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 | *(подпись)* | Копейко И.В. |
| Работу принял | *(подпись)* | Ананьевский М.С. |

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.

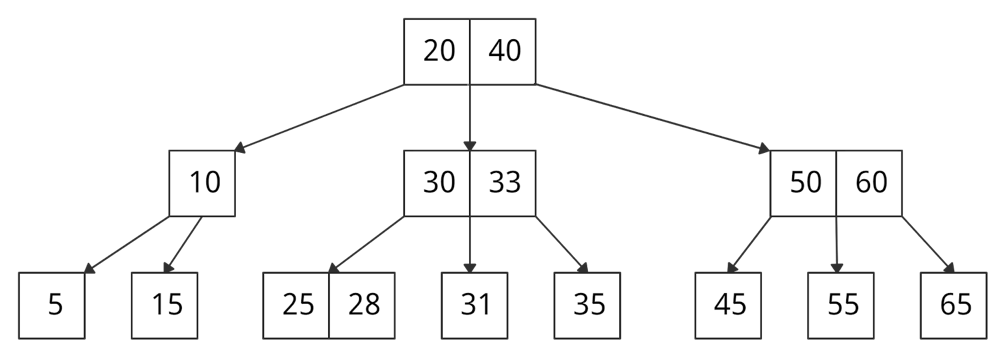


Рисунок 1 – Пример структуры B-дерева

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

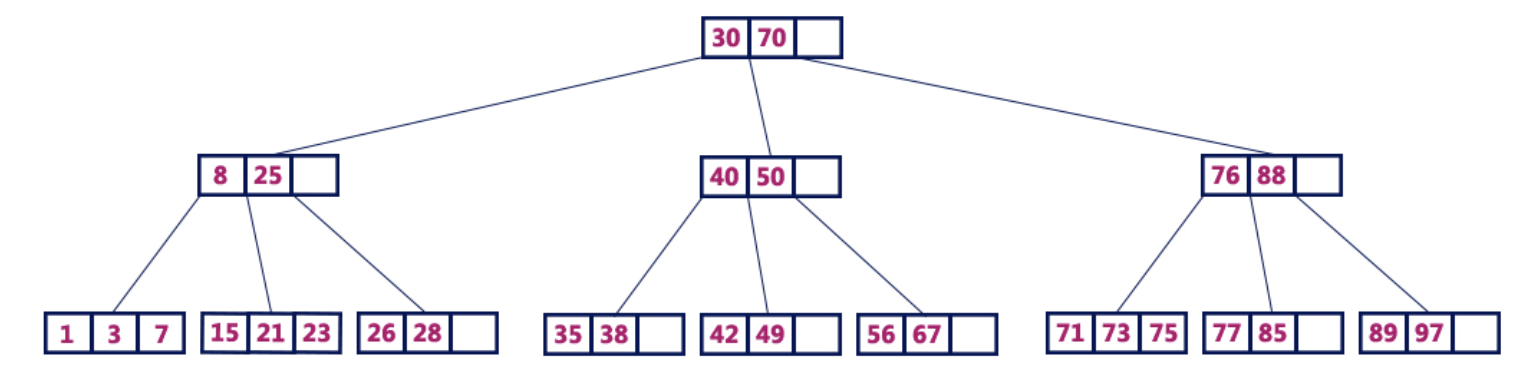
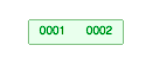


Рисунок 2 – B-Дерево с Т = 4

**Визуальная демонстрация алгоритма добавления**

*Пример сценария 1(обычное добавление):*



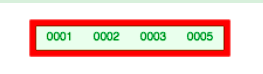
Добавляем 3



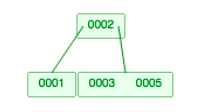
*Пример сценария 2 (деление коренного узла):*

**

Добавляем 5

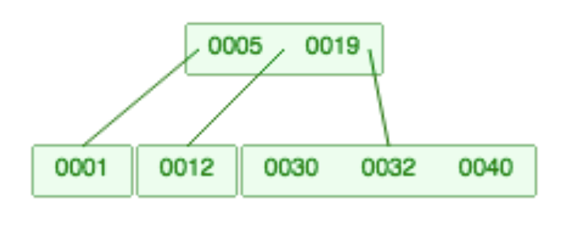


Узел переполнен, он делится

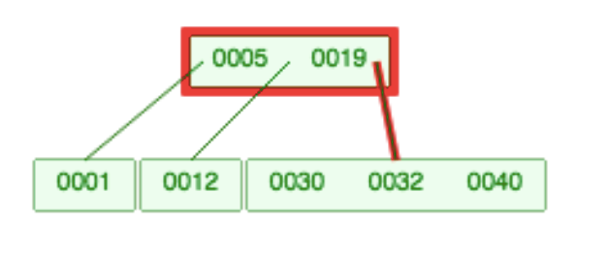


*Пример сценария 3 (деление узла):*

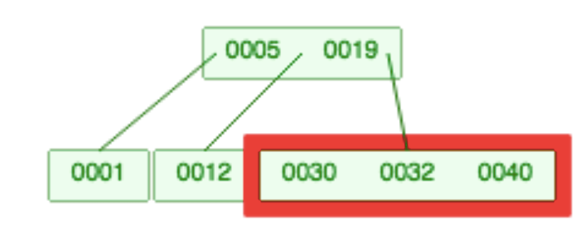
Имеется другое B-дерево с T = 4, решаем добавить ключ 25.



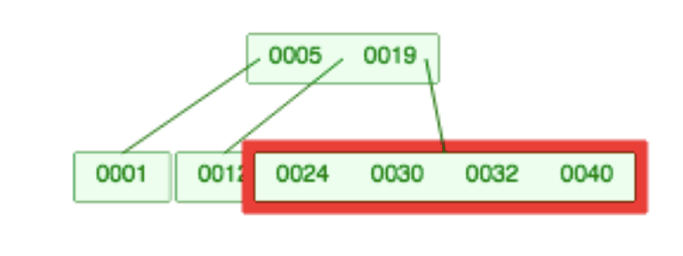
Мы переходим в правый узел



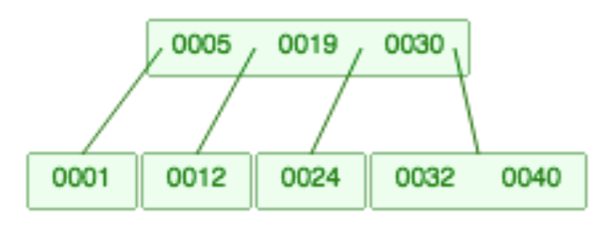
Добрались до нужного узла



Узел переполнен, его разбивают

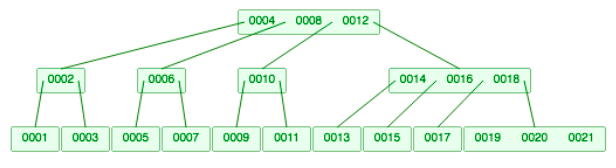


Итог операции добавления

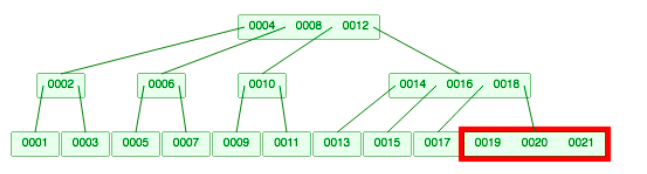


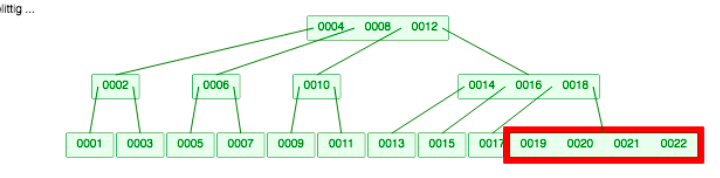
*Пример сценария 4 (деление узла рекурсивно):*

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

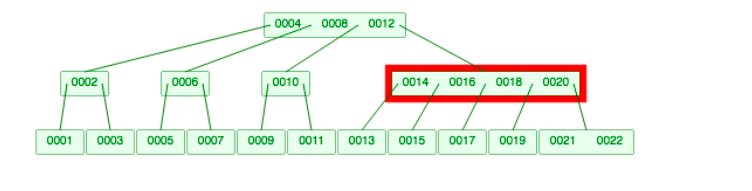


Добавляем 22 в нижний узел

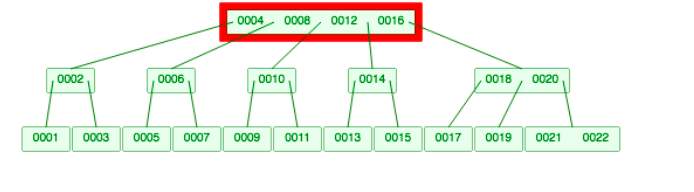




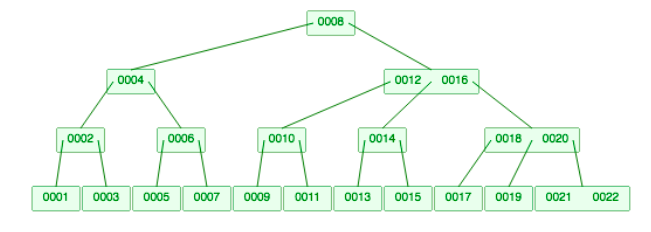
Узел переполнен, он делится



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



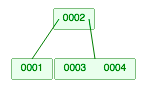
То же самое, делится коренной узел



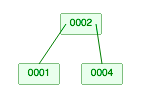
Дерево всегда растет вверх!

**Визуальная демонстрация алгоритма удаления**

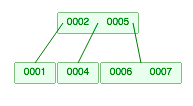
*Пример сценария 1 (простое удаление):*



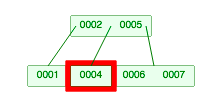
Удалим ключ 3



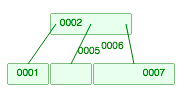
*Пример сценария 2 (взятие ключа у брата):*

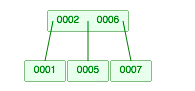


Удаляем ключ 4

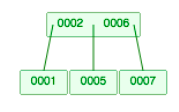


Он единственный в своем узле

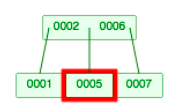


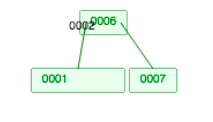


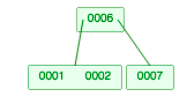
*Пример сценария 3 (объединяем узлы):*



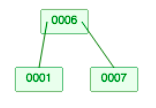
Удаляем ключ 5







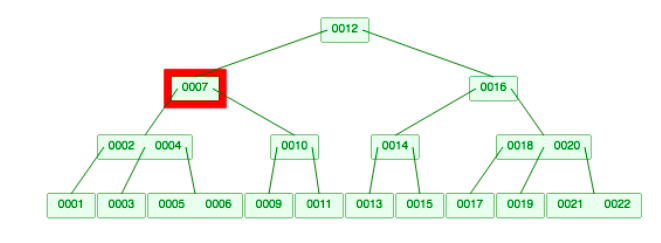
*Пример сценария 4 (объединяем коренной узел):*



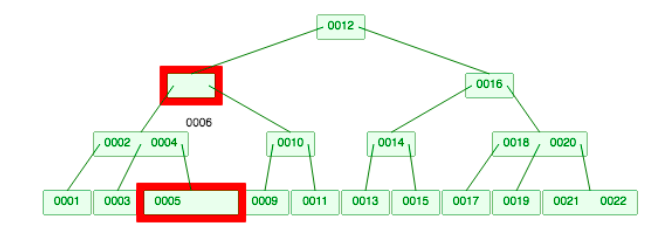
Удаляем ключ 1



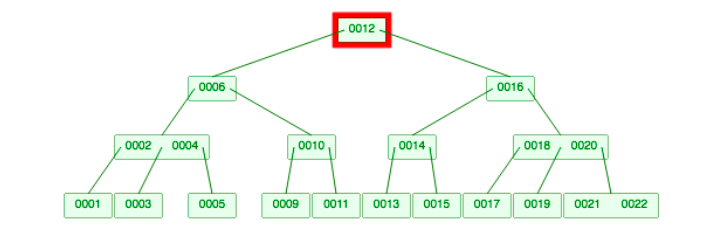
*Пример сценария 5 (берем ключ из нижнего узла):*

**

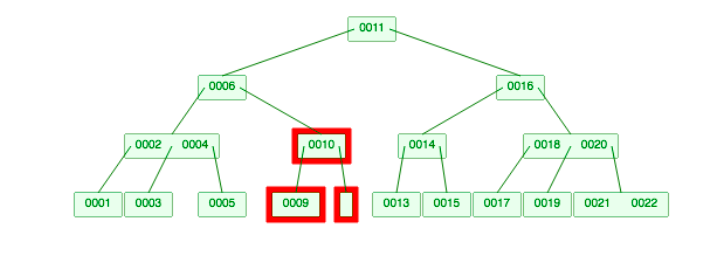
Удаляем ключ 7



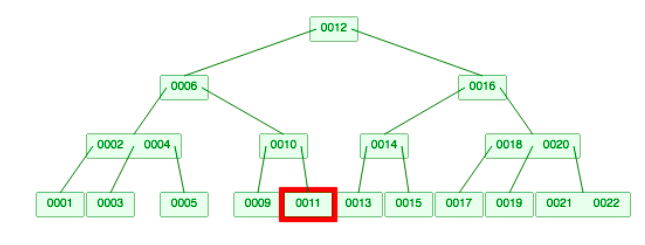
*Пример сценария 6 ( забираем ключи из нижнего узла):*



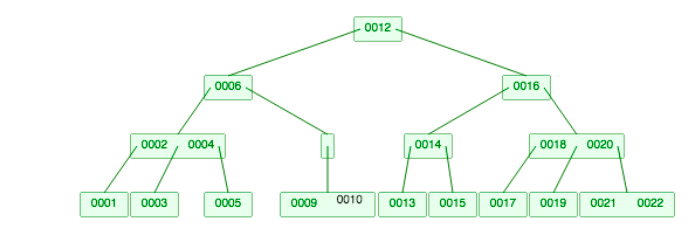
Удаляем ключ 12

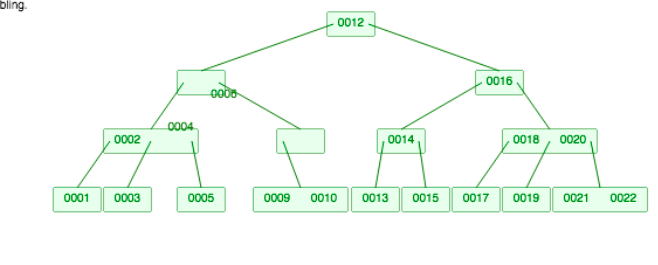


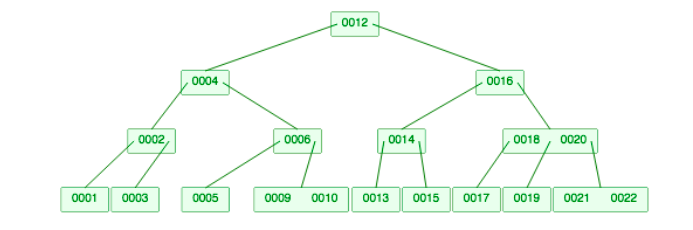
*Пример сценария 7 (рекурсивно забираем ключи):*

**

Удаляем ключ 11

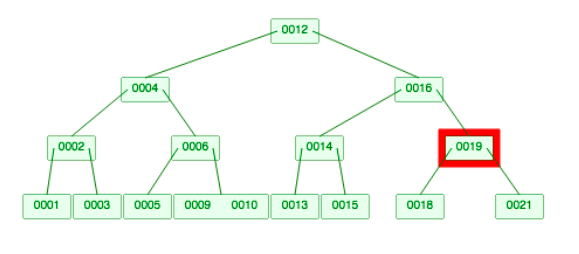




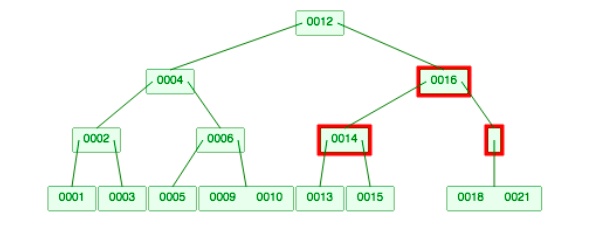


Вместе с ключом был передан еще и указатель

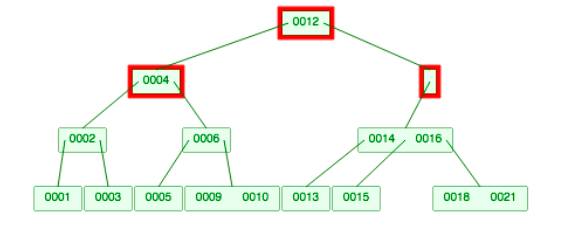
*Пример сценария 8 (рекурсивно забираем ключи):*

**

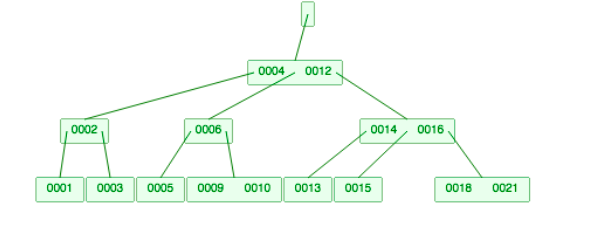
Удаляем ключ 19

**

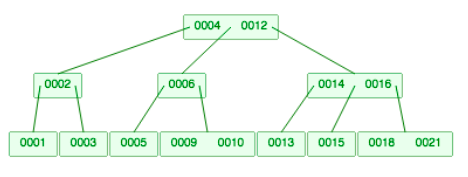
Нижние узлы под ключом 19 объединились, теперь будет происходить объединение с братом

**

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))

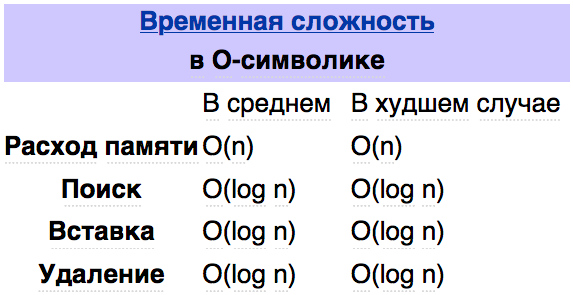


Рисунок 3 – Временная сложность алгоритма

Итак, исследуем временную сложность *алгоритма при добавлении*:

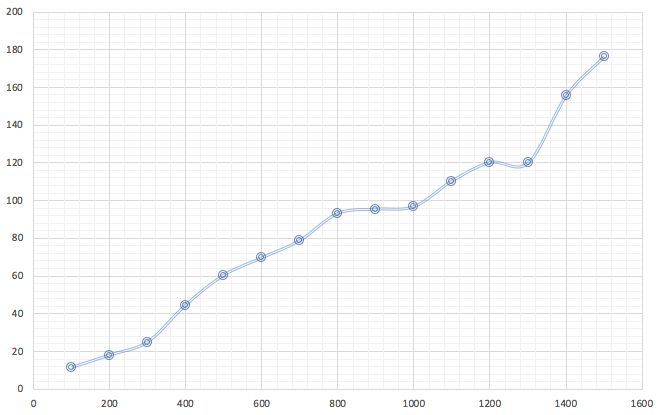


Рисунок 4 – Временная сложность алгоритма при добавлении на малом промежутке

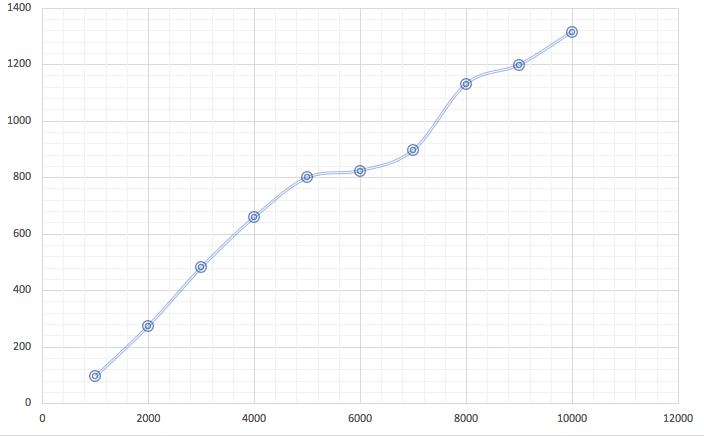
****

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:*

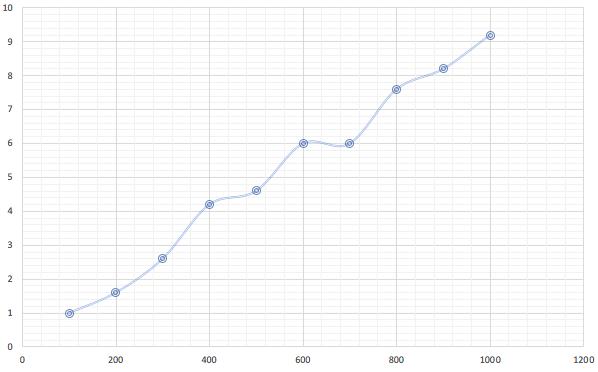


Рисунок 6 – Временная сложность алгоритма при удалении

Исследуем временную сложность *алгоритма при поиске:*

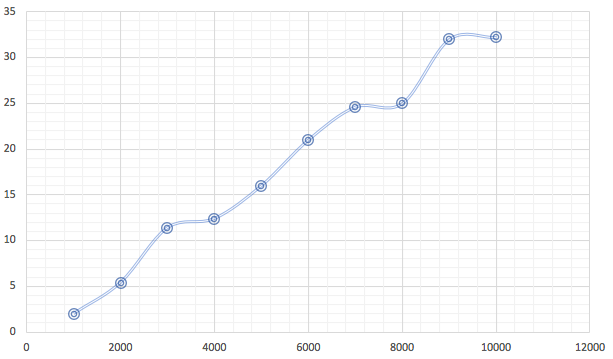


Рисунок 7 – Временная сложность алгоритма при поиске

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;  
 }  
 }  
  
 //cначала 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,PARENT\_HAS\_MORE\_KEYS);  
 } else if (parent == root) {  
 node\_with\_key->merge\_nodes\_brothers\_with\_pointers(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\_with\_pointers(key,B\_factor,node\_with\_key,parent,root,PARENT\_HAS\_ONE\_KEY);  
 }  
 }  
}