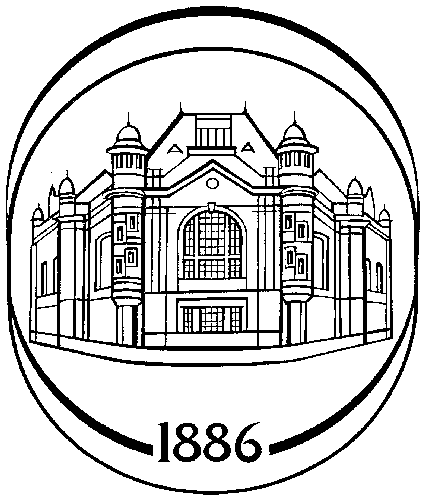
МИНОБРНАУКИ РОССИИ

Федеральное государственное автономное образовательное учреждение высшего образования

**Санкт-Петербургский государственный электротехнический университет “ЛЭТИ” им. В. И. Ульянова (Ленина) (СПбГЭТУ «ЛЭТИ»)**



Кафедра САПР

Лабораторная работа № 3

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

Тема: «Двоичные деревья»

Вариант № 23 (1)

|  |  |  |
| --- | --- | --- |
| Студентка гр. 9301, ФКТИ |  | Синицкая В. А. |
| Преподаватель |  | Тутуева А. В. |

Санкт-Петербург

2020 г.

Постановка задачи

Задача для 1 варианта:

Двоичное дерево поиска:

1. bool contains(int); // поиск элемента в дереве по ключу
2. void insert(int); // добавление элемента в дерево по ключу. Должен работать за O(logN)
3. void remove(int); // удаление элемента дерева по ключу
4. Iterator create\_dft\_iterator(); // создание итератора, реализующего один из методов обхода в глубину (depth-first traverse)
5. Iterator create\_bft\_iterator() // создание итератора, реализующего методы обхода в ширину (breadth-first traverse)

Итераторы на основе собственной очереди и стека.

Описание класса и методов

В проекте содержатся классы Queue, Stack, Element, Node, BinarySearchTree, Iterator. Файлы: QueueAndStack.h, QueueAndStack.cpp, BinarySearchTree.h, BinarySearchTree.cpp, UnitTestForQueueAndStack.cpp, UnitTestForBinarySearchTree.cpp. BinarySearchTree содержит BreadthFirstTraverse\_Iterator, PreOrderedDepthFirstTraverse\_Iterator, InOrderedDepthFirstTraverse\_Iterator, PostOrderedDepthFirstTraverse\_Iterator – классы, наследованные от Iterator.

Node – это узел, содержит в себе ключ типа int и два указателя на левого и правого ребенка соответственно.

Element – это элемент для стека и очереди, поле дата у него типа Node, а также присутствует поле следующего элемента.

Queue содержит поля head и tail, реализованы методы:

1. int size() – возвращает количество элементов в очереди
2. bool is\_empty() – возвращает правду, если очередь пуста, иначе – ложь
3. Element\* top() – возвращает без удаления из очереди тот элемент, который будет обработан первым
4. Element\* pop() – удаляет и возвращает первый в очереди элемент
5. Element\* push(Node\*) – добавляет элемент с заданной датой в конец очереди
6. Element\* back() – возвращает без удаления последний элемент в очереди

Stack содержит поле top, реализованы методы:

1. int size() – возвращает количество элементов в стеке
2. bool is\_empty() – возвращает правду, если стек пуст, иначе – ложь
3. Element\* peek() – возвращает без удаления верхний элемент
4. Element\* pop() – удаляет и возвращает верхний элемент
5. Element\* push(Node\*) – добавляет наверх стека элемент с заданной датой

Iterator содержит поле current и методы:

1. next() – вернуть текущий, перейти на следующий
2. has\_next() – узнать, есть ли следующий

BinarySearchTree содержит корень root, в нем реализованы:

1. bool is\_empty() – проверка на пустоту
2. Node\* get\_root() – узнать корень
3. void insert(int) - добавление элемента в дерево по ключу
4. bool contains(int) - поиск элемента в дереве по ключу
5. void remove(int) - удаление элемента дерева по ключу
6. Iterator\* create\_breadth\_first\_traverse\_iterator() – итератор для обхода дерева в ширину
7. Iterator\* create\_pre\_ordered\_depth\_first\_traverse\_iterator() – итератор для прямого обхода дерева в глубину
8. Iterator\* create\_in\_ordered\_depth\_first\_traverse\_iterator() – итератор для обратного обхода дерева в глубину
9. Iterator\* create\_post\_ordered\_depth\_first\_traverse\_iterator() – итератор для концевого обхода дерева в глубину

Оценка временной сложности

Queue:

1. int size() – O(n)
2. bool is\_empty() – O(1)
3. Element\* top() – O(1)
4. Element\* pop() – O(n)
5. Element\* push(Node\*) – O(1)
6. Element\* back() – O(1)

Stack:

1. int size() – O(n)
2. bool is\_empty() – O(1)
3. Element\* peek() – O(1)
4. Element\* pop() – O(1)
5. Element\* push(Node\*) – O(1)

BinarySearchTree:

1. bool is\_empty() – O(1)
2. Node\* get\_root() – O(1)
3. void insert(int) – O(log(n)) (log(n) – количество вариантов выбора, слева или справа)
4. bool contains(int) - O(log(n)) (log(n) – количество вариантов выбора, слева или справа)
5. void remove(int) - O(log(n)) + O(log(n)) => O(log(n))

BreadthFirstTraverse\_Iterator :

1. next() – 3\*O(1) + O(n) => O(n)
2. has\_next() – O(1)

PreOrderedDepthFirstTraverse\_Iterator:

1. next() – 3\*O(1) => O(1)
2. has\_next() – O(1)

InOrderedDepthFirstTraverse\_Iterator:

1. next() – O(1) + O(n) => O(n) (может быть левое дерево)
2. has\_next() – O(1)

PostOrderedDepthFirstTraverse\_Iterator:

1. O(n) (ни один спуск из текущего узла не может превысить n, учитывая однозначный выбор ветви)
2. has\_next() – O(1)

Описание реализованных unit-тестов

Для стека и очереди реализовано 20 тестов для проверки правильности работы всех функций стека и очереди на разном количестве элементов, на обработку исключений.

Для итераторов и бинарного дерева поиска реализовано 38 тестов на правильную работу всех функций дерева на разных размерах, для проверки исключений, для проверки работы всех итераторов на разных деревьях.

Пример работы

При функции main():

1. int main()
2. {
3. BinarySearchTree BST(8);
4. BST.insert(3); BST.insert(10); BST.insert(14); BST.insert(13);
5. BST.insert(6); BST.insert(1); BST.insert(7); BST.insert(4);
6. Iterator\* post\_dft\_iterator = BST.create\_post\_ordered\_depth\_first\_traverse\_iterator();
7. Iterator\* pre\_dft\_iterator = BST.create\_pre\_ordered\_depth\_first\_traverse\_iterator();
8. Iterator\* in\_dft\_iterator = BST.create\_in\_ordered\_depth\_first\_traverse\_iterator();
9. Iterator\* bft\_iterator = BST.create\_breadth\_first\_traverse\_iterator();
10. cout << "Tree, passed by breadth-first iterator: " << endl;
11. while (bft\_iterator->has\_next())
12. cout << bft\_iterator->next()->get\_key() << " ";
13. cout << endl;
14. cout << "Tree, passed by pre-ordered depth-first iterator: " << endl;
15. while (pre\_dft\_iterator->has\_next())
16. cout << pre\_dft\_iterator->next()->get\_key() << " ";
17. cout << endl;
18. cout << "Tree, passed by in-ordered depth-first iterator: " << endl;
19. while (in\_dft\_iterator->has\_next())
20. cout << in\_dft\_iterator->next()->get\_key() << " ";
21. cout << endl;
22. cout << "Tree, passed by post-ordered depth-first iterator: " << endl;
23. while (post\_dft\_iterator->has\_next())
24. cout << post\_dft\_iterator->next()->get\_key() << " ";
25. cout << endl;
26. return 0;
27. }

Результат, выведенный в консоль:

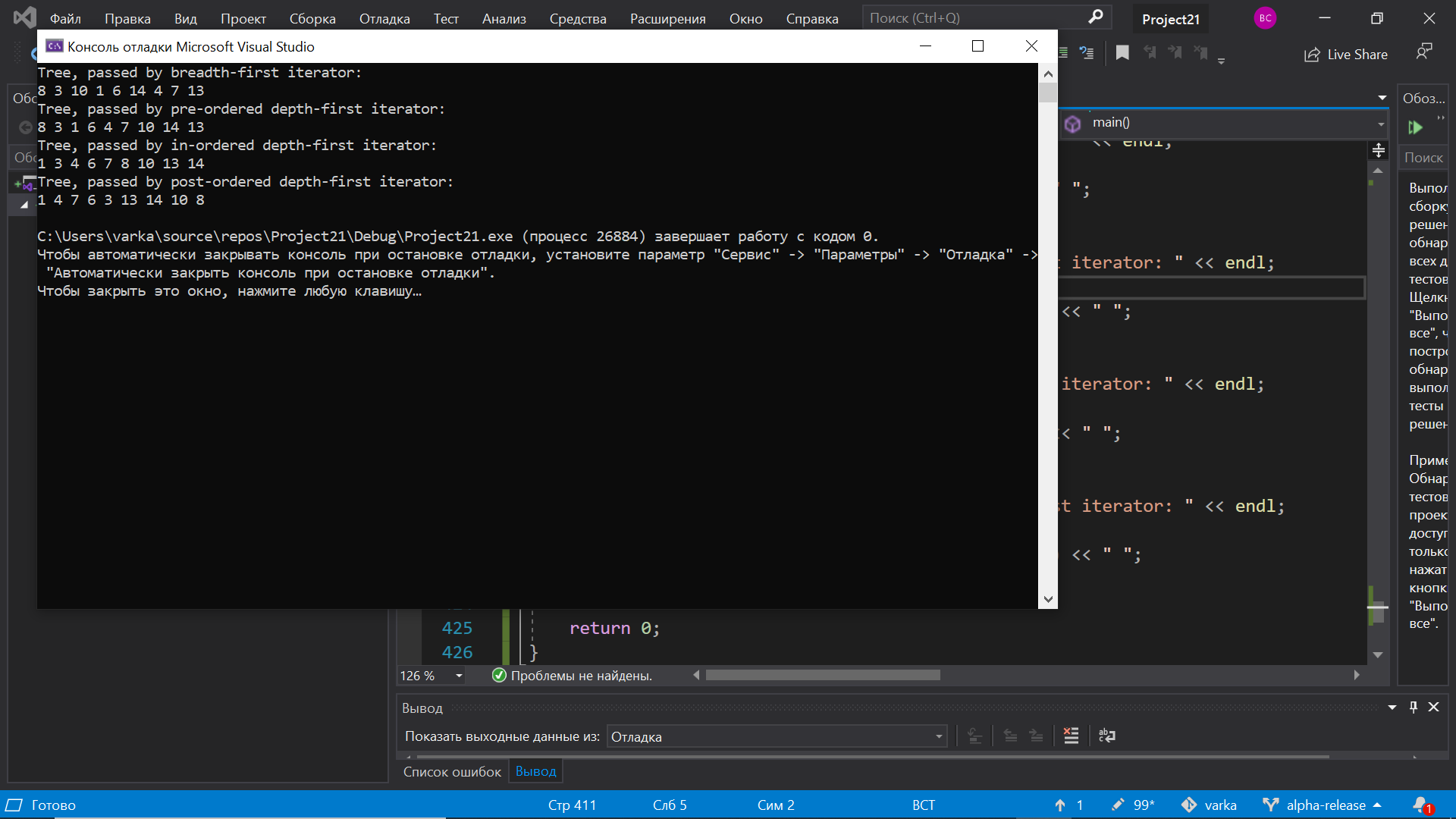


Рис. 1.1 – Пример работы программы

Листинг

QueueAndStack.h:

1. #pragma once
2. #include <iostream>
3. #include <stdexcept>
4. using namespace std;
5. class Node
6. {
7. private:
8. int key;
9. Node\* left\_child;
10. Node\* right\_child;
11. Node\* set\_left\_child(int);
12. Node\* set\_right\_child(int);
13. void set\_key(int);
14. public:
15. Node();
16. Node(int);
17. friend class BinarySearchTree;
18. friend class Element;
19. friend class Queue;
20. friend class Stack;
21. Node\* get\_left\_child();
22. Node\* get\_right\_child();
23. int get\_key();
24. ~Node();
25. };
26. class Element
27. {
28. Element\* next;
29. Node\* data;
30. void set\_next(Element\*);
31. void set\_data(Node\*);
32. public:
33. Element();
34. Element\* get\_next();
35. Node\* get\_data();
36. ~Element();
37. friend class Queue;
38. friend class Stack;
39. };
40. class Queue
41. {
42. private:
43. Element\* head; // the first in queue, leaves first
44. Element\* tail; // the last, new in queue
45. void set\_head(Element\*);
46. void set\_tail(Element\*);
47. public:
48. Queue();
49. int size(); // return a number of elements in queue
50. bool is\_empty(); // returns true if queue is empty
51. Element\* top(); // to know who will leave first
52. Element\* pop(); // to delete first in queue and return it to user
53. Element\* push(Node\*); // to add in the end of the queue the element with key-data and return it to user
54. Element\* back(); // to know who is the last in queue and return it to user
55. ~Queue();
56. };
57. class Stack
58. {
59. private:
60. Element\* top;
61. void set\_top(Element\*);
62. public:
63. Stack();
64. bool is\_empty();
65. int size();
66. Element\* peek(); // to show who's on the top
67. Element\* pop(); // to delte top and show it
68. Element\* push(Node\*); // to push on top and show it
69. ~Stack();
70. };

BinarySearchTree.h:

1. #pragma once
2. #include "QueueAndStack.h"
3. using namespace std;
4. class Iterator
5. {
6. public:
7. virtual Node\* next() = 0; // return current element and goes to the next
8. virtual bool has\_next() = 0; // return true if next exists
9. };
10. class BinarySearchTree // BST
11. {
12. private:
13. Node\* root;
14. Node\* set\_root(int);
15. public:
16. BinarySearchTree();
17. BinarySearchTree(int);
18. bool is\_empty();
19. Node\* get\_root();
20. void insert(int); // add element with given key
21. bool contains(int); // search element with given key
22. void remove(int); // delete element with given key
23. Iterator\* create\_breadth\_first\_traverse\_iterator(); // to realise this method of passing through elements
24. class BreadthFirstTraverse\_Iterator : public Iterator
25. {
26. private:
27. Node\* current;
28. Queue queue;
29. public:
30. BreadthFirstTraverse\_Iterator(Node\* start);
31. Node\* next() override;
32. bool has\_next() override;
33. };
34. Iterator\* create\_pre\_ordered\_depth\_first\_traverse\_iterator(); // to realise this method of passing through elements
35. class PreOrderedDepthFirstTraverse\_Iterator : public Iterator
36. {
37. private:
38. Node\* current;
39. Stack stack;
40. public:
41. PreOrderedDepthFirstTraverse\_Iterator(Node\* start);
42. Node\* next() override;
43. bool has\_next() override;
44. };
45. Iterator\* create\_in\_ordered\_depth\_first\_traverse\_iterator(); // to realise this method of passing through elements
46. class InOrderedDepthFirstTraverse\_Iterator : public Iterator
47. {
48. private:
49. Node\* current;
50. Stack stack;
51. public:
52. InOrderedDepthFirstTraverse\_Iterator(Node\* start);
53. Node\* next() override;
54. bool has\_next() override;
55. };
56. Iterator\* create\_post\_ordered\_depth\_first\_traverse\_iterator(); // to realise this method of passing through elements
57. class PostOrderedDepthFirstTraverse\_Iterator : public Iterator
58. {
59. private:
60. Node\* current;
61. Stack stack;
62. public:
63. PostOrderedDepthFirstTraverse\_Iterator(Node\* start);
64. Node\* next() override;
65. bool has\_next() override;
66. };
67. ~BinarySearchTree();
68. };

QueueAndStack.cpp:

1. #include "QueueAndStack.h"
2. Node::Node()
3. {
4. key = 0;
5. left\_child = nullptr;
6. right\_child = nullptr;
7. }
8. Node::Node(int key\_number)
9. {
10. key = key\_number;
11. left\_child = nullptr;
12. right\_child = nullptr;
13. }
14. Node\* Node::set\_left\_child(int child\_key)
15. {
16. Node\* new\_node = new Node(child\_key);
17. left\_child = new\_node;
18. return left\_child;
19. }
20. Node\* Node::set\_right\_child(int child\_key)
21. {
22. Node\* new\_node = new Node(child\_key);
23. right\_child = new\_node;
24. return right\_child;
25. }
26. void Node::set\_key(int key\_number) { key = key\_number; }
27. Node\* Node::get\_left\_child() { return left\_child; }
28. Node\* Node::get\_right\_child() { return right\_child; }
29. int Node::get\_key() { return key; }
30. Node::~Node()
31. {
32. delete left\_child;
33. delete right\_child;
34. left\_child = right\_child = nullptr;
35. }
36. Element::Element()
37. {
38. data = 0;
39. next = nullptr;
40. }
41. Element\* Element::get\_next() { return next; }
42. Node\* Element::get\_data() { return data; }
43. void Element::set\_next(Element\* new\_element) { next = new\_element; }
44. void Element::set\_data(Node\* information) { data = information; }
45. Element::~Element() { delete next; next = nullptr; }
46. Queue::Queue()
47. {
48. head = nullptr;
49. tail = nullptr;
50. }
51. void Queue::set\_head(Element\* key) { head = key; }
52. void Queue::set\_tail(Element\* key) { tail = key; }
53. bool Queue::is\_empty() // returns true if queue is empty
54. {
55. if (tail == nullptr)
56. return true;
57. return false;
58. }
59. int Queue::size() // return a number of elements in queue
60. {
61. if (is\_empty())
62. return 0;
63. int size = 1;
64. for (Element\* now = tail; now->get\_next() != nullptr; now = now->get\_next())
65. size++;
66. return size;
67. }
68. Element\* Queue::top() // to know who will leave first
69. { return head; }
70. Element\* Queue::push(Node\* data\_key) // to add in the end of the queue the element with key-data
71. {
72. Element\* new\_element = new Element;
73. new\_element->set\_data(data\_key);
74. if (is\_empty())
75. head = tail = new\_element;
76. else
77. {
78. new\_element->set\_next(tail);
79. tail = new\_element;
80. }
81. return new\_element;
82. }
83. Element\* Queue::pop() // to delete first in queue and return it's key-data
84. {
85. Element\* to\_delete;
86. if (is\_empty())
87. throw out\_of\_range("the queue is empty");
88. else if (size() == 1)
89. {
90. to\_delete = head;
91. head = tail = nullptr;
92. }
93. else
94. {
95. to\_delete = head;
96. for (Element\* now = tail; now->get\_next() != nullptr; now = now->get\_next())
97. head = now;
98. head->next = nullptr;
99. }
100. return to\_delete;
101. }
102. Element\* Queue::back() // to know who is the last in queue
103. { return tail; }
104. Queue::~Queue()
105. {
106. while (!is\_empty())
107. pop();
108. }
109. void Stack::set\_top(Element\* top\_element) { top = top\_element; }
110. Stack::Stack()
111. {
112. top = nullptr;
113. }
114. bool Stack::is\_empty()
115. {
116. if (top == nullptr)
117. return true;
118. return false;
119. }
120. int Stack::size()
121. {
122. int size = 1;
123. if (is\_empty())
124. return 0;
125. for (Element\* now = top; now->get\_next() != nullptr; now = now->get\_next())
126. size++;
127. return size;
128. }
129. Element\* Stack::peek() // to show who's on the top
130. {
131. return top;
132. }
133. Element\* Stack::pop() // to delte top and show it
134. {
135. Element\* to\_delete = top;
136. if (is\_empty())
137. throw out\_of\_range("the stack is empty");
138. else
139. top = top->get\_next();
140. return to\_delete;
141. }
142. Element\* Stack::push(Node\* data) // to push on top and show it
143. {
144. Element\* new\_element = new Element;
145. new\_element->set\_data(data);
146. if (is\_empty())
147. top = new\_element;
148. else
149. {
150. new\_element->set\_next(top);
151. top = new\_element;
152. }
153. return new\_element;
154. }
155. Stack::~Stack()
156. {
157. while (!is\_empty())
158. pop();
159. }

BinarySearchTree.cpp:

1. #include "/Users/varka/source/repos/BinarySearchTree/BinarySearchTree.h"
2. #include "/Users/varka/source/repos/BinarySearchTree/QueueAndStack.h"
3. BinarySearchTree::BinarySearchTree()
4. { root = nullptr; }
5. BinarySearchTree::BinarySearchTree(int key\_number)
6. {
7. Node\* new\_node = new Node;
8. new\_node->key = key\_number;
9. root = new\_node;
10. }
11. bool BinarySearchTree::is\_empty() { return (root == nullptr); }
12. Node\* BinarySearchTree::set\_root(int root\_key)
13. {
14. Node\* new\_node = new Node(root\_key);
15. new\_node->key = root\_key;
16. root = new\_node;
17. return root;
18. }
19. Node\* BinarySearchTree::get\_root() { return root; }
20. void BinarySearchTree::insert(int vertex\_key)
21. {
22. if (is\_empty())
23. set\_root(vertex\_key);
24. else
25. {
26. Node\* now = root;
27. while (true) // while did not set element
28. {
29. if (vertex\_key <= now->get\_key())
30. { // search for the place in the left branch
31. if (now->get\_left\_child() == nullptr)
32. {
33. now->set\_left\_child(vertex\_key);
34. break;
35. }
36. now = now->get\_left\_child();
37. }
38. else
39. { // search for the place in the right branch
40. if (now->get\_right\_child() == nullptr)
41. {
42. now->set\_right\_child(vertex\_key);
43. break;
44. }
45. now = now->get\_right\_child();
46. }
47. }
48. }
49. }
50. bool BinarySearchTree::contains(int vertex\_key)
51. {
52. if (is\_empty())
53. throw out\_of\_range("search in the empty tree");
54. Node\* now = root;
55. while (vertex\_key != now->get\_key())
56. {
57. if (vertex\_key < now->get\_key())
58. { // search in the left branch
59. if (now->get\_left\_child() == nullptr)
60. return false; // false case, element doesn't exist
61. now = now->get\_left\_child();
62. }
63. else
64. { // search in the right branch
65. if (now->get\_right\_child() == nullptr)
66. return false; // false case, element doesn't exist
67. now = now->get\_right\_child();
68. }
69. }
70. return true;
71. }
72. void BinarySearchTree::remove(int vertex\_key)
73. {
74. if (is\_empty())
75. throw out\_of\_range("remove from empty tree");
76. // we should find this element
77. Node\* to\_delete = root;
78. Node\* parent\_of\_to\_delete = root;
79. while (vertex\_key != to\_delete->get\_key())
80. {
81. if (vertex\_key < to\_delete->get\_key())
82. { // search in the left branch
83. if (to\_delete->get\_left\_child() == nullptr)
84. {
85. to\_delete = nullptr;
86. break;
87. }// false case, element doesn't exist
88. parent\_of\_to\_delete = to\_delete;
89. to\_delete = to\_delete->get\_left\_child();
90. }
91. else
92. { // search in the right branch
93. if (to\_delete->get\_right\_child() == nullptr)
94. {
95. to\_delete = nullptr;
96. break;
97. }// false case, element doesn't exist
98. parent\_of\_to\_delete = to\_delete;
99. to\_delete = to\_delete->get\_right\_child();
100. }
101. }
102. if (to\_delete == nullptr)
103. throw out\_of\_range("remove not-existing element");
104. if (to\_delete->get\_left\_child() == nullptr)
105. {
106. if (to\_delete->get\_right\_child() == nullptr)
107. {//case 1: a leaf
108. if (to\_delete == root)
109. { // root is the only element in the tree
110. delete root;
111. root = nullptr;
112. return;
113. }
114. if (parent\_of\_to\_delete->left\_child == to\_delete)
115. {
116. delete parent\_of\_to\_delete->left\_child;
117. parent\_of\_to\_delete->left\_child = nullptr;
118. return;
119. }
120. if (parent\_of\_to\_delete->right\_child == to\_delete)
121. {
122. delete parent\_of\_to\_delete->right\_child;
123. parent\_of\_to\_delete->right\_child = nullptr;
124. return;
125. }
126. }
127. else
128. {// case 2: to\_delete has only right child
129. if (to\_delete == root)
130. root = to\_delete->right\_child;
131. else if (parent\_of\_to\_delete->left\_child == to\_delete)
132. parent\_of\_to\_delete->left\_child = to\_delete->right\_child;
133. else if (parent\_of\_to\_delete->right\_child == to\_delete)
134. parent\_of\_to\_delete->right\_child = to\_delete->right\_child;
135. to\_delete = nullptr;
136. return;
137. }
138. }
139. else if (to\_delete->get\_right\_child() == nullptr)
140. { // case 3: to\_delete has only left child
141. if (to\_delete == root)
142. root = to\_delete->left\_child;
143. else if (parent\_of\_to\_delete->left\_child == to\_delete)
144. parent\_of\_to\_delete->left\_child = to\_delete->left\_child;
145. else if (parent\_of\_to\_delete->right\_child == to\_delete)
146. parent\_of\_to\_delete->right\_child = to\_delete->left\_child;
147. to\_delete = nullptr;
148. return;
149. }
150. else
151. {// case 4: to\_delete has both childs
152. Node\* change\_leaf = to\_delete->right\_child;
153. Node\* change\_leaf\_parent = to\_delete;
154. while (change\_leaf->left\_child != nullptr)
155. { // go to the left leaf
156. change\_leaf\_parent = change\_leaf;
157. change\_leaf = change\_leaf->left\_child;
158. }
159. if (to\_delete == root)
160. root->key = change\_leaf->key;
161. else
162. to\_delete->key = change\_leaf->key;
163. if (change\_leaf != to\_delete->right\_child)
164. {
165. delete change\_leaf\_parent->left\_child;
166. change\_leaf\_parent->left\_child = nullptr;
167. }
168. else
169. {
170. delete change\_leaf\_parent->right\_child;
171. change\_leaf\_parent->right\_child = nullptr;
172. }
173. }
174. }
175. BinarySearchTree::~BinarySearchTree()
176. {
177. while (root != nullptr)
178. remove(root->get\_key());
179. }
180. Iterator\* BinarySearchTree::create\_breadth\_first\_traverse\_iterator()
181. { return new BreadthFirstTraverse\_Iterator(root); }
182. BinarySearchTree::BreadthFirstTraverse\_Iterator::BreadthFirstTraverse\_Iterator(Node\* start)
183. {
184. current = start;
185. queue.push(current);
186. }
187. bool BinarySearchTree::BreadthFirstTraverse\_Iterator::has\_next()
188. { return (!queue.is\_empty()); }
189. Node\* BinarySearchTree::BreadthFirstTraverse\_Iterator::next()
190. {
191. if (queue.is\_empty())
192. return current = nullptr;
193. current = queue.top()->get\_data();
194. if (current->get\_left\_child() != nullptr)
195. queue.push(current->get\_left\_child());
196. if (current->get\_right\_child() != nullptr)
197. queue.push(current->get\_right\_child());
198. return queue.pop()->get\_data();
199. }
200. Iterator\* BinarySearchTree::create\_pre\_ordered\_depth\_first\_traverse\_iterator()
201. { return new PreOrderedDepthFirstTraverse\_Iterator(root); }
202. BinarySearchTree::PreOrderedDepthFirstTraverse\_Iterator::PreOrderedDepthFirstTraverse\_Iterator(Node\* start)
203. {
204. current = start;
205. stack.push(current);
206. }
207. bool BinarySearchTree::PreOrderedDepthFirstTraverse\_Iterator::has\_next()
208. { return (!stack.is\_empty()); }
209. Node\* BinarySearchTree::PreOrderedDepthFirstTraverse\_Iterator::next()
210. {
211. if (stack.is\_empty())
212. return current = nullptr;
213. Node\* to\_return = new Node;
214. to\_return = current;
215. if (current->get\_right\_child() != nullptr)
216. stack.push(current->get\_right\_child());
217. if (current->get\_left\_child() != nullptr)
218. current = current->get\_left\_child();
219. else if (!stack.is\_empty())
220. current = stack.pop()->get\_data();
221. return to\_return;
222. }
223. Iterator\* BinarySearchTree::create\_in\_ordered\_depth\_first\_traverse\_iterator()
224. { return new InOrderedDepthFirstTraverse\_Iterator(root); }
225. BinarySearchTree::InOrderedDepthFirstTraverse\_Iterator::InOrderedDepthFirstTraverse\_Iterator(Node\* start)
226. { current = start; }
227. bool BinarySearchTree::InOrderedDepthFirstTraverse\_Iterator::has\_next()
228. { return (current != nullptr); }
229. Node\* BinarySearchTree::InOrderedDepthFirstTraverse\_Iterator::next()
230. {
231. Node\* to\_return = new Node;
232. to\_return = nullptr;
233. if (!stack.is\_empty())
234. {
235. if (stack.peek()->get\_data() != current) // else we skip getting down procedure
236. {
237. stack.push(current);
238. while (current->get\_left\_child() != nullptr)
239. {
240. current = current->get\_left\_child();
241. stack.push(current);
242. }
243. }
244. to\_return = stack.pop()->get\_data();
245. }
246. else if (current != nullptr) // when reach the root stack is empty but we have right branch
247. {
248. stack.push(current);
249. while (current->get\_left\_child() != nullptr)
250. {
251. current = current->get\_left\_child();
252. stack.push(current);
253. }
254. to\_return = stack.pop()->get\_data();
255. }
256. else
257. return nullptr;
258. if (current->get\_right\_child() != nullptr)
259. current = current->get\_right\_child(); // the next is right-child node
260. else if (!stack.is\_empty())
261. current = stack.peek()->get\_data(); // the next is the top from the stack
262. else
263. current = nullptr;
264. return to\_return;
265. }
266. Iterator\* BinarySearchTree::create\_post\_ordered\_depth\_first\_traverse\_iterator()
267. { return new PostOrderedDepthFirstTraverse\_Iterator(root); }
268. BinarySearchTree::PostOrderedDepthFirstTraverse\_Iterator::PostOrderedDepthFirstTraverse\_Iterator(Node\* start)
269. {
270. current = start;
271. stack.push(current);
272. while ((current->get\_left\_child() != nullptr) || (current->get\_right\_child() != nullptr))
273. {
274. if (current->get\_left\_child() != nullptr)
275. {
276. if (current->get\_right\_child() != nullptr)
277. {
278. stack.push(current->get\_right\_child());
279. stack.push(current);
280. }
281. current = current->get\_left\_child();
282. stack.push(current);
283. }
284. else if (current->get\_right\_child() != nullptr)
285. {
286. current = current->get\_right\_child();
287. if (((current->get\_left\_child() != nullptr) && (current->get\_right\_child() == nullptr))|| ((current->get\_right\_child() != nullptr) && (current->get\_left\_child() == nullptr)))
288. stack.push(current); // if has only one child
289. }
290. }
291. }
292. bool BinarySearchTree::PostOrderedDepthFirstTraverse\_Iterator::has\_next()
293. { return (current != nullptr); }
294. Node\* BinarySearchTree::PostOrderedDepthFirstTraverse\_Iterator::next()
295. {
296. Node\* to\_return = new Node;
297. to\_return = nullptr;
298. if (!stack.is\_empty())
299. {
300. if ((current->get\_right\_child() == stack.peek()->get\_data()) || (current->get\_left\_child() == stack.peek()->get\_data()))
301. { // if current has childs
302. current = stack.peek()->get\_data(); // take the one that wasn't considered
303. while ((current->get\_left\_child() != nullptr)||(current->get\_right\_child() != nullptr))
304. {
305. if (current->get\_left\_child() != nullptr)
306. { // if left exists add right and go to left
307. if (current->get\_right\_child() != nullptr)
308. {
309. stack.push(current->get\_right\_child());
310. stack.push(current);
311. }
312. current = current->get\_left\_child();
313. stack.push(current);
314. } // if left doesn't exist, but right exist
315. else if (current->get\_right\_child() != nullptr)
316. {
317. current = current->get\_right\_child();
318. if (((current->get\_left\_child() != nullptr) && (current->get\_right\_child() == nullptr)) || ((current->get\_right\_child() != nullptr) && (current->get\_left\_child() == nullptr)))
319. stack.push(current);
320. }
321. }
322. to\_return = stack.pop()->get\_data();
323. }
324. else
325. {
326. if (current == stack.peek()->get\_data())
327. to\_return = stack.pop()->get\_data();
328. else
329. to\_return = current;
330. }
331. }
332. else if (current != nullptr)
333. {
334. to\_return = current;
335. current = nullptr;
336. }
337. if (!stack.is\_empty())
338. current = stack.pop()->get\_data(); // the next is the top from the stack
339. return to\_return;
340. }
341. int main()
342. {
343. BinarySearchTree BST(8);
344. BST.insert(3); BST.insert(10); BST.insert(14); BST.insert(13);
345. BST.insert(6); BST.insert(1); BST.insert(7); BST.insert(4);
346. Iterator\* post\_dft\_iterator = BST.create\_post\_ordered\_depth\_first\_traverse\_iterator();
347. Iterator\* pre\_dft\_iterator = BST.create\_pre\_ordered\_depth\_first\_traverse\_iterator();
348. Iterator\* in\_dft\_iterator = BST.create\_in\_ordered\_depth\_first\_traverse\_iterator();
349. Iterator\* bft\_iterator = BST.create\_breadth\_first\_traverse\_iterator();
350. cout << "Tree, passed by breadth-first iterator: " << endl;
351. while (bft\_iterator->has\_next())
352. cout << bft\_iterator->next()->get\_key() << " ";
353. cout << endl << "Tree, passed by pre-ordered depth-first iterator: " << endl;
354. while (pre\_dft\_iterator->has\_next())
355. cout << pre\_dft\_iterator->next()->get\_key() << " ";
356. cout << endl << "Tree, passed by in-ordered depth-first iterator: " << endl;
357. while (in\_dft\_iterator->has\_next())
358. cout << in\_dft\_iterator->next()->get\_key() << " ";
359. cout << endl << "Tree, passed by post-ordered depth-first iterator: " << endl;
360. while (post\_dft\_iterator->has\_next())
361. cout << post\_dft\_iterator->next()->get\_key() << " ";
362. cout << endl;
363. return 0;
364. }

UnitTestForQueueAndStack.cpp:

1. #include "pch.h"
2. #include "CppUnitTest.h"
3. #include "/Users/varka/source/repos/BinarySearchTree/QueueAndStack.h"
4. using namespace Microsoft::VisualStudio::CppUnitTestFramework;
5. namespace UnitTestForQueueAndStack
6. {
7. TEST\_CLASS(UnitTestForQueueAndStack)
8. {
9. public:
10. TEST\_METHOD(TestEmptyQueue)
11. {
12. Queue queue;
13. Assert::AreEqual(queue.is\_empty(), true);
14. }
15. TEST\_METHOD(TestEmptyStack)
16. {
17. Stack stack;
18. Assert::AreEqual(stack.is\_empty(), true);
19. }
20. TEST\_METHOD(TestEmptyQueuePop)
21. {
22. Queue queue;
23. try
24. {
25. queue.pop();
26. }
27. catch (const exception & message)
28. {
29. Assert::AreEqual(message.what(), "the queue is empty");
30. }
31. }
32. TEST\_METHOD(TestEmptyStackPop)
33. {
34. Stack stack;
35. try
36. {
37. stack.pop();
38. }
39. catch (const exception & message)
40. {
41. Assert::AreEqual(message.what(), "the stack is empty");
42. }
43. }
44. TEST\_METHOD(TestEmptyQueueSize)
45. {
46. Queue queue;
47. Assert::AreEqual(queue.size(), 0);
48. }
49. TEST\_METHOD(TestEmptyStackSize)
50. {
51. Stack stack;
52. Assert::AreEqual(stack.size(), 0);
53. }
54. TEST\_METHOD(TestOneElementQueueIsEmpty)
55. {
56. Queue queue;
57. Node new\_node(3);
58. queue.push(&new\_node);
59. Assert::AreEqual(queue.is\_empty(), false);
60. }
61. TEST\_METHOD(TestOneElementQueue)
62. {
63. Queue queue;
64. Node new\_node(3);
65. queue.push(&new\_node);
66. Assert::AreEqual(queue.top()->get\_data()->get\_key(), 3);
67. }
68. TEST\_METHOD(TestOneElementStack)
69. {
70. Stack stack;
71. Node new\_node(3);
72. stack.push(&new\_node);
73. Assert::AreEqual(stack.is\_empty(), false);
74. }
75. TEST\_METHOD(TestOneElementQueueSize)
76. {
77. Queue queue;
78. Node new\_node(3);
79. queue.push(&new\_node);
80. Assert::AreEqual(queue.size(), 1);
81. }
82. TEST\_METHOD(TestOneElementStackSize)
83. {
84. Stack stack;
85. Node new\_node(3);
86. stack.push(&new\_node);
87. Assert::AreEqual(stack.size(), 1);
88. }
89. TEST\_METHOD(TestOneElementQueuePop)
90. {
91. Queue queue;
92. Node new\_node(3);
93. queue.push(&new\_node);
94. Assert::AreEqual(queue.pop()->get\_data()->get\_key(), 3);
95. }
96. TEST\_METHOD(TestOneElementStackPop)
97. {
98. Stack stack;
99. Node new\_node(3);
100. stack.push(&new\_node);
101. Assert::AreEqual(stack.pop()->get\_data()->get\_key(), 3);
102. }
103. TEST\_METHOD(TestTwoElementQueuePop)
104. {
105. Queue queue;
106. Node new\_node(3);
107. queue.push(&new\_node);
108. Node new\_node2(4);
109. queue.push(&new\_node2);
110. Assert::AreEqual(queue.pop()->get\_data()->get\_key(), 3);
111. Assert::AreEqual(queue.pop()->get\_data()->get\_key(), 4);
112. }
113. TEST\_METHOD(TestTwoElementStackPop)
114. {
115. Stack stack;
116. Node new\_node(3);
117. stack.push(&new\_node);
118. Node new\_node2(4);
119. stack.push(&new\_node2);
120. Assert::AreEqual(stack.pop()->get\_data()->get\_key(), 4);
121. Assert::AreEqual(stack.pop()->get\_data()->get\_key(), 3);
122. }
123. TEST\_METHOD(TestTwoElementQueueSize)
124. {
125. Queue queue;
126. Node new\_node(3);
127. queue.push(&new\_node);
128. Node new\_node2(1);
129. queue.push(&new\_node2);
130. Assert::AreEqual(queue.size(), 2);
131. }
132. TEST\_METHOD(TestTwoElementStackSize)
133. {
134. Stack stack;
135. Node new\_node(3);
136. stack.push(&new\_node);
137. Node new\_node2(1);
138. stack.push(&new\_node2);
139. Assert::AreEqual(stack.size(), 2);
140. }
141. TEST\_METHOD(TestOneElementQueueTop)
142. {
143. Queue queue;
144. Node new\_node(3);
145. queue.push(&new\_node);
146. Node new\_node2(5);
147. queue.push(&new\_node2);
148. Node new\_node3(7);
149. queue.push(&new\_node3);
150. Node new\_node4(1);
151. queue.push(&new\_node4);
152. Assert::AreEqual(queue.top()->get\_data()->get\_key(), 3);
153. }
154. TEST\_METHOD(TestOneElementStackPeek)
155. {
156. Stack stack;
157. Node new\_node(3);
158. stack.push(&new\_node);
159. Node new\_node2(5);
160. stack.push(&new\_node2);
161. Node new\_node3(7);
162. stack.push(&new\_node3);
163. Node new\_node4(1);
164. stack.push(&new\_node4);
165. Assert::AreEqual(stack.peek()->get\_data()->get\_key(), 1);
166. }
167. TEST\_METHOD(TestOneElementQueueBack)
168. {
169. Queue queue;
170. Node new\_node(3);
171. queue.push(&new\_node);
172. Node new\_node2(5);
173. queue.push(&new\_node2);
174. Node new\_node3(7);
175. queue.push(&new\_node3);
176. Node new\_node4(1);
177. queue.push(&new\_node4);
178. Assert::AreEqual(queue.back()->get\_data()->get\_key(), 1);
179. }
180. };
181. }

UnitTestsForBinarySearchTree.cpp:

1. #include "pch.h"
2. #include "CppUnitTest.h"
3. #include "/Users/varka/source/repos/BinarySearchTree/BinarySearchTree.h"
4. using namespace Microsoft::VisualStudio::CppUnitTestFramework;
5. namespace UnitTestsForBinarySearchTree
6. {
7. TEST\_CLASS(UnitTestsForBinarySearchTree)
8. {
9. public:
10. TEST\_METHOD(TestBinarySearchTreeEmpty)
11. {
12. BinarySearchTree BST;
13. Assert::AreEqual(BST.is\_empty(), true);
14. }
15. TEST\_METHOD(TestBinarySearchTreeContainsError)
16. {
17. BinarySearchTree BST;
18. try
19. {
20. BST.contains(2);
21. }
22. catch (const exception & message)
23. {
24. Assert::AreEqual(message.what(), "search in the empty tree");
25. }
26. }
27. TEST\_METHOD(TestBinarySearchTreeRemoveFromEmptyTree)
28. {
29. BinarySearchTree BST(2);
30. BST.remove(2);
31. try
32. {
33. BST.remove(2);
34. }
35. catch (const exception & message)
36. {
37. Assert::AreEqual(message.what(), "remove from empty tree");
38. }
39. }
40. TEST\_METHOD(TestBinarySearchTreeRemoveError)
41. {
42. BinarySearchTree BST(8);
43. BST.insert(2), BST.insert(3);
44. BST.remove(2);
45. try
46. {
47. BST.contains(2);
48. }
49. catch (const exception & message)
50. {
51. Assert::AreEqual(message.what(), "remove not-existing element");
52. }
53. }
54. TEST\_METHOD(TestBinarySearchTreeNotEmpty)
55. {
56. BinarySearchTree BST(1);
57. Assert::AreEqual(BST.is\_empty(), false);
58. }
59. TEST\_METHOD(TestBinarySearchTreeIntConstructor)
60. {
61. BinarySearchTree BST(1);
62. Assert::AreEqual(BST.contains(1), true);
63. BinarySearchTree BST1(1);
64. BinarySearchTree BST2;
65. BST2.insert(1);
66. Assert::AreEqual(BST1.contains(1), BST2.contains(1));
67. }
68. TEST\_METHOD(TestBinarySearchForExistingElement)
69. {
70. BinarySearchTree BST(8);
71. BST.insert(3); BST.insert(10);
72. Assert::AreEqual(BST.contains(8), true);
73. }
74. TEST\_METHOD(TestBinarySearchForNotExistingElement)
75. {
76. BinarySearchTree BST(8);
77. BST.insert(3); BST.insert(10);
78. Assert::AreEqual(BST.contains(4), false);
79. }
80. TEST\_METHOD(TestBinarySearchBFTIteratorAndInsert)
81. {
82. BinarySearchTree BST(8);
83. BST.insert(3); BST.insert(10); BST.insert(14); BST.insert(13);
84. BST.insert(6); BST.insert(1); BST.insert(7); BST.insert(4);
85. Iterator\* iterator = BST.create\_breadth\_first\_traverse\_iterator();
86. if (iterator->has\_next())
87. Assert::AreEqual(iterator->next()->get\_key(), 8);
88. if (iterator->has\_next())
89. Assert::AreEqual(iterator->next()->get\_key(), 3);
90. if (iterator->has\_next())
91. Assert::AreEqual(iterator->next()->get\_key(), 10);
92. if (iterator->has\_next())
93. Assert::AreEqual(iterator->next()->get\_key(), 1);
94. if (iterator->has\_next())
95. Assert::AreEqual(iterator->next()->get\_key(), 6);
96. if (iterator->has\_next())
97. Assert::AreEqual(iterator->next()->get\_key(), 14);
98. if (iterator->has\_next())
99. Assert::AreEqual(iterator->next()->get\_key(), 4);
100. if (iterator->has\_next())
101. Assert::AreEqual(iterator->next()->get\_key(), 7);
102. if (iterator->has\_next())
103. Assert::AreEqual(iterator->next()->get\_key(), 13);
104. Assert::AreEqual(iterator->has\_next(), false);
105. }
106. TEST\_METHOD(TestBinarySearchPreOrderDFTIteratorAndInsert)
107. {
108. BinarySearchTree BST(8);
109. BST.insert(3); BST.insert(10); BST.insert(14); BST.insert(13);
110. BST.insert(6); BST.insert(1); BST.insert(7); BST.insert(4);
111. Iterator\* iterator = BST.create\_pre\_ordered\_depth\_first\_traverse\_iterator();
112. if (iterator->has\_next())
113. Assert::AreEqual(iterator->next()->get\_key(), 8);
114. if (iterator->has\_next())
115. Assert::AreEqual(iterator->next()->get\_key(), 3);
116. if (iterator->has\_next())
117. Assert::AreEqual(iterator->next()->get\_key(), 1);
118. if (iterator->has\_next())
119. Assert::AreEqual(iterator->next()->get\_key(), 6);
120. if (iterator->has\_next())
121. Assert::AreEqual(iterator->next()->get\_key(), 4);
122. if (iterator->has\_next())
123. Assert::AreEqual(iterator->next()->get\_key(), 7);
124. if (iterator->has\_next())
125. Assert::AreEqual(iterator->next()->get\_key(), 10);
126. if (iterator->has\_next())
127. Assert::AreEqual(iterator->next()->get\_key(), 14);
128. if (iterator->has\_next())
129. Assert::AreEqual(iterator->next()->get\_key(), 13);
130. Assert::AreEqual(iterator->has\_next(), false);
131. }
132. TEST\_METHOD(TestBinarySearchInOrderDFTIteratorAndInsert)
133. {
134. BinarySearchTree BST(8);
135. BST.insert(3); BST.insert(10); BST.insert(14); BST.insert(13);
136. BST.insert(6); BST.insert(1); BST.insert(7); BST.insert(4);
137. Iterator\* iterator = BST.create\_in\_ordered\_depth\_first\_traverse\_iterator();
138. if (iterator->has\_next())
139. Assert::AreEqual(iterator->next()->get\_key(), 1);
140. if (iterator->has\_next())
141. Assert::AreEqual(iterator->next()->get\_key(), 3);
142. if (iterator->has\_next())
143. Assert::AreEqual(iterator->next()->get\_key(), 4);
144. if (iterator->has\_next())
145. Assert::AreEqual(iterator->next()->get\_key(), 6);
146. if (iterator->has\_next())
147. Assert::AreEqual(iterator->next()->get\_key(), 7);
148. if (iterator->has\_next())
149. Assert::AreEqual(iterator->next()->get\_key(), 8);
150. if (iterator->has\_next())
151. Assert::AreEqual(iterator->next()->get\_key(), 10);
152. if (iterator->has\_next())
153. Assert::AreEqual(iterator->next()->get\_key(), 13);
154. if (iterator->has\_next())
155. Assert::AreEqual(iterator->next()->get\_key(), 14);
156. Assert::AreEqual(iterator->has\_next(), false);
157. }
158. TEST\_METHOD(TestBinarySearchPostOrderDFTIteratorAndInsert)
159. {
160. BinarySearchTree BST(8);
161. BST.insert(3); BST.insert(10); BST.insert(14); BST.insert(13);
162. BST.insert(6); BST.insert(1); BST.insert(7); BST.insert(4);
163. Iterator\* iterator = BST.create\_post\_ordered\_depth\_first\_traverse\_iterator();
164. if (iterator->has\_next())
165. Assert::AreEqual(iterator->next()->get\_key(), 1);
166. if (iterator->has\_next())
167. Assert::AreEqual(iterator->next()->get\_key(), 4);
168. if (iterator->has\_next())
169. Assert::AreEqual(iterator->next()->get\_key(), 7);
170. if (iterator->has\_next())
171. Assert::AreEqual(iterator->next()->get\_key(), 6);
172. if (iterator->has\_next())
173. Assert::AreEqual(iterator->next()->get\_key(), 3);
174. if (iterator->has\_next())
175. Assert::AreEqual(iterator->next()->get\_key(), 13);
176. if (iterator->has\_next())
177. Assert::AreEqual(iterator->next()->get\_key(), 14);
178. if (iterator->has\_next())
179. Assert::AreEqual(iterator->next()->get\_key(), 10);
180. if (iterator->has\_next())
181. Assert::AreEqual(iterator->next()->get\_key(), 8);
182. Assert::AreEqual(iterator->has\_next(), false);
183. }
184. TEST\_METHOD(TestBinarySearchRemoveOneElement)
185. {
186. BinarySearchTree BST(8);
187. BST.remove(8);
188. Assert::AreEqual(BST.is\_empty(), true);
189. }
190. TEST\_METHOD(TestBinarySearchRemoveOneElementFromThreeBFTIterator)
191. {
192. BinarySearchTree BST(8);
193. BST.insert(14); BST.insert(13);
194. BST.remove(14);
195. Iterator\* iterator = BST.create\_breadth\_first\_traverse\_iterator();
196. if (iterator->has\_next())
197. Assert::AreEqual(iterator->next()->get\_key(), 8);
198. if (iterator->has\_next())
199. Assert::AreEqual(iterator->next()->get\_key(), 13);
200. Assert::AreEqual(iterator->has\_next(), false);
201. }
202. TEST\_METHOD(TestBinarySearchThreeBFTIterator)
203. {
204. BinarySearchTree BST(8);
205. BST.insert(13); BST.insert(4);
206. Iterator\* iterator = BST.create\_breadth\_first\_traverse\_iterator();
207. if (iterator->has\_next())
208. Assert::AreEqual(iterator->next()->get\_key(), 8);
209. if (iterator->has\_next())
210. Assert::AreEqual(iterator->next()->get\_key(), 4);
211. if (iterator->has\_next())
212. Assert::AreEqual(iterator->next()->get\_key(), 13);
213. Assert::AreEqual(iterator->has\_next(), false);
214. }
215. TEST\_METHOD(TestBinarySearchThreeBFTIteratorLeft)
216. {
217. BinarySearchTree BST(8);
218. BST.insert(13); BST.insert(14);
219. Iterator\* iterator = BST.create\_breadth\_first\_traverse\_iterator();
220. if (iterator->has\_next())
221. Assert::AreEqual(iterator->next()->get\_key(), 8);
222. if (iterator->has\_next())
223. Assert::AreEqual(iterator->next()->get\_key(), 13);
224. if (iterator->has\_next())
225. Assert::AreEqual(iterator->next()->get\_key(), 14);
226. Assert::AreEqual(iterator->has\_next(), false);
227. }
228. TEST\_METHOD(TestBinarySearchThreeBFTIteratorRight)
229. {
230. BinarySearchTree BST(8);
231. BST.insert(6); BST.insert(3);
232. Iterator\* iterator = BST.create\_breadth\_first\_traverse\_iterator();
233. if (iterator->has\_next())
234. Assert::AreEqual(iterator->next()->get\_key(), 8);
235. if (iterator->has\_next())
236. Assert::AreEqual(iterator->next()->get\_key(), 6);
237. if (iterator->has\_next())
238. Assert::AreEqual(iterator->next()->get\_key(), 3);
239. Assert::AreEqual(iterator->has\_next(), false);
240. }
241. TEST\_METHOD(TestBinarySearchThreeBFTIteratorLeftRight)
242. {
243. BinarySearchTree BST(8);
244. BST.insert(5); BST.insert(7);
245. Iterator\* iterator = BST.create\_breadth\_first\_traverse\_iterator();
246. if (iterator->has\_next())
247. Assert::AreEqual(iterator->next()->get\_key(), 8);
248. if (iterator->has\_next())
249. Assert::AreEqual(iterator->next()->get\_key(), 5);
250. if (iterator->has\_next())
251. Assert::AreEqual(iterator->next()->get\_key(), 7);
252. Assert::AreEqual(iterator->has\_next(), false);
253. }
254. TEST\_METHOD(TestBinarySearchThreeBFTIteratorRightLeft)
255. {
256. BinarySearchTree BST(8);
257. BST.insert(10); BST.insert(9);
258. Iterator\* iterator = BST.create\_breadth\_first\_traverse\_iterator();
259. if (iterator->has\_next())
260. Assert::AreEqual(iterator->next()->get\_key(), 8);
261. if (iterator->has\_next())
262. Assert::AreEqual(iterator->next()->get\_key(), 10);
263. if (iterator->has\_next())
264. Assert::AreEqual(iterator->next()->get\_key(), 9);
265. Assert::AreEqual(iterator->has\_next(), false);
266. }
267. TEST\_METHOD(TestBinarySearchThreePreOrderedDFTIterator)
268. {
269. BinarySearchTree BST(8);
270. BST.insert(13); BST.insert(4);
271. Iterator\* iterator = BST.create\_pre\_ordered\_depth\_first\_traverse\_iterator();
272. if (iterator->has\_next())
273. Assert::AreEqual(iterator->next()->get\_key(), 8);
274. if (iterator->has\_next())
275. Assert::AreEqual(iterator->next()->get\_key(), 4);
276. if (iterator->has\_next())
277. Assert::AreEqual(iterator->next()->get\_key(), 13);
278. Assert::AreEqual(iterator->has\_next(), false);
279. }
280. TEST\_METHOD(TestBinarySearchThreePreOrderedDFTIteratorLeft)
281. {
282. BinarySearchTree BST(8);
283. BST.insert(13); BST.insert(14);
284. Iterator\* iterator = BST.create\_pre\_ordered\_depth\_first\_traverse\_iterator();
285. if (iterator->has\_next())
286. Assert::AreEqual(iterator->next()->get\_key(), 8);
287. if (iterator->has\_next())
288. Assert::AreEqual(iterator->next()->get\_key(), 13);
289. if (iterator->has\_next())
290. Assert::AreEqual(iterator->next()->get\_key(), 14);
291. Assert::AreEqual(iterator->has\_next(), false);
292. }
293. TEST\_METHOD(TestBinarySearchThreePreOrderedDFTIteratorRight)
294. {
295. BinarySearchTree BST(8);
296. BST.insert(6); BST.insert(3);
297. Iterator\* iterator = BST.create\_pre\_ordered\_depth\_first\_traverse\_iterator();
298. if (iterator->has\_next())
299. Assert::AreEqual(iterator->next()->get\_key(), 8);
300. if (iterator->has\_next())
301. Assert::AreEqual(iterator->next()->get\_key(), 6);
302. if (iterator->has\_next())
303. Assert::AreEqual(iterator->next()->get\_key(), 3);
304. Assert::AreEqual(iterator->has\_next(), false);
305. }
306. TEST\_METHOD(TestBinarySearchThreePreOrderedDFTIteratorLeftRight)
307. {
308. BinarySearchTree BST(8);
309. BST.insert(5); BST.insert(7);
310. Iterator\* iterator = BST.create\_pre\_ordered\_depth\_first\_traverse\_iterator();
311. if (iterator->has\_next())
312. Assert::AreEqual(iterator->next()->get\_key(), 8);
313. if (iterator->has\_next())
314. Assert::AreEqual(iterator->next()->get\_key(), 5);
315. if (iterator->has\_next())
316. Assert::AreEqual(iterator->next()->get\_key(), 7);
317. Assert::AreEqual(iterator->has\_next(), false);
318. }
319. TEST\_METHOD(TestBinarySearchThreePreOrderedDFTIteratorRightLeft)
320. {
321. BinarySearchTree BST(8);
322. BST.insert(10); BST.insert(9);
323. Iterator\* iterator = BST.create\_pre\_ordered\_depth\_first\_traverse\_iterator();
324. if (iterator->has\_next())
325. Assert::AreEqual(iterator->next()->get\_key(), 8);
326. if (iterator->has\_next())
327. Assert::AreEqual(iterator->next()->get\_key(), 10);
328. if (iterator->has\_next())
329. Assert::AreEqual(iterator->next()->get\_key(), 9);
330. Assert::AreEqual(iterator->has\_next(), false);
331. }
332. TEST\_METHOD(TestBinarySearchThreeInOrderedDFTIterator)
333. {
334. BinarySearchTree BST(8);
335. BST.insert(13); BST.insert(4);
336. Iterator\* iterator = BST.create\_in\_ordered\_depth\_first\_traverse\_iterator();
337. if (iterator->has\_next())
338. Assert::AreEqual(iterator->next()->get\_key(), 4);
339. if (iterator->has\_next())
340. Assert::AreEqual(iterator->next()->get\_key(), 8);
341. if (iterator->has\_next())
342. Assert::AreEqual(iterator->next()->get\_key(), 13);
343. Assert::AreEqual(iterator->has\_next(), false);
344. }
345. TEST\_METHOD(TestBinarySearchThreeInOrderedDFTIteratorLeft)
346. {
347. BinarySearchTree BST(8);
348. BST.insert(13); BST.insert(14);
349. Iterator\* iterator = BST.create\_in\_ordered\_depth\_first\_traverse\_iterator();
350. if (iterator->has\_next())
351. Assert::AreEqual(iterator->next()->get\_key(), 8);
352. if (iterator->has\_next())
353. Assert::AreEqual(iterator->next()->get\_key(), 13);
354. if (iterator->has\_next())
355. Assert::AreEqual(iterator->next()->get\_key(), 14);
356. Assert::AreEqual(iterator->has\_next(), false);
357. }
358. TEST\_METHOD(TestBinarySearchThreeInOrderedDFTIteratorRight)
359. {
360. BinarySearchTree BST(8);
361. BST.insert(6); BST.insert(3);
362. Iterator\* iterator = BST.create\_in\_ordered\_depth\_first\_traverse\_iterator();
363. if (iterator->has\_next())
364. Assert::AreEqual(iterator->next()->get\_key(), 3);
365. if (iterator->has\_next())
366. Assert::AreEqual(iterator->next()->get\_key(), 6);
367. if (iterator->has\_next())
368. Assert::AreEqual(iterator->next()->get\_key(), 8);
369. Assert::AreEqual(iterator->has\_next(), false);
370. }
371. TEST\_METHOD(TestBinarySearchThreeInOrderedDFTIteratorLeftRight)
372. {
373. BinarySearchTree BST(8);
374. BST.insert(5); BST.insert(7);
375. Iterator\* iterator = BST.create\_in\_ordered\_depth\_first\_traverse\_iterator();
376. if (iterator->has\_next())
377. Assert::AreEqual(iterator->next()->get\_key(), 5);
378. if (iterator->has\_next())
379. Assert::AreEqual(iterator->next()->get\_key(), 7);
380. if (iterator->has\_next())
381. Assert::AreEqual(iterator->next()->get\_key(), 8);
382. Assert::AreEqual(iterator->has\_next(), false);
383. }
384. TEST\_METHOD(TestBinarySearchThreeInOrderedDFTIteratorRightLeft)
385. {
386. BinarySearchTree BST(8);
387. BST.insert(10); BST.insert(9);
388. Iterator\* iterator = BST.create\_in\_ordered\_depth\_first\_traverse\_iterator();
389. if (iterator->has\_next())
390. Assert::AreEqual(iterator->next()->get\_key(), 8);
391. if (iterator->has\_next())
392. Assert::AreEqual(iterator->next()->get\_key(), 9);
393. if (iterator->has\_next())
394. Assert::AreEqual(iterator->next()->get\_key(), 10);
395. Assert::AreEqual(iterator->has\_next(), false);
396. }
397. TEST\_METHOD(TestBinarySearchThreePostOrderedDFTIterator)
398. {
399. BinarySearchTree BST(8);
400. BST.insert(13); BST.insert(4);
401. Iterator\* iterator = BST.create\_post\_ordered\_depth\_first\_traverse\_iterator();
402. if (iterator->has\_next())
403. Assert::AreEqual(iterator->next()->get\_key(), 4);
404. if (iterator->has\_next())
405. Assert::AreEqual(iterator->next()->get\_key(), 13);
406. if (iterator->has\_next())
407. Assert::AreEqual(iterator->next()->get\_key(), 8);
408. Assert::AreEqual(iterator->has\_next(), false);
409. }
410. TEST\_METHOD(TestBinarySearchThreePostOrderedDFTIteratorLeft)
411. {
412. BinarySearchTree BST(8);
413. BST.insert(13); BST.insert(14);
414. Iterator\* iterator = BST.create\_post\_ordered\_depth\_first\_traverse\_iterator();
415. if (iterator->has\_next())
416. Assert::AreEqual(iterator->next()->get\_key(), 14);
417. if (iterator->has\_next())
418. Assert::AreEqual(iterator->next()->get\_key(), 13);
419. if (iterator->has\_next())
420. Assert::AreEqual(iterator->next()->get\_key(), 8);
421. Assert::AreEqual(iterator->has\_next(), false);
422. }
423. TEST\_METHOD(TestBinarySearchThreePostOrderedDFTIteratorRight)
424. {
425. BinarySearchTree BST(8);
426. BST.insert(6); BST.insert(3);
427. Iterator\* iterator = BST.create\_post\_ordered\_depth\_first\_traverse\_iterator();
428. if (iterator->has\_next())
429. Assert::AreEqual(iterator->next()->get\_key(), 3);
430. if (iterator->has\_next())
431. Assert::AreEqual(iterator->next()->get\_key(), 6);
432. if (iterator->has\_next())
433. Assert::AreEqual(iterator->next()->get\_key(), 8);
434. Assert::AreEqual(iterator->has\_next(), false);
435. }
436. TEST\_METHOD(TestBinarySearchThreePostOrderedDFTIteratorLeftRight)
437. {
438. BinarySearchTree BST(8);
439. BST.insert(5); BST.insert(7);
440. Iterator\* iterator = BST.create\_post\_ordered\_depth\_first\_traverse\_iterator();
441. if (iterator->has\_next())
442. Assert::AreEqual(iterator->next()->get\_key(), 7);
443. if (iterator->has\_next())
444. Assert::AreEqual(iterator->next()->get\_key(), 5);
445. if (iterator->has\_next())
446. Assert::AreEqual(iterator->next()->get\_key(), 8);
447. Assert::AreEqual(iterator->has\_next(), false);
448. }
449. TEST\_METHOD(TestBinarySearchThreePostOrderedDFTIteratorRightLeft)
450. {
451. BinarySearchTree BST(8);
452. BST.insert(10); BST.insert(9);
453. Iterator\* iterator = BST.create\_post\_ordered\_depth\_first\_traverse\_iterator();
454. if (iterator->has\_next())
455. Assert::AreEqual(iterator->next()->get\_key(), 9);
456. if (iterator->has\_next())
457. Assert::AreEqual(iterator->next()->get\_key(), 10);
458. if (iterator->has\_next())
459. Assert::AreEqual(iterator->next()->get\_key(), 8);
460. Assert::AreEqual(iterator->has\_next(), false);
461. }
462. TEST\_METHOD(TestBinarySearchTwoPostOrderedDFTIterator)
463. {
464. BinarySearchTree BST(8);
465. BST.insert(3);
466. Iterator\* iterator = BST.create\_post\_ordered\_depth\_first\_traverse\_iterator();
467. if (iterator->has\_next())
468. Assert::AreEqual(iterator->next()->get\_key(), 3);
469. if (iterator->has\_next())
470. Assert::AreEqual(iterator->next()->get\_key(), 8);
471. Assert::AreEqual(iterator->has\_next(), false);
472. }
473. TEST\_METHOD(TestBinarySearchRemoveOneElementFromThreeDFTIterator)
474. {
475. BinarySearchTree BST(8);
476. BST.insert(14); BST.insert(13);
477. BST.remove(14);
478. Iterator\* iterator = BST.create\_pre\_ordered\_depth\_first\_traverse\_iterator();
479. if (iterator->has\_next())
480. Assert::AreEqual(iterator->next()->get\_key(), 8);
481. if (iterator->has\_next())
482. Assert::AreEqual(iterator->next()->get\_key(), 13);
483. Assert::AreEqual(iterator->has\_next(), false);
484. }
485. TEST\_METHOD(TestBinarySearchRemoveAndInsertBFTIterator)
486. {
487. BinarySearchTree BST(8);
488. BST.insert(3); BST.insert(10); BST.insert(14); BST.insert(13);
489. BST.insert(6); BST.insert(1); BST.insert(7); BST.insert(4);
490. BST.remove(6); BST.remove(10); BST.remove(1);
491. Iterator\* iterator = BST.create\_breadth\_first\_traverse\_iterator();
492. if (iterator->has\_next())
493. Assert::AreEqual(iterator->next()->get\_key(), 8);
494. if (iterator->has\_next())
495. Assert::AreEqual(iterator->next()->get\_key(), 3);
496. if (iterator->has\_next())
497. Assert::AreEqual(iterator->next()->get\_key(), 14);
498. if (iterator->has\_next())
499. Assert::AreEqual(iterator->next()->get\_key(), 7);
500. if (iterator->has\_next())
501. Assert::AreEqual(iterator->next()->get\_key(), 13);
502. if (iterator->has\_next())
503. Assert::AreEqual(iterator->next()->get\_key(), 4);
504. Assert::AreEqual(iterator->has\_next(), false);
505. }
506. TEST\_METHOD(TestBinarySearchRemoveAndInsertDFTIterator)
507. {
508. BinarySearchTree BST(8);
509. BST.insert(3); BST.insert(10); BST.insert(14); BST.insert(13);
510. BST.insert(6); BST.insert(1); BST.insert(7); BST.insert(4);
511. BST.remove(6); BST.remove(10); BST.remove(1);
512. Iterator\* iterator = BST.create\_pre\_ordered\_depth\_first\_traverse\_iterator();
513. if (iterator->has\_next())
514. Assert::AreEqual(iterator->next()->get\_key(), 8);
515. if (iterator->has\_next())
516. Assert::AreEqual(iterator->next()->get\_key(), 3);
517. if (iterator->has\_next())
518. Assert::AreEqual(iterator->next()->get\_key(), 7);
519. if (iterator->has\_next())
520. Assert::AreEqual(iterator->next()->get\_key(), 4);
521. if (iterator->has\_next())
522. Assert::AreEqual(iterator->next()->get\_key(), 14);
523. if (iterator->has\_next())
524. Assert::AreEqual(iterator->next()->get\_key(), 13);
525. Assert::AreEqual(iterator->has\_next(), false);
526. }
527. };
528. }