Санкт-Петербургский политехнический университет Петра великого
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

Курсовой проект
по дисциплине «Объектно-ориентированное программирование»
«B+-дерево»

Выполнил
студент гр. 3331506/90401 Яковлев Д.Э.

Работу принял Ананьевский М.С.

Санкт-Петербург
2022

Оглавление

Введение

B+ дерево — это оптимизация B-дерева, сбалансированное n-арное дерево поиска с переменным, но зачастую большим количеством потомков в узле, где значения хранятся только в листьях, а в узлах копии этих значений. В этой оптимизации больше ключей умещается в узел, что увеличивает степень ветвления. Пример B+-дерева представлен на рисунке 1.

Изначально структура предназначалась для хранения данных в целях эффективного поиска в блочно-ориентированной среде хранения — в частности, для файловых систем; применение связано с тем, что в отличие от бинарных деревьев поиска, B^+-деревья имеют очень высокий коэффициент ветвления (число указателей из родительского узла на дочерние — обычно порядка 100 или более), что снижает количество операций ввода-вывода, требующих поиска элемента в дереве.

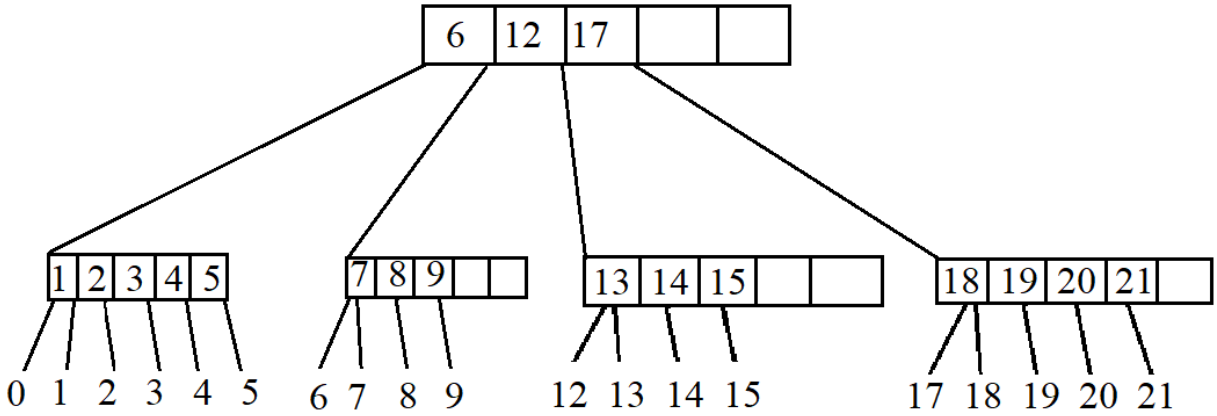

Рисунок 1 – Пример B+-дерева

Описание алгоритма

B+-дерево состоит из корня, внутренних узлов и листьев.

Имеются следующие правила:

• Ключи содержат только листья. Узлы содержат копии ключей.

• В каждом узле содержатся минимум (t-1) копий и t указателей. Максимум копий в узле – (2t-1), указателей – 2t. Корень же содержит от

1 до (2t-1) копий и от 2 до 2t указателей. (Указатели – ссылки на узлов-потомков/листья). Все копии и указатели упорядочены по возрастанию. Здесь t — параметр дерева определяющий количество копий и указателей, иногда именуемый b-фактором, коэффициентом ветвления, не меньший 2 (и обычно принимающий значения от 50 до 2000. Поэтому B+-дерево сильноветвистое).

• У листьев нет потомков (указателей).

• Глубина всех листьев одинакова.

• Листья имеют ссылку на соседа, позволяющую быстро обходить дерево в порядке возрастания ключей, и ссылки на данные.

Описание реализация алгоритма

B+-дерево – структура данных, предназначенная для эффективного доступа к информации. При работе алгоритма реализуются добавление, удаление и поиск ключа.

Поиск

Отправная точка – Корень B+-дерева. Начинаем с него:

1. Проходимся по копиям пока не найдем копию больше искомого значения, тогда переходим к потомку перед этой копией.

1.2. Если значение нашего листа больше всех имеющихся копий – переходим к последнему сыну в данном узле.

2. Повторять эти действия пока не дойдем до листа

Добавление

Для добавления нового листа в первую очередь необходимо найти предлистовой узел, в который его необходимо добавить. В этом случае алгоритм таков:

• Если предлистовой узел полностью не заполнен (количество указателей после вставки не более чем **2t**, то создать лист и добавить ссылку на него, а также копию значения листа.

• В противном случае необходимо расщепить узел:

• создать новый узел, затем переместить половину элементов из текущего в новый;

• создать копию наименьшего ключа из нового узла и адрес на него (узел) в родительский;

• если родительский узел заполнен, аналогично разделить его;

• повторять до тех пор, когда родительский узел не будет нуждаться в расщеплении.

• Если расщепляется корень — создать новый корень, имеющий одну копию и два указателя.

Пример операции добавления представлен на рисунке 2.

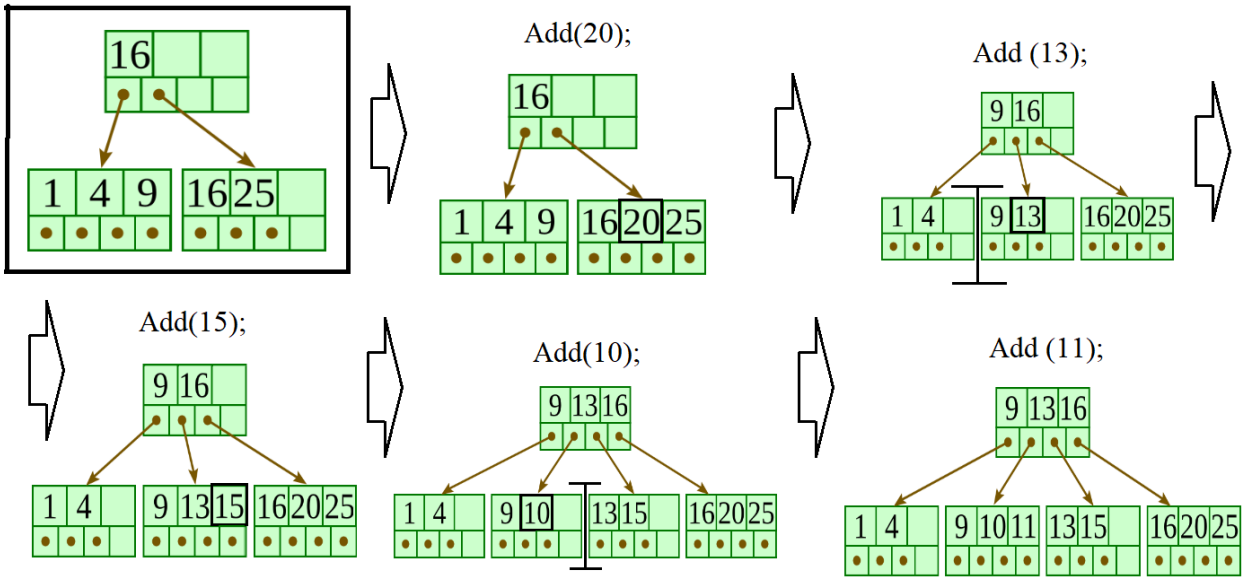

Рисунок 2 – Операции добавления узла

Удаление

Для удаления листа в первую очередь необходимо найти предлистовой узел, в котором есть ссылка на искомый лист. Алгоритм удаления от предлистового узла:

• Если узел хотя бы наполовину заполнен — завершение алгоритма;

• Если узел имеет меньше элементов, то:

• если «брат» — элемент с общим предком, заполнен больше чем на половину выполнить попытку перераспределения элементов, то есть добавить в узел элемент из «брата» .

• если выполнить перераспределение не удалось, объединить узел с «братом».

• Если произошло объединение, удалить копию ключа и указатель, которые ссылаются на удалённый узел или его «брата» из родительского узла.

Объединение может распространяться на корень, тогда происходит уменьшение высоты дерева.

Исследование алгоритма

1. Теоретическая временная сложность представлена на рисунке 3.

B+ дерево

Тип Дерево (структура данных)

Временная сложность

Алгоритм	Среднее	Худший случай
Поиск	O(log *n*)	O(log *n* + log *L*)
Вставить	O(log *n*)	O(M*log *n* + log *L*)
Удалить	O(log *n*)	O(M*log *n* + log *L*)

Рисунок 3 – Временная сложность, заявленная в Википедии

Проверим экспериментально время выполнения алгоритма при t=5.

2. Усредненные данные времени выполнения операций поиска, создания n-количества листов, удаления представлены в таблицах 1-3 соответственно.

Таблица 1 – Временная сложность операции поиска

Искомое значение:	Случайное	Последний лист	Первый лист
Количество листов	Затраченное время, нс		
10	400	600	300
100	800	500	600
1 000	1 100	1 200	1 000
10 000	1 600	800	1 300
100 000	1 900	600	1 700
1 000 000	2 200	800	2 300

Таблица 2 – Временная сложность операции добавления n – листов

Значение в ключах:	Случайное	Непрерывно растет	Непрерывно падает
Количество создаваемых листов	Затраченное время, мкс		
1	11	10	9
10	33	35	27
100	124	424	180
1 000	343	2 551	2 434
10 000	3 765	23 268	16 909
100 000	28 492	180 887	135 031
1 000 000	183 017	1 850 540	1 378 438

Таблица 3 – Временная сложность операции удаления

Значение в ключах:	Случайное	Непрерывно растет	Непрерывно падает
Количество созданных листов	Затраченное время, нс		
1	5600	5800	4900
10	4800	5900	5900
100	4800	7000	6600
1 000	7600	6500	7800
10 000	6800	8400	8200
100 000	7300	8200	9400
1 000 000	5600	5800	4900

Все выполненные измерения проводились с помощью библиотеки «chrono».

3. Графики времени выполнения операций поиска, создания n-количества листов, удаления представлены на рисунках 4–6 соответственно.

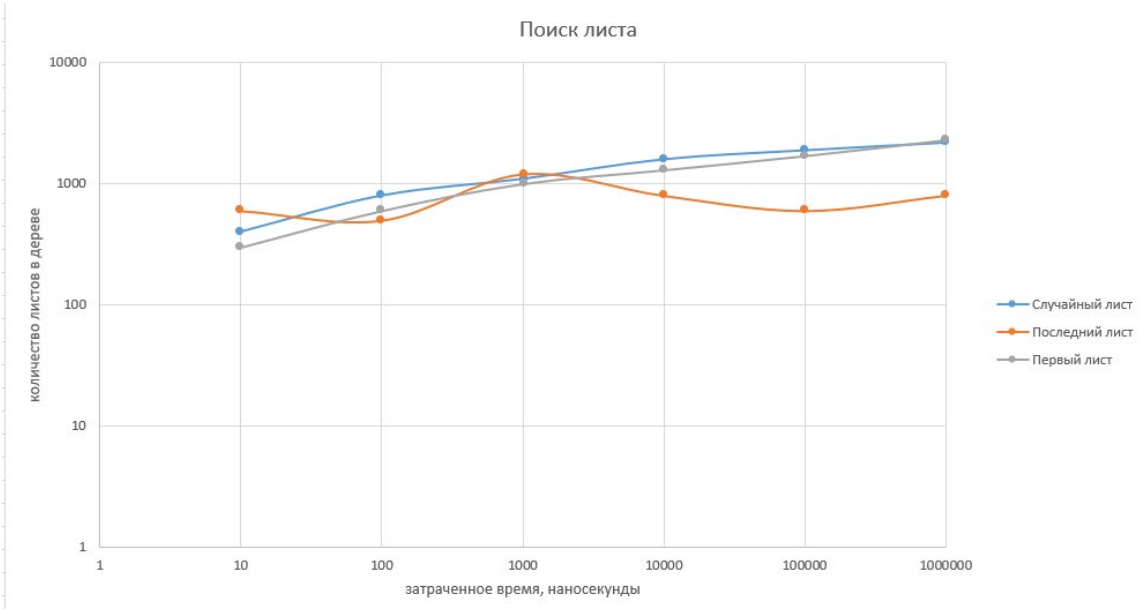

Рисунок 4 – Временная сложность операции поиска

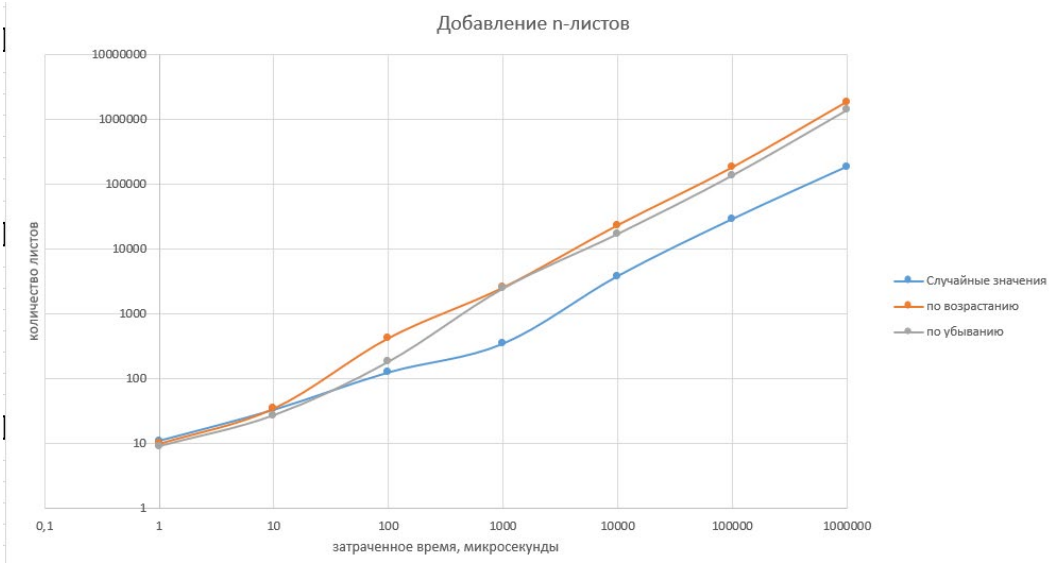

Рисунок 5 – Временная сложность операции добавления n – листов

Из графиков мы можем сделать вывод, что в моей работе временная сложность у функция имеет линейную зависимость, нежели заявленную логарифмическую.

Область применения

Файловые системы ReiserFS, NSS, XFS, JFS, ReFS и BFS используют этот тип дерева для индексации метаданных; BFS также использует деревья B+ для хранения каталогов. NTFS использует деревья B+ для индексации каталогов и метаданных, связанных с безопасностью. APFS использует деревья B+ для хранения сопоставлений идентификаторов объектов файловой системы с их расположениями на диске и для хранения записей файловой системы (включая каталоги), хотя в листовых узлах этих деревьев отсутствуют родственные указатели. Системы управления реляционными базами данных, такие как IBM Db2, Informix, Microsoft SQL Server, Oracle 8, Sybase ASE, и SQLite поддерживают этот тип дерева для индексов таблиц. Системы управления базами данных типа "ключ-значение", такие как CouchDB и Tokyo Cabinet, поддерживают этот тип дерева для доступа к данным.

Заключение

Был изучен и написан алгоритм B+-дерево, а также проведены исследования этого алгоритма. Реализация его оказалась довольна трудна вследствие предусмотра большого количества критичных сценариев, однако несмотря на громоздкий код алгоритм B+-дерево прост и понятен.

Список литературы

1. B+ дерево [Электронный ресурс] : Материал из Википедии — свободной энциклопедии: — Режим доступа: https://ru.wikipedia.org/?curid=2069640&oldid=119870344
2. B+ дерево [Электронный ресурс] : Материал из Национальной библиотеки им. Н. Э. Баумана — Режим доступа: https://ru.bmstu.wiki/B+_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE
3. B+ дерево [Литрература] : Дональд Кнут. Генерация всех деревьев. История комбинаторной генерации // Искусство программирования = The Art of Computer Programming. — М.: «Вильямс», 2007. — Т. 4. — С. 160.

Приложение

Код из файла libraryseul.h

```
#ifndef TOKIO3_LIBRARYSEUL_H
#define TOKIO3_LIBRARYSEUL_H

#include <iostream>
#include <cstring>
#include <vector>

#define MAX 2147483647
#define MIN -2147483648

using namespace std;

class BPTree;
class Node;

class Node{
    friend class BPTree;
private:                    //для узла
    int max_ref_child_id;               //заполненность
    vector <Node*> child_array;         // массив со ссылками на детей
    int *key_copy;                      //массив копий ключей
    int size_array;                     //размер массивов, равный бфактору
private:                  //для листа
    bool leaf;                      //лист если тру
    Node* neighbour;                //ссылка на соседа
    int key_data;                   //значение в листе
private:
    Node(int data_leaf_or_bfactor,bool leaf_or_node);
    Node(const Node &node);      //конструктор копирования
    Node(Node &&node) noexcept;  //конструктор перемещений
    ~Node();                     //деструктор
};

class BPTree{
    friend class Node;
private:
    int b_factor;               //ветвистость = 2t-1 так как нумерация с нуля
    Node* root;                //ссылка на текущий корень дерева
    Node* newroot;              //временный корень, так как у меня там что-то
все ломается при делении корня
private:                            //вспомогательные плюшки
    int H;                     //высота дерева
    bool Hplus;                 //высота изменилась
    Node* *ptr_path_array;     //массив ссылок узлов от корня до нужного
листа
public:
    BPTree(int t);
    virtual ~BPTree();
public:
    bool search(int leaf);
    void* add(int leaf); //работает, но с рандомом при миллионе листов
ломается
    void* del(int leaf);
    void print();
private://for search
    bool search(int leaf, Node* ref_node, int floor);
private://for add
    int search_place_for_add(int leaf, Node* &ref_node, int floor, int
floors_for_separation);
```

```
    void* node_separation(Node* &ref_node,Node* &ref_parent_node, int floor,
int leaf);
private://for add and dell
    int search_for_the_first_leaf_of_this_branch(Node* ref_node, int floor);
    Node* search_neighbour_left(int leaf, Node* &ref_node, int floor);
    Node* search_neighbour_right(int leaf, Node* &ref_node, int floor,Node*
minimally_larger_node, int mln_floor);
    Node* search_leaf_neighbour(Node* ref_node, int floor);
private://for dell
    bool search_place_for_del(int leaf, Node* &ref_node, int floor);
    void* tree_edits_after_deletion(Node* &ref_node,Node* &ref_parent_node,
int floor);
private://for print
    Node* search_once_leaf(Node* ref_node, int floor);
    Node* print_leaf(Node* ref_leaf);
};

#endif //TOKIO3_LIBRARYSEUL_H
```

Код из файла lidraryseul.cpp

```
#include "libraryseul.h"

Node::Node(int data_leaf_or_bfactor,bool leaf_or_node) {
    if (leaf_or_node) {
        key_data = data_leaf_or_bfactor;
        neighbour = nullptr;
    }
    else {
        size_array = data_leaf_or_bfactor+1;
        max_ref_child_id = -1;
        key_copy = new int[size_array];
        child_array.resize(size_array, nullptr);
        key_copy[0] = MIN;
        for (int i = 1; i < size_array; i++) {
            key_copy[i] = MAX;
        }
    }
    leaf = leaf_or_node;
}
Node::Node(const Node &node){
    this-> size_array = node.size_array;
    if (leaf) {
        this-> key_data = node.key_data;
        this-> neighbour = node.neighbour;
    }
    else {
        this-> max_ref_child_id = node.max_ref_child_id;
        key_copy = new int[size_array];
        child_array.reserve(node.child_array.size());
        memcpy(key_copy, node.key_copy,sizeof(int)*(size_array));
        memcpy(&child_array.at(0), &node.child_array.at(0),
node.child_array.size());
    }
    this-> leaf = node.leaf;
}
Node::Node(Node &&node) noexcept {
    size_array = node.size_array;
    leaf = node.leaf;
    node.size_array = false;
    node.leaf = false;
    if (leaf) {
        key_data = node.key_data;
        neighbour = node.neighbour;
        node.key_data = 0;
        node.neighbour = nullptr;
    }
    else {
        key_copy = node.key_copy;
        child_array = node.child_array;
        max_ref_child_id = node.max_ref_child_id;
        node.key_copy = nullptr;
        node.child_array.clear();
        node.max_ref_child_id = 0;
    }
}
Node::~Node() {
    if (leaf) {
        key_data = 0;
        neighbour = nullptr;
    }
    else {
        delete key_copy ;
```

```
        child_array.clear();
        max_ref_child_id = 0;
    }
}

BPTree::BPTree(int t) {
    H = 0;
    Hplus = false;
    b_factor = 2*t-1;
    root = new Node(b_factor,false);
}
BPTree::~BPTree(){
    while(root!= nullptr){
        del(search_for_the_first_leaf_of_this_branch(root, 0));
    }
    b_factor=0;
    H=0;
    Hplus=false;
}

///готово
bool BPTree::search(int leaf){
    if (H==0) if (root->child_array[0] == nullptr) return
false;//единственный случай пустоты - ни один лист не создан
    return search(leaf, root, 0);
}
bool BPTree::search(int leaf,Node* ref_node, int floor) {
    if (floor < H){//до предлистового узла поднимаемся
        int i = 1;
        while ((i <= ref_node->max_ref_child_id) and (leaf > ref_node-
>key_copy[i])) i++;//проход по текущему узлу сравнивая ключи
        if(leaf == ref_node->key_copy[i]) return true;
        return search(leaf, ref_node->child_array[i-1], floor+1); //переход к
следующему узлу рекурсией
    }
    else {//мы в листе
        if(leaf == ref_node->key_data) return true;
    }
    return false;
};

///готово
void* BPTree::add(int leaf) {
    //cout << endl<<"_____ Добавление " << leaf <<" _____"<< endl;
    Hplus = false;
    ///0. отработка исключения
    if (root->max_ref_child_id == -1){      // если нет ни одного еще листа
        root->child_array[0] = new Node(leaf,true);
        root->max_ref_child_id++;
        H++;
        //cout <<"_____ Добавили первый лист номиналом " << leaf << " _____"
<< endl<< endl;
        return nullptr;
    }
/// 1. поиск последнего узла, в который добавим лист
    ptr_path_array = new Node*[H];
    ptr_path_array[0] = root;
    int floors_for_separation = search_place_for_add(leaf, root, 0,
0);//сколько ярусов нужно разделить пополам по пути добавления листа
    /// двигаемся вправо пока не достигнем места куда вставим наш лист
    int i_new_leaf = 0; //будущее место нового листа
    while ((i_new_leaf <= ptr_path_array[H-1]->max_ref_child_id)and(leaf >
ptr_path_array[H-1]->child_array[i_new_leaf]->key_data)){// убрать =
        i_new_leaf++;
```

```
    }
    /// проверяем, есть ли уже такой лист
    if (floors_for_separation == -1){
        //cout << "_____ Такой лист уже существует = " << leaf  << " _____"
<< endl;
        delete ptr_path_array;
        return nullptr;
    }
    if (i_new_leaf <= ptr_path_array[H-1]->max_ref_child_id){
        if(ptr_path_array[H-1]->child_array[i_new_leaf]->key_data == leaf){
            //cout << "_____ Такой лист уже существует = " << leaf << "
_____" << endl;
            delete ptr_path_array;
            return nullptr;
        }
    }
    /// 2. подготовить дерево к добавлению узла
    while(floors_for_separation > 0){                   //если предлистовой
узел переполнен и далее в сторону корня
        if (H == floors_for_separation){            //разделение корня, если
надо
            ///1. Создаем новый корень
            Hplus = true;
            newroot = new Node(b_factor, false);
            newroot->child_array[0] = root;
            newroot->max_ref_child_id++;
            ///2. разделяем значения
            node_separation(root,newroot, 0, leaf);
            root = newroot;
            newroot = nullptr;
            floors_for_separation--;
        }
        else{
            node_separation(ptr_path_array[H-floors_for_separation],
ptr_path_array[H-floors_for_separation-1], H-floors_for_separation, leaf);
            floors_for_separation--;
        }
    }
///3. подготовка предлистового узла и добавление листа
    Node *here = ptr_path_array[H-1];
    int i = here->max_ref_child_id + 1;
    while (i > i_new_leaf) {// тут смещение ссылок и копий на 1 вправо до
места добавления листа
        here->key_copy[i] = here->key_copy[i-1];
        here->child_array[i] = here->child_array[i-1];
        i--;
    }
    if (leaf < here->child_array[0]->key_data) {                        //если
новый лист меньше нулевого
        here->key_copy[1] = here->child_array[0]->key_data;
        here->child_array[1] = here->child_array[0];
        here->child_array[0] = new Node(leaf, true);
    }
    else {
        here -> child_array[i] = new Node(leaf, true);
        here -> key_copy[i] = leaf;
    }
    here -> max_ref_child_id++;
    if (Hplus) {H++; Hplus = false;}//если в процессе был создан новый
корень, то высота увеличилась
///4. Поправляем ссылки на соседеей
    Node* left_node = search_neighbour_left(leaf, root, 0);
    if (left_node-> key_data < leaf) {
        //cout << "Сосед слева "<< left_node->key_data << endl;
```

```
        left_node->neighbour = here->child_array[i]; //добавка предыдущему
листу ссылку на текущий
    }
    Node* right_node = search_neighbour_right(leaf, root, 0, nullptr, 0);
    if (right_node != nullptr){
        //cout << "Сосед справа "<< right_node->key_data << endl;
        here->child_array[i] -> neighbour = right_node;//добавка текущему
листу ссылку на следующий
    }
    //cout <<"_____ Добавление " << leaf <<  " успешно _____" << endl<< endl;
    delete ptr_path_array;
    return nullptr;
}
int BPTree::search_place_for_add(int leaf, Node* &ref_node, int floor, int
floors_for_separation){
    if (floor < H-1){ //до предлистового узла поднимаемся
        int i = 0;
        while ((i <= ref_node->max_ref_child_id) and (leaf > ref_node-
>key_copy[i])) i++;
        if(leaf == ref_node->key_copy[i]) return -1;//такой лист уже есть
        if (ref_node->max_ref_child_id >= b_factor)
floors_for_separation++;//нужно ли разделять узел
        else floors_for_separation = 0;
        floor++;
        ptr_path_array[floor] = ref_node->child_array[i-1];
        return search_place_for_add(leaf, ptr_path_array[floor], floor,
floors_for_separation);
    } //мы в листе:
    if (ref_node->max_ref_child_id >= b_factor) floors_for_separation++;
    else floors_for_separation = 0;
    return floors_for_separation;
}
void* BPTree::node_separation(Node* &ref_node,Node* &ref_parent_node, int
floor, int leaf){
        ///1. подвинуть у родителя узлы для добавления
    int i_place_node = ref_parent_node-> max_ref_child_id;
    while ((i_place_node > 0) and (ref_parent_node ->
child_array[i_place_node] != ref_node)){
        ref_parent_node->key_copy[i_place_node+1] = ref_parent_node-
>key_copy[i_place_node] ;
        ref_parent_node->child_array[i_place_node+1] = ref_parent_node-
>child_array[i_place_node];
        i_place_node--;
    } i_place_node++;
        ///2. создаем новый узел. он будет справа от переполненного.
    ref_parent_node->child_array[i_place_node] = new Node(b_factor, false);
    ref_parent_node->max_ref_child_id++;
       ///3. непосредственно разделяем узлы
    int half = 1+b_factor/2;//аккуратный перенос 0-1 позиций. а дальше в
цикле
    ref_parent_node->child_array[i_place_node]->child_array[0] = ref_node-
>child_array[half];
    ref_parent_node->child_array[i_place_node]->child_array[1] = ref_node-
>child_array[half+1];
    if(floor == (H-1)){     //если мы на предлистовом узле
        ref_parent_node->child_array[i_place_node]->key_copy[1] = ref_node-
>child_array[half+1]->key_data;
    }
    else{                   //иначе мы выше
        ref_parent_node->child_array[i_place_node]->key_copy[1] = ref_node-
>key_copy[half];
    }
    ref_node->child_array[half] = nullptr;
    ref_node->child_array[half+1] = nullptr;
```

```
    ref_node->key_copy[half] = MAX;
    ref_node->key_copy[half+1] = MAX;
    int k = 2;
    while (k < half){
        ref_parent_node->child_array[i_place_node]->child_array[k] =
ref_node->child_array[half+k];
        ref_parent_node->child_array[i_place_node]->key_copy[k] = ref_node-
>key_copy[half+k];
        ref_node->child_array[half+k] = nullptr;
        ref_node->key_copy[half+k] = MAX;
        k++;
    }
    int new_key_copy_parent =
search_for_the_first_leaf_of_this_branch(ref_parent_node-
>child_array[i_place_node]->child_array[0], floor+1);
    ref_parent_node->key_copy[i_place_node] = new_key_copy_parent;
    ref_node -> max_ref_child_id = b_factor/2;
    ref_parent_node->child_array[i_place_node] -> max_ref_child_id = ref_node
-> max_ref_child_id;
    if(leaf > new_key_copy_parent) ptr_path_array[floor] = ref_parent_node-
>child_array[i_place_node];// перезапись пути
    return nullptr;
}

///отточены))
int BPTree::search_for_the_first_leaf_of_this_branch(Node* ref_node, int
floor){//дохожу до листа, беру значение для разделения
    if(floor<H){
        floor++;
        return search_for_the_first_leaf_of_this_branch(ref_node-
>child_array[0],floor);
    }
    return ref_node->key_data;
}
Node* BPTree::search_neighbour_left(int leaf, Node* &ref_node, int floor){
    if (floor < H-1){//до листа
        int i = 0;
        while ((i <= ref_node->max_ref_child_id) and (leaf > ref_node-
>key_copy[i])) i++;
        return search_neighbour_left(leaf, ref_node->child_array[i-1],
floor+1);
    }
    int i = 0;
    while ((i <= ref_node->max_ref_child_id) and (leaf > ref_node-
>key_copy[i])) i++;
    return ref_node->child_array[i-1];
}
Node* BPTree::search_neighbour_right(int leaf, Node* &ref_node, int floor,
Node* minimally_larger_node, int mln_floor){
    if (floor < H-1){//до предлистового узла
        int i = 0;
        while ((i < ref_node->max_ref_child_id) and (leaf > ref_node-
>key_copy[i+1])) i++;
        floor++;
        if (i < ref_node->max_ref_child_id) {minimally_larger_node =
ref_node->child_array[i+1]; mln_floor = floor;}//если это был не максимальный
номер, записываем следующий узел ,чтобы потом если че на него перейти
        return search_neighbour_right(leaf, ref_node->child_array[i], floor,
minimally_larger_node, mln_floor);
    }
    else if (floor == H-1) {//мы на предлистовом узле
        int i = 0;
        while ((i < ref_node->max_ref_child_id) and (leaf >= ref_node-
>key_copy[i])) i++;
```

```
        floor++;
        return search_neighbour_right(leaf, ref_node->child_array[i], floor,
minimally_larger_node, mln_floor);
    }
    if (leaf == ref_node->key_data){//если текущий лист меньше нашего
        if (minimally_larger_node != nullptr){//но в соседних узлах есть еще
листы
            return search_leaf_neighbour(minimally_larger_node, mln_floor);
        }
        return nullptr;
    }
    return ref_node;
}
Node* BPTree::search_leaf_neighbour(Node* ref_node, int floor){//дохожу до
листа, беру значение для разделения
    if(floor<H){
        floor++;
        return search_leaf_neighbour(ref_node->child_array[0],floor);
    }
    return ref_node;
}

///подредактировал. пока ошибок не всплывало
void* BPTree::del(int leaf) {
    Hplus = false;
/// 1. поиск последнего узла, в котором удалим лист
    ptr_path_array = new Node*[H];
    ptr_path_array[0] = root;
    bool change_node = search_place_for_del(leaf, root, 0);//нужно ли
начинать перестройку узлов после удаления листа
    Node *here = ptr_path_array[H-1];//предлистовой узел
    int i = 0;
    while ((i < here->max_ref_child_id) and (leaf > here->child_array[i]-
>key_data)){
        i++;
    }
    if (here->child_array[i]->key_data != leaf){
        delete ptr_path_array;
        return nullptr;
    }
///2. поправка соседей
    Node* left_node = search_neighbour_left(leaf, root, 0);
    Node* right_node = search_neighbour_right(leaf, root, 0, nullptr, 0);

    if ((left_node != here->child_array[i])and(right_node != nullptr)){//если
они существуют оба
        left_node -> neighbour = right_node;
    }
///3. удаление и наведение порядка в текущем узле предлистовом
    here -> max_ref_child_id--;
    delete here->child_array[i];    //удаление листа поэтому соседи и были
раньше
    while (i <= here->max_ref_child_id) {   // тут смещение ссылок и копий на
1 влево с места удаления листа
        here->key_copy[i] = here->key_copy[i+1];
        here->child_array[i] = here->child_array[i+1];
        i++;
    }
    if (i<=b_factor){// на всякий случай. потом проверю надо ли
********************************************************** * * *
        here->key_copy[i] = MAX;
        here->child_array[i] = nullptr;
    }
```

```
///4. А теперь настройка дерева. Объединение узлов / одалживание у
соседей.....................рекурсия по дереву к корню
    if (change_node){
        tree_edits_after_deletion(ptr_path_array[H-1],  ptr_path_array[H-2],
H-1);
    }
    delete ptr_path_array;
    return nullptr;
}
void* BPTree::tree_edits_after_deletion(Node* &ref_node,Node*
&ref_parent_node, int floor){//перестройка этажей, бабочки летают..
    int i = 0;
    bool node_neighbour_left = false;
    ///1. Проходим по листу родителя пока не дойдем до узла просящего добавки
узлов.
    while (ref_parent_node -> child_array[i] != ref_node) i++;
    ///1.2. если он последний - обратимся к соседу слева, иначе к соседу
справа
    if (i== ref_parent_node ->max_ref_child_id) {i--; node_neighbour_left=
true;}else i++;
    ///2. проверка колва детей. если их больше половина минус 1, то просто
возьмем крайний узел и подвинем остальных - проверено
    if (ref_parent_node -> child_array[i] -> max_ref_child_id > b_factor/2-
1){
        if (node_neighbour_left){   //взаимствование у соседа слева
            ///двигаем узлы вправо для освобождения первого места добавления
            int n = ref_node -> max_ref_child_id+1;
            while (n > 1) {
                ref_node->key_copy[n] = ref_node->key_copy[n-1];
                ref_node->child_array[n] = ref_node->child_array[n-1];
                n--;
            }
            ref_node-
>key_copy[1]=search_for_the_first_leaf_of_this_branch(ref_node-
>child_array[0], floor+1);
            /// берем у соседа лист и очищаем у соседа информацию о нем
            ref_node->child_array[0] = ref_parent_node -> child_array[i]-
>child_array[ref_parent_node -> child_array[i] -> max_ref_child_id];
            ref_parent_node -> child_array[i]->child_array[ref_parent_node ->
child_array[i] -> max_ref_child_id] = nullptr;
            ref_parent_node -> child_array[i]->key_copy[ref_parent_node ->
child_array[i] -> max_ref_child_id] = MAX;
            ///поправляем родительскую копию
            ref_parent_node -> key_copy[i+1] =
search_for_the_first_leaf_of_this_branch(ref_node, floor);
        }
        else{//взаимствование у соседа справа
            ///собстна само взаимствование нулевого элемента
            ref_node->child_array[ref_node->max_ref_child_id+1] =
ref_parent_node -> child_array[i]->child_array[0];
            ref_node->key_copy[ref_node->max_ref_child_id+1] =
search_for_the_first_leaf_of_this_branch(ref_node->child_array[ref_node-
>max_ref_child_id+1], floor+1);
            ref_parent_node -> child_array[i]->child_array[0] =
ref_parent_node -> child_array[i]->child_array[1];
            int n = 1;
            while (n <= ref_parent_node -> child_array[i] ->
max_ref_child_id) {// тут смещение ссылок и копий на 1 влево у соседа
                ref_parent_node -> child_array[i]->key_copy[n] =
ref_parent_node -> child_array[i]->key_copy[n+1];
                ref_parent_node -> child_array[i]->child_array[n] =
ref_parent_node -> child_array[i]->child_array[n+1];
                n++;
            }
```

```
            ///поправляем родительскую копию
            ref_parent_node -> key_copy[i] =
search_for_the_first_leaf_of_this_branch(ref_parent_node->child_array[i],
floor);
        }
        ref_parent_node -> child_array[i]-> max_ref_child_id--;
        ref_node->max_ref_child_id++;
    }
    ///2.2 если их меньше или равно половина минус 1, то объединим узлы
    else{//хехе
        Node* left_node;
        Node* right_node;
        if (node_neighbour_left) {left_node = ref_parent_node ->
child_array[i]; right_node = ref_node; }
        else  {left_node = ref_node; right_node = ref_parent_node ->
child_array[i]; i--;}
        ///объединение
        int half = b_factor;
        left_node->child_array[half] = right_node ->child_array[0];
        left_node->key_copy[half] =
search_for_the_first_leaf_of_this_branch(left_node->child_array[half],
floor+1);
        int n = 1;
        while (n < half){
            left_node->child_array[n+half] = right_node ->child_array[n];
            left_node->key_copy[n+half] = right_node -> key_copy[n];
            right_node ->child_array[n] = nullptr;
            right_node -> key_copy[n] = 0;
        }
        /// и еще поправить ссылки у родителя и копии тоже
        while ( i < ref_parent_node -> max_ref_child_id ){
            ref_parent_node->child_array[i] = ref_parent_node -
>child_array[i+1];
            ref_parent_node->key_copy[i] = ref_parent_node -> key_copy[i+1];
            i++;
        }
        ref_parent_node->child_array[i] = nullptr;
        ref_parent_node->key_copy[i] = MAX;
        ref_parent_node -> max_ref_child_id--;
    }
    ///3. идем в следующий узел, если требуется
    floor--;
    if (floor==0){// если мы в корне можем оказаться
        if ((ref_parent_node -> max_ref_child_id==0)and(H>1)){ //если у него
только 1 ссылка и он не предлистовой узел
            root = ref_parent_node ->child_array[0];
            delete ref_parent_node;//сработает ?
        }
        else if ((ref_parent_node -> max_ref_child_id==0)and(H==1)){//если же
он оказался при этом предлистовым...
            delete ref_parent_node;//сработает ?
            root = nullptr;
        }
        return nullptr;
    }
    if (ref_parent_node -> max_ref_child_id == b_factor/2-2){
        return tree_edits_after_deletion(ptr_path_array[floor-1],
ptr_path_array[floor], floor);
    }
    return nullptr;
}
bool BPTree::search_place_for_del(int leaf, Node* &ref_node, int floor){
    if (floor < H-1){
        int i = 1;
```

```
        while ((i <= ref_node->max_ref_child_id) and (leaf > ref_node->key_copy[i])) i++;
        ptr_path_array[floor+1] = ref_node->child_array[i-1];
        return search_place_for_del(leaf, ref_node->child_array[i-1], floor+1);
    }
    bool combining = false;
    if (ref_node->max_ref_child_id == b_factor/2-1) combining = true;
    return combining;
}///требуется улучшение как у "соседа справа"
///готово
void BPTree::print(){
    cout << "B+ - tree:"<< endl;
    Node* once_leaf = search_once_leaf(root, 0);
    print_leaf(once_leaf);
}
Node* BPTree::search_once_leaf(Node* ref_node, int floor){//поиск первого листа
    if (floor < H){
        floor++;
        return search_once_leaf(ref_node->child_array[0], floor);
    }
    return ref_node;
}
Node* BPTree::print_leaf(Node* ref_leaf){//печать всех листов
    cout << ref_leaf->key_data << "  ";
    if (ref_leaf->neighbour == nullptr){
        cout << "Конец" << endl;
        return nullptr;
    }
    return print_leaf(ref_leaf->neighbour);
}
```