorColorBlack)
30           {
31                   Node<int, int> node(10, 12, Color::black);
32                   Assert::IsTrue(node.get_color() == Color::black);
33           }
34           TEST_METHOD(TestNodeConstructorParent)
35           {
36                   Node<int, int>* Parent = new Node<int, int>;
37                   Node<int, int> node(10, 12, Color::red, nullptr, nullptr, Parent);
38                   Assert::IsTrue(node.get_parent() == Parent);
39           }
40           TEST_METHOD(TestInsert)
41           {
42                   RB_Tree<int, int> tree;
43                   tree.insert(5, 10); // key, value
44                   Assert::IsTrue((*tree.get_root()).get_color() == Color::black);
45                   Assert::AreEqual((*tree.get_root()).get_key(), 5);
46           }
47           TEST_METHOD(TestInsertSecondNodeRightColor)
48           {
49                   RB_Tree<int, int>* tree = new RB_Tree<int, int>();
50                   tree->insert(5, 10); // key, value
51                   tree->insert(8, 3);
52                   Node<int, int>* test = tree->get_root();
53                   Assert::IsTrue(test->get_right()->get_color() == Color::red);
54           }
55           TEST_METHOD(TestInsertSecondNodeLeftColor)
56           {
57                   RB_Tree<int, int>* tree = new RB_Tree<int, int>;
58                   tree->insert(5, 10); // key, value
59                   tree->insert(1, 3);
60                   Node<int, int>* test = tree->get_root();
```

```
61                      Assert::IsTrue(test->get_left()->get_color() == Color::red);
62              }
63              TEST_METHOD(TestInsertThirdNodeColor)
64              {
65                      RB_Tree<int, int>* tree = new RB_Tree<int, int>;
66                      tree->insert(5, 10); // key, value
67                      tree->insert(8, 3);
68                      tree->insert(1, 4);
69                      Node<int, int>* test = tree->get_root();
70                      Assert::IsTrue(test->get_left()->get_color() == Color::red);
71              }
72              TEST_METHOD(TestInsertLeftTurnCheck)
73              {
74
75                      RB_Tree<int, int>* tree = new RB_Tree<int, int>;
76                      tree->insert(5, 5); // key, value
77                      tree->insert(10, 10);
78                      tree->insert(12, 12);
79                      Assert::AreEqual(tree->get_root()->get_key(), 10);
80                      Assert::IsTrue(tree->get_root()->get_color() == Color::black);
81                      Assert::AreEqual(tree->get_root()->get_left()->get_key(), 5);
82                      Assert::AreEqual(tree->get_root()->get_right()->get_key(), 12);
83                      Assert::IsTrue(tree->get_root()->get_left()-
    >get_color() == Color::red);
84                      Assert::IsTrue(tree->get_root()->get_right()-
    >get_color() == Color::red);
85              }
86              TEST_METHOD(TestInsertRightTurnCheck)
87              {
88
89                      RB_Tree<int, int>* tree = new RB_Tree<int, int>;
90                      tree->insert(5, 5); // key, value
91                      tree->insert(4, 4);
92                      tree->insert(3, 3);
93                      Assert::AreEqual(tree->get_root()->get_key(), 4);
94              }
95              TEST_METHOD(test_insert_1_2_3_4)
96              {
97                      RB_Tree<int, int>* tree = new RB_Tree<int, int>;
98                      tree->insert(1, 1); // key, value
99                      tree->insert(2, 2);
100                     tree->insert(3, 3);
101                     tree->insert(4, 4);
102                     Assert::AreEqual(tree->get_root()->get_key(), 2);
103                     Assert::IsTrue(tree->get_root()->get_color() == Color::black);
104                     Assert::AreEqual(tree->get_root()->get_left()->get_key(), 1);
105                     Assert::IsTrue(tree->get_root()->get_left()-
    >get_color() == Color::black);
106                     Assert::AreEqual(tree->get_root()->get_right()->get_key(), 3);
107                     Assert::IsTrue(tree->get_root()->get_right()-
    >get_color() == Color::black);
108                     Assert::AreEqual(tree->get_root()->get_right()->get_right()-
    >get_key(), 4);
109                     Assert::IsTrue(tree->get_root()->get_right()->get_right()-
    >get_color() == Color::red);
110             }
111
112             TEST_METHOD(test_insert_1_2_3_4_5)
113             {
114                     RB_Tree<int, int>* tree = new RB_Tree<int, int>;
115                     tree->insert(1, 1); // key, value
```

```
116                     tree->insert(2, 2);
117                     tree->insert(3, 3);
118                     tree->insert(4, 4);
119                     tree->insert(5, 5);
120                     Assert::AreEqual(tree->get_root()->get_key(), 2);
121                     Assert::IsTrue(tree->get_root()->get_color() == Color::black);
122                     Assert::AreEqual(tree->get_root()->get_left()->get_key(), 1);
123                     Assert::IsTrue(tree->get_root()->get_left()-
    >get_color() == Color::black);
124                     Assert::AreEqual(tree->get_root()->get_right()->get_key(), 4);
125                     Assert::IsTrue(tree->get_root()->get_right()-
    >get_color() == Color::black);
126                     Assert::AreEqual(tree->get_root()->get_right()->get_right()-
    >get_key(), 5);
127                     Assert::IsTrue(tree->get_root()->get_right()->get_right()-
    >get_color() == Color::red);
128                     Assert::AreEqual(tree->get_root()->get_right()->get_left()-
    >get_key(), 3);
129                     Assert::IsTrue(tree->get_root()->get_right()->get_left()-
    >get_color() == Color::red);
130             }
131
132             TEST_METHOD(test_insert_1_2_3_4_5_6)
133             {
134                     RB_Tree<int, int>* tree = new RB_Tree<int, int>;
135                     tree->insert(1, 1); // key, value
136                     tree->insert(2, 2);
137                     tree->insert(3, 3);
138                     tree->insert(4, 4);
139                     tree->insert(5, 5);
140                     tree->insert(6, 6);
141                     Assert::AreEqual(tree->get_root()->get_key(), 2);
142                     Assert::IsTrue(tree->get_root()->get_color() == Color::black);
143                     Assert::AreEqual(tree->get_root()->get_left()->get_key(), 1);
144                     Assert::IsTrue(tree->get_root()->get_left()-
    >get_color() == Color::black);
145                     Assert::AreEqual(tree->get_root()->get_right()->get_key(), 4);
146                     Assert::IsTrue(tree->get_root()->get_right()-
    >get_color() == Color::red);
147                     Assert::AreEqual(tree->get_root()->get_right()->get_right()-
    >get_key(), 5);
148                     Assert::IsTrue(tree->get_root()->get_right()->get_right()-
    >get_color() == Color::black);
149                     Assert::AreEqual(tree->get_root()->get_right()->get_left()-
    >get_key(), 3);
150                     Assert::IsTrue(tree->get_root()->get_right()->get_left()-
    >get_color() == Color::black);
151                     Assert::AreEqual(tree->get_root()->get_right()->get_right()-
    >get_right()->get_key(), 6);
152                     Assert::IsTrue(tree->get_root()->get_right()->get_right()-
    >get_right()->get_color() == Color::red);
153             }
154             TEST_METHOD(BigInsert)
155             {
156                     RB_Tree<int, int>* tree = new RB_Tree<int, int>;
157                     tree->insert(5, 5); // key, value
158                     tree->insert(10, 10);
159                     tree->insert(12, 12);
160                     tree->insert(14, 14);
161                     tree->insert(7, 7);
162                     tree->insert(11, 11);
```

```
163                     tree->insert(19, 19);
164                     tree->insert(3, 3);
165                     tree->insert(8, 8);
166                     tree->insert(6, 6);
167                     Assert::AreEqual(tree->get_root()->get_key(), 10);
168             }
169
170             TEST_METHOD(test_delete_root)
171             {
172                     RB_Tree<int, int>* tree = new RB_Tree<int, int>;
173                     tree->insert(5, 5); // key, value
174                     tree->remove(5);
175                     Assert::IsTrue(tree->get_root() == tree->get_Nil());
176             }
177             TEST_METHOD(test_delete_root_two_nodes_in_tree)
178             {
179                     RB_Tree<int, int>* tree = new RB_Tree<int, int>;
180                     tree->insert(5, 5); // key, value
181                     tree->insert(3, 3);
182                     tree->remove(5);
183                     Assert::AreEqual(tree->get_root()->get_key(), 3);
184                     Assert::IsTrue((tree->get_root()->get_left() == tree-
    >get_Nil()) && (tree->get_root()->get_right() == tree->get_Nil()));
185             }
186             TEST_METHOD(test_delete_not_root_two_nodes_in_tree)
187             {
188                     RB_Tree<int, int>* tree = new RB_Tree<int, int>;
189                     tree->insert(5, 5); // key, value
190                     tree->insert(3, 3);
191                     tree->remove(3);
192                     Assert::AreEqual(tree->get_root()->get_key(), 5);
193                     Assert::IsTrue((tree->get_root()->get_left() == tree-
    >get_Nil()) && (tree->get_root()->get_right() == tree->get_Nil()));
194             }
195             TEST_METHOD(test_delete_root_two_nodes_in_tree_right_child)
196             {
197                     RB_Tree<int, int>* tree = new RB_Tree<int, int>;
198                     tree->insert(5, 5); // key, value
199                     tree->insert(7, 7);
200                     tree->remove(5);
201                     Assert::AreEqual(tree->get_root()->get_key(), 7);
202                     Assert::IsTrue((tree->get_root()->get_left() == tree-
    >get_Nil()) && (tree->get_root()->get_right() == tree->get_Nil()));
203             }
204             TEST_METHOD(test_delete_not_root_two_nodes_in_tree_right_child)
205             {
206                     RB_Tree<int, int>* tree = new RB_Tree<int, int>;
207                     tree->insert(5, 5); // key, value
208                     tree->insert(7, 7);
209                     tree->remove(7);
210                     Assert::AreEqual(tree->get_root()->get_key(), 5);
211                     Assert::IsTrue((tree->get_root()->get_left() == tree-
    >get_Nil()) && (tree->get_root()->get_right() == tree->get_Nil()));
212             }
213             TEST_METHOD(iterator_test)
214             {
215                     RB_Tree<int, int>* tree = new RB_Tree<int, int>;
216                     tree->insert(1, 1); // key, value
217                     tree->insert(2, 2);
218                     tree->insert(3, 3);
219                     tree->insert(4, 4);
```

```
220                 tree->insert(5, 5);
221                 tree->insert(6, 6);
222                 Iterator<int, int>* iterator = tree-
    >create_breadth_first_traverse_iterator();
223                 if (iterator->has_next())
224                         Assert::AreEqual(iterator->next()->get_key(), 2);
225                 if (iterator->has_next())
226                         Assert::AreEqual(iterator->next()->get_key(), 1);
227                 if (iterator->has_next())
228                         Assert::AreEqual(iterator->next()->get_key(), 4);
229                 if (iterator->has_next())
230                         Assert::AreEqual(iterator->next()->get_key(), 3);
231                 if (iterator->has_next())
232                         Assert::AreEqual(iterator->next()->get_key(), 5);
233                 if (iterator->has_next())
234                         Assert::AreEqual(iterator->next()->get_key(), 6);
235                 Assert::AreEqual(iterator->has_next(), false);
236         }
237         TEST_METHOD(print_test)
238         {
239                 RB_Tree<int, int>* tree = new RB_Tree<int, int>;
240                 tree->insert(1, 1); // key, value
241                 tree->insert(2, 2);
242                 tree->insert(3, 3);
243                 tree->insert(4, 4);
244                 tree->insert(5, 5);
245                 tree->insert(6, 6);
246                 tree->print("../out.txt"); // check file "..UnitTest_RB_Tree/out.txt"
247                 tree->print();
248                 Assert::IsTrue(true);
249         }
250
251         TEST_METHOD(get_keys_test)
252         {
253                 RB_Tree<int, int>* tree = new RB_Tree<int, int>;
254                 tree->insert(1, 1); // key, value
255                 tree->insert(2, 2);
256                 tree->insert(3, 3);
257                 tree->insert(4, 4);
258                 tree->insert(5, 5);
259                 tree->insert(6, 6);
260                 int* keys = tree->get_keys();
261                 Assert::AreEqual(keys[0], 2);
262                 Assert::AreEqual(keys[1], 1);
263                 Assert::AreEqual(keys[2], 4);
264                 Assert::AreEqual(keys[3], 3);
265                 Assert::AreEqual(keys[4], 5);
266                 Assert::AreEqual(keys[5], 6);
267         }
268
269         TEST_METHOD(get_values_test)
270         {
271                 RB_Tree<int, int>* tree = new RB_Tree<int, int>;
272                 tree->insert(1, 1); // key, value
273                 tree->insert(2, 2);
274                 tree->insert(3, 3);
275                 tree->insert(4, 4);
276                 tree->insert(5, 5);
277                 tree->insert(6, 6);
278                 int* values = tree->get_values();
279                 Assert::AreEqual(values[0], 2);
```

```
280                     Assert::AreEqual(values[1], 1);
281                     Assert::AreEqual(values[2], 4);
282                     Assert::AreEqual(values[3], 3);
283                     Assert::AreEqual(values[4], 5);
284                     Assert::AreEqual(values[5], 6);
285             }
286
287             TEST_METHOD(clear_test)
288             {
289                     RB_Tree<int, int>* tree = new RB_Tree<int, int>;
290                     tree->insert(1, 1); // key, value
291                     tree->insert(2, 2);
292                     tree->insert(3, 3);
293                     tree->insert(4, 4);
294                     tree->insert(5, 5);
295                     tree->insert(6, 6);
296                     tree->clear();
297                     Assert::IsTrue(tree->get_root() == tree->get_Nil());
298             }
299     };
300 }
```