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-дерево

Студент гр. 9382 _____ Субботин М. О.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург

2020

Цель работы.

Реализовать АВЛ-дерево, разобраться со свойствами дерева и научиться производить вставку элементов.

Задание

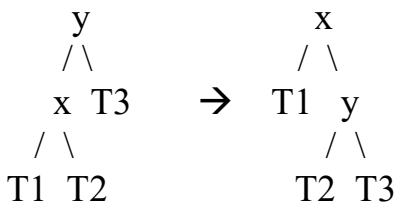
15. БДП: АВЛ-дерево; действие: 1+2а
По заданной последовательности элементов Elem построить структуру данных определённого типа – БДП или хеш-таблицу
а) Для построенной структуры данных проверить, входит ли в неё элемент e типа Elem, и если входит, то в скольких экземплярах. Добавить элемент e в структуру данных. Предусмотреть возможность повторного выполнения с другим элементом.

Описание АВЛ-дерева и функций

АВЛ-дерево в первую очередь бинарное дерево поиска, т.е. для каждой вершины справедливо: все элементы правого поддерева строго больше, чем вершина, и все элементы из левого поддерева строго меньше, чем вершина. Кроме этого, АВЛ-дерево сбалансированно и существует показатель баланса для каждой вершины. Он высчитывается как разность между высотой левого поддерева и высотой правого поддерева. Тем самым, дерево считается сбалансированным, если для каждой вершины этот критерий равен -1, 0, 1, в противном случае дерево не сбалансированно и над ним надо производить необходимые преобразования.

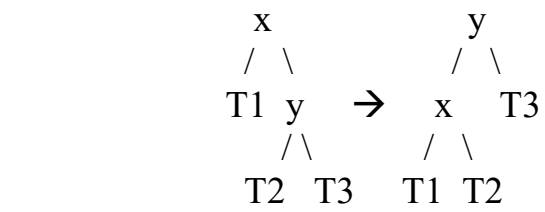
Следует определить два преобразования дерева: поворот направо и поворот налево.

Поворот направо:



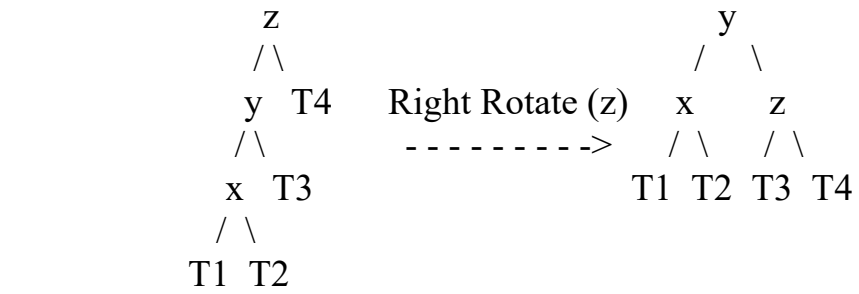
T1, T2, T3 – поддеревья.
Мы буквально взяли, подвесили это дерево за вершину x, и немного передвинули поддерево T2.

Поворот налево:



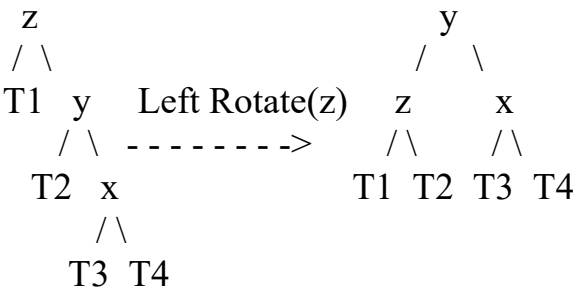
Теперь же мы как будто подвесили дерево за вершину y и переместили T2.

Если какая-то из вершин не сбалансированно, может получиться 4 случая.
Left Left:



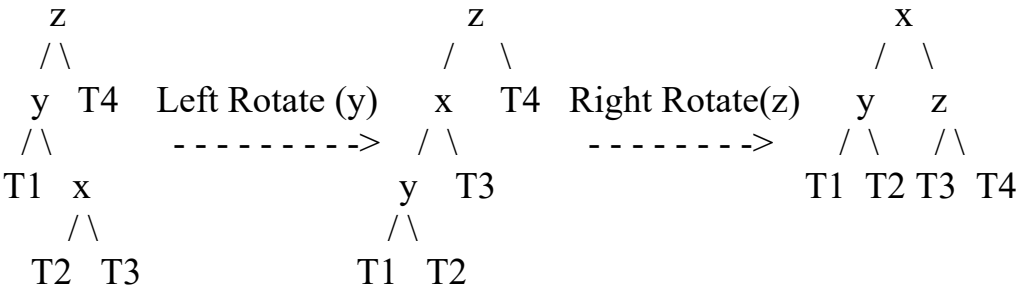
В этом случае вершина z имеет критерий равный 2, и она не сбалансирована. Для того, чтобы сбалансировать эту вершину, следует повернуть дерево направо в этой вершине.

Right Right:



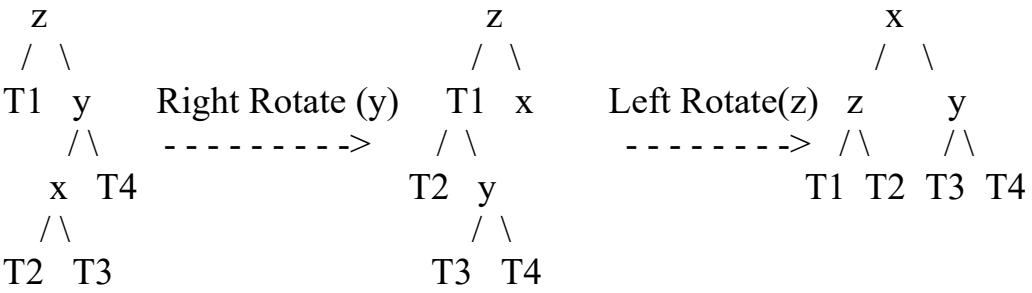
Здесь такая же ситуация с вершиной z, только теперь ее критерий равен -2. Для того, чтобы сбалансировать в таком случае, следует повернуть дерево в этой вершине налево.

Left Right:



z имеет критерий баланса равный 2, так же как и в случае LL, но как тогда их различать? Дело в том, что дерево может стать не сбалансированным только после вставки элемента. В этом случае x это и есть тот самый вставляемый элемент. Если он находится слева от левого ребенка вершины z, то это случай LL, если справа, то LR.
Случай LR решается с помощью двух поворотов, сначала левый поворот для вершины y, затем правый для вершины z.

Right Left:



Случай RL решается также с помощью двух поворотов: правый для вершины y и левый для вершины z.

Вставка элемента в АВЛ-дерево происходит таким образом. Сначала новый элемент вставляется, как и в бинарном дереве поиска, т.е. если значение текущего элемент меньше, чем вставляемого – идем в правого ребенка, если больше, то в левого, как только оказались на пустом месте, вставляем на это место нашу новую вершину. Мы могли также наткнуться на вершину с равным вставляемому элементом, тогда мы бы просто увеличил количество дубликатов этой вершины на 1. В случае же, когда мы все-таки вставили новый экземпляр вершины в дереве, нужно проверить, не поломали ли мы баланс. Для этого, наверное, лучшим решением будет вставка элемента с помощью рекурсии и вот когда мы будем возвращаться обратно по этой рекурсии и будем проверять баланс вершин. Как-бы нам нужно проверить ту “часть” дерева, которая могла сломаться, нет смысла проверять критерий баланса для каждой вершины дерева. Возвращаясь по рекурсии, будем проверять критерии вершин, если они

не сбалансированы, то применяем вышеописанные методы. Таким образом мы сбалансируем все дерево.

Описание структуры АВЛ-дерева

```
template<typename Elem>
class Node{
public:
   Elem key;
   Node *left;
   Node *right;
   int height;
   int count;
};
```

Elem key – значение самого элемента вершины.
Node *left – указатель на левого ребенка вершины.
Node *right – указатель на правого ребенка вершины.
int height – высота вершины.
int count – количество “экземпляров” данного значения в дереве.

Описание функций АВЛ-дерева

Node<Elem>* newNode(Elem key) – функция создания новой вершины.

Elem key – элемент для вставки в вершину.

int height(Node<Elem> *node) – функция для вычисления высоты вершины

Node<Elem> *node – указатель на вершину, для которой считается высота

Node<Elem> *rightRotate(Node<Elem> *y, int depth) – функция поворота дерева направо

Node<Elem> *y – указатель на вершину с которой производится поворот.

int depth – глубина рекурсии, для корректного отображения промежуточных вычислений.

Node<Elem> *leftRotate(Node<Elem> *x, int depth) – функция для поворота дерева налево
Node<Elem> *x – указатель на вершину, с которой будет производится поворот
int depth – глубина рекурсии, для корректного отображения промежуточных вычислений.

int getBalance(Node<Elem> *node) – функция вычисляет критерий баланса для заданной вершины Node<Elem> *node

Node<Elem>* insert(Node<Elem> *node, Elem key, int depth) – функция вставляет элемент в дерево
Node<Elem> *node – текущая вершина, которая участвует в поиске места для вставки элемента.
Elem key – значение вставляемого элемента
int depth - глубина рекурсии, для корректного отображения промежуточных вычислений.

Node<Elem>* countElemAndInsert(Node<Elem>* root, Elem elem) – функция для определения есть ли вставляемый элемент в дереве, определения количества таких элементов и вставка элемента
Node<Elem>* root – корень дерева
Elem elem – вставляемый элемент

void preOrder(Node<Elem>* root) – функция для отображения дерева
Node<Elem>* root – элемент с которого начинается отображение.

Тестирование.

№	Входные данные	Выходные данные
1	3 3 2 1	2(1) 1(1) # # 3(1) # #
2	3 4 2 3	3(1) 2(1) # # 4(1) # #
3	3 1 2 3	2(1) 1(1) # # 3(1) # #
4	3 1 3 2	2(1) 1(1) # # 3(1) # #
5	7 10 20 30 40 50 25 10	30(1) 20(1) 10(2) # # 25(1) # # 40(1) # 50(1) # #
6	5 10 10 10 10 10	10(5) # #
7	-1	Некорректное значение количества вершин в дереве!

Обработка результатов тестирования.

Программа выдает корректные результаты на всех тестах. Дерево в выходных данных представлено с помощью прямого обхода, где символом # обозначены пустые листья. Также в скобках указано количество дупликатов вершин данного значения. В первых четырех тестах было проверено, как работают повороты в 4 случаях. 5 тест проверял как программа отработает на дереве с несколькими операциями перебалансировки. 6 тест предназначен для проверки работы с дупликатами и 7 тест был на некорректных данных.

Выводы.

Было реализовано АВЛ-дерево, были разобраны свойства дерева и реализована вставка элемента в дерево.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

template<typename Elem>
class Node{
public:
    Elem key;
    Node *left;
    Node *right;
    int height;
    int count;
};
template<typename Elem>
void preOrder(Node<Elem>* root);

void shift(int depth){
    for(int i = 0; i<depth; i++){
        cout << "   ";
    }
}

int max(int a, int b){
    return (a>b)? a : b;
}

template<typename Elem>
int height(Node<Elem> *node){
    if(node == nullptr){
        return 0;
    }
    return node->height;
}

template<typename Elem>
Node<Elem>* newNode(Elem key){
    Node<Elem>* node = new Node<Elem>();
    node->key = key;
```

```
    node->left = nullptr;
    node->right = nullptr;
    node->height = 1;
    node->count = 1;
    return node;
}
/*
 * Функция правого поворота
 *          y                              x
 *         / \                            /  \
 *        x  T3       --->               T1   y
 *      /  \                                /  \
 *     T1  T2                              T2  T3
 */
template<typename Elem>
Node<Elem> *rightRotate(Node<Elem> *y, int depth){
    Node<Elem> *x = y->left;
    Node<Elem> *T2 = x->right;

    //поворачиваем
    x->right = y;
    y->left = T2;

    //подгоняем высоту
    y->height = max(height(y->left), height(y->right) + 1);
    x->height = max(height(x->left), height(x->right) + 1);

    shift(depth);
    cout << "Повернутое дерево вправо: ";
    preOrder(x);
    cout << endl;

    return x;
}
/*
 * Функция левого поворота
 *        x                             y
 *      /   \                         /   \
 *     T1    y      --->             x     T3
 *          / \                     /   \
 *        T2   T3                 T1    T2
 *
```

```
 */
template<typename Elem>
Node<Elem> *leftRotate(Node<Elem> *x, int depth){
    Node<Elem> *y = x->right;
    Node<Elem> *T2 = y->left;

    //поворачиваем
    y->left = x;
    x->right = T2;

    //подгоняем высоту
    x->height = max(height(x->left), height(x->right) + 1);
    y->height = max(height(y->left), height(y->right) + 1);

    shift(depth);
    cout << "Повернутое дерево влево: ";
    preOrder(y);
    cout << endl;

    return y;
}

template<typename Elem>
int getBalance(Node<Elem> *node){
    if(node ==  nullptr){
        return 0;
    }
    return height(node->left) - height(node->right);
}


template<typename Elem>
Node<Elem>* insert(Node<Elem> *node, Elem key, int depth){
    //Если мы в несуществующей вершине, то на ее место вставим новую
    if(node == nullptr){
        shift(depth);
        cout << "Вставляем вершину " << key << endl;
        return newNode<Elem>(key);
    }

    //Если вставляемый элемент уже есть в дереве, то просто инкрементим
поле count
```

```
        if(key == node->key){
            node->count +=1;
            shift(depth);
            cout << "Такая вершина уже существует в дереве, всего вершин с ключём "<< key <<" = " << node->count << endl;
            return node;
        }

        //Здесь мы передвигаемся по правилам BST
        if(key < node->key){
            shift(depth);
            cout << "Значение " << key << " новой вершины меньше, чем у текущей вершины " << node->key << " ,идем к левому ребенку "<< endl;
            node->left = insert(node->left, key, depth+1);
        }
        else if(key > node->key){
            shift(depth);
            cout << "Значение " << key << " новой вершины больше, чем у текущей вершины " << node->key << " ,идем к правому ребенку "<< endl;
            node->right = insert(node->right, key, depth+1);
        }

        //Задаем высоту  вершины
        node->height = 1 + max(height(node->left), height(node->right));

        int balance = getBalance(node);
        shift(depth);
        cout << "Критерий баланса вершины " << node->key << " равен " << balance << endl;

        /*
         * Если balance > 1, значит проблема в левом поддереве
         * Если ключ меньше чем левый ребенок рассматриваемой вершины то это
         * случай Left Left
         */
        if(balance > 1 && key < node->left->key){
            shift(depth);
            cout << "LL случай, в поддереве: ";
            preOrder(node);
            cout << "поворачиваем дерево вправо с началом в вершине: " << node->key<< endl;
            return rightRotate(node, depth);
```

```
        }

        /*
         * Если balance < -1, значит проблема в правом поддереве
         * Если ключ больше чем правый ребенок рассматриваемой вершины то это
         * случай Right Right
         */
        if(balance < -1 && key > node->right->key){
            shift(depth);
            cout << "RR случай, в поддереве: ";
            preOrder(node);
            cout << " поворачиваем дерево влево с началом в вершине: " <<
node->key<< endl;
            return leftRotate(node, depth);
        }

        /*
         * Если balance > 1, значит проблема в левом поддереве
         * Если ключ больше чем левый ребенок рассматриваемой вершины то это
         * случай Left Right
         */
        if(balance > 1 && key > node->left->key){
            shift(depth);
            cout << "LR случай, в поддереве: ";
            preOrder(node);
            cout << " с началом в вершине: " << node->key<< endl;
            shift(depth);
            cout << "Сначала поворачиваем дерево влево в вершине " << node-
>left->key << endl;
            node->left = leftRotate(node->left, depth);
            shift(depth);
            cout << "Затем поворачиваем дерево вправо в вершине " << node->key
<< endl;
            return rightRotate(node, depth);
        }

        /*
         * Если balance < -1, значит проблема в правом поддереве
         * Если ключ меньше чем правый ребенок рассматриваемой вершины то это
         * случай Right Left
         */
        if(balance < -1 && key < node->right->key){
```

```
            shift(depth);
            cout << "RL случай, в поддереве: ";
            preOrder(node);
            cout << "с началом в вершине: " << node->key<< endl;
            shift(depth);
            cout << "Сначала поворачиваем дерево вправо в вершине " << node-
>right->key << endl;
            node->right = rightRotate(node->right, depth);
            shift(depth);
            cout << "Затем поворачиваем дерево влево в вершине " << node->key
<< endl;
            return leftRotate(node, depth);
        }
        shift(depth);
        cout << "Вершина сбалансированна" << endl;
        return node;
    }

    template<typename Elem>
    Node<Elem>* countElemAndInsert(Node<Elem>* root, Elem elem){
        Node<Elem> *tempNode = root;
        //Передвигаемся по дереву и ищем дубликат
        while(tempNode!=nullptr){
            if(tempNode->key == elem){
                cout << "Вершина " << elem << " уже существует и ее дупликатов
было " << tempNode->count << endl;
                tempNode->count++;
                return root;
            }
            if(tempNode->key < elem){
                tempNode = tempNode->right;
            }
            else if(tempNode->key > elem){
                tempNode = tempNode->left;
            }
        }
        //вставляем новый ключ
        cout << "Такой вершины " << elem << " нет в дереве, вставляем ее в
дерево: " << endl;
        preOrder(root);
        cout << endl;
        root = insert(root, elem, 0);
```

```
    cout << "Дерево после вставки: ";
    preOrder(root);
    return root;
}

template<typename Elem>
void preOrder(Node<Elem>* root){
    if(root!=nullptr){
        cout << root->key << "(" << root->count << ") ";
        preOrder(root->left);
        preOrder(root->right);
    }
    else {
        cout << "# ";
    }
}


int main() {
    Node<int> *root = nullptr;

    string ch = "0";
    while(ch!="f" && ch!="c"){
        cout <<"Введите c/f для ввода из консоли/файла: ";
        cin >> ch;
    }

    if(ch == "f") {
        string filename = "../in.txt";
        ifstream myfile(filename);

        int n;
        myfile >> n;
        if( n < 0) {
            cout << "Некорректное значение количества вершин в дереве!";
            return 0;
        }
        for(int i = 0; i<n; i++){
            int key;
            myfile >> key;
            cout << "Хотим добавить вершину с ключём " << key << endl;
            root = insert(root, key, 1);
```

```
                cout << "Сбалансированное дерево после вставки: ";
                preOrder(root);
                cout << endl << endl;
            }

            myfile.close();
        }
        else if(ch == "c"){
            cout << "Введите сначала количество элементов в дереве и затем
сами вершины: ";
            int n;
            cin >> n;
            if( n < 0) {
                cout << "Некорректное значение количества вершин в дереве!";
                return 0;
            }
            for(int i = 0; i<n; i++){
                int key;
                cin >> key;
                cout << "Хотим добавить вершину с ключём " << key << endl;
                root = insert(root, key, 1);
                cout << "Сбалансированное дерево после вставки: ";
                preOrder(root);
                cout << endl << endl;
            }
        }

        ch = "y";
        while(ch == "y") {
            cout << endl << "Если хотите/не хотите добавить вершину нажмите
y/n: ";
            cin >> ch;
            if( ch == "y"){
                cout << "Введите вершину, которую хотите добавить: ";
                int elem;
                cin >> elem;
                root = countElemAndInsert(root,elem);
            }
        }

        return 0;
    }
```