ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
«РОССИЙСКИЙ УНИВЕРСИТЕТ ТРАНСПОРТА»  
(РУТ (МИИТ))

Институт транспортной техники и систем управления

Кафедра «Управление и защита информации»

ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 2

по дисциплине «Системы управления базами данных»

Выполнил: ст. гр. ТКИ – 441

Гаврилин-Кирьянов И.А.

Проверил: к.т.н., доц.

Васильева М. А.

Москва 2022

ОГЛАВЛЕНИЕ

[ОГЛАВЛЕНИЕ 2](#_Toc124969078)

[ЗАДАНИЕ 3](#_Toc124969079)

[СТРУКТУРА ПРОЕКТА 3](#_Toc124969080)

[1 ЛИСТИНГ ПРОГРАММЫ 4](#_Toc124969081)

[1.1 Файл Node.cpp» 4](#_Toc124969082)

[1.2 Файл «Binary\_tree.cpp» 9](#_Toc124969083)

[1.3 Файл «test\_tree.cpp» 14](#_Toc124969084)

[1.4 Файл «Makefile» 16](#_Toc124969085)

[2 ДИАГРАММЫ структур 17](#_Toc124969086)

[3 РЕЗУЛЬТАТЫ ПРОХОЖДЕНИЙ ТЕСТОВ 18](#_Toc124969087)

ЗАДАНИЕ

1. Для структуры «Двоичное дерево» разработать API;
2. На языке «C++» написать конструктор и деструктор класса разрабатываемой структуры для типа «int»;
3. Разработать метод вывода структуры в строку;
4. Реализовать основные функции для работы.
5. Разработать тесты.

СТРУКТУРА ПРОЕКТА

* Task\_2
  + source
    - Binary\_tree.cpp
    - Binary\_tree.h
    - main.cpp
    - Node.cpp
    - Node.h
  + tests
    - test\_tree.cpp
    - test\_node.cpp
  + Makefile

# ЛИСТИНГ ПРОГРАММЫ

## Файл Node.h»

1. #pragma once
2. #include <iostream>

5. **struct** Node;
7. /\*
8. \* \brief Метод преобразования узла в строку
9. \* \param Узел дерева
10. \* \return Данные в виде строки
11. \*/
12. std::wstring ToString(**const** Node& node);
14. /\*
15. \* \brief Узел дерева
16. \*/
17. **struct** Node {
18. /\*
19. \* \brief Данные для узла
20. \*/
21. **int** key;
23. /\*
24. \* \brief Указатель на левый узел
25. \*/
26. Node\* left = nullptr;
28. /\*
29. \* \brief Указатель на правый узел
30. \*/
31. Node\* right = nullptr;

34. /\*
35. \* \brief Конструктор без параметров
36. \*/
37. Node();
39. /\*
40. \* \brief Конструктор с параметрами
41. \* \param Данные узла
42. \*/
43. Node(**const** **int** key);
45. /\*
46. \* \brief Деструктор
47. \*/
48. ~Node();
50. /\*
51. \* \brief Оператор вывода в стандартный поток данных
52. \* \param Стандартный поток данных
53. \* \param Узел дерева
54. \*/
55. friend std::ostream& **operator**<<(std::ostream& **out**, **const** Node& node);
57. /\*
58. \* \brief Оператор вывода в стандартный поток данных
59. \* \param Стандартный поток данных
60. \* \param Узел дерева
61. \*/
62. friend std::wostream& **operator**<<(std::wostream& **out**, **const** Node& node);
64. /\*
65. \* \brief Оператор ввода из стандартного потока данных
66. \* \param Стандартный поток данных
67. \* \param Узел дерева
68. \*/
69. friend std::istream& **operator**>>(std::istream& **in**, Node& node);
71. /\*
72. \* \brief Оператор равенства двух узлов
73. \* \param Левый узел
74. \* \param Правый узел
75. \* \return true, если равны
76. \*/
77. friend **bool** **operator**==(**const** Node& left, **const** Node& right);
79. /\*
80. \* \brief Оператор неравенства двух узлов
81. \* \param Левый узел
82. \* \param Правый узел
83. \* \return true, если не равны
84. \*/
85. friend **bool** **operator**!=(**const** Node& left, **const** Node& right);
87. /\*
88. \* \brief Оператор больше
89. \* \param Левый узел
90. \* \param Правый узел
91. \* \return true, если левый узел больше
92. \*/
93. friend **bool** **operator**>(**const** Node& left, **const** Node& right);
95. /\*
96. \* \brief Оператор больше или равно
97. \* \param Левый узел
98. \* \param Правый узел
99. \* \return true, если левый узел больше
100. \*/
101. friend **bool** **operator**>=(**const** Node& left, **const** Node& right);
103. /\*
104. \* \brief Оператор меньше
105. \* \param Левый узел
106. \* \param Правый узел
107. \* \return true, если левый узел меньше
108. \*/
109. friend **bool** **operator**<(**const** Node& left, **const** Node& right);
111. /\*
112. \* \brief Оператор меньше
113. \* \param Левый узел
114. \* \param Правый узел
115. \* \return true, если левый узел меньше
116. \*/
117. friend **bool** **operator**<=(**const** Node& left, **const** Node& right);
118. };

## Файл «Node.cpp»

1. #pragma once
2. #include "Node.h"
3. #include <string>
4. #include <sstream>
6. Node::Node()
7. : Node(0)
8. {
9. }
11. Node::Node(**const** **int** key)
12. : key(key), left(nullptr), right(nullptr)
13. {
14. }
16. Node::~Node() {
17. left  = nullptr;
18. right = nullptr;
19. }
21. std::wstring ToString(**const** Node& node)
22. {
23. std::wstringstream **out**;
24. **out** << node;
25. **return** **out**.str();
26. }
28. std::ostream& **operator**<<(std::ostream& **out**, **const** Node& leaf)
29. {
30. **out** << leaf.key;
31. **return** **out**;
32. }
34. std::wostream& **operator**<<(std::wostream& **out**, **const** Node& leaf)
35. {
36. **out** << leaf.key;
37. **return** **out**;
38. }
40. std::istream& **operator**>>(std::istream& **in**, Node& node)
41. {
42. **if** (**in**.peek() == '\n' || **in**.eof())
43. {
44. **return** **in**;
45. }
47. **in** >> node.key;
48. **return** **in**;
49. }
51. **bool** **operator**==(**const** Node& left, **const** Node& right)
52. {
53. **return** left.key == right.key;
54. }
56. **bool** **operator**!=(**const** Node& left, **const** Node& right)
57. {
58. **return** left.key != right.key;
59. }
61. **bool** **operator**>(**const** Node& left, **const** Node& right)
62. {
63. **return** left.key > right.key;
64. }
66. **bool** **operator**>=(**const** Node& left, **const** Node& right)
67. {
68. **return** left.key >= right.key;
69. }
71. **bool** **operator**<(**const** Node& left, **const** Node& right)
72. {
73. **return** left.key < right.key;
74. }
76. **bool** **operator**<=(**const** Node& left, **const** Node& right)
77. {
78. **return** left.key <= right.key;
79. }

## Файл «Binary\_tree.h»

1. #pragma once
2. #include <iostream>
3. #include "Node.h"
5. **class** Tree;
7. /\*
8. \* \brief Класс дерево
9. \*/
10. **class** Tree
11. {
12. **public**:
14. /\*
15. \* \brief Конструктор класса
16. \*/
17. Tree();
19. /\*
20. \* \brief Деструктор класса
21. \*/
22. ~Tree();
24. /\*
25. \* \brief Корень дерева
26. \*/
27. Node\* root;
29. /\*
30. \* \brief Публичнная функция проверки дерева на пустоту
31. \* \return true, если дерево является пустым
32. \*/
33. **bool** IsEmpty();
35. /\*
36. \* \brief Публичная функция добавления элемента в дерево
37. \* \param Добавляемое значение
38. \*/
39. **void** insert(**int** key);
41. /\*
42. \* \brief Публичная функция удаления дерева
43. \*/
44. **void** deleteTree();
46. /\*
47. \* \brief Публичная ункция вывода дерева в упорядоченном состоянии
48. \*/
49. **void** printTree();
51. /\*
52. \* \brief Публичная функция замены первого найденного узла указанным
53. \* \param Заменяемое значение
54. \* \param Новое значение
55. \*/
56. **void** changeNode(**int** old\_key, **int** new\_key);
58. /\*
59. \* \brief Публичная функция удаления первого найденного узла с указанным значением из дерева
60. \* \param Удаляемое значение
61. \* \return
62. \*/
63. **void** deleteNode(**int** key);
65. **private**:
67. /\*
68. \* \brief Приватная функция добавления элемента в дерево
69. \* \param Добавляемое значение
70. \* \return Измененное дерево с добавленным значением
71. \*/
72. Node\* insert(**int** key, Node\* leaf);
74. /\*
75. \* \brief Приватная вспомогательная функция добавления элемента в дерево
76. \* \param Добавляемое значение
77. \* \return Измененное дерево с добавленным значением
78. \*/
79. Node\* newNode(**int** key);
81. /\*
82. \* \brief Приватная функция удаления первого найденного узла с указанным значением из дерева
83. \* \param Удаляемое значение
84. \* \return Измененное дерево с удаленным значением
85. \*/
86. Node\* deleteNode(**int** key, Node\* leaf);
88. /\*
89. \* \brief Приватная вспомогательная функция для нахождения минимального элемента в дереве
90. \* \return Узел с наименьшим значением
91. \*/
92. Node\* minValueNode(Node\* leaf);
94. /\*
95. \* \brief Приватная функция удаления дерева
96. \* \param Указатель на вершину дерева
97. \*/
98. **void** deleteTree(Node\* leaf);
100. /\*
101. \* \brief Приватная функция вывода дерева в упорядоченном состоянии
102. \* \param Указатель на вершину дерева
103. \* \return Строка с упорядоченным перечислением элементов дерева
104. \*/
105. std::**string** ToString(Node\* leaf);
107. /\*
108. \* \brief Приватная функция замены первого найденного узла указанным
109. \* \param Заменяемое значение
110. \* \param Новое значение
111. \* \param Указатель на вершину дерева
112. \*/
113. **void** changeNode(**int** old\_key, **int** new\_key, Node\* root);
114. };

## Файл «Binary\_tree.cpp»

1. #pragma once
2. #include "Binary\_tree.h"
3. #include <sstream>
4. #include <iostream>

7. Tree::Tree() : root(nullptr)
8. {
9. }
11. Tree::~Tree()
12. {
13. deleteTree(root);
14. root = nullptr;
15. }
17. Node\* Tree::minValueNode(Node\* leaf) {
18. Node\* current = leaf;
20. // Ищем крайний левый лист — он и будет inorder-преемником
21. **while** (current && current->left != NULL) {
22. current = current->left;
23. }
24. **return** current;
25. }
27. **void** Tree::deleteNode(**int** key) {
28. root = deleteNode(key, root);
29. }
31. Node\* Tree::deleteNode(**int** key, Node\* leaf) {
32. // Возвращаем, если дерево пустое
33. **if** (leaf == NULL) {
34. **return** leaf;
35. }
37. // Ищем узел, который хотим удалить
38. **if** (key < leaf->key) {
39. leaf->left = deleteNode(key, leaf->left);
40. }
41. **else** **if** (key > leaf->key) {
42. leaf->right = deleteNode(key, leaf->right);
43. }
44. **else** {
45. // Если у узла один дочерний элемент или их нет
46. **if** (leaf->left == NULL) {
47. Node\* temp = leaf->right;
48. free(leaf);
49. **return** temp;
50. }
51. **else** **if** (leaf->right == NULL) {
52. Node\* temp = leaf->left;
53. free(leaf);
54. **return** temp;
55. }
57. // Если у узла два дочерних элемента
58. Node\* temp = minValueNode(leaf->right);
60. // Помещаем inorder-преемника на место узла, который хотим удалить
61. leaf->key = temp->key;
63. // Удаляем inorder-преемника
64. leaf->right = deleteNode(temp->key, leaf->right);
65. }
66. **return** leaf;
67. }
69. **void** Tree::deleteTree()
70. {
71. deleteTree(root);
72. **this**->root = nullptr;
73. }
75. **void** Tree::deleteTree(Node\* leaf)
76. {
77. **if** (leaf != nullptr)
78. {
79. deleteTree(leaf->left);
80. deleteTree(leaf->right);
81. delete leaf;
82. }
83. }
85. **bool** Tree::IsEmpty() {
86. **return** **this**->root == nullptr;
87. }
89. **void** Tree::insert(**int** key)
90. {
91. root = insert(key, root);
92. }
94. Node\* Tree::newNode(**int** key) {
95. Node\* temp = **new** Node;
96. temp->key = key;
97. **return** temp;
98. }
100. Node\* Tree::insert(**int** key, Node\* leaf) {
101. // Возвращаем новый узел, если дерево пустое
102. **if** (leaf == nullptr) {
103. **return** newNode(key);
104. }
106. // Проходим в нужное место и вставляет узел
107. **if** (key <= leaf->key) {
108. leaf->left = insert(key, leaf->left);
109. }
110. **else** {
111. leaf->right = insert(key, leaf->right);
112. }
113. **return** leaf;
114. }
116. **void** Tree::printTree() {
117. std::cout << ToString(**this**->root);
118. }
120. std::**string** Tree::ToString(Node\* root) {
121. **if** (root == nullptr) {
122. **return** std::**string**("");
123. }
124. std::stringstream ss;
125. ss << ToString(root->left) << " " << root->key << " " << ToString(root->right);
126. **return** ss.str();
127. }
129. **void** Tree::changeNode(**int** old\_key, **int** new\_key) {
130. changeNode(old\_key, new\_key, **this**->root);
131. }
133. **void** Tree::changeNode(**int** old\_key, **int** new\_key, Node\* root) {
134. **if** (root == NULL) {
135. **return**;
136. }
137. **if** (root->key == old\_key) {
138. deleteNode(old\_key, **this**->root);
139. insert(new\_key, **this**->root);
140. **return**;
141. }
142. **if** (old\_key <= root->key) {
143. changeNode(old\_key, new\_key, root->left);
144. }
145. **else** {
146. changeNode(old\_key, new\_key, root->right);
147. }
148. }

## Файл «test\_node.cpp»

1. #include <gtest/gtest.h>
2. #include <gmock/gmock.h>
4. #include <Node.cpp>

7. **using** **namespace** std;

10. TEST(NodeTestGroup, NodeTest) {
11. // Arange
12. **const** **int** key = 4;
14. // Act
15. Node\* node = **new** Node(key);
17. // Assert
18. ASSERT\_TRUE(node->key == key);
19. }
21. **int** main(**int** argc, **char** \*\*argv)
22. {
23. ::testing::InitGoogleTest(&argc, argv);
25. **return** RUN\_ALL\_TESTS();
26. }

## Файл «test\_tree.cpp»

1. #include <gtest/gtest.h>
2. #include <gmock/gmock.h>
4. #include <Binary\_tree.cpp>

7. **using** **namespace** std;

10. TEST(BinaryTreeTestGroup, BinaryTree) {
11. // Arange
12. **const** **int** expected\_data = 10;
13. Tree\* myTree = **new** Tree();
15. // Act
16. myTree->root->key = expected\_data;
18. // Assert
19. ASSERT\_TRUE(myTree->root->key == expected\_data);
20. }
22. TEST(BinaryTreeTestGroup, BinaryTreeInsertKey) {
23. // Arange
24. **const** **int** expected\_data = 3;
25. Tree\* myTree = **new** Tree();
27. // Act
28. myTree->insert(expected\_data);
30. // Assert
31. ASSERT\_TRUE(myTree->root->key == expected\_data);
32. }
34. TEST(BinaryTreeTestGroup, BinaryTreeInsertNode) {
35. // Arange
36. **const** **int** expected\_key = 3;
37. Tree\* myTree = **new** Tree();
38. Node\* myNode = **new** Node(4);
40. // Act
41. myTree->insert(expected\_key, myNode);
42. Node\* treeNode = myTree->root->right;
44. // Assert
45. ASSERT\_TRUE(treeNode->key == expected\_key);
46. }
48. TEST(BinaryTreeTestGroup, BinaryTreeDelete) {
49. // Arange
50. Tree\* myTree = **new** Tree();
52. // Act
53. myTree->deleteTree();
55. // Assert
56. ASSERT\_TRUE(myTree->root == nullptr);
57. }
59. TEST(BinaryTreeTestGroup, BinaryTreeEmpty) {
60. // Arange
61. **const** **int** expected\_data = 3;
62. Tree\* myTree = **new** Tree();
64. // Act
65. myTree->insert(expected\_data);
66. **bool** isEmpty = myTree->IsEmpty();
68. // Assert
69. ASSERT\_FALSE(isEmpty == **true**);
70. }
72. **int** main(**int** argc, **char** \*\*argv)
73. {
74. ::testing::InitGoogleTest(&argc, argv);
76. **return** RUN\_ALL\_TESTS();
77. }

## Файл «Makefile»

1. FILENAME = main
3. install:
4. mkdir -p build
5. g++ -g source/$(FILENAME).cpp -o build/$(FILENAME) -I source/
6. g++ -g tests/test\_node.cpp -o build/test\_node -I source/  -lgtest -lpthread
7. g++ -g tests/test\_tree.cpp -o build/test\_tree -I source/  -lgtest -lpthread
8. build/test\_node
9. build/test\_tree

# ДИАГРАММЫ структуры и класса

Text

Description automatically generated

Рисунок 1. Диаграмма методов структуры Node

Text

Description automatically generated

Рисунок 2. Диаграмма методов класса Binary\_tree

# РЕЗУЛЬТАТЫ ПРОХОЖДЕНИЙ ТЕСТОВ

![Table

Description automatically generated]()

Рисунок 3. Результаты прохождения тестов