**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 3](#_Toc67609164)

[2 Реализуемые классы и методы 3](#_Toc67609165)

[3 Оценка временной сложности каждого метода 5](#_Toc67609166)

[4 Описание реализованных Unit test 6](#_Toc67609167)

[5 Пример работы программы 6](#_Toc67609168)

[6 Листинг 7](#_Toc67609169)

[6.1 Element.h 7](#_Toc67609170)

[6.2 Node.h 7](#_Toc67609171)

[6.3 Iterator.h 8](#_Toc67609172)

[6.4 Queue.h 9](#_Toc67609173)

[6.5 Stack.h 10](#_Toc67609174)

[6.6 RB\_Tree.h 11](#_Toc67609175)

[6.7 UnitTest\_RB\_Tree.cpp 20](#_Toc67609176)

# 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

# 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.

# 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.

# 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.

# 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:

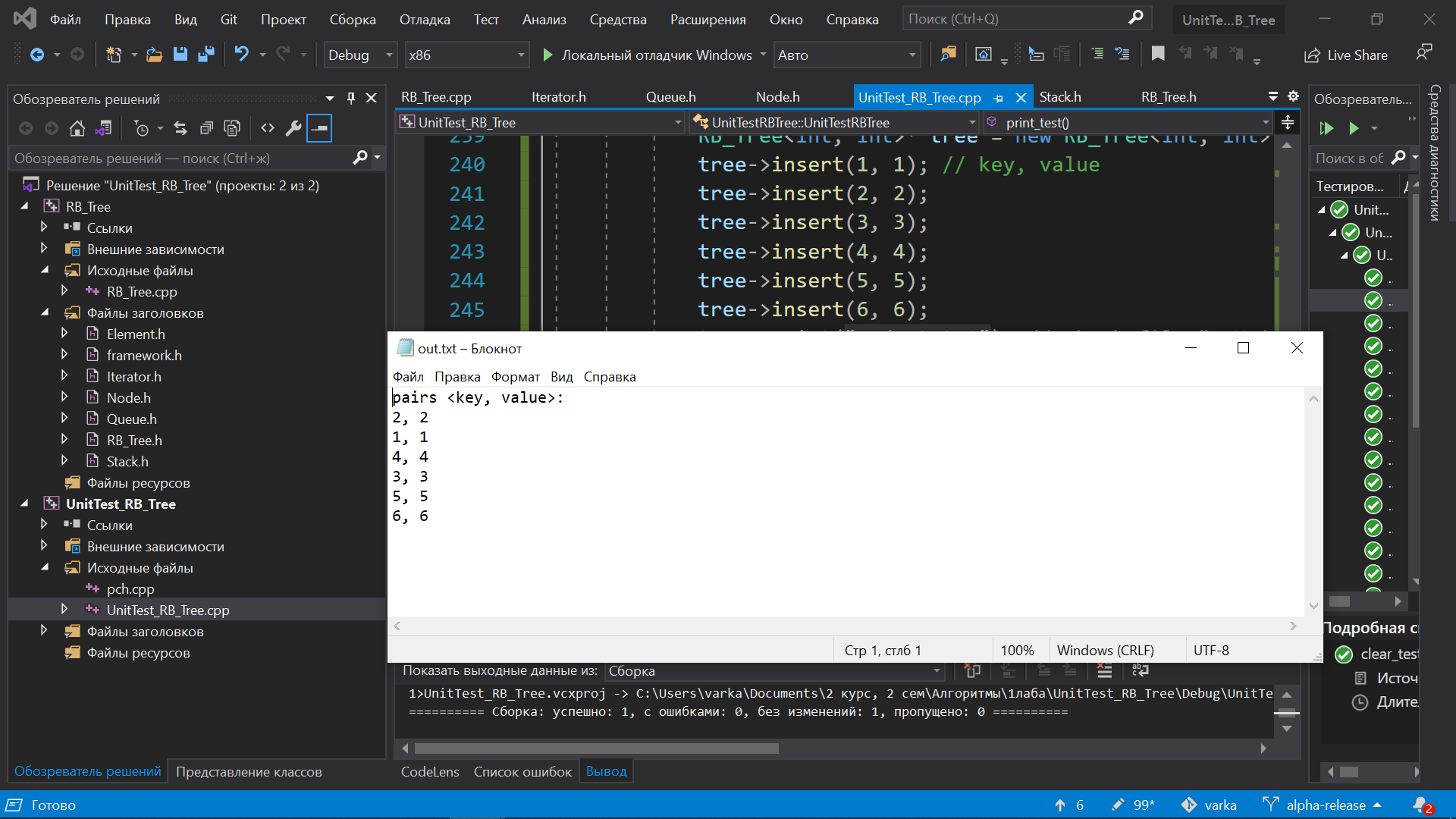


Рис. 5.1 — пример работы программы.

# Листинг

## 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; }
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. };

## Node.h

1. #pragma once
2. #include <stdexcept>
3. #include <iostream> // is needed?
5. using namespace std;
6. enum class Color { red, black };
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; }
24. public:
25. Node(void) {}
26. Node(Key\_Type new\_key, Value\_Type new\_value, Color new\_color = Color::red, Node<Key\_Type, Value\_Type>\* new\_left = nullptr, Node<Key\_Type, Value\_Type>\* new\_right = nullptr, Node<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. };

## 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 the next
8. virtual bool has\_next() = 0; // return true if next exists
9. };

## 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; }
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. }
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. }
69. ~Queue()
70. {
71. while (!is\_empty())
72. pop();
73. }
74. };

## 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. };

## RB\_Tree.h

1. #pragma once
2. #include "Queue.h"
3. #include "Stack.h"
4. #include "Iterator.h"
5. #include <fstream>
7. template <class Key\_Type, class Value\_Type>
8. class RB\_Tree {
9. private:
11. Node<Key\_Type, Value\_Type>\* root = nullptr;
12. Node<Key\_Type, Value\_Type>\* Nil = nullptr; // node after all lists, one big common child 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; }
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. }
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. }
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. }
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. }
246. public:
248. RB\_Tree()
249. {
250. Nil = new Node<Key\_Type, Value\_Type>;
251. Nil->set\_color(Color::black);
252. set\_root(get\_Nil());
253. }
255. int get\_size() { return size; }
257. Node<Key\_Type, Value\_Type>\* get\_Nil() { return Nil; }
259. Node<Key\_Type, Value\_Type>\* get\_root() { return root; }
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. }
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\_value, 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 restoring
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 reds and of black height
302. }
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. }
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()); }
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. };
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. }
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. }
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. }
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. }
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. };

## UnitTest\_RB\_Tree.cpp

1. #include "pch.h"
2. #include "CppUnitTest.h"
3. #include "../RB\_Tree/RB\_Tree.h"
5. using namespace Microsoft::VisualStudio::CppUnitTestFramework;
7. namespace UnitTestRBTree
8. {
9. TEST\_CLASS(UnitTestRBTree)
10. {
11. public:
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. }
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. {
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. {
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. }
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. }
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. }
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. }
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. }
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. }
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. }