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

КУРСОВАЯ РАБОТА

ПО ДИСЦИПЛИНЕ «АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ»

ТЕМА: ИССЛЕДОВАНИЕ AVL И B ДЕРЕВЬЕВ

Студент гр. 1384 _____ Шушков Е.В.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург
2022

ЗАДАНИЕ

НА КУРСОВУЮ РАБОТУ

Студент Шушков Е.В.

Группа 1384

Тема работы: исследование AVL и B деревьев

Задание: "Исследование" - реализация требуемых структур данных/алгоритмов; генерация входных данных (вид входных данных определяется студентом); использование входных данных для измерения количественных характеристик структур данных, алгоритмов, действий; сравнение экспериментальных результатов с теоретическими. Вывод промежуточных данных не является строго обязательным, но должна быть возможность убедиться в корректности алгоритмов.

Студент_____Шушков Е.В.

Преподаватель_____Иванов Д.В.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В ходе данной работы мы изучим и сравним такие структуры данных, как: ABV-дерево и B-дерево. Данные деревья являются сбалансированными и имеют сложность базовых операций вставки, удаления и поиска O(n).

В результате работы мы узнаем, чем отличаются эти деревья и насколько они эффективны на практике с помощью их реализации на языке программирования C++. Также сравним результаты их работы.

1. ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ ДАННЫЕ

1.1) B-дерево.

B-дерево – это сбалансированное дерево, особенности которого заключаются в том, что каждый узел может хранить более одного ключа и иметь более двух дочерних элементов. Благодаря таким особенностям структуры, высота дерева значительно уменьшается, что обеспечивает более быстрый доступ к диску.

Основные свойства, используемые в работе:

- Ключи в каждом узле упорядочены по неубыванию.
- В каждом узле есть логическое значение leaf. Оно истинно, если узел является листом.
- Если узел является листом, то у него не может быть потомков.
- Если узел не является листом, то у него (n+1) потомков, где n – кол-во элементов.
- Если у узла есть дети, то первый ребёнок содержит значения от (-inf, k1), следующие от (ki,k(i+1)), последний (kn, +inf), где ki - ключи родителя, а n – кол-во элементов в родителе.
- Каждые узел, кроме корня, содержит не менее t-1 ключей, а каждый внутренний узел имеет как минимум t дочерних узлов, где t – минимальная степень B-дерева.
- Каждый узел содержит не более 2t-1 ключей, а внутренний – не более 2t.
- Все листья находятся на одном уровне, т.е. обладают одинаковой глубиной, равной высоте дерева.

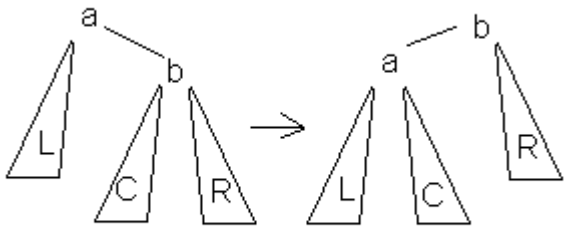
При операциях удаления и вставки может происходить разбиение нашего узла таким образом, чтобы можно было передать значения детям и все свойства B-дерева сохранились. Операции вставка, удаление и поиск выполняются за O(log(N)).

1.2) AVL-дерево.

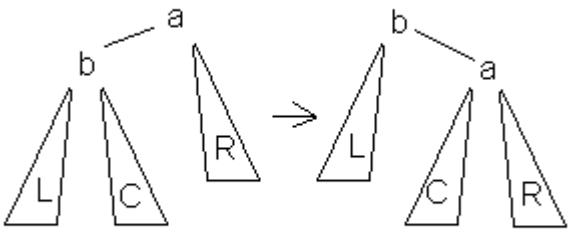
AVL-дерево – сбалансированное по высоте бинарное дерево поиска, такое что для каждой вершины высота двух её поддеревьев различается не более, чем на 1.

Для балансировки дерева используются 4 способа:

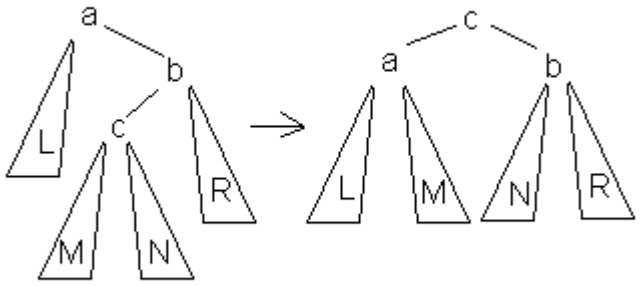
1) Малое левое вращение – используется тогда, когда разница высот a-поддерева и b-поддерева равна 2 и высота C <= высота R.



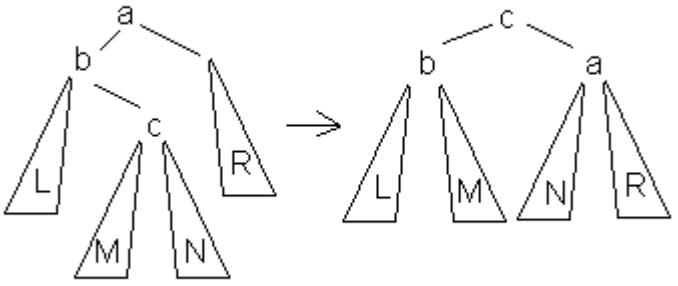
2) Малое правое вращение – используется тогда, когда разница высот a-поддерева и b-поддерева равна 2 и высота C <= высота L.



3) Большое левое вращение – используется тогда, когда разница высот a-поддерева и b-поддерева равна 2 и высота c-поддерева > высота R.



4) Большое правое вращение – используется тогда, когда разница высот a-поддерева и b-поддерева равна 2 и высота c-поддерева > высота L.



В каждом случае операция балансировки уменьшает полную высоту не более чем на 1, тем более, не может увеличивать её.

Также можно заметить, что большие вращения – это комбинация малых для разных поддеревье. Этот факт помогает упростить реализацию AVL-дерева.

Т. к. дерево сбалансировано, высота дерева – это O(log(N)), где N – количество вершин. Таким образом, операция добавления требует O(log(N)) операций.

2. РЕАЛИЗАЦИЯ

2.1) Реализация B-дерева:

Struct BtreeNode – класс узла B-дерева, содержит в себе такие поля, как bool leaf – является ли узел листом, int* keys – список ключей, BTreeNode** childrens – список детей, int size – размер текущего списка ключей. В целях удобства при использовании рекурсии все основные методы реализованы через этот класс.

Методы:

- void splitChild(int idx, BtreeNode* node) – метод с помощью, которого заполненный узел разбивается на два незаполненных, при этом средний ключ попадает в узел-родитель.
- void insertNonFull(int k) – метод, который вставляет ключ в незаполненный узел. При этом может использоваться метод splitChild с целью сохранить свойства дерева.
- BtreeNode* search(int key) – метод, который обходит дерево и ищет в нём ключ. Возвращается указатель на узел, где находится ключ или NULL, если такого нет.
- void deletion(int key) – метод, который удаляет ключ из дерева. Может быть несколько ситуаций в зависимости от расположения ключа в дереве:
    1) Ключ находится в листе. Если удаление не нарушает свойство минимальных количества элементов, просто убираем. Если нарушает, то заимствуется ключ из ближайшего соседнего дочернего узла в порядке слева направо. Если оба соседа имеют минимальное кол-во элементов, то происходит слияние с одним из них через родительский узел.
    2) Ключ находится не в листе. В такой ситуации внутренний узел, который удаляется, заменяется предшественником по порядку, если у левого дочернего элемента больше минимального количества ключей. Или приемником по порядку, если аналогичное условие у

правого дочернего элемента. Если оба элемента имеют минимально кол-во ключей, то происходит их слияние.

3) Два предыдущих случая не изменяют высоту дерева. Если же удаление минимального элемента влечёт её уменьшение, то ищем приемника или предшественник с помощью in-order обхода и заимствуем. Если оба имеют минимальное количество элементов, то ищем брата для заимствования. Если и братья имеют минимальное количество элементов, то тогда объединяются узел с родственным узлом вместе с родительским.

Следующие методы используются для удаления ключа:

- void removeFromLeaf(int idx);
- void removeFromNonLeaf(int idx);
- int getPredecessor(int idx);
- int getSuccessor(int idx);
- int findIdxKeyInNode(int key);
- void merge(int idx);
- void fill(int idx);
- void borrowFromPrev(int idx);
- void borrowFromNext(int idx);

- void traverse() – метод, который выводит все значения в дереве.

Class BTree хранит в себе корень дерева root и отвечает за применение операций вставки, удаления, поиска для корня.

2.2) Реализация AVL-дерева

Struct AVLnode хранит в себе ключ key, ссылки на левого и правого (left и right) ребёнка, а также height – высота дерева.

Class AVLtree хранит в себе корень дерева, а также имеет методы, реализовывающие функционал дерева, при этом большинство функций не изменяет дерево, а просто возвращает копию его изменённого корня.

- int depth(AVLnode* node) – проверяет глубину листа, чтобы избежать обращения к NULL детям.
- int getBalance(AVLnode* node) – возвращает коэффициент баланса, т. е. разницу между высотой левого и правого поддерева. От него зависит выбор поворота при балансировке.
- AVLnode* rightRotate(AVLnode* y_node) – малый правый поворот.
- AVLnode* leftRotate(AVLnode* x_node) – малый левый поворот
- AVLnode* minValueNode(AVLnode* node) – ищет минимальный ключ, который находится в самом левом листе. Этот метод будет использовать при удалении элемента.
- AVLnode* insert(AVLnode* node, int key) – вставка ключа в дерево.
- AVLnode* search(AVLnode* node, int key) – обход дерева для нахождения узла, где хранится ключ. Если ключа нет, то возвращается NULL.
- AVLnode* deletion(AVLnode* root, int key) – удаление элемента из дерева.
- void inOrderPrint(AVLnode* root) и void print(AVLnode* root, int lvl) – методы необходимые для вывод дерева. Первый совершает in-order обход и выводит ключи в данном порядке. Второй метод имеет по сути такой же обход, но выводит дерево визуально в горизонтальном направление.

3. ТЕОРЕТИЧЕСКАЯ ОЦЕНКА СТРУКТУР

3.1) B-дерево

	В лучшем	В среднем	В худшем
Вставка	O(1)	O(log(n))	O(log(n))
Удаление	O(1)	O(log(n))	O(log(n))
Поиск	O(1)	O(log(n))	O(log(n))

Все операции зависят от высоты дерева h, а т.к. дерево сбалансированное, то сложность всех операция равна O(h) = O(log(n)). Лучшая при пустом дереве или высота равна 1.

3.2) AVL-дерево

	В лучшем	В среднем	В худшем
Вставка	O(1)	O(log(n))	O(log(n))
Удаление	O(1)	O(log(n))	O(log(n))
Поиск	O(1)	O(log(n))	O(log(n))

Аналогичная ситуация для AVL-дерева.

Но сложность не означает, что время выполнения операций у этих двух структур одинаковое. Это можно легко увидеть при анализе экспериментальных данных.

4. СРАВНЕНИЕ С ЭКСПЕРИМЕНТАЛЬНЫМИ ЗНАЧЕНИЯМИ

Сначала рассмотрим графики каждой структуры отдельно. Каждый график представляет зависимость времени выполнения одной операции от количества уже обработанных данных. Эксперимент проводился при случайных значениях, которые получали операции структур с помощью функции rand(). Графики строились в терминальной утилите для создания двух- и трёхмерных графиков gnuplot.

4.1) B-дерево

Вставка (см. прил. А - рис. 1) – ограничена 60log(x)

Удаление (см. прил. А - рис. 2) – ограничено 25log(x)

Поиск (см. прил. А - рис. 3) – ограничено 12log(x)

Во всех 3 графиках, как и полагается, прослеживается функция логарифма. Таким образом наша теоретическая оценка операций является верной. В среднем, операция вставки не превосходит 500 нс, удаления – 200 нс, как и поиске .

В графике удаления наблюдается много скачущих точек из-за сложности самого алгоритма, т.к. он имеет много случае, которые требуют разный подход.

4.2) AVL-дерево

Вставка (см. прил. А - рис. 4) – ограничение 66log(x)

Удаление (см. прил. А - рис. 5) – ограниченно 42log(x)

Поиск (см. прил. А - рис. 6) – ограничен 21log(x)

Ситуация аналогичная, графики приблизительно похожи на графики логарифмов.

4.3) Сравнение структур.

По функция, ограничивающим графики, и при прямом сравнении графиков мы можем обнаружить такое соотношение между операциями:

Вставка: AVL-дерево работает медленнее, чем B-дерево, так как перебалансировка AVL-дерева сделана сложнее, чем у B-дерева

Поиск: Структуры имеют примерно одинаковое время поиска значения, потому что перебалансировка при поиске не активируется, а алгоритмы поиска примерно одинаковы.

Удаление: Ситуация аналогична ситуации со вставкой.

ЗАКЛЮЧЕНИЕ

В ходе работы были исследованы такие структуры данных как АВЛ-дерево и B-дерево, несмотря на то, что теоретически операции этих двух деревьев имеют одинаковую сложность, но практика показала, что B-дерево имеет лучшие результаты. Но это зависит от значения t B-дерева. Если взять достаточно больше значение, то B-дерево будет похоже на массив.

Данное исследование показывает, что в разных ситуациях следует применять разные структуры данных, чтобы извлечь максимальную производительность.

ПРИЛОЖЕНИЕ А

РИСУНКИ

Рис 1. Вставка B-дерева

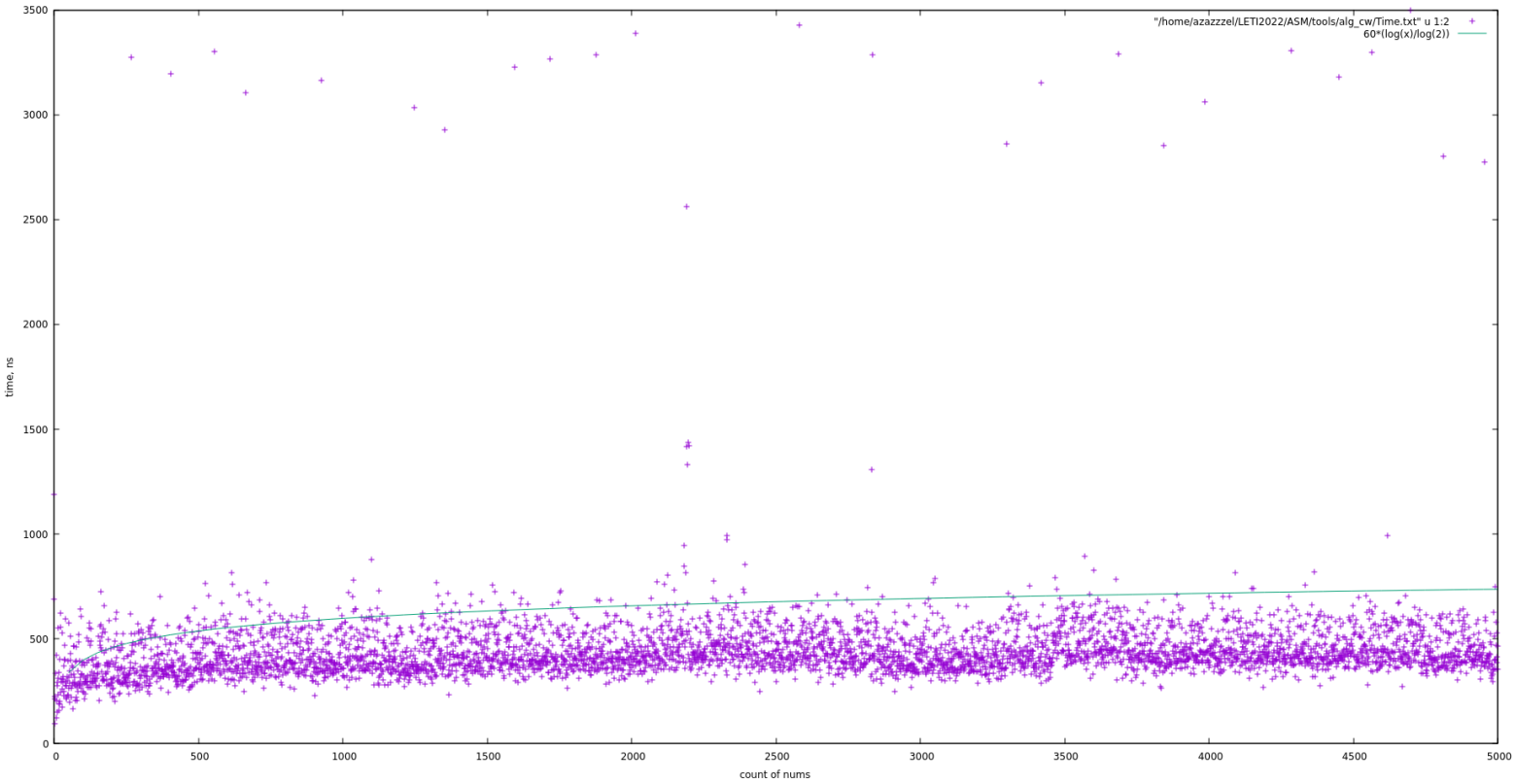


Рис. 2. Удаление B-дерева

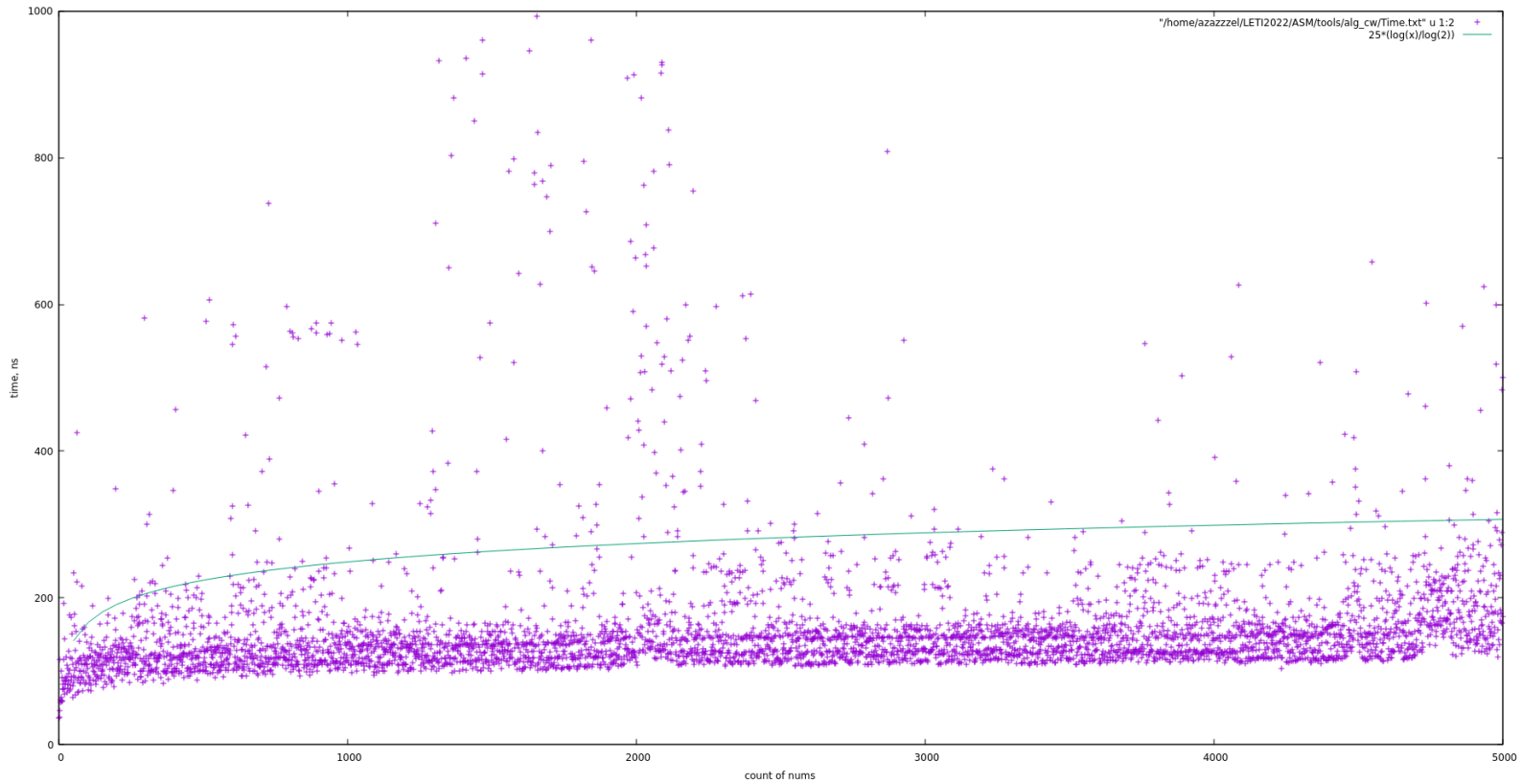


Рис. 3. Поиск B-дерева

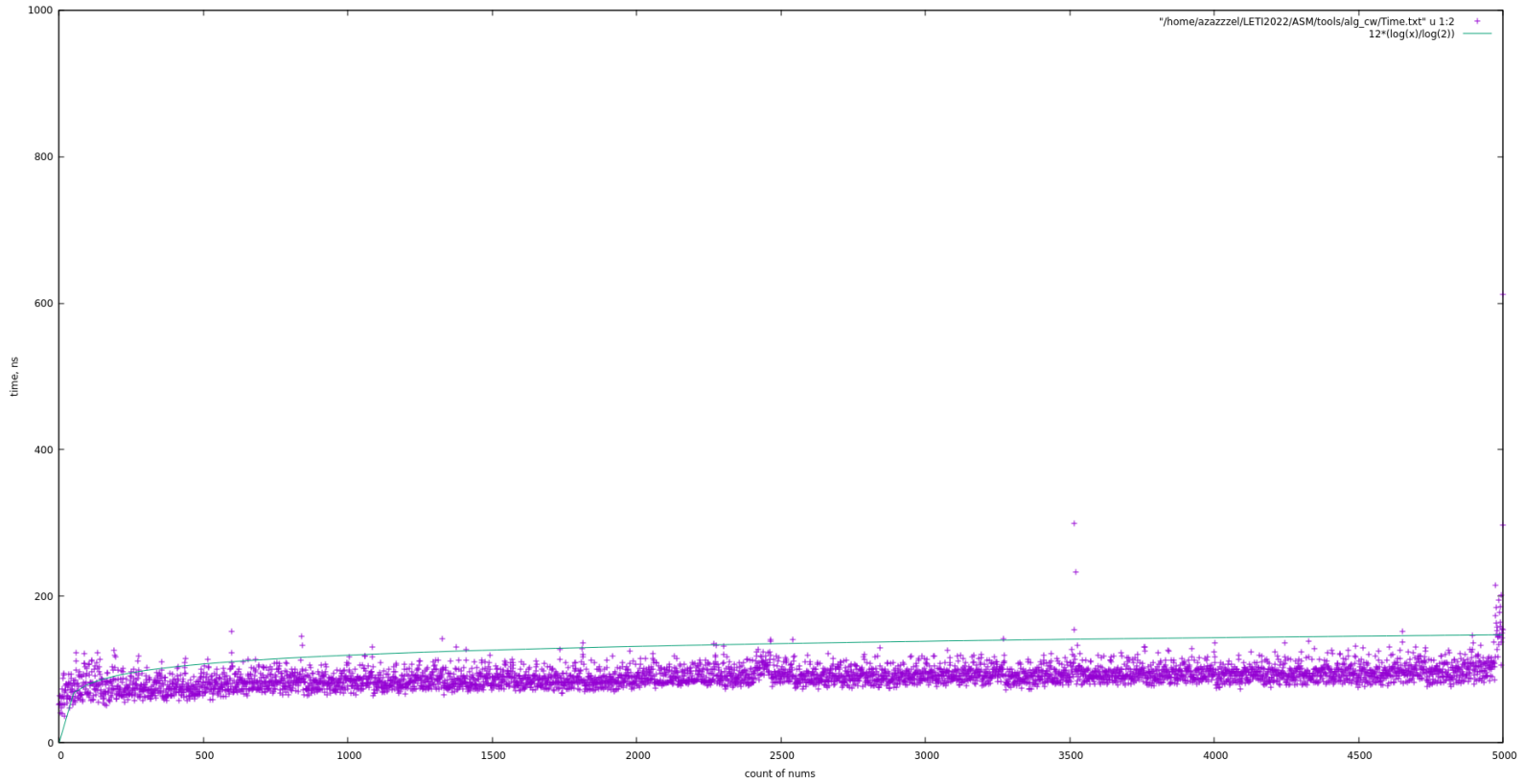


Рис. 4. Вставка AVL-дерева

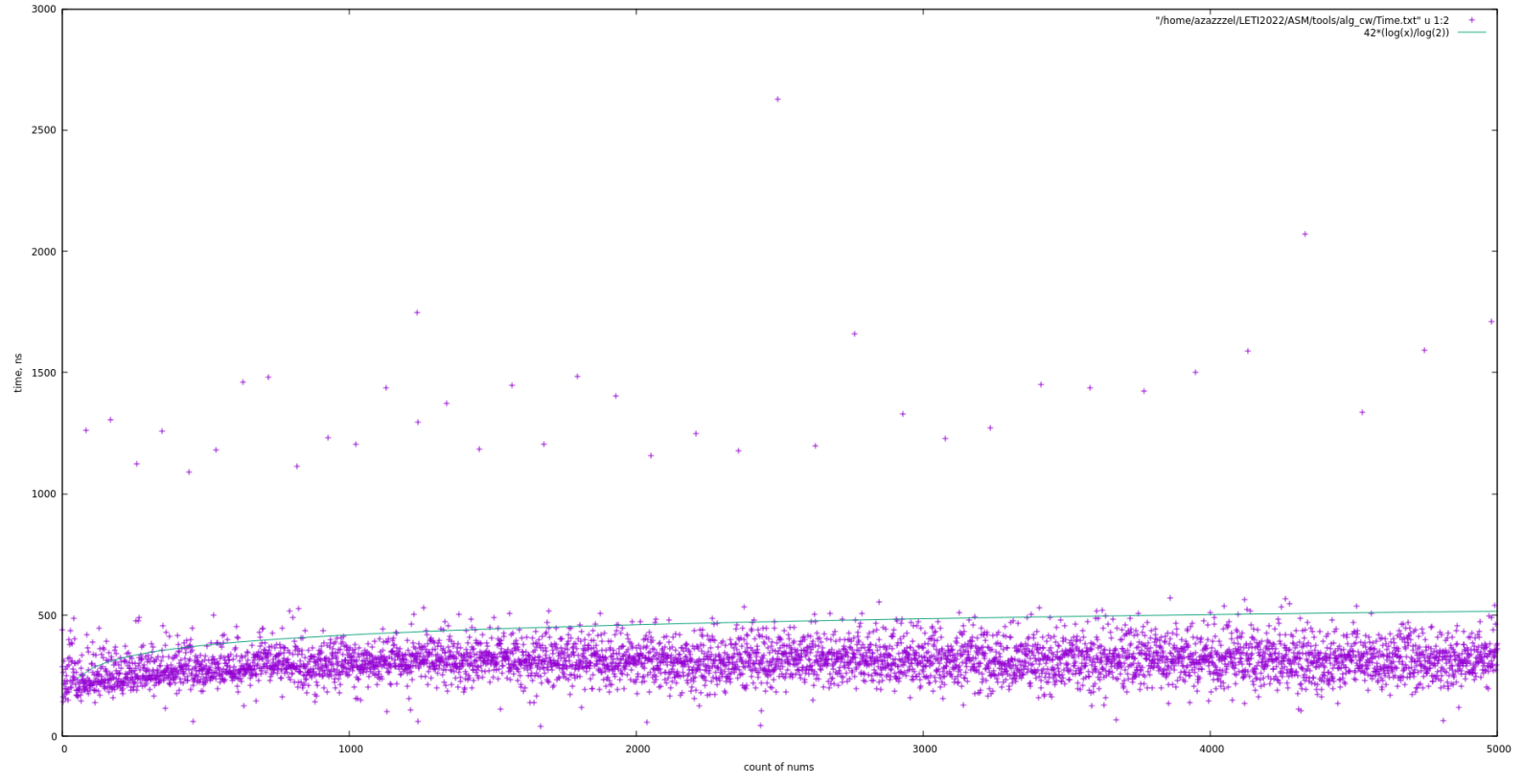


Рис. 5. Удаление AVL-дерева

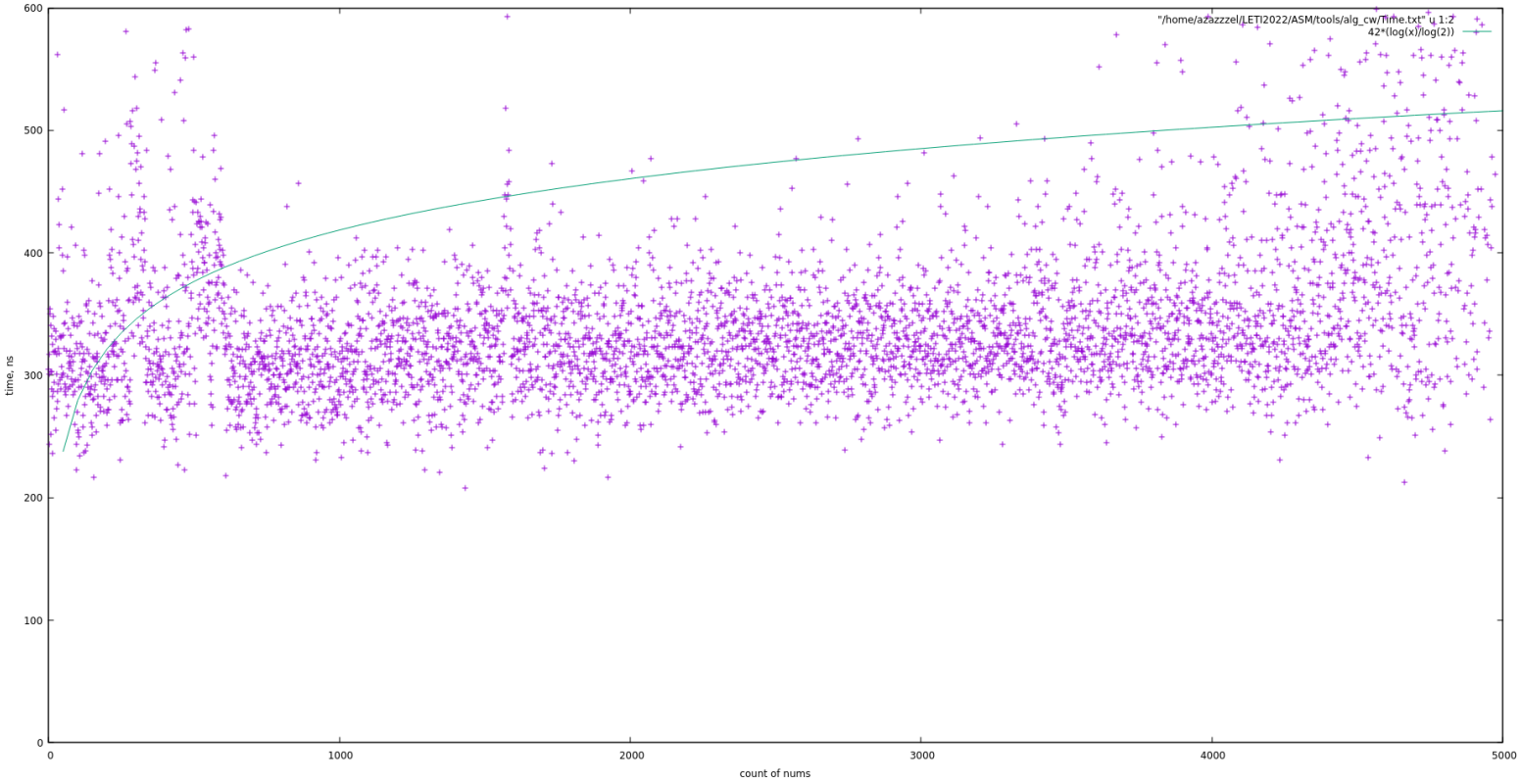
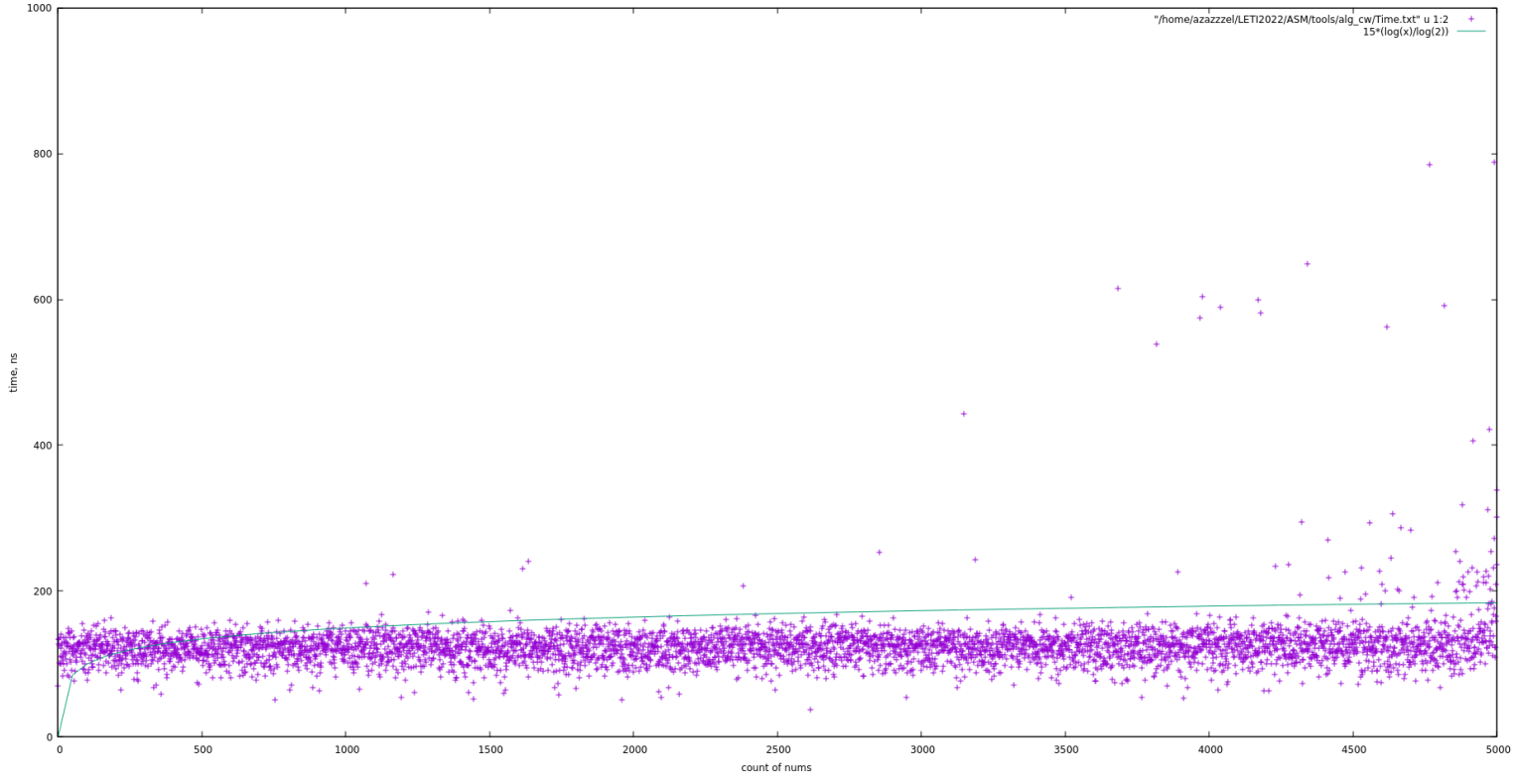


Рис. 6. Поиск AVL-дерева



ПРИЛОЖЕНИЕ Б

КОД ПРОГРАММЫ

Файл AVLtree.h

```
#ifndef ALG_CW_AVLTREE_H
#define ALG_CW_AVLTREE_H
#include <iostream>
using namespace std;

struct AVLnode{
    int key;
    AVLnode* left = NULL;
    AVLnode* right = NULL;
    int height = 0;
    AVLnode(int _key): key(_key){}
};

class AVLtree{
public:
    AVLtree(){
        root = NULL;
    }

    AVLnode* root;
    int depth(AVLnode* node);
    int getBalance(AVLnode* node);
    AVLnode* rightRotate(AVLnode* y_node);
    AVLnode* leftRotate(AVLnode* x_node);
    AVLnode* minValueNode(AVLnode* node);

    AVLnode* insert(AVLnode* node, int key);
    AVLnode* search(AVLnode* node, int key);
    AVLnode* deletion(AVLnode* root, int key);


    void inOrderPrint(AVLnode* root);
    void print(AVLnode* root, int lvl);
};


#endif
```

Файл AVLtree.cpp

```
#include "AVLtree.h"
void AVLtree::inOrderPrint(AVLnode *root) {
    if(root != NULL){
        inOrderPrint(root->left);
        cout << root->key << " ";
        inOrderPrint(root->right);
    }
}

int AVLtree::depth(AVLnode* node){
    if(!node)
```

```
        return 0;
    return node->height;
}

int AVLtree::getBalance(AVLnode *node) {
    if(!node)
        return 0;
    return depth(node->left) - depth(node->right);
}

AVLnode* AVLtree::rightRotate(AVLnode* y_node) {
    AVLnode* x_node = y_node->left;
    AVLnode* tmp = x_node->right;

    x_node->right = y_node;
    y_node->left = tmp;

    y_node->height = max(depth(y_node->left), depth(y_node->right)+1);
    x_node->height = max(depth(x_node->left), depth(x_node->right)+1);

    return x_node;
}

AVLnode* AVLtree::leftRotate(AVLnode* x_node) {
    AVLnode* y_node = x_node->right;
    AVLnode* tmp = y_node->left;

    y_node->left = x_node;
    x_node->right = tmp;

    y_node->height = max(depth(y_node->left), depth(y_node->right)+1);
    x_node->height = max(depth(x_node->left), depth(x_node->right)+1);

    return y_node;
}

AVLnode* AVLtree::insert(AVLnode* node, int key){
    if(node == NULL){
        return new AVLnode(key);
    }
    if(key<node->key)
        node->left = insert(node->left, key);
    else if(key>node->key)
        node->right = insert(node->right, key);
    else
        return node;

    node->height = 1 + max(depth(node->left), depth(node->right));

    int balance = getBalance(node);

    if(balance > 1 && key < node->left->key)
        return rightRotate(node);

    if(balance < -1 && key > node->right->key)
        return leftRotate(node);

    if(balance > 1 && key > node->left->key){
```

```
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
    if(balance < -1 && key < node->right->key){
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }
    return node;
}

AVLnode *AVLtree::search(AVLnode *node, int key) {
    if (node == NULL) return NULL;
    if (node->key == key) return node;

    if (node->key > key) return search(node->left, key);
    return search( node->right, key);
}


AVLnode *AVLtree::minValueNode(AVLnode *node) {
    AVLnode* cur = node;
    while(cur->left != NULL)
        cur = cur->left;
    return cur;
}


AVLnode* AVLtree::deletion(AVLnode* node, int key) {
    if(node == NULL)
        return node;

    if(key < node->key)
        node->left = deletion(node->left, key);
    else if(key > node->key)
        node->right = deletion(node->right,key);
    else{
        if(node->left == NULL || node->right == NULL){
            AVLnode* tmp = node->left ? node->left : node->right;
            if(tmp == NULL){
                tmp = node;
                node = NULL;
            }else{
                *node = *tmp;
                delete tmp;
            }
        }else{
            AVLnode* tmp = minValueNode(node->right);
            node->key = tmp->key;
            node->right = deletion(node->right, tmp->key);
        }
    }

    if(node == NULL)
        return node;

    node->height = 1 + max(depth(node->left), depth(node->right));

    int balance = getBalance(node);
```

```
    if(balance > 1 && getBalance(node->left) >= 0)
        return rightRotate(node);

    if(balance < -1 && getBalance(node->right) <= 0)
        return leftRotate(node);

    if(balance > 1 && getBalance(node->left) < 0){
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
    if(balance < -1 && getBalance(node->right) > 0){
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}

void AVLtree::print(AVLnode *root, int lvl) {
    if(root != NULL)
    {
        print(root->right,lvl + 1);
        for(int i = 0;i < lvl;i++) cout<<"   ";
        cout << root->key << endl;
        print(root->left,lvl + 1);
    }
}
```

Файл BTree.h

```
#ifndef ALG_CW_BTREE_H
#define ALG_CW_BTREE_H

#include <iostream>

#define BTreeOrder 3
using namespace std;

struct BtreeNode{
    BtreeNode(bool _leaf): leaf(_leaf){}
    int keys [2*BTreeOrder-1];
    BtreeNode* childrens[2*BTreeOrder];
    int size = 0;
    bool leaf;

    void splitChild(int idx, BtreeNode* node);
    void insertNonFull(int k);
    BtreeNode* search(int key);
    void deletion(int key);
    void removeFromLeaf(int idx);
    void removeFromNonLeaf(int idx);
    int getPredecessor(int idx);
    int getSuccessor(int idx);
    int findIdxKeyInNode(int key);
    void merge(int idx);
    void fill(int idx);
```

```
    void borrowFromPrev(int idx);
    void borrowFromNext(int idx);
    void traverse();
};

class BTree{
private:
    BtreeNode* root;
public:
    BTree(){root = NULL;}
    void insert(int key);
    void traverse();
    void deletion(int key);
    BtreeNode* search(int key);
};


#endif
```

Файл BTree.cpp

```
#include "BTree.h"

void BtreeNode::splitChild(int idx, BtreeNode *node) {
    BtreeNode* NewNode = new BtreeNode(node->leaf);
    NewNode -> size = BTreeOrder-1;

    for(int i = 0; i < BTreeOrder - 1; ++i)
        NewNode->keys[i] = node->keys[i+BTreeOrder];

    if(node->leaf == false)
        for(int i = 0; i < BTreeOrder; ++i)
            NewNode->childrens[i] = node->childrens[i + BTreeOrder];

    node->size = BTreeOrder - 1;
    for(int i = size; i >= idx + 1; --i)
        childrens[i+1] = childrens[i];

    childrens[idx + 1] = NewNode;

    for(int i = size - 1; i >= idx; --i)
        keys[i+1] = keys[i];

    keys[idx] = node->keys[BTreeOrder - 1];
    ++size;
}

void BtreeNode::insertNonFull(int key) {
    int i = size - 1;
    if(leaf) {
        while (i >= 0 && keys[i] > key) {
            keys[i + 1] = keys[i];
            --i;
        }
        keys[i + 1] = key;
        ++size;
    }else{
        while (i>=0 && keys[i] > key)
```

```
            --i;

        if(childrens[i+1]->size == 2 * BTreeOrder - 1){
            splitChild(i+1, childrens[i+1]);
            if(keys[i+1] < key)
                ++i;
        }
        childrens[i+1]->insertNonFull(key);
    }
}

BtreeNode *BtreeNode::search(int key){
    int i = 0;
    while(i < size && key > keys[i])
        ++i;

    if(keys[i]==key)
        return this;

    if(leaf)
        return NULL;

    return childrens[i]->search(key);
}


void BtreeNode::traverse() {
    int i;
    for (i = 0; i < size; i++) {
        if (leaf == false)
            childrens[i]->traverse();
        cout << " " << keys[i];
    }

    if (leaf == false)
        childrens[i]->traverse();
}

int BtreeNode::findIdxKeyInNode(int key) {
    int idx = 0;
    while(idx < size && keys[idx] < key)
        ++idx;
    return idx;
}

int BtreeNode::getPredecessor(int idx) {
    BtreeNode* cur = childrens[idx];
    while (!cur->leaf)
        cur = cur->childrens[cur->size];

    return cur->keys[cur->size -1];
}
int BtreeNode::getSuccessor(int idx) {
    BtreeNode* cur = childrens[idx+1];
    while (!cur->leaf)
        cur = cur->childrens[0];
    return cur->keys[0];
}
```

```
void BtreeNode::removeFromLeaf(int idx) {
    for(int i = idx + 1; i < size; ++i)
        keys[i-1] = keys[i];
    --size;
}

void BtreeNode::merge(int idx) {
    BtreeNode* child = childrens[idx];
    BtreeNode* sibling = childrens[idx+1];

    child->keys[BTreeOrder-1] = keys[idx];

    for(int i = 0; i < sibling->size; ++i)
        child->keys[i+BTreeOrder] = sibling->keys[i];

    if(!child->leaf)
        for(int i = 0; i <= sibling->size; ++i)
            child -> childrens[i+BTreeOrder] = sibling -> childrens[i];

    for(int i = idx + 1; i < size; ++i)
        keys[i-1] = keys[i];

    for(int i = idx + 2; i<=size; ++i)
        childrens[i-1] = childrens[i];

    child->size += sibling->size + 1;
    --size;

    delete(sibling);
}

void BtreeNode::removeFromNonLeaf(int idx) {
    int key = keys[idx];
    if(childrens[idx]->size >= BTreeOrder){
        int pred = getPredecessor(idx);
        keys[idx] = pred;
        childrens[idx]->deletion(pred);
    }
    else if(childrens[idx + 1]->size >= BTreeOrder){
        int succ = getSuccessor(idx);
        keys[idx] = succ;
        childrens[idx + 1]->deletion(succ);
    }
    else{
        merge(idx);
        childrens[idx]->deletion(key);
    }
}

void BtreeNode::borrowFromPrev(int idx) {
    BtreeNode* child = childrens[idx];
    BtreeNode* sibiling = childrens[idx - 1];

    for(int i = child->size - 1; i >= 0; --i)
        child -> keys[i+1] = child -> keys[i];

    if(!child->leaf){
```

```
        for(int i = child -> size; i>=0; --i)
            child->childrens[i+1] = child->childrens[i];
    }

    child -> keys[0] = keys[idx - 1];

    if(!child->leaf)
        child->childrens[0] = sibiling->childrens[sibiling->size];

    keys[idx - 1] = sibiling->keys[sibiling->size-1];

    ++child->size;
    --sibiling->size;
}

void BtreeNode::borrowFromNext(int idx) {
    BtreeNode* child = childrens[idx];
    BtreeNode* sibiling = childrens[idx + 1];

    child->keys[child->size] = keys[idx];

    if(!child->leaf)
        child->childrens[child->size + 1] = sibiling->childrens[0];

    keys[idx] = sibiling -> keys[0];

    if(!sibiling->leaf)
        for(int i = 1; i <= sibiling->size; ++i)
            sibiling -> childrens[i-1] = sibiling->childrens[i];

    ++child->size;
    --sibiling->size;
}

void BtreeNode::fill(int idx) {
    if(idx != 0  && childrens[idx - 1]->size >= BTreeOrder)
        borrowFromPrev(idx);
    else if(idx != size  && childrens[idx + 1]->size >= BTreeOrder)
        borrowFromNext(idx);

    else if(idx != size)
        merge(idx);
    else
        merge(idx-1);
}

void BtreeNode::deletion(int key) {
    int idx = findIdxKeyInNode(key);

    if(idx < size && keys[idx] == key){
        if(leaf)
            removeFromLeaf(idx);
        else
            removeFromNonLeaf(idx);
    }else{
        if(leaf){
            cout<<"Данный ключ "<<key<<" не существует в дереве\n";
            return;
```

```
        }

        bool flag =((idx == size) ? true : false);

        if(childrens[idx]->size < BTreeOrder)
            fill(idx);

        if(flag && idx > size)
            childrens[idx - 1]->deletion(key);
        else
            childrens[idx]->deletion(key);
    }
}

BtreeNode *BTree::search(int key) {
    return (root == NULL) ? NULL : root->search(key);
}

void BTree::insert(int key){
    if(root == NULL){
        root = new BtreeNode(true);
        root->keys[0] = key;
        root->size = 1;
    }else if(root->size == 2*BTreeOrder - 1){
        BtreeNode* NewRoot = new BtreeNode(false);
        NewRoot -> childrens[0] = root;
        NewRoot ->splitChild(0,root);

        int i = NewRoot->keys[0] < key ? 1 : 0;
        NewRoot->childrens[i]->insertNonFull(key);

        root = NewRoot;
    }else
        root->insertNonFull(key);
}

void BTree::deletion(int key) {
    if(!root){
        cout<<"Это дерево пустое!\n";
        return;
    }
    root->deletion(key);

    if(root->size == 0){
        BtreeNode* tmp = root;
        if(root->leaf)
            root=NULL;
        else
            root=root->childrens[0];
        delete tmp;
    }
}

void BTree::traverse() {
    if(root != NULL)
        root -> traverse();
}
```

Файл main.cpp

```
#include "BTree.h"
#include "AVLtree.h"
#include "gnuplot-iostream.h"

#include <time.h>

void test();
int main() {
    test();
}

void test(){
    BTree t;
    t.insert(8);
    t.insert(9);
    t.insert(10);
    t.insert(11);
    t.insert(15);
    t.insert(16);
    t.insert(17);
    t.insert(18);
    t.insert(20);
    t.insert(23);

    cout << "B-дерево: ";
    t.traverse();
     (t.search(17) != NULL) ? cout<<endl<<"17 найдено" : cout<<endl<<"17
не найдено";
    cout<<'\n';
    t.deletion(33);
    t.deletion(17);
    cout<<"Новое B-дерево: ";
    t.traverse();
     (t.search(17) != NULL) ? cout<<endl<<"17 найдено" : cout<<endl<<"17
не найдено";

    AVLtree tr;
    cout<<"\n\n\n"<<"AVL-дерево:";
    tr.root = tr.insert(tr.root, 10);
    tr.root =tr.insert(tr.root, 20);
    tr.root = tr.insert(tr.root, 12);
    tr.root =    tr.insert(tr.root, 36);
    tr.root =tr.insert(tr.root, 30);
    tr.root =tr.insert(tr.root, 40);
    tr.root = tr.insert(tr.root, 28);
    tr.root =  tr.insert(tr.root, 22);
    tr.root =  tr.insert(tr.root, 48);
    tr.root =  tr.insert(tr.root, 38);
    tr.inOrderPrint(tr.root);
    cout<<endl;

    tr.print(tr.root,0);

       (tr.search(tr.root,22)  !=  NULL)  ?  cout<<endl<<"22  найдено"  :
cout<<endl<<"22 не найдено";
    tr.root = tr.deletion(tr.root, 22);
```

```
	(tr.search(tr.root,22) != NULL) ? cout<<endl<<"22 найдено" :
cout<<endl<<"22 не найдено";
	(tr.search(tr.root,30) != NULL) ? cout<<endl<<"30 найдено" :
cout<<endl<<"30 не найдено";
}
```