Санкт-Петербургский политехнический университет Петра великого

Институт машиностроения, материалов и транспорта

Высшая школа автоматизации и робототехники

Курсовой проект

по дисциплине «Объектно-ориентированное программирование»

«В+-дерево»

Выполнил

студент гр. 3331506/90401 Яковлев Д.Э.

Работу принял Ананьевский М.С.

Санкт-Петербург

2022

Оглавление

[**Введение** 3](#_Toc105140964)

[**Описание алгоритма** 3](#_Toc105140965)

[**Описание реализация алгоритма** 4](#_Toc105140966)

[**Поиск** 4](#_Toc105140967)

[**Добавление** 4](#_Toc105140968)

[**Удаление** 5](#_Toc105140969)

[**Исследование алгоритма** 6](#_Toc105140970)

[**Область применения** 9](#_Toc105140971)

[**Заключение** 10](#_Toc105140972)

[**Список литературы** 11](#_Toc105140973)

[**Приложение** 12](#_Toc105140974)

**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.

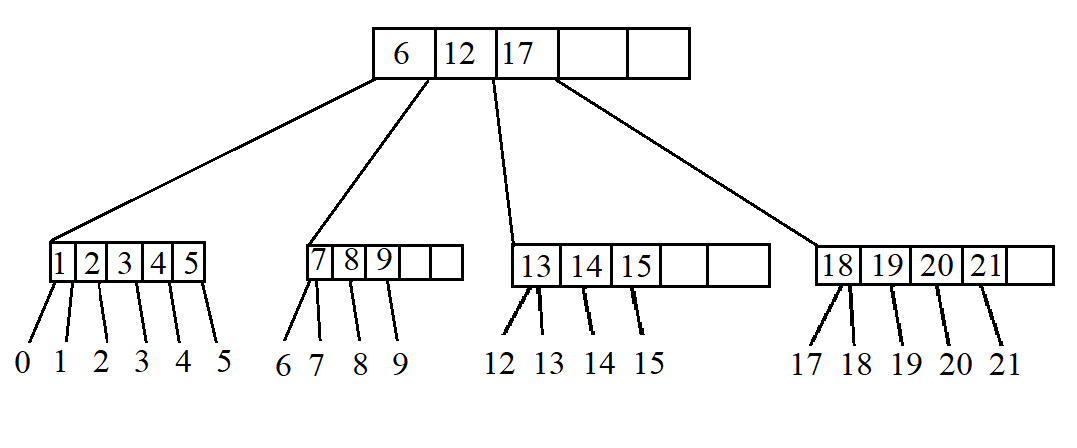


Рисунок 1 – Пример B+-дерева

**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**

В+-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 **2tt**, 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.

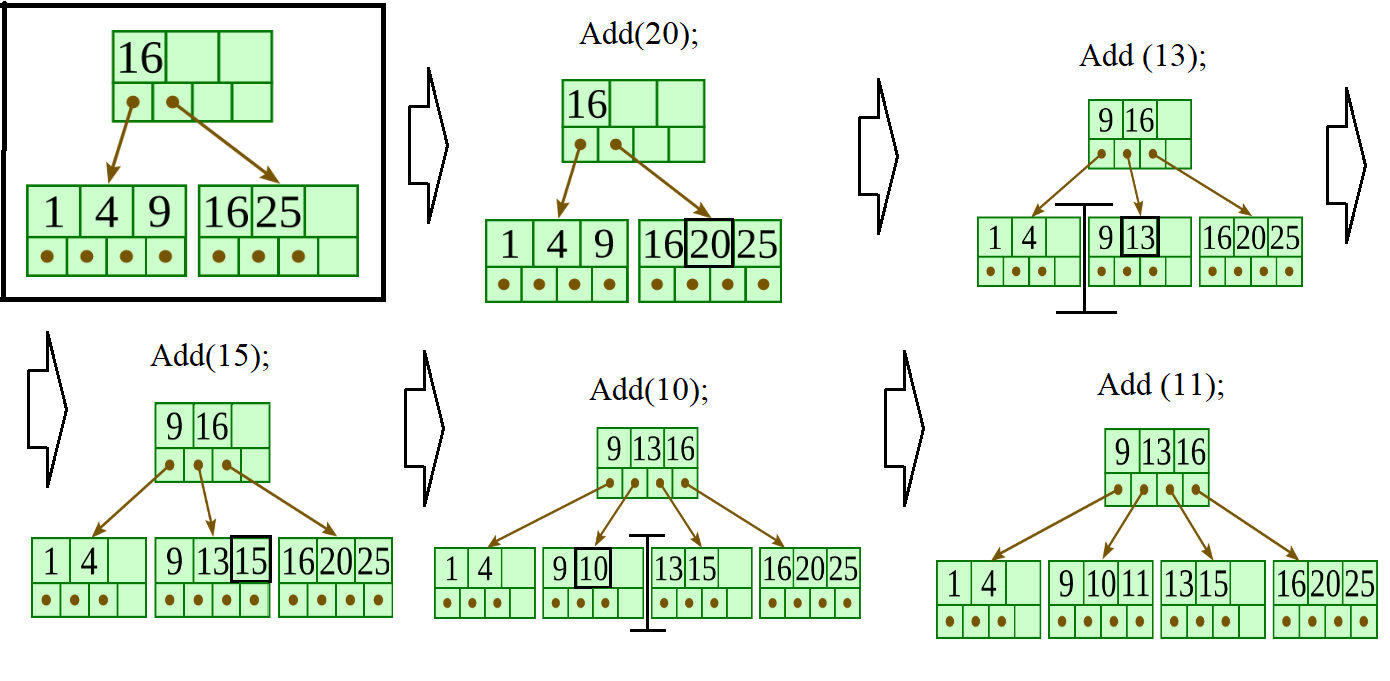


Рисунок 2 – Операции добавления узла

**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.

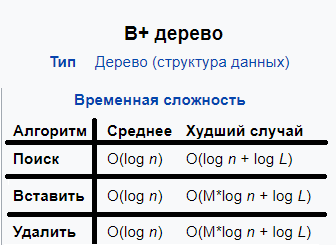


Рисунок 3 – Временная сложность, заявленная в Википедии

Проверим экспериментально время выполнения алгоритма при t=5.

1. 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».

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

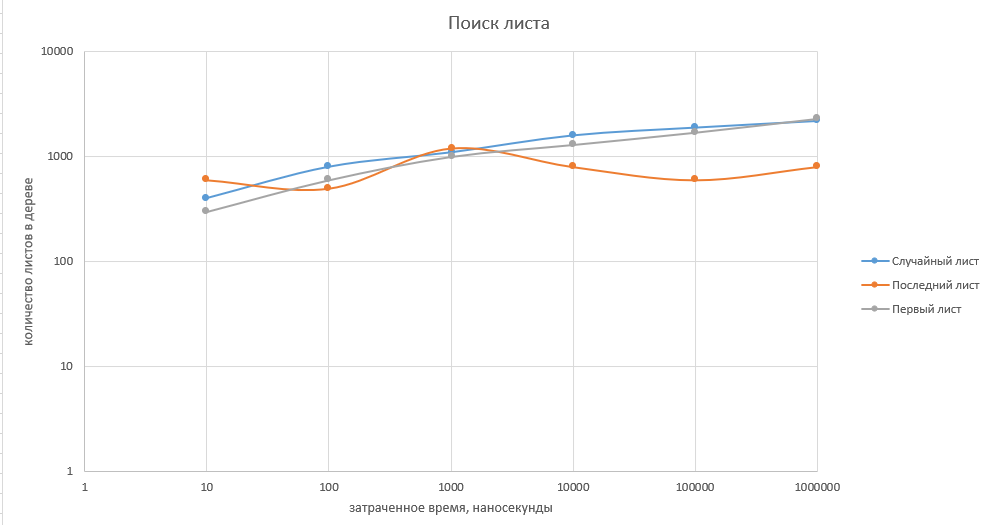


Рисунок 4 – Временная сложность операции поиска

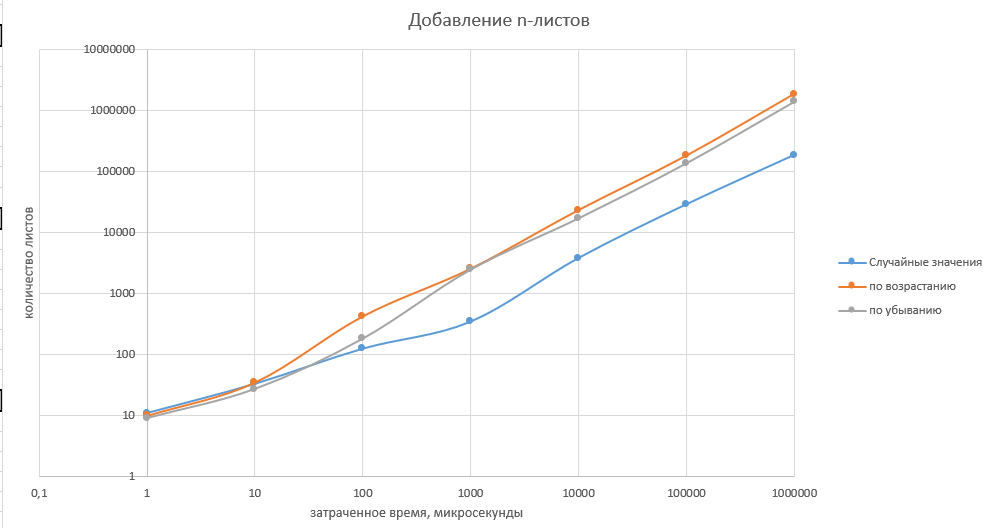
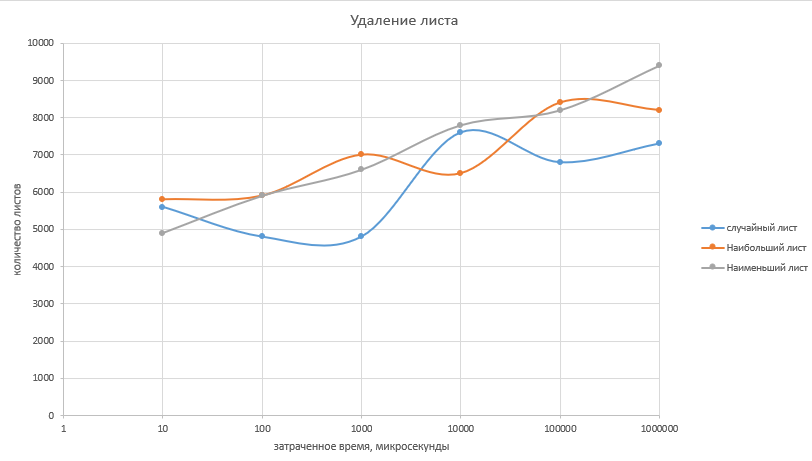


Рисунок 5 – Временная сложность операции добавления n – листов

 Рисунок 6 – Временная сложность операции удаления

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 В+-дерево, а также проведены исследования этого алгоритма. Реализация его оказалась довольна трудна вследствие предусмотра большого количества критичных сценариев, однако несмотря на громоздкий код алгоритм В+-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);  
}