МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №5

по дисциплине «Алгоритмы и структуры данных»

Тема: АВЛ-деревья

Студент гр. 9381 _____ Аухадиев А.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Познакомиться со структурой данных АВЛ-деревья и алгоритмами работы с ними, реализовать алгоритм добавления элемента в АВЛ-дерево на языке C++.

Задание.

15. БДП: АВЛ-дерево; действие: 1+2a

По заданной последовательности элементов Elem построить структуру данных определённого типа – БДП или хеш-таблицу

Для построенной структуры данных проверить, входит ли в неё элемент e типа Elem, и если входит, то в скольких экземплярах. Добавить элемент e в структуру данных. Предусмотреть возможность повторного выполнения с другим элементом.

Описание структуры данных и алгоритма.

1. АВЛ-дерево.

АВЛ-дерево - бинарное дерево поиска, для которого выполняются два требования:

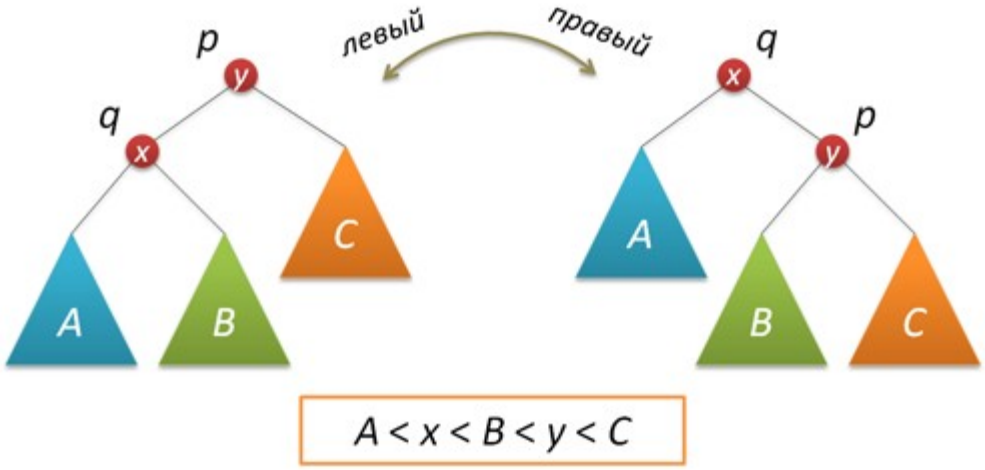
1. Для каждой вершины все вершины из ее левого поддерева меньше по величине, чем ее значение, а все вершины из правого поддерева - больше.

2. Разность высот левого и правого поддерева текущей вершины должно находится в диапазоне от -1 до 1. При нарушении требования происходит перебалансировка вершин, то есть правые и левые повороты.

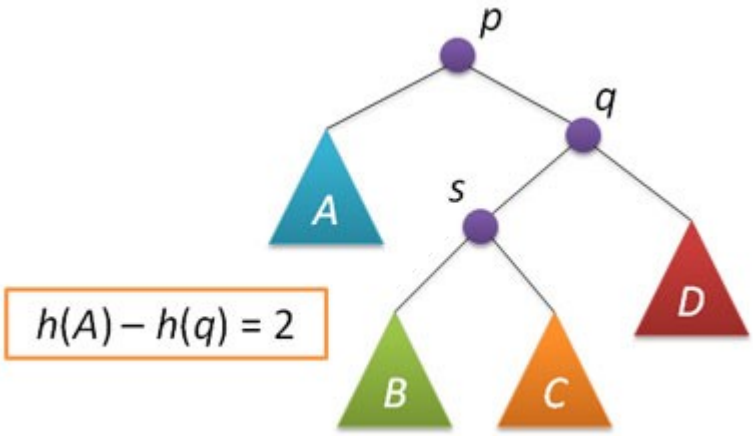
2. Алгоритм поворота дерева.

В процессе добавления или удаления узлов в АВЛ-дереве возможно возникновение ситуации, когда баланс некоторых узлов (разница между высотами левого и правого сыновей) оказывается равными 2 или -2, т.е. возникает расбалансировка поддерева. Для выправления ситуации применяются повороты вокруг тех или иных узлов дерева.

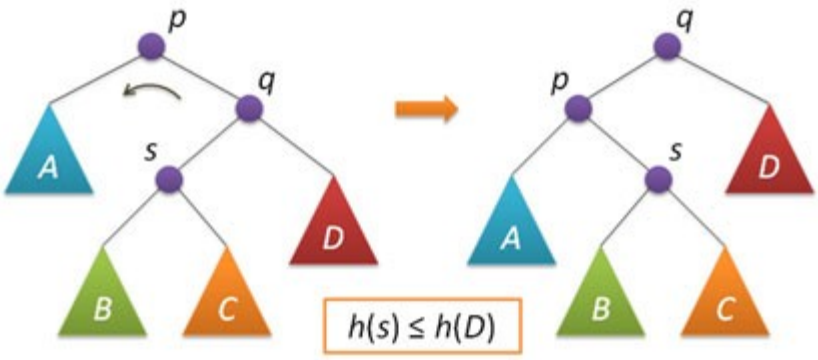
Простой поворот вправо (влево) производит следующую трансформацию дерева:



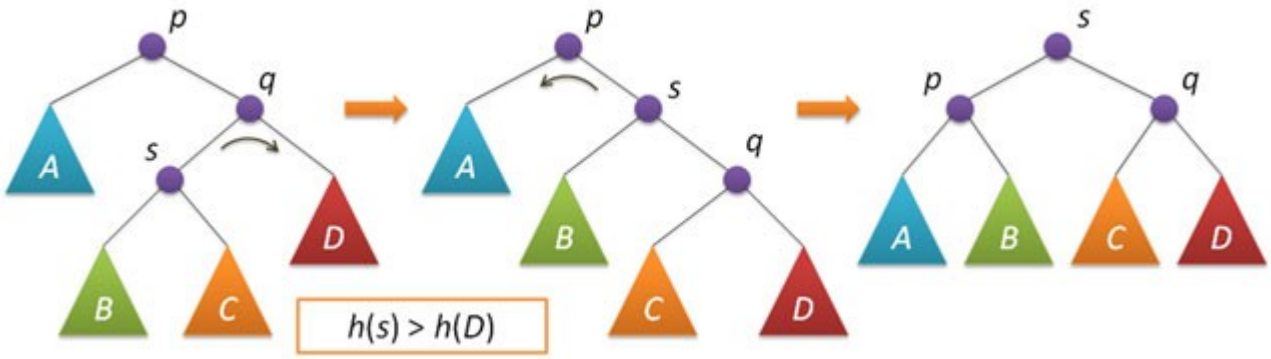
Существует ситуация дисбаланса, когда высота правого поддерева узла p на 2 больше высоты левого поддерева (обратный случай является симметричным и реализуется аналогично). Пусть q — правый дочерний узел узла p, а s — левый дочерний узел узла q.



Для исправления расбалансировки в узле p достаточно выполнить либо простой поворот влево вокруг p, либо так называемый большой поворот влево вокруг того же p. Простой поворот выполняется при условии, что высота левого поддерева узла q больше высоты его правого поддерева: h(s)≤h(D).



Большой поворот применяется при условии h(s)>h(D) и сводится в данном случае к двум простым — сначала правый поворот вокруг q и затем левый вокруг p.



Описанные алгоритмы поворотов и балансировки не содержат ни циклов, ни рекурсии, а значит выполняются за постоянное время, не зависящее от размера АВЛ-дерева.

В коде алгоритмы поворота вправо и влево осуществляют функции rightRotate() и leftRotate() (Описание функций см. ниже).

3. Алгоритм добавления элемента.

Вставка нового ключа в АВЛ-дерево выполняется так же, как это делается в простых деревьях поиска: осуществляется проход вниз по дереву, выбираются правое или левое направление движения в зависимости от результата сравнения ключа в текущем узле и вставляемого ключа. Единственное отличие заключается в том, что при возвращении из рекурсии (т.е. после того, как ключ

вставлен либо в правое, либо в левое поддерево, и это дерево сбалансировано) выполняется балансировка текущего узла.

В коде алгоритм добавления элемента в дерево осуществляется с помощью функции insert() (Описание функции см. ниже).

Выполнение работы.

1.1. Структура данных

class Node - класс, описывающий основные свойства узла АВЛ-дерева.

Поля класса:

1. int key_ - значение ключа узла
2. unsigned char height_ - высота узла
3. Node* left_ - левый сын
4. Node* right_ - правый сын
5. int count_ - количество узлов с таким же значением ключа

Конструктор:

Node(int key): key_(key), height_(1), left_(nullptr), right_(nullptr), count_(1) {};

1.2. Функции для работы с АВЛ-деревьями

1. void printTree(Node* root, int n, int flagRight = 0) - функция вывода АВЛ-дерева на экран, "лежащим на левом боку". Принимает указатель на АВЛ-дерево, количество табуляций и флаг, указывающий место для вывода ребра между узлами.

2. int height(Node* N) - возвращает значение высоты дерева N.

3. int max(int a, int b) - вспомогательная функция, возвращающая большее из чисел a и b.

4. Node* rightRotate(Node* N) - функция правого поворота дерева N, возвращает дерево после правого поворота. (Описание алгоритма см. ниже)

5. Node* leftRotate(Node* N) - функция левого поворота дерева N, возвращает дерево после левого поворота. (Описание алгоритма см. ниже)

6. int getBalance(Node* N) - возвращает баланс узла N (баланс - разница между высотами левого и правого сыновей.

7. Node* insert(Node* node, int key) - функция вставки элемента с ключом key в дерево node, возвращает указатель на дерево с добавленным элементом. (Описание алгоритма см. ниже)

8. Node* minValueNode(Node* node) - возвращает узел с минимальным значением ключа (самый левый лист) дерева node.

9. Node* deleteNode(Node* root, int key) - функция удаления элемента из дерева root по ключу key, возвращает дерево с удалённым элементом. (Описание алгоритма см. ниже)

10. void clearTree(Node* root) - рекурсивное высвобождение выделенной памяти под узлы дерева root.

11. bool isDigit(std::string key) - проверка строки key, если она является числом, возвращает true, иначе - false.

Разработанный программный код см. в приложении А.

Результаты тестирования см. в приложении Б.

Выводы.

В ходе работы была изучена структура данных АВЛ-дерево, освоены алгоритмы работы с этой структурой данных, написана программа на языке программирования C++, способная добавлять элементы АВЛ-дерева и выводить результат в консоли.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <iostream>
#include <cctype>
#include <string>
#include <sstream>
#include <fstream>

class Node *globalTree = nullptr;

class Node {
public:
    int key_;       //Значение в узле
    unsigned char height_; //высота узла
    Node* left_;         //левый сын
    Node* right_;        //правый сын
    int count_;             //Количество узлов с одинаковым ключом
    Node(int key): key_(key), height_(1), left_(nullptr), right_(nullptr),
count_(1){};
};

void printTree(Node* root, int n = 0, int flagRight = 0, int key = -1)
{   //Вывод дерева, "лежащего на левом боку"
    if(!root){
        std::cout << "Дерево пусто\n";
        return;
    }
    if(root->right_)
        printTree(root->right_, n+4, 1, key);
    for(auto i = 0; i<n; i++)
        std::cout << " ";
    if(root->key_ == key)
        std::cout << "\x1b[31m" << root->key_ << "(" << root->count_ <<
")" << "\x1b[0m" << '\n';
    else
        std::cout << root->key_ << '(' << root->count_ << ")\n";
    if(flagRight || root->left_) {
        for (auto i = 0; i < n + 1; i++) {
            if (i == n - 3 && flagRight)
                std::cout << '/';
            if (i == n && root->left_)
                std::cout << "\\";
            std::cout << ' ';
        }
        std::cout << '\n';
    }
    if(root->left_)
        printTree(root->left_, n+4, 0, key);
}

int height(Node* N){ //получение высоты дерева
    if (!N)
        return 0;

    return N->height_;
}

int max(int a, int b){ //Получение максимума из двух чисел
    return (a > b) ? a : b;
}
```

```
      Node* rightRotate(Node* N) //Правый поворот
      {
          std::cout << "Выполнение правого поворота относительно узла с ключом "
<< N->key_ << '\n';
          printTree(globalTree, 0, 0, N->key_);
          Node* x = N->left_;
          Node* y = x->right_;

          // Выполнить вращение
          x->right_ = N;
          N->left_ = y;

          // Обновление высоты
          N->height_ = max(height(N->left_), height(N->right_)) + 1;
          x->height_ = max(height(x->left_), height(x->right_)) + 1;

          std::cout << "\nДерево после поворота:\n";
          printTree(x);
          // Возвращаем новый корень
          return x;
      }

      Node* leftRotate(Node* N) //Левый поворот
      {
          std::cout << "Выполнение левого поворота относительно узла с ключом "
<< N->key_ << "\n";
          printTree(globalTree, 0, 0, N->key_);
          Node* x = N->right_;
          Node* y = x->left_;

          // Выполнить вращение
          x->left_ = N;
          N->right_ = y;

          // Обновление высоты
          N->height_ = max(height(N->left_), height(N->right_)) + 1;
          x->height_ = max(height(x->left_), height(x->right_)) + 1;

          std::cout << "\nДерево после поворота:\n";
          printTree(x);

          return x;
      }

      int getBalance(Node* N) //Баланс узла
      {
          if (!N)
              return 0;

          return height(N->left_) - height(N->right_);
      }

      Node* insert(Node* node, int key) {

          if (!node) {
              std::cout << "Добавляем ключ " << key << '\n';
              return (new Node(key));
          }

          // Если ключ уже существует, увеличить счетчик и вернуть
          if (key == node->key_) {
              (node->count_)++;
              return node;
          }
```

```
        /* В противном случае движемся вниз по дереву */
        if (key < node->key_) {
            std::cout << "Добавляемый ключ " << key << " меньше узла с ключом
" << node->key_ << ", движемся влево\n";
            node->left_ = insert(node->left_, key);
        } else{
            std::cout << "Добавляемый ключ " << key << " больше узла с ключом
" << node->key_ << ", движемся вправо\n";
            node->right_ = insert(node->right_, key);
        }
        /* Обновить высоту этого узла-предка */
        node->height_ = max(height(node->left_), height(node->right_)) + 1;

        /* Получить коэффициент баланса этого узла-предка, чтобы проверить
уравновешенность*/
        int balance = getBalance(node);

        // Если этот узел становится несбалансированным, то есть 4 случая
        // Левый левый регистр
        if (balance > 1 && key < node->left_->key_) {
            std::cout << "В узле с ключом " << node->key_ << ": \nбаланс = "
<< balance
                      << " и ключ добаемого элемента " << key << " меньше
ключа левого сына "
                      << node->left_->key_ << " => выполняем поворот вправо\
n";
            return rightRotate(node);
        }

        // правый правый случай
        if (balance < -1 && key > node->right_->key_) {
            std::cout << "В узле с ключом " << node->key_ << ": \nбаланс = "
<< balance
                      << " и ключ добавляемого элемента " << key << " больше
ключа правого сына "
                      << node->right_->key_ << " => выполняем поворот влево\
n";
            return leftRotate(node);
        }

        // левый правый регистр
        if (balance > 1 && key > node->left_->key_) {
            std::cout << "В узле с ключом " << node->key_ << ": \nбаланс = "
<< balance <<
                      " и ключ добавляемого элемента " << key << " больше
ключа левого сына"
                      << node->left_->key_ << " => выполняем поворот левого
сына влево\n";
            node->left_ = leftRotate(node->left_);
            std::cout << "В узле с ключом " << node->key_ << ": \nбаланс = "
<< balance <<
                      " => выполнем поворот вправо\n";
            return rightRotate(node);
        }

        // Правый левый регистр
        if (balance < -1 && key < node->right_->key_) {
            std::cout << "В узле с ключом " << node->key_ << ": \nбаланс = "
<< balance <<
                      " и ключ добавляемого элемента " << key << " меньше
ключа правого сына"
                      << node->right_->key_ << " => выполнем поворот правого
сына вправо\n";
```

```
            node->right_ = rightRotate(node->right_);
            std::cout << "В узле с ключом " << node->key_ << ": \nбаланс = "
<< balance <<
                      " => выполнем поворот влево\n";
            return leftRotate(node);
        }

        /* вернуть (неизмененный) указатель узла */
        return node;
    }

    // Если дано непустое двоичное дерево, вернуть узел с минимальным
значением ключа, найденного в этом дереве.
    Node* minValueNode(Node* node){

        Node* current = node;

        //поиск самого левого листа

        while (current->left_ != NULL)
            current = current->left_;

        return current;
    }

    void clearTree(Node* root){
        if(!root)
            return;
        if(root->left_)
            clearTree(root->left_);
        if(root->right_)
            clearTree(root->right_);
        delete root;
    }

    bool isDigit(std::string key) {
        for (auto i = 0; i < key.length(); i++)
            if (!std::isdigit(key[i]))
                return false;
        return true;
    }

    int main()
    {
        char chooseInput = 0;
        char chooseAction = 0;
        std::string key;
        std::string keys;
        std::string fileName;
        std::cout << "Выберите формат ввода(0 - консоль, 1 - файл)\n";
        std::cin >> chooseInput;
        switch(chooseInput) {
            case '0':
                while (chooseAction != '*') {
                    std::cout << "Выберите действие: + - добавление элемента,
* - выход из программы\n";
                    std::cin >> chooseAction;
                    switch (chooseAction) {
                        case '+':
                            std::cout << "Введите значение ключа добавляемого
элемента\n";
                            std::cin >> key;
                            if (!isDigit(key)) {
```

```
                                std::cout << "Ключом должно быть целое
положительное число\n";
                                continue;
                            }
                            globalTree = insert(globalTree,
atoi(key.c_str()));
                            std::cout << "Получившееся дерево:\n";
                            printTree(globalTree, 0);
                            break;
                        case '*':
                            std::cout << "Выход из программы\n";
                            break;
                        default:
                            std::cout << "Данное действие не поддерживается
программой\n";
                            break;
                    }
                }
                break;
            case '1': {
                std::cout << "Введите название файла\n";
                std::cin >> fileName;
                std::fstream file(fileName);
                if (!file.is_open()) {
                    std::cout << "Файл не может быть открыт\n";
                    break;
                }
                std::getline(file, keys);
                if(keys.empty()){
                    std::cout << "Строка пуста\n";
                    break;
                }
                std::cout << "Элементы дерева:\n" << keys << '\n';
                file.close();
                std::istringstream stream(keys);
                int x = 0;
                while(stream >> x) {
                    std::cout << "Добавляемый элемент: " << x << '\n';
                    globalTree = insert(globalTree, x);
                    std::cout << "Получившееся дерево:\n";
                    printTree(globalTree, 0);
                }
                break;
            }
            default:
                std::cout << "Данный формат ввода не поддерживается
программой\n";
                break;
        }
        clearTree(globalTree);
        return 0;
    }
```

ПРИЛОЖЕНИЕ Б
ТЕСТИРОВАНИЕ

№ п/п	Входные данные	Выходные данные
1.	1 2 3	Добавляемый элемент: 1 Добавляем ключ 1 Получившееся дерево: 1(1) Добавляемый элемент: 2 Добавляемый ключ 2 больше узла с ключом 1, движемся вправо Добавляем ключ 2 Получившееся дерево: 2(1) / 1(1) Добавляемый элемент: 3 Добавляемый ключ 3 больше узла с ключом 1, движемся вправо Добавляемый ключ 3 больше узла с ключом 2, движемся вправо Добавляем ключ 3 В узле с ключом 1: баланс = -2 и ключ добавляемого элемента 3 больше ключа правого сына 2 => выполняем поворот влево Выполнение левого поворота относительно узла с ключом 1 3(1) / 2(1) / 1(1) Дерево после поворота: 3(1) / 2(1) \ 1(1) Получившееся дерево: 3(1) / 2(1) \ 1(1)
2.	3 2 1	Добавляемый элемент: 3 Добавляем ключ 3 Получившееся дерево: 3(1)

		Добавляемый элемент: 2 Добавляемый ключ 2 меньше узла с ключом 3, движемся влево Добавляем ключ 2 Получившееся дерево: 3(1) \ 2(1) Добавляемый элемент: 1 Добавляемый ключ 1 меньше узла с ключом 3, движемся влево Добавляемый ключ 1 меньше узла с ключом 2, движемся влево Добавляем ключ 1 В узле с ключом 3: баланс = 2 и ключ добаемого элемента 1 меньше ключа левого сына 2 => выполняем поворот вправо Выполнение правого поворота относительно узла с ключом 3 3(1) \ 2(1) \ 1(1) Дерево после поворота: 3(1) / 2(1) \ 1(1) Получившееся дерево: 3(1) / 2(1) \ 1(1)
3.	2 3 1 6 5	Добавляемый элемент: 2 Добавляем ключ 2 Получившееся дерево: 2(1) Добавляемый элемент: 3 Добавляемый ключ 3 больше узла с ключом 2, движемся вправо Добавляем ключ 3 Получившееся дерево: 3(1) / 2(1) Добавляемый элемент: 1

		Добавляемый ключ 1 меньше узла с ключом 2, движемся влево Добавляем ключ 1 Получившееся дерево: 3(1) / 2(1) \ 1(1) Добавляемый элемент: 6 Добавляемый ключ 6 больше узла с ключом 2, движемся вправо Добавляемый ключ 6 больше узла с ключом 3, движемся вправо Добавляем ключ 6 Получившееся дерево: 6(1) / 3(1) / 2(1) \ 1(1) Добавляемый элемент: 5 Добавляемый ключ 5 больше узла с ключом 2, движемся вправо Добавляемый ключ 5 больше узла с ключом 3, движемся вправо Добавляемый ключ 5 меньше узла с ключом 6, движемся влево Добавляем ключ 5 В узле с ключом 3: баланс = -2 и ключ добавляемого элемента 5 меньше ключа правого сына6 => выполнем поворот правого сына вправо Выполнение правого поворота относительно узла с ключом 6 6(1) / \ 5(1) 3(1) / 2(1) \ 1(1) Дерево после поворота: 6(1) / 5(1) В узле с ключом 3: баланс = -2 => выполнем поворот влево Выполнение левого поворота относительно узла с ключом 3

		6(1) / 5(1) / 3(1) / 2(1) \ 1(1) Дерево после поворота: 6(1) / 5(1) \ 3(1) Получившееся дерево: 6(1) / 5(1) / \ 3(1) 2(1) \ 1(1)
4.	9 10 8 1 5	Добавляемый элемент: 9 Добавляем ключ 9 Получившееся дерево: 9(1) Добавляемый элемент: 10 Добавляемый ключ 10 больше узла с ключом 9, движемся вправо Добавляем ключ 10 Получившееся дерево: 10(1) / 9(1) Добавляемый элемент: 8 Добавляемый ключ 8 меньше узла с ключом 9, движемся влево Добавляем ключ 8 Получившееся дерево: 10(1) / 9(1) \ 8(1)

Добавляемый элемент: 1
Добавляемый ключ 1 меньше узла с ключом 9, движемся влево
Добавляемый ключ 1 меньше узла с ключом 8, движемся влево
Добавляем ключ 1
Получившееся дерево:
```
    10(1)
  /
9(1)
\
    8(1)
    \
       1(1)
```
Добавляемый элемент: 5
Добавляемый ключ 5 меньше узла с ключом 9, движемся влево
Добавляемый ключ 5 меньше узла с ключом 8, движемся влево
Добавляемый ключ 5 больше узла с ключом 1, движемся вправо
Добавляем ключ 5
В узле с ключом 8:
баланс = 2 и ключ добавляемого элемента 5 больше ключа левого сына1 => выполняем поворот левого сына влево
Выполнение левого поворота относительно узла с ключом 1
```
    10(1)
  /
9(1)
\
    8(1)
    \
          5(1)
        /
       1(1)
```

Дерево после поворота:
```
5(1)
\
    1(1)
```
В узле с ключом 8:
баланс = 2 => выполнем поворот вправо
Выполнение правого поворота относительно узла с ключом 8
```
    10(1)
  /
9(1)
\
    8(1)
    \
       5(1)
```

		\ 1(1) Дерево после поворота: 8(1) / 5(1) \ 1(1) Получившееся дерево: 10(1) / 9(1) \ 8(1) / 5(1) \ 1(1)
5.	9 10 q e	Добавляемый элемент: 9 Добавляем ключ 9 Получившееся дерево: 9(1) Добавляемый элемент: 10 Добавляемый ключ 10 больше узла с ключом 9, движемся вправо Добавляем ключ 10 Получившееся дерево: 10(1) / 9(1)