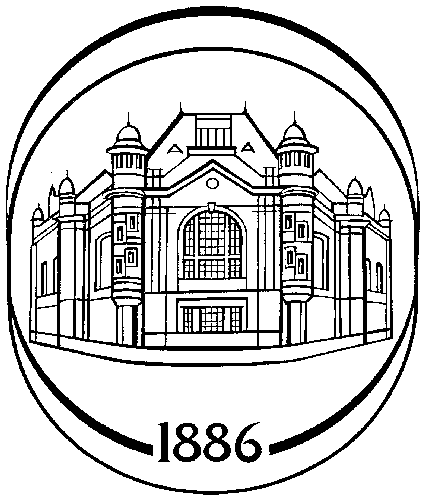
МИНОБРНАУКИ РОССИИ

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 с соответствующими заголовками. Файлы: Queue.cpp, Stack.cpp, Element.cpp, Node.cpp, BinarySearchTree.cpp, main.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\* pop() – удаляет и возвращает первый в очереди элемент
4. Element\* push(Node\*) – добавляет элемент с заданной датой в конец 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\* pop() – O(n)
4. Element\* push(Node\*) – 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)) в случае идеального и полного двоичного дерева, а также близких к ним, в случае перекошенных и вырожденных деревьев будет работать за О(n). В процессе выполнения операции рассматривается каждая новая выбранная по правилам двоичного дерева поиска вершина, сложность зависит от количества в итоге рассмотренных вершин (Можно переписать данный алгоритм рекурсивно, тогда сложность можно будет описать как количество вызовов функции)
4. bool contains(int) – аналогично операции insert
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 18 тестов для проверки правильности работы всех функций стека и очереди на разном количестве элементов, на обработку исключений.

Для итераторов и бинарного дерева поиска реализовано 37 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:

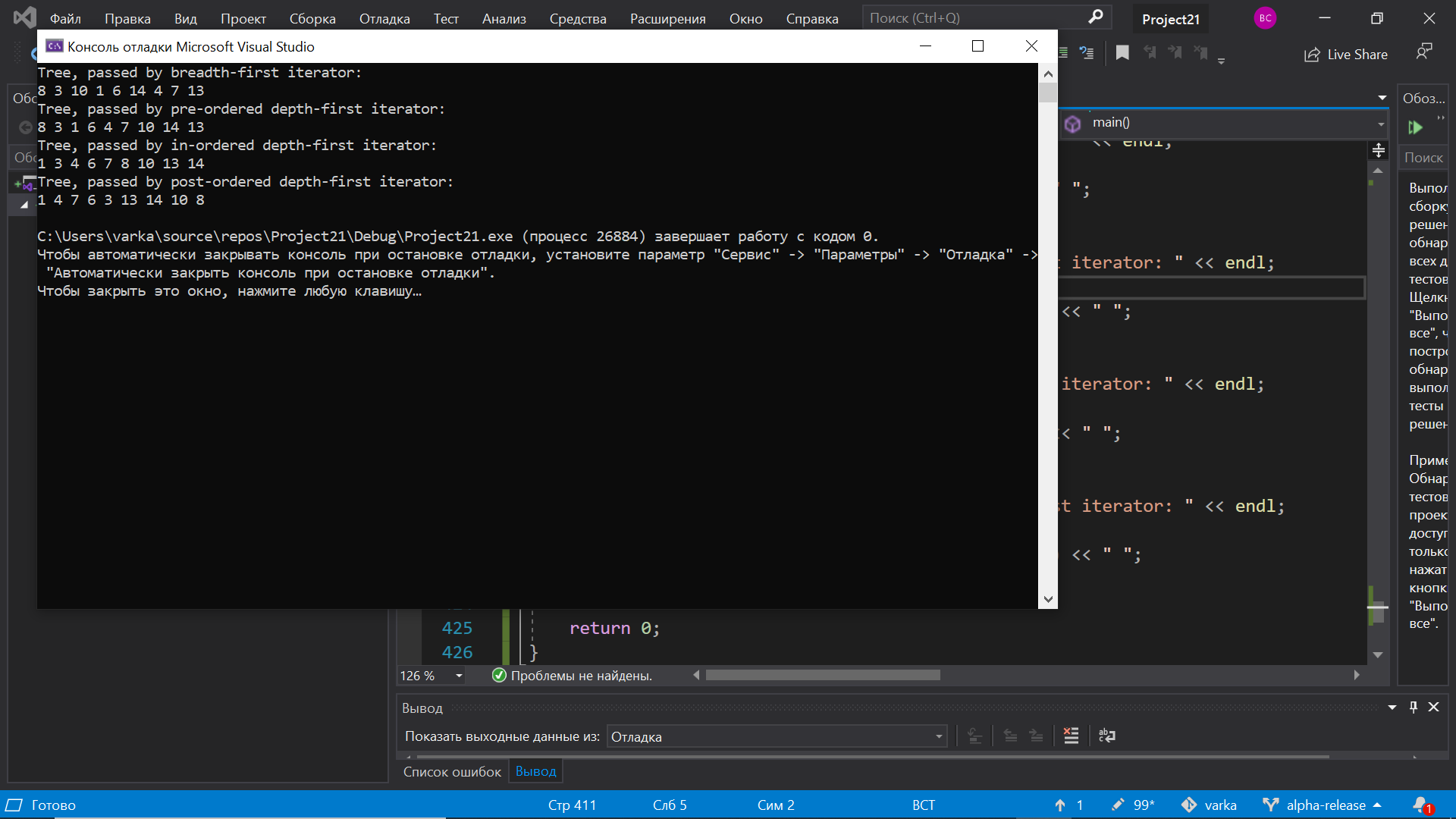


Рис. 1.1 – Пример работы программы

Листинг

Node.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. };

Element.h:

1. #pragma once
2. #include "Node.h"
3. class Element
4. {
5. Element\* next;
6. Node\* data;
7. void set\_next(Element\*);
8. void set\_data(Node\*);
9. public:
10. Element();
11. Element\* get\_next();
12. Node\* get\_data();
13. ~Element();
14. friend class Queue;
15. friend class Stack;
16. };

Queue.h:

1. #pragma once
2. #include "Element.h"
3. class Queue
4. {
5. private:
6. Element\* head; // the first in queue, leaves first
7. Element\* tail; // the last, new in queue
8. void set\_head(Element\*);
9. void set\_tail(Element\*);
10. public:
11. Queue();
12. int size(); // return a number of elements in queue
13. bool is\_empty(); // returns true if queue is empty
14. Element\* pop(); // to delete first in queue and return it to user
15. Element\* push(Node\*); // to add in the end of the queue the element with key-data and return it to user
16. ~Queue();
17. };

Stack.h:

1. #pragma once
2. #include "Element.h"
3. class Stack
4. {
5. private:
6. Element\* top;
7. void set\_top(Element\*);
8. public:
9. Stack();
10. bool is\_empty();
11. int size();
12. Element\* peek(); // to show who's on the top
13. Element\* pop(); // to delte top and show it
14. Element\* push(Node\*); // to push on top and show it
15. ~Stack();
16. };

Iterator.h:

1. #pragma once
2. #include "Node.h"
3. class Iterator
4. {
5. public:
6. virtual Node\* next() = 0; // return current element and goes to the next
7. virtual bool has\_next() = 0; // return true if next exists
8. };

BinarySearchTree.h:

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

Node.cpp:

1. #include "Node.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. }

Element.cpp:

1. #include "Element.h"
2. Element::Element()
3. {
4. data = 0;
5. next = nullptr;
6. }
7. Element\* Element::get\_next() { return next; }
8. Node\* Element::get\_data() { return data; }
9. void Element::set\_next(Element\* new\_element) { next = new\_element; }
10. void Element::set\_data(Node\* information) { data = information; }
11. Element::~Element() { delete next; next = nullptr; }

Queue.cpp:

1. #include "Queue.h"
2. Queue::Queue()
3. {
4. head = nullptr;
5. tail = nullptr;
6. }
7. void Queue::set\_head(Element\* key) { head = key; }
8. void Queue::set\_tail(Element\* key) { tail = key; }
9. bool Queue::is\_empty() // returns true if queue is empty
10. {
11. if (tail == nullptr)
12. return true;
13. return false;
14. }
15. int Queue::size() // return a number of elements in queue
16. {
17. if (is\_empty())
18. return 0;
19. int size = 1;
20. for (Element\* now = tail; now->get\_next() != nullptr; now = now->get\_next())
21. size++;
22. return size;
23. }
24. Element\* Queue::push(Node\* data\_key) // to add in the end of the queue the element with key-data
25. {
26. Element\* new\_element = new Element;
27. new\_element->set\_data(data\_key);
28. if (is\_empty())
29. head = tail = new\_element;
30. else
31. {
32. new\_element->set\_next(tail);
33. tail = new\_element;
34. }
35. return new\_element;
36. }
37. Element\* Queue::pop() // to delete first in queue and return it's key-data
38. {
39. Element\* to\_delete;
40. if (is\_empty())
41. throw out\_of\_range("the queue is empty");
42. else if (size() == 1)
43. {
44. to\_delete = head;
45. head = tail = nullptr;
46. }
47. else
48. {
49. to\_delete = head;
50. for (Element\* now = tail; now->get\_next() != nullptr; now = now->get\_next())
51. head = now;
52. head->next = nullptr;
53. }
54. return to\_delete;
55. }
56. Queue::~Queue()
57. {
58. while (!is\_empty())
59. pop();
60. }

Stack.cpp:

1. #include "Stack.h"
2. void Stack::set\_top(Element\* top\_element) { top = top\_element; }
3. Stack::Stack()
4. { top = nullptr; }
5. bool Stack::is\_empty()
6. {
7. if (top == nullptr)
8. return true;
9. return false;
10. }
11. int Stack::size()
12. {
13. int size = 1;
14. if (is\_empty())
15. return 0;
16. for (Element\* now = top; now->get\_next() != nullptr; now = now->get\_next())
17. size++;
18. return size;
19. }
20. Element\* Stack::peek() // to show who's on the top
21. { return top; }
22. Element\* Stack::pop() // to delte top and show it
23. {
24. Element\* to\_delete = top;
25. if (is\_empty())
26. throw out\_of\_range("the stack is empty");
27. else
28. top = top->get\_next();
29. return to\_delete;
30. }
31. Element\* Stack::push(Node\* data) // to push on top and show it
32. {
33. Element\* new\_element = new Element;
34. new\_element->set\_data(data);
35. if (is\_empty())
36. top = new\_element;
37. else
38. {
39. new\_element->set\_next(top);
40. top = new\_element;
41. }
42. return new\_element;
43. }
44. Stack::~Stack()
45. {
46. while (!is\_empty())
47. pop();
48. }

BinarySearchTree.cpp:

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

main.cpp:

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

UnitTestForQueueAndStack.cpp:

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

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. }