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

ОТЧЕТ

По лабораторной работе №5
по дисциплине «Алгоритмы и структуры данных»
Тема: **АВЛ-деревья**

Студент гр. 9381 Судаков Е.В.

Преподаватель _____ Фирсов М.А.

_____

Санкт-Петербург
2020

1. Цель работы.

Познакомиться со структурой данных “АВЛ-дерево”

2. Задание.

Вариант 16.

16. БДП: АВЛ-дерево; действие: 1+2б

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

2) Б) Для построенной структуры данных проверить, входит ли в неё элемент e типа Elem, и если входит, то удалить элемент e из структуры данных (первое обнаруженное

вхождение). Предусмотреть возможность повторного выполнения с другим элементом.

3. Основные теоретические положения.

АВЛ-дерево — сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на 1.

АВЛ — аббревиатура, образованная первыми буквами фамилий создателей (советских учёных) Георгия Максимовича Адельсон-Вельского и Евгения Михайловича Ландиса.

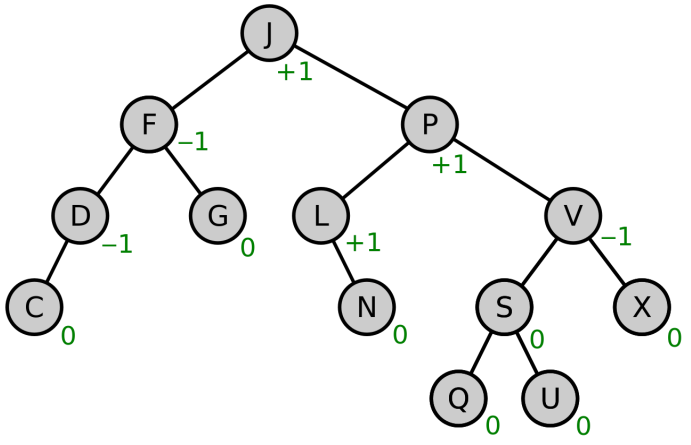

Рисунок 1. Пример АВЛ-дерева

4. Описание алгоритма

Алгоритм добавления элемента следующий:

1. Вставка элемента происходит почти также, как и обычном БДП. Спускаемся по дереву вниз, сравнивая элемент для вставки с элементами дерева.
2. После вставки необходимо **сбалансировать дерево**.

Балансировка дерева происходит, когда разница между высотами поддеревьев одного элемента становится равной 2. В таком случае, в зависимости от конфигурации, необходимо провести серию **вращений**.

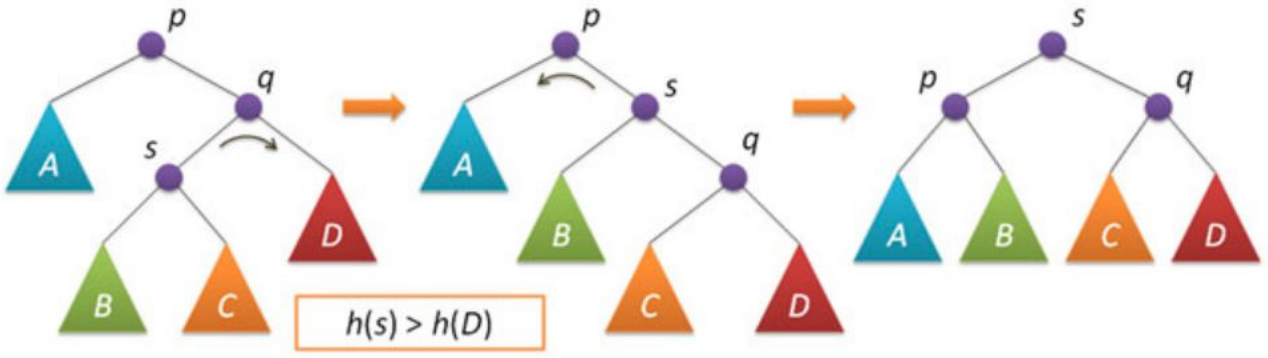

Рисунок 2. Балансировка дерева с помощью правого и левого вращения

Алгоритм удаления элемента :

Находим узел p с заданным ключом k (если не находим, то делать ничего не надо), в правом поддереве находим узел min с наименьшим ключом и заменяем удаляемый узел p на найденный узел min. При каждом выходе из рекурсии необходимо ребалнсировать дерево.

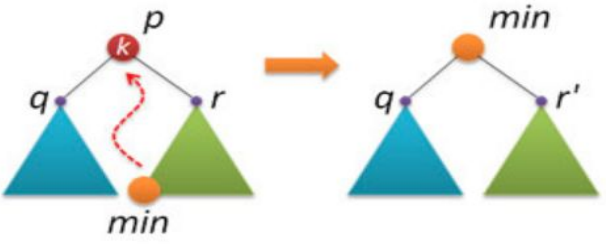

Рисунок 3. Алгоритм удаления элемента

5. Пример работы программы:

Основной тест №1:

Входные данные: Создать дерево с корнем 30. Вставить 10. Вставить 11. Вставить 40. Вставить 35. Вставить 5. Вставить 4. Удалить 11. Вставить 3.

Выходные данные (с промежуточной информацией):

Обратите внимание : дерево выводится *слева-направо*.

```
====================================

Создано АВЛ-дерево с корнем 30


30


==================

Вставка элемента 10 в дерево с корнем 30


30
   10


==================


==================

Вставка элемента 11 в дерево с корнем 30


   30
11
   10


==================


==================

Вставка элемента 40 в дерево с корнем 11
```

```
      40
   30
11
   10

==================

==================
Вставка элемента 35 в дерево с корнем 11

      40
   35
      30
11
   10

==================

==================
Вставка элемента 5 в дерево с корнем 11

      40
   35
      30
11
   10
      5

==================
```

```
==================
Вставка элемента 4 в дерево с корнем 11

      40
   35
      30
11
      10
   5
      4

==================

==================
Удаление элемента 11 из дерева с корнем 11

      40
   35
30
      10
   5
      4

==================

==================

Вставка элемента 3 в дерево с корнем 30
```

```
      40
   35
30
      10
   5
      4
         3

==================

Process finished with exit code 0
```

6. Выполнение программы:

1. Программа в диалоговом режиме предлагает пользователю на выбор одно из трех действий :
   (1) Вставить элемент
   (2) Удалить элемент
   (3) Выход

Рекомендуется работать только с консолью, так как она позволяет использовать цвета, что в данной работе активно используется при выводе информации.

7. Описание функций:

Функции описаны в исходном коде в стиле Javadoc:

int Node::bFactor() - Функция поиска разности между высотами поддеревьем элемента

@return Разница между высотами левого и правого поддерева

`void Node::updateHeight()` - После каждой вставки/балансировки/удаления нужно обновлять высоту дерева

`Node *Node::rotateRight()-` Правое вращение.

@return Node* p - новый корень полученного дерева

`Node *Node::rotateLeft()` - Функция левого вращения.

@return новый корень дерева

`Node *Node::balance()` - Функция балансировки АВЛ дерева.Балансировка нужна в случае когда разница высот левого и правого поддеревьев становится == |2|

@return указатель на самого себя(узла)

`AVLTree::AVLTree(int k)` - Конструктор АВЛ-дерева

@param k ключ для рута

`void AVLTree::printTree(Node *node, int level)` - Служебная функция вывода дерева. Выводит дерево не сверху-вниз, а слева-направо

@param node корень выводимого поддерева

@param level уровень рекурсии для индентации

`Node *AVLTree::insert(Node *node, int key)` - Вставка элемента. Единственное отличие от вставки в простое БДПв том, что в конце необходимо балансировать.

@param node корень дерева, куда добавляем

@param key ключ элемента

@return Корень сбалансированного дерева

Node *AVLTree::remove(Node *node, int key) - Функция удаления элемента с заданным ключом находим узел p с заданным ключом k (если не находим, то делать ничего не надо), в правом поддереве находим узел min с наименьшим ключом и заменяем удаляемый узел p на найденный узел min.

@param node корень дерева, в котором происходит удаление элемента

@param key ключ для удаления

@return ребалансированный корень дерева

Node *AVLTree::findMin(Node *node) - Функция поиска минимального элемента в (под)дереве

@param node корень дерева, где ищется минимум

@return указатель на элемент с наименьшим ключем

Node *AVLTree::removeMin(Node *node)- Удаление минимального элемента из заданного дерева. По свойству АВЛ-дерева у минимального элемента справа либо подвешен узел, либо там пусто. В обоих случаях надо просто вернуть указатель на правый узел и по пути назад (при возвращении из рекурсии) выполнить балансировку

@param node корень (под)дерева, где удаляется минимальный элемент

@return указатель на новый корень после балансировки

Node *AVLTree::insertNode(Node *root, int key) - Служебная функция-обертка над вставком для удобного вывода

@param root корень (под)дерева, куда вставляется элемент

@param key ключ элемента для вставки

@return (возможно обновленный) корень поддерева

`Node *AVLTree::removeNode(Node *root, int key)` - Синтаксический сахар над remove.

@param root (под)дерево, в котором удалется элемент

@param key элемент для удаления

@return корень дерева, где удаляли элемент

8. Описание структур данных

class Node - Представление узла дерева

class AVLTree - Класс представления АВЛ-дерева. Является по-сути всего лишь синтаксическим сахаром над Node.

9. Вывод:

В ходе выполнения лабораторной работы была изучена и реализована на языка C++ структура данных АВЛ-дерево.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл main.cpp:

```
#include <bits/stdc++.h>


using namespace std;


#ifdef __linux__

    #define REMOVE_COLOR "\033[1m\033[31m"

    #define INSERT_COLOR "\033[1m\033[32m"

    #define RESET_COLOR   "\033[0m"


#elif __WIN32

    #define REMOVE_COLOR ""

    #define INSERT_COLOR ""

    #define RESET_COLOR  ""

#endif


/*

* Вариант 16. БДП: АВЛ-дерево

* Действия : 1. По заданной последовательности элементов построить дерево

* 2 б) : Для построенной структуры данных проверить, входит ли в неё элемент, и если

* входит, то удалить элемент из дерева. Предусмотреть возможность повторного выполнения с
другим

* Элементом

*/


/**

 * Представление узла дерева

 */

class Node {
```

```
    int key;
    int height;
    Node *left;
    Node *right;

    Node *rotateRight();

    Node *rotateLeft();

public:
    Node(int k) : key(k), left(0), right(0), height(1) {}

    int getHeight();

    int bFactor();

    void updateHeight();

    Node *balance();

    Node *getRight();

    Node *getLeft();

    void setLeft(Node *node);

    void setRight(Node *node);

    void setKey(int key);

    int getKey();
```

```
};


int Node::getHeight() {
    return this ? this->height : 0;
}


/**
 *
 * @return Разница между высотами левого и правого поддерева
 */
int Node::bFactor() {
    return this->right->getHeight() - this->left->getHeight();
}


/**
 * После каждой вставки/балансировки/удаления нужно
 * обновлять высоту дерева
 */
void Node::updateHeight() {
    int hl = this->left->getHeight();
    int hr = this->right->getHeight();
    this->height = max(hl, hr) + 1;
}


/**
 * Правое вращение.
 *      y                               x
```

```
 *     / \     Правое вращение         /  \
 *    x   T3   - - - - - - - >        T1   y
 *   / \       < - - - - - - -            / \
 *  T1  T2     Левое вращение           T2  T3
 * @return Node* p - новый корень полученного дерева
 */
Node *Node::rotateRight() {
    Node *newRoot = this->left;
    this->left = newRoot->right;
    newRoot->right = this;
    this->updateHeight();
    newRoot->updateHeight();
    return newRoot;
}



/**
 * Функция левого вращения.
 * @return новый корень дерева
 */
Node *Node::rotateLeft() {
    Node *newRoot = this->right;
    this->right = newRoot->left;
    newRoot->left = this;
    this->updateHeight();
    newRoot->updateHeight();
    return newRoot;
}

/**
 * Функция балансировки АВЛ дерева.
```

```
 * Балансировка нужна в случае когда разница высот левого и

 * правого поддеревьев становится == |2|

 * @return указатель на самого себя(узла)

 */

Node *Node::balance() {

    this->updateHeight();

    int diff = this->bFactor();

    if (diff == 2) {

        if (this->right->bFactor() < 0) { // высота правого внука меньше высоты левого
внука

            this->right = this->right->rotateRight();

        }

        return this->rotateLeft(); // иначе - правый внук больше либо равен левому и
простое вращение

    } else if (diff == -2) {

        if (this->left->bFactor() > 0) {

            this->left = this->left->rotateLeft();

        }

        return this->rotateRight();

    }

    return this;

}




Node *Node::getRight() {

    return this ? this->right : nullptr;

}




Node *Node::getLeft() {

    return this ? this->left : nullptr;

}
```

```
int Node::getKey() {
    return this ? this->key : 0;
}


void Node::setLeft(Node *node) {
    if (this)
        this->left = node;
}


void Node::setRight(Node *node) {
    if (this)
        this->right = node;
}

void Node::setKey(int key) {
    if (this)
        this->key = key;
}


/**
 * Класс представления АВЛ-дерева. Является по-сути всего лишь
 * синтаксическим сахаром над Node.
 */
class AVLTree {
public:
    Node *root;
```

```
    AVLTree(int k);

    void printTree(Node *node, int level);

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

    Node *findMin(Node *node);

    Node *removeMin(Node *node);

    Node *remove(Node *node, int key);

    Node *insertNode(Node *root, int key);

    Node *removeNode(Node *root, int key);
};

/**
 * Конструктор АВЛ-дерева
 * @param k ключ для рута
 */
AVLTree::AVLTree(int k) {
    cout << INSERT_COLOR << "Создано АВЛ-дерево с корнем " << k << "\n\n";
    this->root = new Node(k);
    this->printTree(this->root, 0);
    cout << RESET_COLOR;
}

/**
 * Служебная функция вывода дерева.
```

```
 * Выводит дерево не сверху-вниз, а слева-направо
 * @param node корень выводимого поддерева
 * @param level уровень рекурсии для индентации
 */
void AVLTree::printTree(Node *node, int level) {
    if (node) {
        printTree(node->getRight(), level + 1);
        for (int i = 0; i < level; i++) cout << "   ";
        cout << node->getKey() << endl;
        printTree(node->getLeft(), level + 1);
    }
}



/**
 * Вставка элемента. Единственное отличие от вставки в простое БДП
 * в том, что в конце необходимо балансировать.
 * @param node корень дерева, куда добавляем
 * @param key  ключ элемента
 * @return Корень сбалансированного дерева
 */
Node *AVLTree::insert(Node *node, int key) {
    if (node == nullptr) return new Node(key);
    if (key < node->getKey()) {
        node->setLeft(insert(node->getLeft(), key));
    } else if (key > node->getKey()) { // не нужно вставлять дубликаты, согласно
варианту.
        node->setRight(insert(node->getRight(), key));
    }
    return node->balance();
}
```

```
/**
 * Функция удаления элемента с заданным ключиом
 * находим узел p с заданным ключом k
 * (если не находим, то делать ничего не надо),
 * в правом поддереве находим узел min с наименьшим ключом
 * и заменяем удаляемый узел p на найденный узел min.
 * @param node корень дерева, в котором происходит удаление элемента
 * @param key ключ для удаления
 * @return ребалансированный корень дерева
 */
Node *AVLTree::remove(Node *node, int key) {
    if (node == nullptr) {
        return nullptr;
    }
    if (key < node->getKey()) {
        node->setLeft(removeMin(node->getLeft()));
    } else if (key > node->getKey()) {
        node->setRight(removeMin(node->getRight()));
    } else { // key == node->getKey()
        Node *right = node->getRight();
        Node *left = node->getLeft();
        delete node;
        if (!right) return left;
        Node *min = findMin(right);
        min->setRight(removeMin(right));
        min->setLeft(left);
        return min->balance();
    }
    return node->balance();
}
```

```
/**
 * Функцяи поиска минимального элемента в (под)дереве
 * @param node корень дерева, где ищется минимум
 * @return указатель на элемент с наименьшим ключем
 */
Node *AVLTree::findMin(Node *node) {
    return node->getLeft() ? findMin(node->getLeft()) : node;
}

/**
 * Удаление минимального элемента из заданного дерева.
 * по свойству АВЛ-дерева у минимального элемента справа
 * либо подвешен  узел, либо там пусто.
 * В обоих случаях надо просто вернуть указатель на правый
 * узел и по пути назад (при возвращении из рекурсии)
 * выполнить балансировку
 * @param node корень (под)дерева, где удаляется минимальный элемент
 * @return указатель на новый корень после балансировки
 */
Node *AVLTree::removeMin(Node *node) {
    if (node->getLeft() == nullptr) {
        return node->getRight();
    }
    node->setLeft(removeMin(node->getLeft()));
    return node->balance();
}

/**
 * Служебная функция-обертка над вставком для удобного вывода
```

```
 * @param root корень (под)дерева, куда вставляется элемент

 * @param key  ключ элемента для вставки

 * @return  (возможно обновленный) корень поддерева

 */

Node *AVLTree::insertNode(Node *root, int key) {

    cout << INSERT_COLOR;

    cout << "\n==================\n";

    cout << "Вставка элемента " << key << " в дерево с корнем " << root->getKey() <<
"\n\n";

    root = this->insert(root, key);

    this->printTree(root, 0);

    cout << "\n==================\n" << RESET_COLOR;

    return root;

}



/**

 * Синтаксический сахар над remove.

 * @param root (под)дерево, в котором удалется элемент

 * @param key элемент для удаления

 * @return корень дерева, где удаляли элемент

 */

Node *AVLTree::removeNode(Node *root, int key) {

    cout << REMOVE_COLOR;

    cout << "\n==================\n";

    cout << "Удаление элемента " << key << " из дерева с корнем " << root->getKey() <<
"\n\n";

    root = this->remove(root, key);

    this->printTree(root, 0);

    cout << "\n==================\n" << RESET_COLOR;

    return root;

}
```

```
void printMenu() {
    cout << "\n\n==================\n"
            "(1) Вставить элемент\n"
            "(2) Удалить элемент\n"
            "(3) Выход\n\n";
}

AVLTree *processUserInput(AVLTree *tree) {
    int f, userKey;
    printMenu();
    cin >> f;
    switch (f) {
        case 1:
            cout << "Введите элемент : \n";
            cin >> userKey;
            if (tree) {
                tree->root = tree->insertNode(tree->root, userKey);
            } else {
                tree = new AVLTree(userKey);
            }
            break;
        case 2:
            if (tree) {
                cout << "Введите элемент : \n";
                cin >> userKey;
                tree->root = tree->removeNode(tree->root, userKey);
            } else cout << "В дереве нет элементов! \n";
            break;
        case 3:
            exit(0);
```

```
    }
    return tree;
}

int main() {
    AVLTree *tree;
    while (true) {
        tree = processUserInput(tree);
    }
    return 0;
}
```