ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

КУРСОВАЯ РАБОТА

Алгоритм «B – Дерево»
по дисциплине «Объектно-ориентированное программирование»

Выполнил
Студент
гр. 3331506/90401 _____ Копейко И.В.
(подпись)

Работу принял _____ Ананьевский М.С.
(подпись)

Санкт-Петербург
2022 г.

Введение

B-дерево (читается как Би-дерево) — это особый тип сбалансированного дерева поиска, в котором каждый узел может содержать более одного ключа и иметь более двух дочерних элементов. Из-за этого свойства B-дерево называют сильноветвящимся.

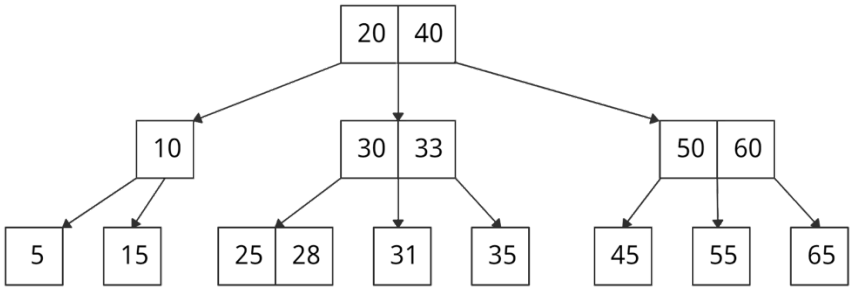


Рисунок 1 – Пример структуры B-дерева

Вторичные запоминающие устройства (жесткие диски, SSD) медленно работают с большим объемом данных. Людям захотелось сократить время доступа к физическим носителям информации, поэтому возникла потребность в таких структурах данных, которые способны это сделать. Помимо этого B-деревья используют:

- В базах данных и файловых системах.
- Для хранения блоков данных (вторичные носители).
- Для многоуровневой индексации.

Двоичное дерево поиска, АВЛ-дерево, красно-черное дерево и т. д. могут хранить только один ключ в одном узле. Если нужно хранить больше, высота деревьев резко начинает расти, из-за этого время доступа сильно увеличивается.

С B-деревом все не так. Оно позволяет хранить много ключей в одном узле и при этом может ссылаться на несколько дочерних узлов. Это значительно уменьшает высоту дерева и, соответственно, обеспечивает более быстрый доступ к диску.

Описание алгоритма

Дерево принимает только единственный параметр «t» или «T», который будет определять количество ключей и указателей в каждом узле. «t» определяет минимальное число указателей в узле. «T» является альтернативой, определяет максимальное число указателей. Иногда эти величины именуют «Б-фактор».

Имеются следующие правила:

1. В каждом узле содержатся минимум (t – 1) ключей и минимум (t) указателей. Все ключи и указатели расположен по возрастанию и чередуются между собой. Максимум ключей (2t – 1), а указателей (2t). Указателей всегда на 1 больше чем ключей.
2. Корень может иметь как минимум один ключ и два указателя, предел такой же как и у других узлов.
3. Потомок, на которого имеется указатель содержит ключи больше чем ключ слева от указателя и меньше чем ключ справа от указателя.
4. Листья потомков (указателей) не имеют.
5. Глубина (число уровней) всех ветвей всегда одинакова.
6. Новый ключ добавляется в самый нижний узел

Основные операция производимые с B-Деревом:

- Добавление ключа
- Поиск ключа
- Удаление ключа

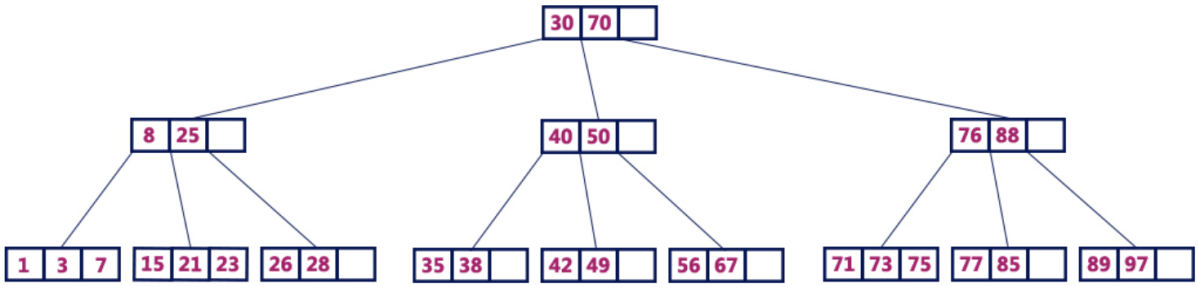
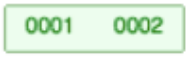


Рисунок 2 – B-Дерево с T = 4

Визуальная демонстрация алгоритма добавления

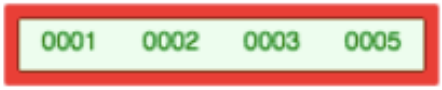
Пример сценария 1(обычное добавление):



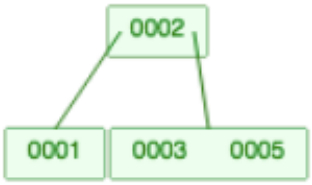
Добавляем 3

Пример сценария 2 (деление коренного узла):

Добавляем 5



Узел переполнен, он делится



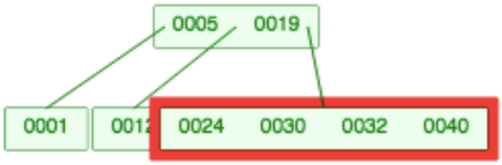
Пример сценария 3 (деление узла):

Имеется другое B-дерево с T = 4, решаем добавить ключ 25.

Мы переходим в правый узел

Добрались до нужного узла

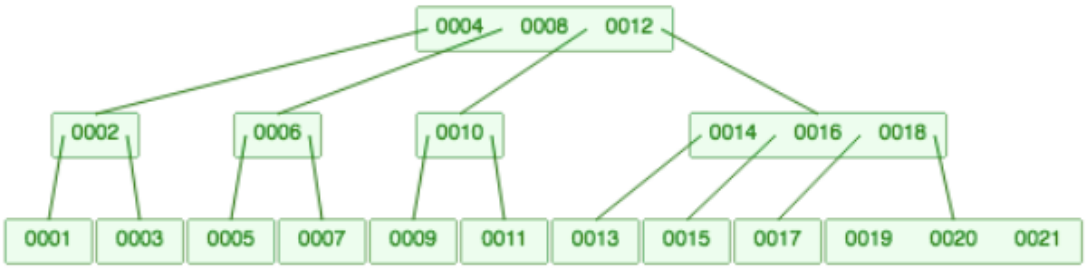
Узел переполнен, его разбивают



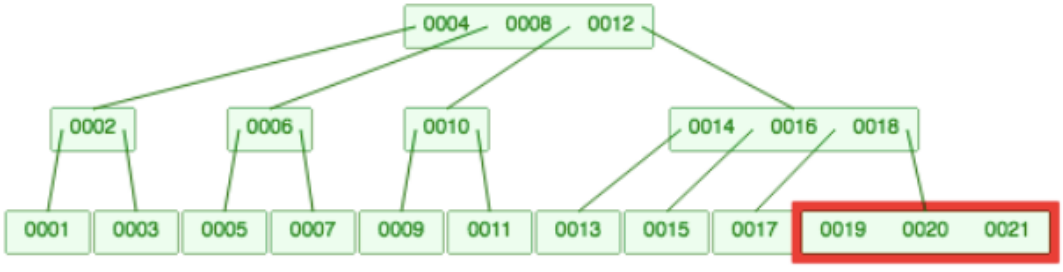
Итог операции добавления

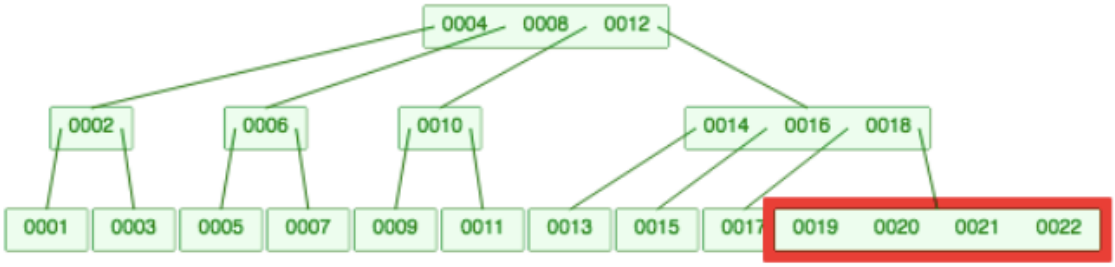
Пример сценария 4 (деление узла рекурсивно):

Имеется другое B-дерево с T = 4, решаем добавить ключ 22:

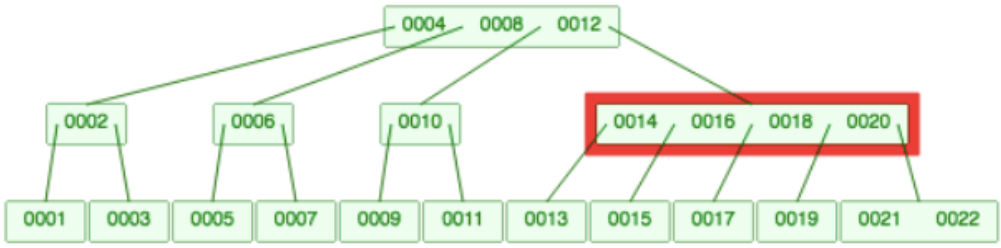


Добавляем 22 в нижний узел

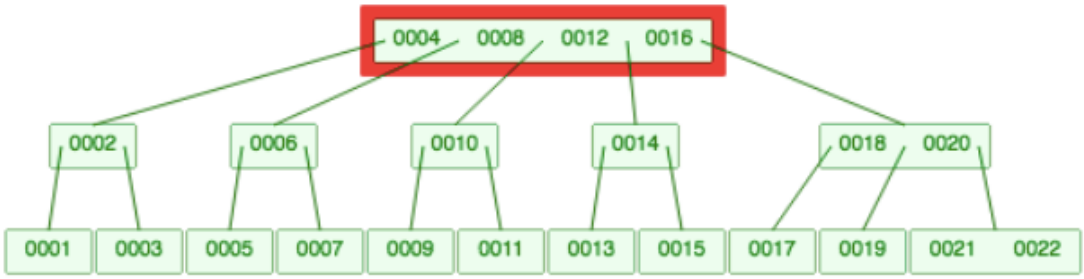




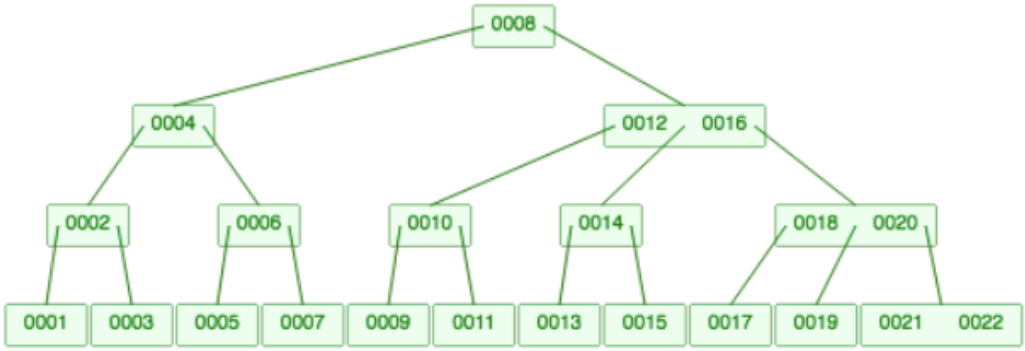
Узел переполнен, он делится



Один ключ отдали в узел уровнем выше, теперь он тоже переполнен и делится



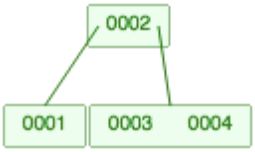
То же самое, делится коренной узел



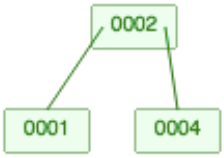
Дерево всегда растет вверх!

Визуальная демонстрация алгоритма удаления

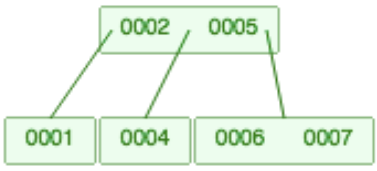
Пример сценария 1 (простое удаление):



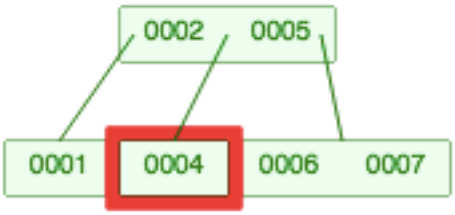
Удалим ключ 3



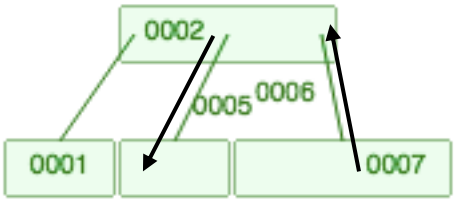
Пример сценария 2 (взятие ключа у брата):



Удаляем ключ 4

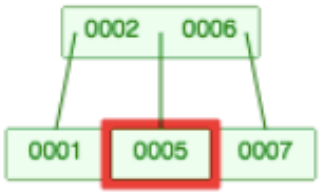


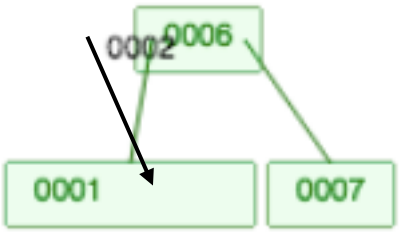
Он единственный в своем узле

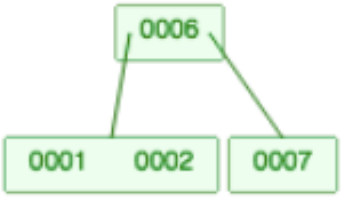


Пример сценария 3 (объединяем узлы):

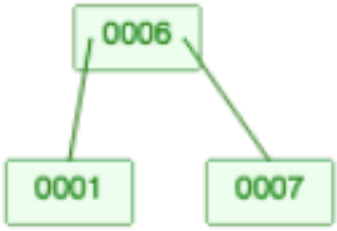
Удаляем ключ 5



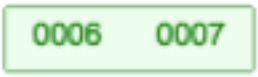




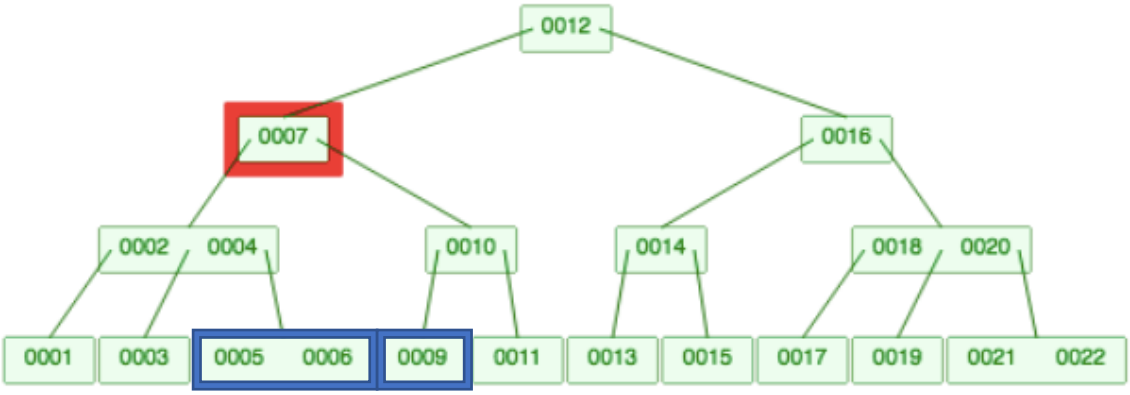
Пример сценария 4 (объединяем коренной узел):



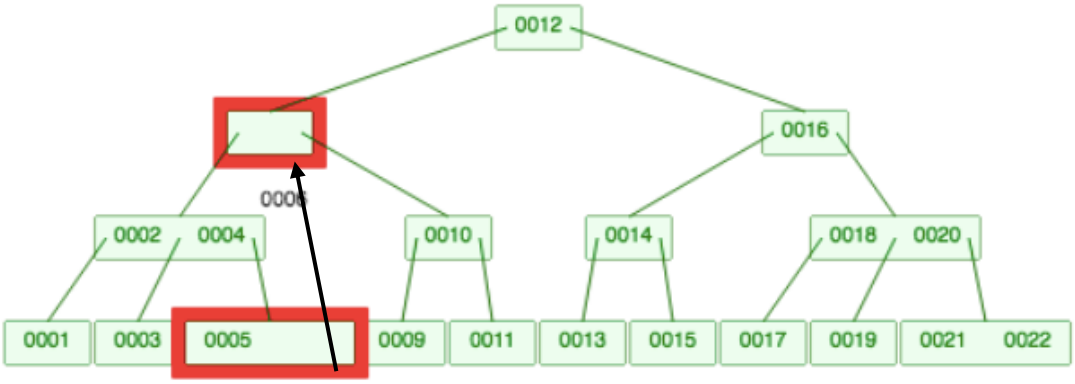
Удаляем ключ 1



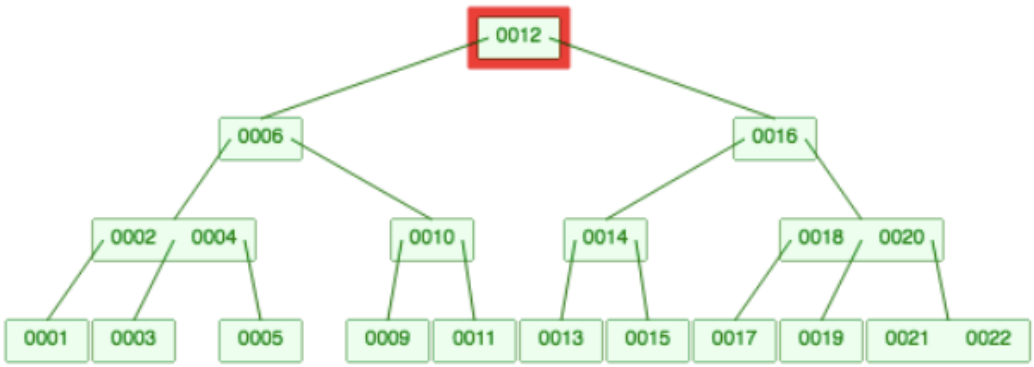
Пример сценария 5 (берем ключ из нижнего узла):



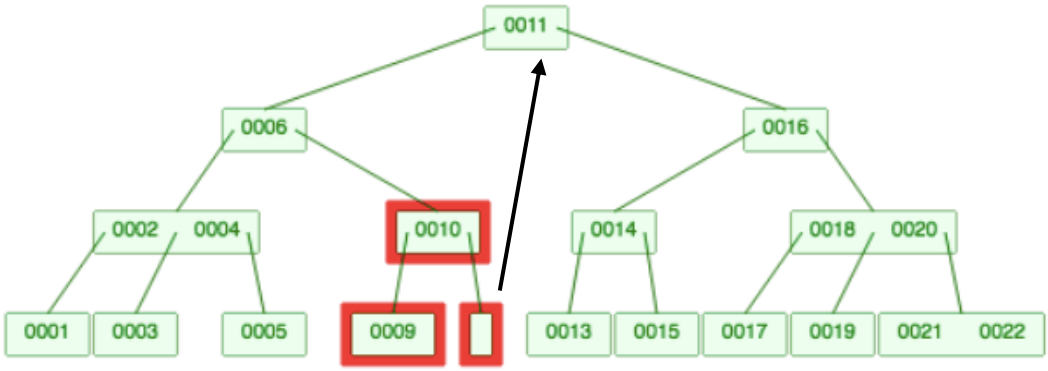
Удаляем ключ 7



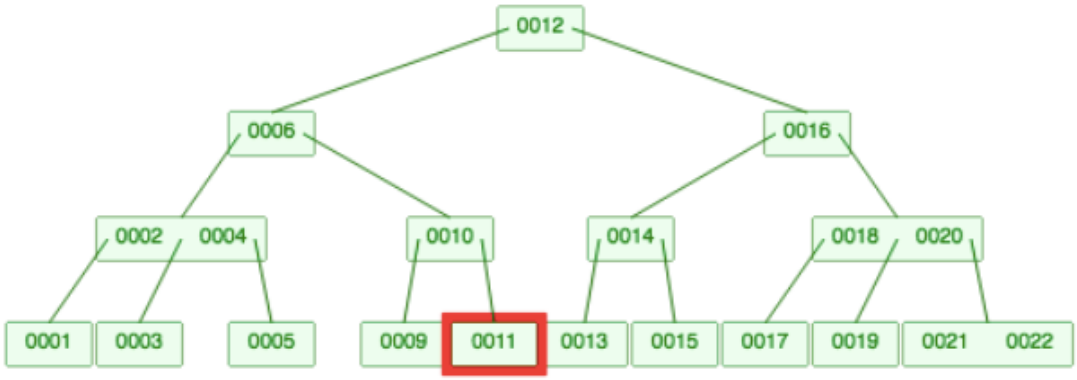
Пример сценария 6 (забираем ключи из нижнего узла):



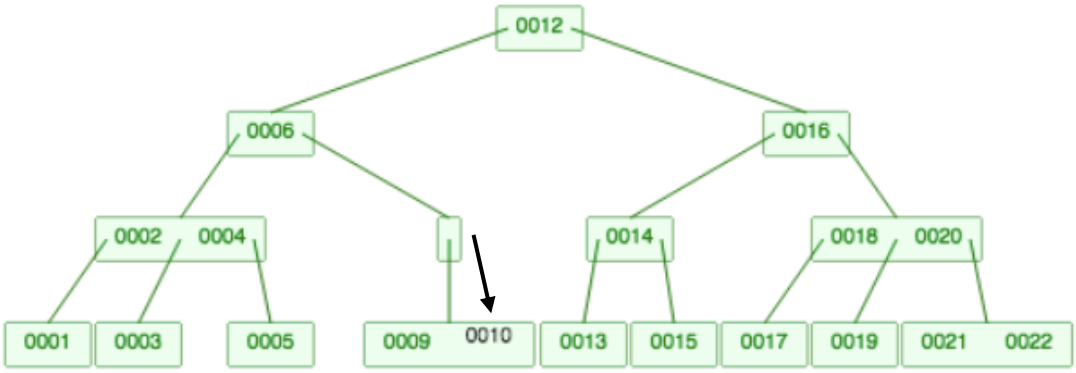
Удаляем ключ 12

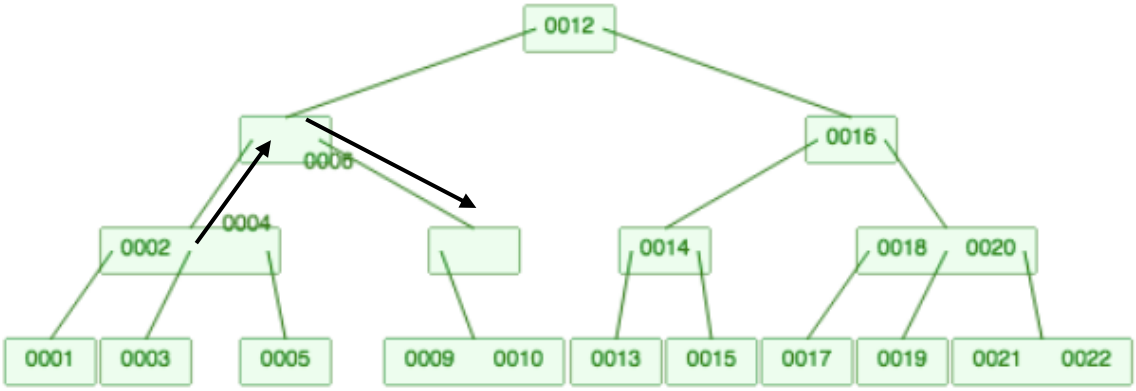


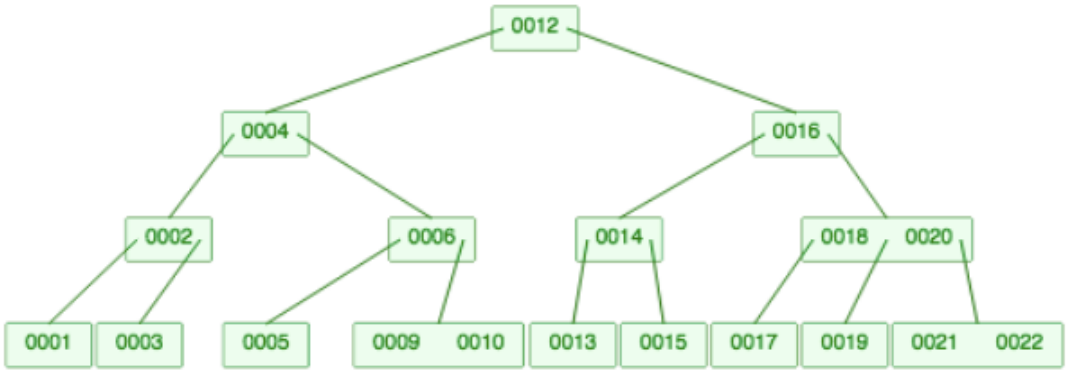
Пример сценария 7 (рекурсивно забираем ключи):



Удаляем ключ 11

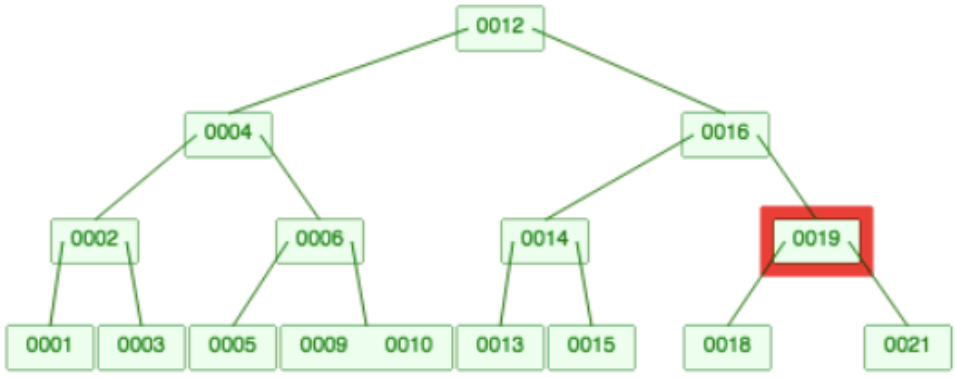




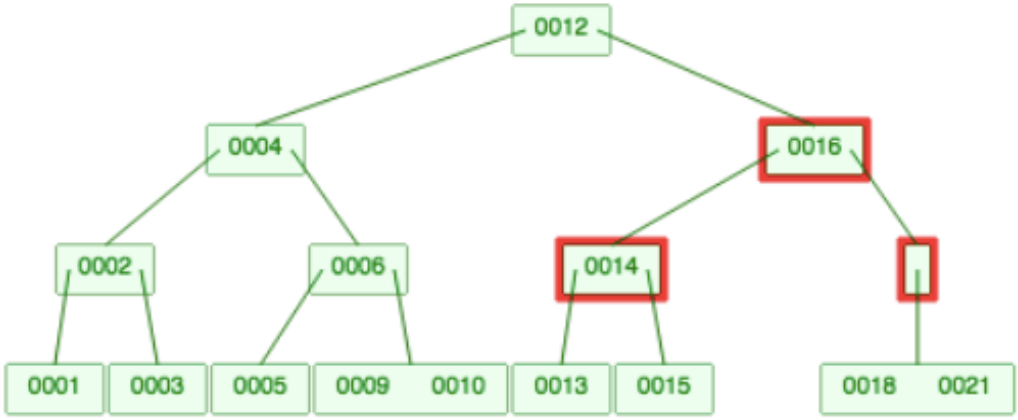


Вместе с ключом был передан еще и указатель

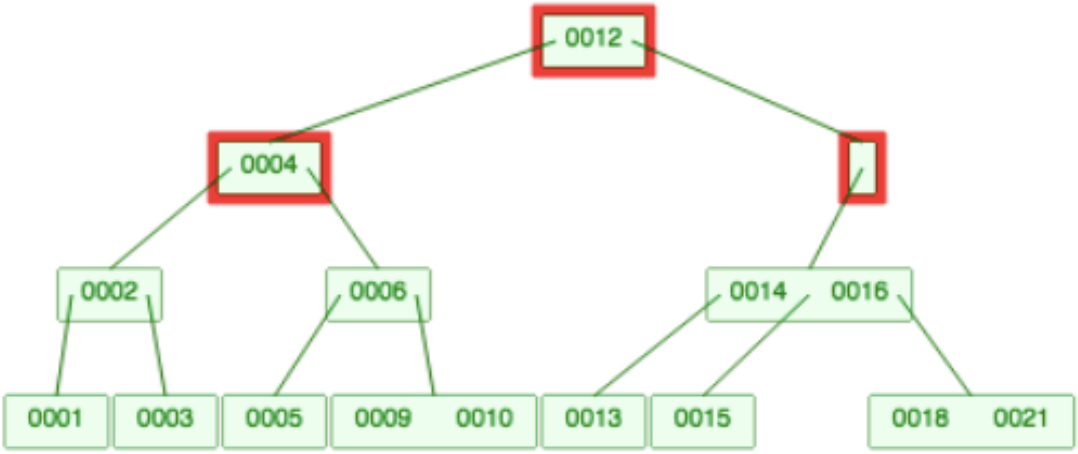
Пример сценария 8 (рекурсивно забираем ключи):



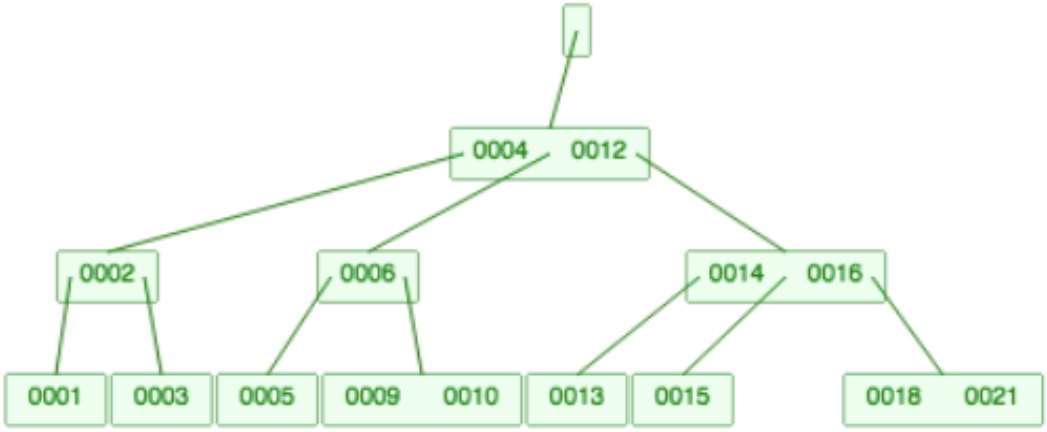
Удаляем ключ 19



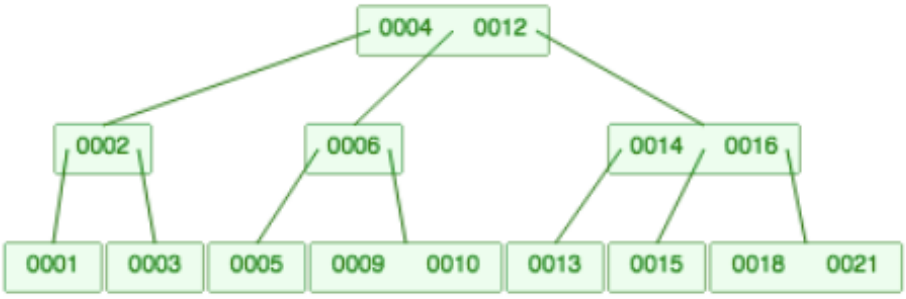
Нижние узлы под ключом 19 объединились, теперь будет происходить объединение с братом



Родителя больше не ключей, он тоже будет объединяться с братом



Так уменьшилась высота всего дерева



Исследование алгоритма

Исходя из интернет-источников временная сложность алгоритма для каждой из операций (вставка, поиск, удаление) равно O(log(n))

Временная сложность в O-символике		
	В среднем	В худшем случае
Расход памяти	O(n)	O(n)
Поиск	O(log n)	O(log n)
Вставка	O(log n)	O(log n)
Удаление	O(log n)	O(log n)

Рисунок 3 – Временная сложность алгоритма

Итак, исследуем временную сложность *алгоритма при добавлении*:

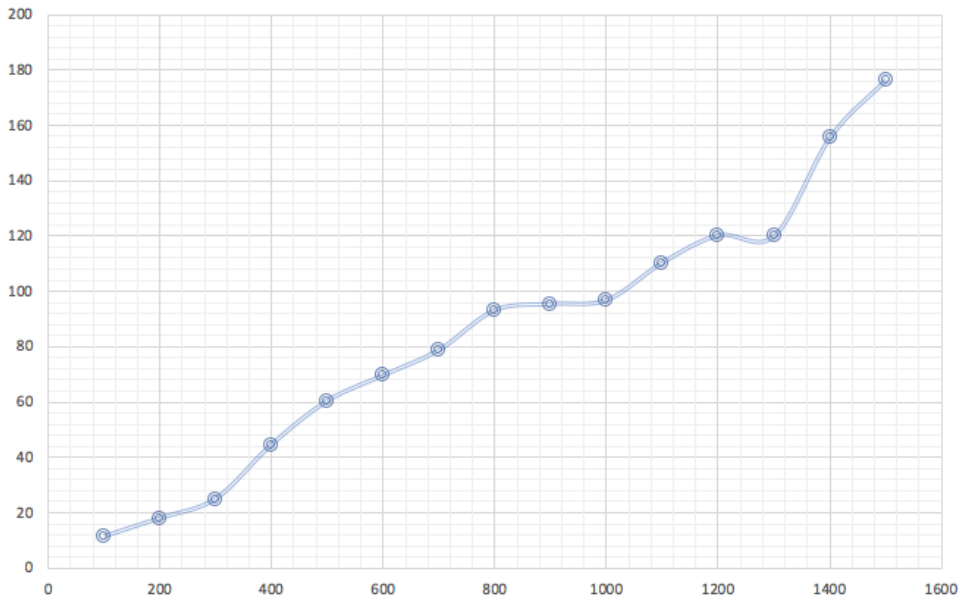


Рисунок 4 – Временная сложность алгоритма при добавлении на малом промежутке

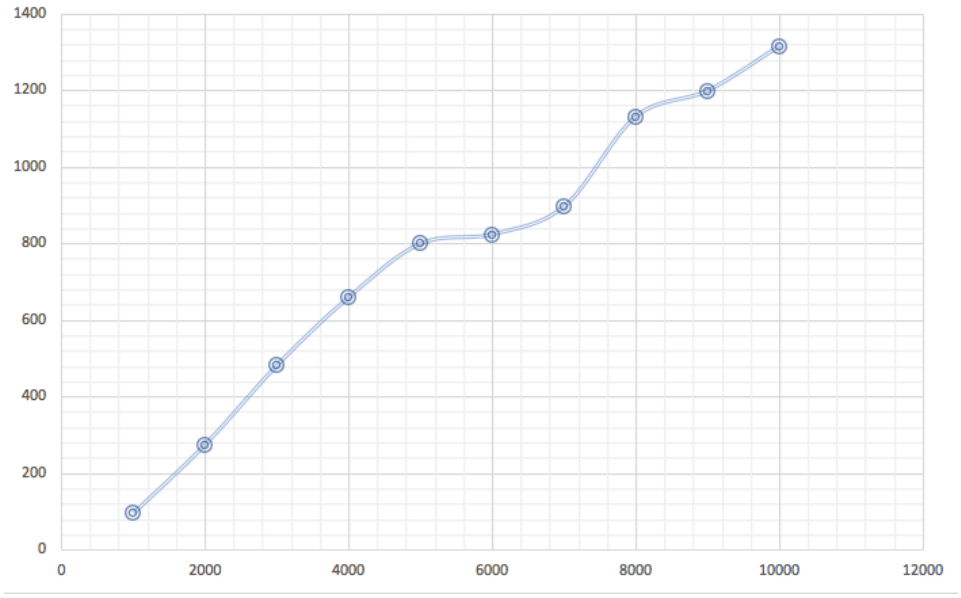


Рисунок 5 – Временная сложность алгоритма при добавлении на большом промежутке

На заявленный изначально график полученные результаты не похожи, скорее изменение происходит по линейному закону.

Исследуем временную сложность *алгоритма при удалении:*

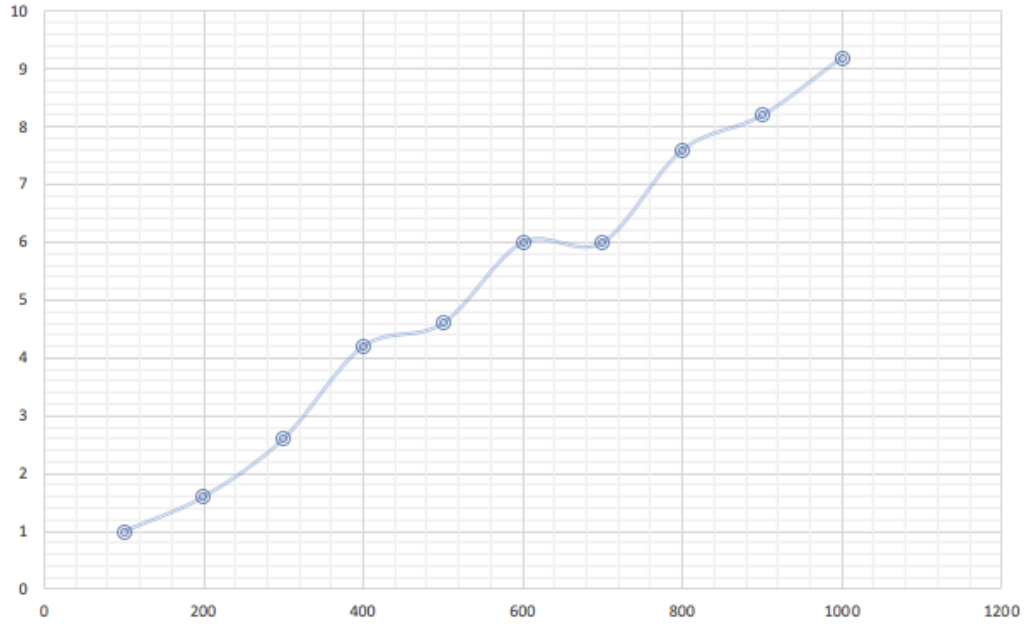


Рисунок 6 – Временная сложность алгоритма при удалении

Исследуем временную сложность *алгоритма при поиске:*

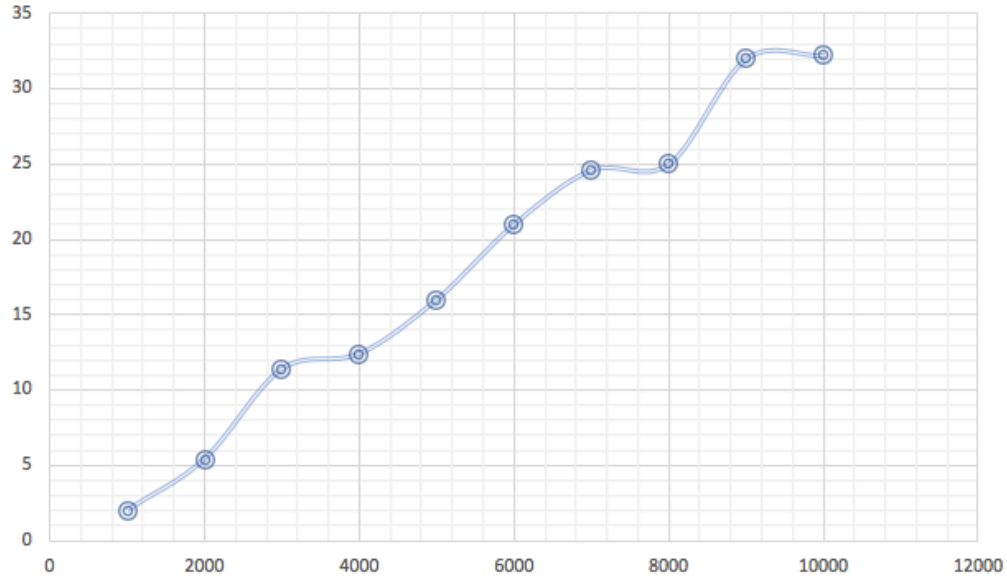


Рисунок 7 – Временная сложность алгоритма при поиске

Как видим все алгоритмы имеют линейную скачкообразную вертикальную функцию. Вероятно, идеальный график достигается на каком-то конкретном промежутке, или же алгоритм построения B-дерева в данной работе несколько отличается от эталона. Тем не менее, алгоритм поиска достаточно простой и должен быть схож с эталоном.

Все выполненные измерения проводились путем использования функции clock() из <ctime>.

Заключение

Алгоритм B – Дерево является довольно простым по своей сути, но имеет трудности на пути его реализации. Это касается того факта, что существуют разные сценарии поведения дерева при добавлении и удалении узла, которые необходимо прописать в коде, по этим причинам код становится довольно большим. На мой взгляд, должны существовать более простые алгоритмы для хранения данных.

Список источников

1. https://youtu.be/WXXetwePSRk
2. https://youtu.be/GKa_t7fF8o0
3. https://www.cs.usfca.edu/~galles/visualization/BTree.html
4. https://codechick.io/tutorials/dsa/dsa-b-tree
5. https://habr.com/ru/post/114154/
6. http://cppstudio.com/post/468/

Приложение

Файл «btree.h»

```
#ifndef BTREE_B_TREE_H
#define BTREE_B_TREE_H

class BTree;
class Node;


class BTree {

private:
    unsigned int B_factor;
    Node* root;
public:
    BTree();
    BTree(int B_factor);
    void count_tree(); //вывод числа ключей в дереве
    void print_keys_in_string();
    void add(int key);
    void print();
    Node* search(int key);
    void delete_key(int key);

    friend class Node;
};


class Node {
private:
    int *keys;
    int *data;
    Node **pointers;
private:
    Node();
    Node(unsigned int factor);
private:
    void root();
    void set(int keys[], Node* pointers[]); //DEBUG
    void print(unsigned int B_factor, Node* root, Node* parent);
    void print_only_this(unsigned int B_factor, Node* root, Node* parent);
//DEBUG
    int count_tree(unsigned int B_factor, Node* root);
    int print_keys_in_string(unsigned int B_factor);
    void add(int key, unsigned int B_factor, Node * active_node, Node * root,
Node * parent);
    void Node_segmentation_root (unsigned int B_factor, Node* active_node,
int key, Node * root, Node * parent);
    void Node_segmentation_round (unsigned int B_factor, Node* active_node,
int key, Node * root, Node * parent);

    Node* search(int key, int level_down, int B_factor, Node * root, Node *
active_node, Node* node_with_key);

    int count_keys(int B_factor);
    int count_pointers(int B_factor);

    void remove_free_pointer(int B_factor);
    void remove_free_place( int B_factor);
    int ask_brother_key(int key, int B_factor, Node * node_with_key, Node*
```

```
parent, Node * root);
    int merge_nodes_brothers(int key, int B_factor, Node* node_with_key,
Node* parent, Node* root, int parent_status);
    int ask_brother_key_with_pointers(int key, int B_factor, Node*
node_with_key, Node*parent, Node* root);
    int lift_up_left (int key, int B_factor, Node * root, Node * parent, Node
* node_with_key, int key_index);
    int lift_up_right (int key, int B_factor, Node * root, Node * parent,
Node * node_with_key, int key_index);
    void merge_nodes_brothers_with_pointers(int key, int B_factor, Node*
node_with_key, Node* parent, Node* root, int parent_status);
    void steal_down_key(int key, int B_factor, Node* node_with_key, Node*
parent, Node* root, int key_index);
    void change_root(Node* new_root, Node* root, int B_factor);

    void del_key_only_this(int key, int B_factor, Node * node_with_key);
    void delete_key(int key, int B_factor, Node * root, Node * parent, Node *
node_with_key, int is_used);

    Node* find_this_parent (int B_factor, Node * active_node, Node * root);

    int count_free_key(int B_factor);
    int count_free_pointer(int B_factor);
    void refresh(unsigned int B_factor, Node * root, Node * parent);
    void add_to_any_child (unsigned int B_factor, int key, int
free_key_count, Node * root);
    void add_only_to_this (unsigned int B_factor, int key, int
free_key_count, Node * root, Node * parent);

    friend class BTree;
};

#endif //BTREE_B_TREE_H
```

Файл «btree.cpp»

```
#include "b_tree.h"
#include <iostream>

#define DEBUG_TREE root->print(B_factor, nullptr, nullptr);

#define TEMP_KEY keys[B_factor-1]
#define TEMP_POINTER pointers[B_factor]

#define USED 1
#define NOT_USED 0

#define PARENT_HAS_ONE_KEY 1
#define PARENT_HAS_MORE_KEYS 2
#define PARENT_ROOT_HAS_ONE 3

static int last_key = -1; //используется в функции print_keys_in_string

//std::cout << "Hello, World!";
//std::cout << "Hello, World!" << std::endl;


Node::Node(){
    int factor = 4;

    //создать массив ключей
```

```
    keys = new int [factor]; //имеется по одному резервному месту под ключ,
под дату, под указатель
    //создать массив из даты
    data = new int [factor];
    //создать массив из указателей
    pointers = new Node* [factor + 1];

    keys[factor] = -1;

    pointers[factor+1] = nullptr;

    //каждому ключу дать номер -1
    for(int i = 0; i < factor; i++) {
        keys[i] = -1;
    }
    //каждому указателю дать nullptr
    for(int i = 0; i <= factor; i++) {
        pointers[i] = nullptr;
    }
}

Node::Node(unsigned int factor) {

    //создать массив ключей
    keys = new int [factor]; //имеется по одному резервному месту под ключ,
под дату, под указатель
    //создать массив из даты
    data = new int [factor];
    //создать массив из указателей
    pointers = new Node* [factor + 1];

    keys[factor] = -1;

    pointers[factor+1] = nullptr;

    //каждому ключу дать номер -1
    for(int i = 0; i < factor; i++) {
        keys[i] = -1;
    }
    //каждому указателю дать nullptr
    for(int i = 0; i <= factor; i++) {
        pointers[i] = nullptr;
    }
}

void Node::root() {
    //заполняем узел с ключом "0"
    keys[0] = 0;
}

void BTree::print(){
    std::cout << " ============= Б-дерево ===============" << std::endl;
    std::cout << "{Адрес узла}[указатель] ключ [указатель] ключ [указатель]"
<< std::endl;
    root->print(B_factor, nullptr, nullptr);
    std::cout << " ============= Конец ===============" << std::endl;
}

void BTree::count_tree(){
    std::cout << " ============= Б-дерево ===============" << std::endl;
    std::cout << " ЧИСЛО КЛЮЧЕЙ" << std::endl;
    root->count_tree(B_factor, nullptr);
    std::cout << root->count_tree(B_factor, nullptr) << std::endl;
    std::cout << " ============= Конец ===============" << std::endl;
```

```
}

int Node::count_tree(unsigned int B_factor, Node* root){
    int number = 0;
    //std::cout << "{" << this << "}";
    for (int i = 0; i < B_factor - 1; i++) {
        if (keys[i] != -1){
            number++;
        }
    }

    for (int i = 0; i < B_factor; i++){
        if (pointers[i] != nullptr) {
            number = number + this->pointers[i]->count_tree(B_factor,
nullptr);
        }
    }
    return number;
}

void Node::print(unsigned int B_factor, Node* root, Node* parent){
    std::cout << "{" << this << "}";
    for (int i = 0; i < B_factor - 1; i++) {
        std::cout << "[" << pointers [i] << "]";
        std::cout << " " << keys[i] << " ";
    }
    std::cout << "["<< pointers[B_factor - 1] << "] ";

    std::cout << "__"<< TEMP_KEY << "__";//DEBUG
    std::cout << "<"<< TEMP_POINTER << "> \n ";

    for (int i = 0; i < B_factor; i++){
        if (pointers[i] != nullptr) {
            this->pointers[i]->print(B_factor, nullptr, nullptr);
        }
    }
}

void BTree::print_keys_in_string(){
    last_key = -1;
    std::cout << " ============= Б-дерево ===============" << std::endl;
    std::cout << "Печать по порядку всех ключей" << std::endl;
    root->print_keys_in_string(B_factor);
    std::cout << " \n============= Конец ===============" << std::endl;
}

int Node::print_keys_in_string(unsigned int B_factor){

    for (int i = 0; i < B_factor - 1; i++) {
        if(pointers[i] != nullptr) {
            pointers[i]->print_keys_in_string(B_factor);
        }
        if(keys[i] != -1) {
            std::cout << keys[i] << " ";
            if(last_key >= keys[i]){
                std::cout << "ERROR ";
            }
            last_key = keys[i];
        }
    }
    if(pointers[B_factor-1] != nullptr) {
        pointers[B_factor-1]->print_keys_in_string(B_factor);
    }
}
```

```
void Node::print_only_this(unsigned int B_factor, Node* root, Node* parent) {
    std::cout << "{" << this << "}";
    for (int i = 0; i < B_factor - 1; i++) {
        std::cout << "[" << pointers [i] << "]";
        std::cout << " " << keys[i] << " ";
    }
    std::cout << "["<< pointers[B_factor - 1] << "] ";

    std::cout << "__"<< TEMP_KEY << "__";//DEBUG
    std::cout << "<"<< TEMP_POINTER << "> \n ";
}

int Node::count_free_key(int B_factor) { //возвращает количество пустых
ключей (-1)
    int free_key_count = 0;
    for (int i = 0; i < B_factor - 1; i++ ) {
        free_key_count = free_key_count + (keys[i] == -1);
    }
    return free_key_count;
}

int Node::count_free_pointer(int B_factor) { //возвращает количество пустых
указателей (nullptr)
    int free_pointer_count = 0;
    for (int i = 0; i < B_factor; i++){
        free_pointer_count = free_pointer_count + (pointers[i] == nullptr);
    }
    return free_pointer_count;
}

void Node::refresh(unsigned int B_factor, Node * root, Node * parent) {
    //функция установит ключ и указатель из резервной позиции в основную для
соблюдения порядка
    //устанавливаем ключи

    int free_key_count = count_free_key(B_factor);
    int free_pointer_count = count_free_pointer(B_factor);

    int temp_key = this->TEMP_KEY;
    if (TEMP_KEY != -1) {
        for (int i = 0; i < B_factor; i++) {
            if (this->TEMP_KEY < keys[i]) {
                for (int k = B_factor - 1; k > i; k = k - 1) {
                    keys[k] = keys[k - 1];
                }
                keys[i] = temp_key;
                //TEMP_KEY = -1;//новые строки
                break;
            }
            if (keys[i] == -1){
                keys[i] = temp_key;
                //TEMP_KEY = -1;//новые строки
                break;
            }
        }
        if (free_key_count > 0) {
            TEMP_KEY = -1;
        }
    }

    //устанавливаем указатели
    if (TEMP_POINTER != nullptr) { //на случай если временного указателя нет
        Node * temp_pointer = this->TEMP_POINTER;
```

```
        for (int i = 0; i <= B_factor; i++) {
            if (pointers[i] == nullptr){

                pointers[i] = temp_pointer;
                //TEMP_POINTER = nullptr; //новые строки
                break;
            }
            if (temp_pointer->keys[0] < pointers[i]->keys[0]) {

                for (int k = B_factor; k > i; k = k - 1) {
                    pointers[k] = pointers[k - 1];
                }
                pointers[i] = temp_pointer;
                //TEMP_POINTER = nullptr;//новые строки
                break;
            }
        }
        if (free_pointer_count > 0) {
            TEMP_POINTER = nullptr; //ВЕРНУТЬ ИЗ КОММЕНТАРИЯ
        }
    }
}

void Node::add_to_any_child (unsigned int B_factor, int key, int
free_key_count, Node * root) {
    //переходим по указателю
    for (int k = 0; k < (B_factor -1) - free_key_count; k++){
        if (key < keys[k]) {
            this->pointers[k]->add( key, B_factor, pointers[k], root, this);
            break;
        }
        if (((B_factor - 2) - free_key_count) == k) { //если это последний
цикл, переходим в последний узел
            this->pointers[k+1]->add( key, B_factor, pointers[k+1] , root,
this);

            break;
        }
    }
}

void Node::add_only_to_this (unsigned int B_factor, int key, int
free_key_count, Node * root, Node * parent) {
    int a = 0;
    for(int i = 0; i < B_factor - 1; i++){
        if (key < keys[i]){
            //сдвинуть ключи и вставить наш.
            for (int j = B_factor - 3; j >= i; j = j - 1/*int j = i; j <
B_factor - 2; j++*/){

                keys[j + 1] = keys[j];
            }
            keys[i] = key;
            break;
        }
        if (keys[i] == -1) { //вставить наш следующим
            keys[i] = key;
            break;
        }
    }
}

Node * Node::find_this_parent(int B_factor, Node * active_node, Node * root){
```

```
    if(this == root){
        return nullptr;
    }
    //this это родителей которого мы ищем. Активный это тот с которого
начинаем
    Node * needed_parent = nullptr;
    for (int i = 0; i < B_factor; i++) {
        if (active_node->pointers[i] == this) {
            return active_node;
        }
    }
    //выберем в какой указатель переходить
    for (int k = 0; k < B_factor; k++){
        if (this->keys[0] <= active_node->keys[k]) {
            needed_parent = this->find_this_parent(B_factor, active_node-
>pointers[k],root);
            break;
        }
        if ((active_node->keys[k] == -1)) { //если это последний цикл,
переходим в последний узел
            needed_parent = this->find_this_parent(B_factor, active_node-
>pointers[k],root);
            break;
        }
    }

    return needed_parent;
}

void Node::Node_segmentation_root (unsigned int B_factor, Node* active_node,
int key, Node * root, Node * parent){
//пока опишем создания узла в случае если у родительского узла есть свободные
места
    int free_pointer_count = count_free_pointer(B_factor);
    this->refresh(B_factor,root, nullptr); //new

    //создаем 2 узла
    Node* a = active_node->pointers[0];
    Node* b = active_node->pointers[1];

    active_node->pointers[0] = new Node (B_factor);
    active_node->pointers[1] = new Node (B_factor);

     //если мы делим узел коренной и ничего не переносим наверх
        //определяем центр
        int center = (B_factor - 1) / 2;
        //заполняем первый до центра
        for (int i = 0; i < center; i++) {
            active_node->pointers[0]->keys[i] = active_node->keys[i];
        }
        //заполняем второй после центра
        for (int i = center + 1; i < (B_factor /*- 1*/); i++) {
            active_node->pointers[1]->keys[i - (center + 1)] = active_node-
>keys[i];
        }
        //сохраняем центр и удаляем все лишнее
        active_node->keys[0] = active_node->keys[center];
        for (int i = 1 /*был центр ну я сделал 1*/; i < B_factor/* - 1*/;
i++) {
            active_node->keys[i] = -1;
        }
        //если у узлов имеются ссылки
        if (free_pointer_count == 0) {
            active_node->pointers[0]->pointers[0] = a; //было active_node-
```

```
>pointers[0]->pointers[0] = a;
            active_node->pointers[0]->pointers[1] = b; //было active_node-
>pointers[0]->pointers[1] = a;

            //заполняем первый до центра
            for (int i = 2; i <= center; i++) {
                active_node->pointers[0]->pointers[i] = active_node-
>pointers[i];
                active_node->pointers[i] = nullptr;
            }
            //заполняем второй после центра
            for (int i = center + 1; i < B_factor /* 1 не было*/ +1; i++) {
                active_node->pointers[1]->pointers[i - (center + 1)] =
active_node->pointers[i];
                active_node->pointers[i] = nullptr;
            }

        }
}


void Node::Node_segmentation_round (unsigned int B_factor, Node* active_node,
int key, Node * root, Node * parent) {

    if (root->TEMP_POINTER != nullptr) {
        root->TEMP_POINTER->print(B_factor, root, nullptr);
    }

    if (this->TEMP_POINTER != nullptr) {
        this->TEMP_POINTER->print(B_factor, root, nullptr);
    }

    int free_pointer_count = count_free_pointer(B_factor);
    int free_key_count = count_free_key(B_factor);
    int center = (B_factor - 1) / 2;

    if (this == root){
        this->Node_segmentation_root(B_factor,this,key,root, nullptr);
        return;
    }

    if (parent->count_free_key(B_factor) > 0) { //если у родителей есть место

        if(this->TEMP_KEY == -1){
            this->TEMP_KEY = key;
        }
        //сначала добавляем резервные места в основу
        this->refresh(B_factor,root,parent);

        parent->TEMP_POINTER = new Node (B_factor); //вставляем в верхний
узел ссылку на новый

        parent->TEMP_KEY = this->keys[center]; //средний увели к родителям
        keys[center] = -1;

        for (int i = center + 1; i < (B_factor); i++) {
            parent->TEMP_POINTER->keys[i - (center + 1)] = this->keys[i];
            this->keys[i] = -1;
        }
        for (int i = center + 1; i <= B_factor; i++) {
            parent->TEMP_POINTER->pointers[i - (center + 1)] = this-
>pointers[i];
            pointers[i] = this->pointers[i] = nullptr;
        }
```

```
        parent->refresh(B_factor,root,nullptr); //средний распределили у
родителей
    } else {

        //сверху нет места, рекурсия

        if(this->TEMP_KEY == -1){
            this->TEMP_KEY = key;
        }
        this->refresh(B_factor,root,parent);

        parent->TEMP_POINTER = new Node (B_factor); //новый узел во временный
адрес наверх
        parent->TEMP_KEY = this->keys[center];//центральный ключ во временный
наверх
        this->keys[center] = -1;

        for (int i = center + 1; i < B_factor; i++) { //переносим ключи после
центрального в новый узел
            parent->TEMP_POINTER->keys[i - (center + 1)] = this->keys[i];
            this->keys[i] = -1;
        }
        // переместим указатели от центрального в новый узел и обнулим их
        for (int i = center + 1; i <= B_factor; i++) {
            parent->TEMP_POINTER->pointers[i - (center + 1)] = this-
>pointers[i];
            this->pointers[i] = nullptr;
        }

        //Рекурсия

        parent->Node_segmentation_round(B_factor, parent, key, root, parent-
>find_this_parent(B_factor,root,root));
    }
}


void Node::set(int keys_in[], Node **pointers_in) {
    keys = keys_in;
    pointers = pointers_in;
}

BTree::BTree() {
    B_factor = 4;
    root = new Node (B_factor);
    root->root();
}

BTree::BTree(int B_factor) {
    this->B_factor = B_factor;
    root = new Node (B_factor);
    root->root();
}

void BTree::add(int key) {
    root->add(key, B_factor, root, root, nullptr);
}

void Node::add(int key, unsigned int B_factor, Node * active_node, Node *
root, Node * parent) {
    int free_pointer_count = count_free_pointer(B_factor);
    int free_key_count = count_free_key(B_factor);

    //Если нет указателей и есть пустые места
```

```
    if ((free_pointer_count == B_factor) && free_key_count){ //добавляем в
этот узел
        //вводим новый ключ
        add_only_to_this(B_factor, key, free_key_count, root, parent);

        // Если есть пустые места, но есть указатели
    } else if ((free_key_count > 0) && (free_pointer_count < B_factor)) { //
выбираем указатель на ребенка и активируем добавление в ребенка

        this->add_to_any_child(B_factor, key, free_key_count, root);

    } else if (free_key_count == 0) { //если все места заняты, но есть пустые
указатели

          //если узел кореенной
        if (this == root) {
            if (((this->TEMP_KEY == -1) && (this->TEMP_POINTER == nullptr))
&& (free_pointer_count != B_factor)) {
                this->add_to_any_child(B_factor,key,free_key_count,root);
            } else {
                this->Node_segmentation_root(B_factor, active_node, key,
root, parent);
                this->add(key, B_factor, this, root, parent);
            }
            // если у родителей есть пустое место
        } else if ((free_key_count == 0) && (free_pointer_count == 0) &&
(TEMP_KEY == -1) && (TEMP_POINTER == nullptr)){
            this->add_to_any_child(B_factor, key, free_key_count, root);
        } else {
            this->Node_segmentation_round(B_factor, this, key, root, parent);
        }
    }
}

Node* BTree::search(int key){
      return root->search(key, 0, B_factor, root, root, nullptr);
}

Node* Node::search(int key, int level_down, int B_factor, Node * root, Node *
active_node, Node* node_with_key){
    //переберем ключи в этом узле
    level_down++;
    for (int i = 0; i < B_factor; i++) {
        if (key == this->keys[i]) {
            node_with_key = this;
            return node_with_key;//level_down;
        }
    }
    if (pointers[0] == nullptr) {
        return nullptr;
    }
    for (int i = 0; i < B_factor; i++) {
        if (key < this->keys[i]){
            node_with_key = pointers[i]->search(key,
level_down,B_factor,root,active_node, nullptr);
            return node_with_key;//level_down;
        }
        if (-1 == this->keys[i]){
            node_with_key = pointers[i]->search(key,
level_down,B_factor,root,active_node, nullptr);
            return node_with_key;//level_down;
        }
    }
```

```
    return node_with_key;//level_down;
}

//УДАЛЕНИЕ УЗЛА

int Node::count_keys(int B_factor) {
    return ((B_factor-1) - this->count_free_key(B_factor));
}

int Node::count_pointers(int B_factor) {
    return (B_factor - this->count_free_pointer(B_factor));
}

void Node::remove_free_pointer(int B_factor){
    for (int i = 0; i < B_factor - 1; i++){
        if(this->pointers[i] == nullptr){
            for (int k = i; k < B_factor - 1; k++){
                this->pointers[k] = this->pointers[k + 1];
            }
            this->pointers[B_factor-1] = nullptr;
        }
    }
}


void Node::remove_free_place(int B_factor ){
    for (int i = 0; i < B_factor - 2; i++){
        if(this->keys[i] == -1){
            for (int k = i; k < B_factor - 2; k++){
                this->keys[k] = this->keys[k + 1];
            }
            this->keys[B_factor-2] = -1;
        }
    }
}


void Node::del_key_only_this(int key, int B_factor, Node * node_with_key){
    for(int i = 0; i < B_factor - 1; i++){
        if(node_with_key->keys[i] == key){
            node_with_key->keys[i] = -1;
            break;
        }
    }
    node_with_key->remove_free_place(B_factor);
}

int Node::ask_brother_key(int key, int B_factor, Node * node_with_key, Node*
parent, Node * root){
    int parent_keys_amount = parent->count_keys(B_factor);
    int node_with_key_index = -1;

    //узнаем каким по порядку идет наш узел
    for (int i = 0; i < B_factor; i++){
        if(parent->pointers[i] == node_with_key){
            node_with_key_index = i;
        }
    }

    //обратимся к левому брату
    if (node_with_key_index > 0){
        int left_brother_key_amount = parent->pointers[node_with_key_index -
1]->count_keys(B_factor);
        if (left_brother_key_amount > 1){
```

```
            //возьмем отсюда крайний ключ правый и отправим родителю в резерв
            parent->TEMP_KEY = parent->pointers[node_with_key_index - 1]-
>keys[left_brother_key_amount - 1];
            parent->pointers[node_with_key_index - 1]-
>keys[left_brother_key_amount - 1] = -1;
            node_with_key->TEMP_KEY = parent->keys[node_with_key_index - 1];
            parent->keys[node_with_key_index - 1] = -1;
            parent->remove_free_place(B_factor);
            parent->refresh(B_factor, root, nullptr);

            node_with_key->refresh(B_factor,root,parent);
            node_with_key->del_key_only_this(key,B_factor,node_with_key);
//закоментим, потому что перекинем проблему
            return 1;
        }
    }

    //обратимся к правому брату
    if (parent->pointers[node_with_key_index + 1] != nullptr){ //обратимся к
правому брату
        if (parent->pointers[node_with_key_index + 1]->count_keys(B_factor) >
1) {
            int right_brother_key_amount = parent-
>pointers[node_with_key_index + 1]->count_keys(B_factor);
            //возьмем отсюда крайний ключ левый и отправим к родителю в
резерв
            parent->TEMP_KEY = parent->pointers[node_with_key_index + 1]-
>keys[0];
            parent->pointers[node_with_key_index + 1]->keys[0] = -1; //вместо
него -1
            parent->pointers[node_with_key_index + 1]-
>remove_free_place(B_factor); //уберем пробел
            parent->refresh(B_factor,root, nullptr);

            node_with_key->TEMP_KEY = parent->keys[node_with_key_index];
            parent->keys[node_with_key_index] = -1;
            parent->remove_free_place(B_factor);
            parent->refresh(B_factor,root, nullptr);

            node_with_key->refresh(B_factor,root,parent);
            node_with_key->del_key_only_this(key,B_factor,node_with_key);
//закоментим потому что перекинем проблему
        } else {
            //братья без лишних ключей
            return 0;
        }
    } else {
        //братья без лишних ключей
        return 0;
    }
}

int Node::merge_nodes_brothers(int key, int B_factor, Node* node_with_key,
Node* parent, Node* root, int parent_status) {
    if (1) {
        //тут уже  мы знаем что братьев можно сливать
        int node_with_key_index = -1;

        //узнаем каким по порядку идет наш узел
        for (int i = 0; i < B_factor; i++) {
            if (parent->pointers[i] == node_with_key) {
                node_with_key_index = i;
            }
        }
```

```
        //сначала проверям левого брата, потом правого
        if (node_with_key_index > 0) {
            int left_brother_key_amount = parent-
>pointers[node_with_key_index - 1]->count_keys(B_factor);
            if (1) {
                //будем скрещивать с левым
                node_with_key->del_key_only_this(key, B_factor,
node_with_key); //удалили ключ

                parent->pointers[node_with_key_index] = nullptr; //удалили
указатель
                parent->pointers[node_with_key_index - 1]->TEMP_KEY = parent-
>keys[node_with_key_index -1]; //левому брату скидываем ключ от родител

                if (parent_status == PARENT_HAS_MORE_KEYS) { //удаляем ключ
если сверху много
                    parent->keys[node_with_key_index - 1] = -1;
                }

                key = parent->keys[node_with_key_index - 1];

                parent->pointers[node_with_key_index - 1]->refresh(B_factor,
root, parent);
                parent->remove_free_place(B_factor);
                parent->remove_free_pointer(B_factor);

                if (parent_status == PARENT_HAS_ONE_KEY){ //перекидываем
проблему
                    parent->delete_key(parent->keys[node_with_key_index -
1],B_factor,root, parent->find_this_parent(B_factor,root,root),parent,USED);
                }
                else if (parent_status == PARENT_ROOT_HAS_ONE){
                    change_root(parent->pointers[0], root, B_factor);
                }
                return key;
            }
        }

        if (parent->pointers[node_with_key_index + 1] != nullptr) {
//обратимся к правому брату
            if (1) {
                node_with_key->del_key_only_this(key, B_factor,
node_with_key); //удалили ключ
                parent->pointers[node_with_key_index] = nullptr; //удалили
указатель
                parent->pointers[node_with_key_index + 1]->TEMP_KEY = parent-
>keys[node_with_key_index]; //правому брату скидываем ключ от родител

                if (parent_status == PARENT_HAS_MORE_KEYS) {
                    parent->keys[node_with_key_index] = -1;
                }

                parent->pointers[node_with_key_index + 1]->refresh(B_factor,
root, parent);
                parent->remove_free_place(B_factor);
                parent->remove_free_pointer(B_factor);

                if (parent_status == PARENT_HAS_ONE_KEY){
                    parent->delete_key(parent-
>keys[node_with_key_index],B_factor,root, parent-
>find_this_parent(B_factor,root,root),parent,USED);
                } else if (parent_status == PARENT_ROOT_HAS_ONE){
                    change_root(parent->pointers[0], root, B_factor);
```

```
                    //root = parent->pointers[0];
                }

                return key;
            }
        }
    }
}

int Node::lift_up_left (int key, int B_factor, Node * root, Node * parent,
Node * node_with_key, int key_index) {
    if (this == node_with_key){
        return this->pointers[key_index]-
>lift_up_left(key,B_factor,root,this,node_with_key,key_index);
    } else if(this->pointers[0] != nullptr){ //отправляем вниз
       return this->pointers[this->count_pointers(B_factor) - 1]-
>lift_up_left(key,B_factor,root,this,node_with_key,key_index);
   } else {
       if (this->count_keys(B_factor) > 1) {
           node_with_key->TEMP_KEY = this->keys[this->count_keys(B_factor) -
1];
           this->keys[this->count_keys(B_factor) - 1] = -1;
           node_with_key->refresh(B_factor, root, nullptr);
           return 1;
       } else {
           return 0;
       }
   }
}

int Node::lift_up_right (int key, int B_factor, Node * root, Node * parent,
Node * node_with_key, int key_index) {
    if (this == node_with_key) {
        return this->pointers[key_index +1]->lift_up_right(key, B_factor,
root, this, node_with_key, key_index);
    } else if(this->pointers[0] != nullptr){ //отправляем вниз
        return this->pointers[0]-
>lift_up_right(key,B_factor,root,this,node_with_key,key_index);
    } else {
        if (this->count_keys(B_factor) > 1) {
            node_with_key->TEMP_KEY = this->keys[0];
            this->keys[0] = -1;
            this->remove_free_place(B_factor);
            node_with_key->refresh(B_factor, root, nullptr);
            return 1;
        } else {
            return 0;
        }
    }
}

void Node::steal_down_key(int key, int B_factor, Node* node_with_key, Node*
parent, Node* root, int key_index){
    if (this == node_with_key){
        this->pointers[key_index]-
>steal_down_key(key,B_factor,node_with_key,this,root,key_index);
    } else if(this->pointers[0] != nullptr){ //отправляем вниз
        this->pointers[this->count_pointers(B_factor) - 1]-
>steal_down_key(key,B_factor,node_with_key,this,root,key_index);
    } else {
        if (1) {
            node_with_key->TEMP_KEY = this->keys[this->count_keys(B_factor) -
1];
            key = this->keys[this->count_keys(B_factor) - 1];
```

```
            node_with_key->refresh(B_factor, root, nullptr);
        }
        //теперь перекидываем удаление ключа вниз
        this->delete_key(key, B_factor, root, this-
>find_this_parent(B_factor,root,root), this, USED);
    }
}

int Node::ask_brother_key_with_pointers(int key, int B_factor, Node*
node_with_key, Node* parent, Node* root){
    //найдем индекс нашего узла сверху
    int node_index = -1;
    for (int i = 0; i < B_factor; i++){
        if(parent->pointers[i] == this){
            node_index = i;
        }
    }

    if (node_index > 0) {//спрашиваем левого брата
        int left_brother_key_amount = parent->pointers[node_index - 1]-
>count_keys(B_factor);
        Node* left_brother = parent->pointers[node_index - 1];
        if (left_brother_key_amount > 1){

            //берем ключ у левого
            //возьмем отсюда крайний ключ правый и отправим родителю в резерв
            parent->TEMP_KEY = left_brother->keys[left_brother_key_amount -
1];
            left_brother->keys[left_brother_key_amount - 1] = -1;

            this->TEMP_KEY = parent->keys[node_index - 1];
            parent->keys[node_index - 1] = -1;
            parent->remove_free_place(B_factor);
            parent->refresh(B_factor, root, nullptr);

            this->TEMP_POINTER = left_brother-
>pointers[left_brother_key_amount];
            left_brother->pointers[left_brother_key_amount] = nullptr;

            this->refresh(B_factor,root,parent);
            this->del_key_only_this(key,B_factor,node_with_key); //даже если
ключа у нас нет, она все равно не должна удалять
            return 1;
        }
    }
    if (parent->pointers[node_index + 1] != nullptr) { //спрашиваем правого
брата
        int right_brother_key_amount = parent->pointers[node_index + 1]-
>count_keys(B_factor);
        Node* right_brother = parent->pointers[node_index + 1];
        if (right_brother_key_amount > 1){
            //берем ключ у правого
            this->del_key_only_this(key,B_factor,node_with_key);
            this->TEMP_KEY = parent->keys[node_index];

            parent->keys[node_index] = -1;
            parent->remove_free_place(B_factor);
            parent->TEMP_KEY = right_brother->keys[0];
            right_brother->keys[0] = -1;
            parent->refresh(B_factor, root, nullptr);
            this->TEMP_POINTER = right_brother->pointers[0];
            right_brother->pointers[0] = nullptr;
            right_brother->remove_free_place(B_factor);
            right_brother->remove_free_pointer(B_factor);
```

```
            this->refresh(B_factor,root,parent);
            return 1;
        }
    }
    return 0;
}

void Node::change_root(Node* new_root, Node* root, int B_factor){
    //передаем ключи
    for (int i = 0; i < B_factor -1; i++){
        root->keys[i] = new_root->keys[i];
    }
    //передаем указатели
    for (int i = 0; i < B_factor; i++){
        root->pointers[i] = new_root->pointers[i];
    }
}

void Node::merge_nodes_brothers_with_pointers(int key, int B_factor, Node*
node_with_key, Node* parent, Node* root, int parent_status){
    int node_with_key_index = -1;

    //узнаем каким по порядку идет наш узел
    for (int i = 0; i < B_factor; i++) {
        if (parent->pointers[i] == node_with_key) {
            node_with_key_index = i;
        }
    }

    if (node_with_key_index > 0) { //работаем с левым
        Node* left_brother = parent->pointers[node_with_key_index -1];

        this->del_key_only_this(key,B_factor,node_with_key);
        this->TEMP_KEY = parent->keys[node_with_key_index - 1];
        this->refresh(B_factor,root,parent);

        if (parent_status == PARENT_HAS_MORE_KEYS) {
            parent->keys[node_with_key_index - 1] = -1;
            parent->remove_free_place(B_factor);
        }
        this->TEMP_KEY = left_brother->keys[0];
        this->TEMP_POINTER = left_brother->pointers[0];
        this->refresh(B_factor,root,parent);
        this->TEMP_POINTER = left_brother->pointers[1];
        this->refresh(B_factor,root,parent);

        parent->pointers[node_with_key_index - 1] = nullptr;
        parent->remove_free_pointer(B_factor);

        //в конце
        key = parent->keys[0];

        if (parent_status == PARENT_HAS_ONE_KEY){
            parent->delete_key(key/*parent-
>keys[node_with_key_index]*/,B_factor,root, parent-
>find_this_parent(B_factor,root,root),parent,USED);
        } else if (parent_status == PARENT_ROOT_HAS_ONE){
            change_root(parent->pointers[0], root, B_factor);
        }
        return;
    }

    if (parent->pointers[node_with_key_index + 1] != nullptr) { //обратимся к
```

```
правому брату
        Node* right_brother = parent->pointers[node_with_key_index + 1];

        this->del_key_only_this(key,B_factor,node_with_key);
        this->TEMP_KEY = parent->keys[node_with_key_index];
        this->refresh(B_factor,root,parent);
        if (parent_status == PARENT_HAS_MORE_KEYS) {
            parent->keys[node_with_key_index] = -1;
            parent->remove_free_place(B_factor);
        }
        this->TEMP_KEY = right_brother->keys[0];
        this->TEMP_POINTER = right_brother->pointers[0];
        this->refresh(B_factor,root,parent);
        this->TEMP_POINTER = right_brother->pointers[1];
        this->refresh(B_factor,root,parent);

        parent->pointers[node_with_key_index + 1] = nullptr;
        parent->remove_free_pointer(B_factor);

        //в конце
        key = parent->keys[0];
        if (parent_status == PARENT_HAS_ONE_KEY){
            parent->delete_key(key/*parent-
>keys[node_with_key_index]*/,B_factor,root, parent-
>find_this_parent(B_factor,root,root),parent,USED);
        } else if (parent_status == PARENT_ROOT_HAS_ONE){
            change_root(parent->pointers[0], root, B_factor);
        }
        return;
    }
}

void BTree::delete_key(int key){
    Node* node_with_key = root->search(key, 0, B_factor, root, root,
nullptr);
    if (node_with_key == nullptr){
        std::cout << "ОШИБКА! НЕТ ТАКОГО КЛЮЧА!" << std::endl;
        return;
    }
    node_with_key->delete_key(key, B_factor, root, node_with_key-
>find_this_parent(B_factor,root,root), node_with_key, NOT_USED);
}

void Node::delete_key(int key, int B_factor, Node * root, Node * parent, Node
* node_with_key, int is_used){
    int success = 0;  // 1 означает что успешно
    //найдем адрес узла с ключем, проверим есть ли такой ключ
    if (node_with_key == nullptr){
        std::cout << "ОШИБКА! НЕТ ТАКОГО КЛЮЧА!" << std::endl;
        return;
    }
    //работа в листьях
    if (node_with_key->pointers[0] == nullptr) { // если нет детей
        if (node_with_key->count_keys(B_factor) > 1) { //если больше 2 ключей
и нет детей
            node_with_key->del_key_only_this(key, B_factor, node_with_key);
            return;
        }
        else if (node_with_key->count_keys(B_factor) == 1){ //остался один
ключ, спросим у братьев соседей
            success = success +
ask_brother_key(key,B_factor,node_with_key,parent,root);
        }
        if ((success) == 0){
```

```
            //объеденимся с братом, даже если у родителя всего один ключ
            if (parent->count_keys(B_factor) > 1) {
                node_with_key->merge_nodes_brothers(key, B_factor,
node_with_key, parent, root, PARENT_HAS_MORE_KEYS);
            } else if (parent == root) {
                node_with_key->merge_nodes_brothers(key, B_factor,
node_with_key, parent, root, PARENT_ROOT_HAS_ONE);
            }else if (parent->count_keys(B_factor) == 1) { //у родителей
СЕЙЧАС КОНЧАТСЯ КЛЮЧИ
                node_with_key->merge_nodes_brothers(key, B_factor,
node_with_key, parent, root, PARENT_HAS_ONE_KEY);
            }
        }
    } else { //узел - ветка
        int key_index = -1;
        for(int i = 0; i < B_factor - 1; i++){
            if(node_with_key->keys[i] == key){
                key_index = i;
            }
        }
        if (is_used == NOT_USED) { //идем вниз воровать ключи
            //идем в вниз до узлов без детей и если возможно берем крайние
ключи
            del_key_only_this(key,B_factor,node_with_key); //удалим ключ
            success = node_with_key-
>lift_up_left(key,B_factor,root,parent,node_with_key, key_index);
            if (success == 0){
                success = node_with_key-
>lift_up_right(key,B_factor,root,parent,node_with_key, key_index);
            }
            if (success == 1){
                return;
            }
            //если success  будет равен 0, то не сработало
        }
        if ((is_used == NOT_USED) && (success == 0)) { //если не удалось
забрать ключи, забираем силой
            //украсть ключ снизу
            node_with_key-
>steal_down_key(key,B_factor,node_with_key,parent,root,key_index);
            return;
            //после того как забрали силой нужно ДЕЛЕГИРОВАТЬ ОТСУТСТВИЕ
КЛЮЧА В САМЫЙ НИЗ
        }
        if (1){ //будем брать ключ у братьев с детьми (если у них есть больше
1)
            success = success + node_with_key-
>ask_brother_key_with_pointers(key,B_factor,node_with_key,parent,root);
            if (success == 1) {
                return;
            }
        }
        if (parent->count_keys(B_factor) > 1) {
            node_with_key-
>merge_nodes_brothers_with_pointers(key,B_factor,node_with_key,parent,root,PA
RENT_HAS_MORE_KEYS);
        } else if (parent == root) {
            node_with_key-
>merge_nodes_brothers_with_pointers(key,B_factor,node_with_key,parent,root,PA
RENT_ROOT_HAS_ONE);
        }else if (parent->count_keys(B_factor) == 1) { //у родителей СЕЙЧАС
КОНЧАТСЯ КЛЮЧИ
            node_with_key-
>merge_nodes_brothers_with_pointers(key,B_factor,node_with_key,parent,root,PA
```

```
RENT_HAS_ONE_KEY);
            }
        }
}
```