**Міністерство освіти і науки України**

**Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"**

**Факультет інформатики та обчислювальної техніки**

**Кафедра інформатики та програмної інженерії**

**Звіт**

з лабораторної роботи № 2 з дисципліни

«Проектування алгоритмів»

„ **Проектування структур даних**”

**Виконав(ла)**

(шифр, прізвище, ім'я, по батькові)

*Спаських М. Д.*

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.Н.*

Київ 2021

Зміст

[1 Мета лабораторної роботи 3](file:///C:\Users\user\Documents\учебники%20и%20задания\2%20курс\теорія%20алгоритмів\лаби\lr2.ukr.docx#_Toc81070524)

[2 Завдання 4](file:///C:\Users\user\Documents\учебники%20и%20задания\2%20курс\теорія%20алгоритмів\лаби\lr2.ukr.docx#_Toc81070525)

[3 Виконання 7](file:///C:\Users\user\Documents\учебники%20и%20задания\2%20курс\теорія%20алгоритмів\лаби\lr2.ukr.docx#_Toc81070526)

[3.1 Псевдокод алгоритмів 7](file:///C:\Users\user\Documents\учебники%20и%20задания\2%20курс\теорія%20алгоритмів\лаби\lr2.ukr.docx#_Toc81070527)

[3.2 Часова складність пошуку 7](file:///C:\Users\user\Documents\учебники%20и%20задания\2%20курс\теорія%20алгоритмів\лаби\lr2.ukr.docx#_Toc81070528)

[3.3 Програмна реалізація 7](file:///C:\Users\user\Documents\учебники%20и%20задания\2%20курс\теорія%20алгоритмів\лаби\lr2.ukr.docx#_Toc81070529)

[3.3.1 Вихідний код 7](file:///C:\Users\user\Documents\учебники%20и%20задания\2%20курс\теорія%20алгоритмів\лаби\lr2.ukr.docx#_Toc81070530)

[3.3.2 Приклади роботи 7](file:///C:\Users\user\Documents\учебники%20и%20задания\2%20курс\теорія%20алгоритмів\лаби\lr2.ukr.docx#_Toc81070531)

[3.4 Тестування алгоритму 8](file:///C:\Users\user\Documents\учебники%20и%20задания\2%20курс\теорія%20алгоритмів\лаби\lr2.ukr.docx#_Toc81070532)

[3.4.1 Часові характеристики оцінювання 8](file:///C:\Users\user\Documents\учебники%20и%20задания\2%20курс\теорія%20алгоритмів\лаби\lr2.ukr.docx#_Toc81070533)

[Висновок 9](file:///C:\Users\user\Documents\учебники%20и%20задания\2%20курс\теорія%20алгоритмів\лаби\lr2.ukr.docx#_Toc81070534)

[Критерії оцінювання 10](file:///C:\Users\user\Documents\учебники%20и%20задания\2%20курс\теорія%20алгоритмів\лаби\lr2.ukr.docx#_Toc81070535)

# Мета лабораторної роботи

Мета роботи – вивчити основні підходи проектування та обробки складних структур даних.

# Завдання

Відповідно до варіанту (таблиця 2.1), записати алгоритми пошуку, додавання, видалення і редагування запису в структурі даних за допомогою псевдокоду (чи іншого способу по вибору).

Записати часову складність пошуку в структурі в асимптотичних оцінках.

Виконати програмну реалізацію невеликої СУБД, з функціями пошуку (алгоритм пошуку у вузлі структури згідно варіанту таблиця 2.1, за необхідності), додавання, видалення та редагування записів (запис складається із ключа і даних, ключі унікальні і цілочисельні, даних може бути декілька полів для одного ключа, але достатньо одного рядка фіксованої довжини). Для зберігання даних використовувати структуру даних згідно варіанту (таблиця 2.1).

Заповнити базу випадковими значеннями до 10000 і зафіксувати середнє (із 10-15 пошуків) число порівнянь для знаходження запису по ключу.

Зробити висновок з лабораторної роботи.

Таблиця 2.1 – Варіанти алгоритмів

|  |  |
| --- | --- |
| **№** | **Структура даних** |
| 1 | Файли з щільним індексом з перебудовою індексної області, бінарний пошук |
| 2 | Файли з щільним індексом з областю переповнення, бінарний пошук |
| 3 | Файли з не щільним індексом з перебудовою індексної області, бінарний пошук |
| 4 | Файли з не щільним індексом з областю переповнення, бінарний пошук |
| 5 | АВЛ-дерево |
| 6 | Червоно-чорне дерево |
| 7 | B-дерево t=10, бінарний пошук |
| 8 | B-дерево t=25, бінарний пошук |
| 9 | B-дерево t=50, бінарний пошук |
| 10 | B-дерево t=100, бінарний пошук |
| 11 | Файли з щільним індексом з перебудовою індексної області, однорідний бінарний пошук |
| 12 | Файли з щільним індексом з областю переповнення, однорідний бінарний пошук |
| 13 | Файли з не щільним індексом з перебудовою індексної області, однорідний бінарний пошук |
| 14 | Файли з не щільним індексом з областю переповнення, однорідний бінарний пошук |
| 15 | АВЛ-дерево |
| 16 | Червоно-чорне дерево |
| 17 | B-дерево t=10, бінарний пошук |
| 18 | B-дерево t=25, бінарний пошук |
| 19 | B-дерево t=50, бінарний пошук |
| 20 | B-дерево t=100, бінарний пошук |
| 21 | Файли з щільним індексом з перебудовою індексної області, метод Шарра |
| 22 | Файли з щільним індексом з областю переповнення, метод Шарра |
| 23 | Файли з не щільним індексом з перебудовою індексної області, метод Шарра |
| 24 | Файли з не щільним індексом з областю переповнення, метод Шарра |
| 25 | АВЛ-дерево |
| 26 | Червоно-чорне дерево |
| 27 | B-дерево t=10, метод Шарра |
| 28 | B-дерево t=25, метод Шарра |
| 29 | B-дерево t=50, метод Шарра |
| 30 | B-дерево t=100, метод Шарра |

# Виконання

**В мене 18 варіант**

B-дерево t=25, бінарний пошук

## Псевдокод алгоритмів

### Структура узла

**struct** Node

**bool** leaf // является ли узел листом

**int** n // количество ключей узла

**int** key[] // ключи узла

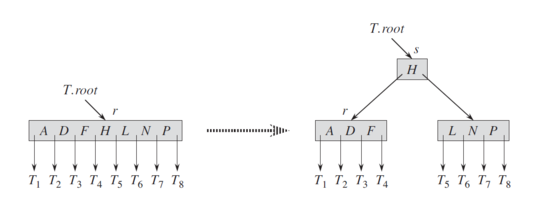
**Node** c[] // указатели на детей узла

### Структура дерева

**struct** BTree

**int** t // минимальная степень дерева

**Node** root // указатель на корень дерева

Ищем лист, в который можно добавить ключ, совершая проход от корня к листьям. Если найденный узел незаполнен, добавляем в него ключ. Иначе разбиваем узел на два узла, в первый добавляем первые t−1t−1 ключей, во второй — последние t−1t−1 ключей. После добавляем ключ в один из этих узлов. Оставшийся средний элемент добавляется в родительский узел, где становится разделительной точкой для двух новых поддеревьев. [](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A4%D0%B0%D0%B9%D0%BB:B3inssp.png)

Если и родительский узел заполнен — повторяем пока не встретим незаполненный узел или не дойдем до корня. В последнем случае корень разбивается на два узла и высота дерева увеличивается. Добавление ключа в B-дереве может быть осуществлена за один нисходящий проход от корня к листу. Для этого не нужно выяснять, требуется ли разбить узел, в который должен вставляться новый ключ. При проходе от корня к листьям в поисках места для нового ключа будут разбиваться все заполненные узлы, которые будут пройдены (включая и сам лист). Таким образом, если надо разбить какой-то полный узел, гарантируется, что его родительский узел не будет заполнен.

Вставка ключа в B-дерево TT высоты hh за один нисходящий проход по дереву потребует O(h)O(h) обращений к диску и O(th)=O(tlogtn)O(th)=O(tlogt⁡n) процессорного времени.

**void** B-Tree-Insert(T: **BTree**, k: **int**):

r = T.root

**if** r.n == 2T.t - 1

s = Allocate-Node()

T.root = s

s.leaf = *false*

s.n = 0

s.c[1] = r

B-Tree-Split-Child(s, T.t, 1)

B-Tree-Insert-Nonfull(s, k, T.t)

**else**

B-Tree-Insert-Nonfull(r, k, T.t)

**void** B-Tree-Insert-Nonfull(x: **Node**, k: **int**, t: **int**):

i = x.n

**if** x.leaf

**while** i >= 1 **and** k < x.key[i]

x.key[i+1] = x.key[i]

i = i - 1

x.key[i+1] = k

x.n = x.n + 1

Disk-Write(x)

**else**

**while** i >= 1 **and** k < x.key[i]

i = i - 1

i = i + 1

Disk-Read(x.c[i])

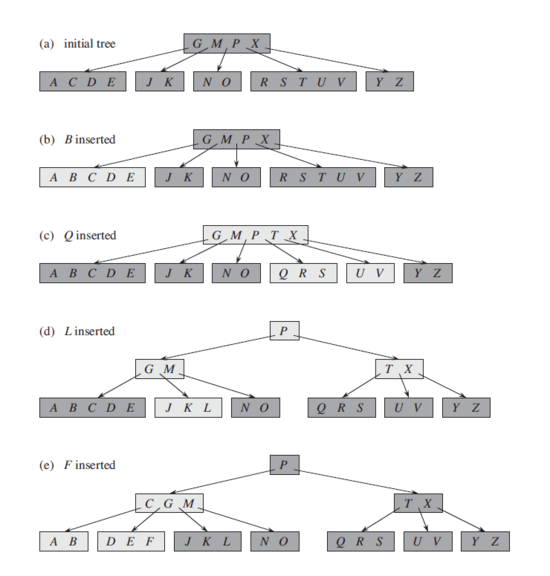
**if** x.c[i].n == 2t - 1

B-Tree-Split-Child(x, t, i)

**if** k > x.key[i]

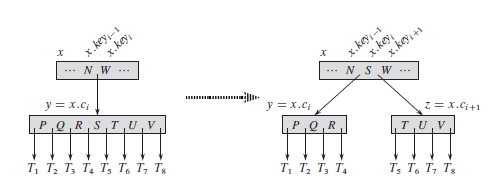
i = i + 1

B-Tree-Insert-Nonfull(x.c[i], k, t)

Функция B-Tree-Insert-NonfullB-Tree-Insert-Nonfull вставляет ключ kk в узел xx, который должен быть незаполненным при вызове. Использование функции B-Tree-Split-ChildB-Tree-Split-Child гарантирует, что рекурсия не встретится с заполненным узлом. Ниже показана вставка ключей BB, QQ, LL и FF в дерево с t=3t=3, т.е. узлы могут содержать не более 55 ключей [](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A4%D0%B0%D0%B9%D0%BB:B3insa.png)

### Разбиение узла

Функция B-Tree-Split-ChildB-Tree-Split-Child получает в качестве входного параметра незаполненный внутренний узел xx (находящийся в оперативной памяти), индекс tt и узел yy (также находящийся в оперативной памяти), такой что y=ci[x]y=ci[x] является заполненным дочерним узлом xx. Процедура разбивает дочерний узел на два и соответствующим образом обновляет поля xx, внося в него информацию о новом дочернем узле. Для разбиения заполненного корневого узла мы сначала делаем корень дочерним узлом нового пустого корневого узла, после чего можно вызвать функцию. При этом высота дерева увеличивается на 11. Отметим, что увеличить высоту B-дерева можно только разбиением.

[](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A4%D0%B0%D0%B9%D0%BB:B3splt.PNG)

**void** B-Tree-Split-Child(x: **Node**, t: **int**, i: **int**):

z = Allocate-Node()

y = x.c[i]

z.leaf = y.leaf

z.n = t - 1

**for** j = 1 **to** t - 1

z.key[j] = y.key[j+t]

**if** not y.leaf

**for** j = 1 **to** t

z.c[j] = y.c[j+t]

y.n = t - 1

**for** j = x.n + 1 **to** i + 1

x.c[j+1] = x.c[j]

x.c[i+1] = z

**for** j = x.n **to** i

x.key[j+1] = x.key[j]

x.key[i] = y.key[t]

x.n = x.n + 1

Disk-Write(y)

Disk-Write(z)

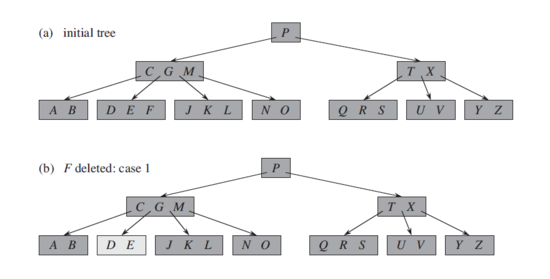
Disk-Write(x)

### Удаление ключа

Операция удаления ключа несколько сложнее, нежели добавление оного, так как необходимо убедиться, что удаляемый ключ находится во внутреннем узле. Процесс похож на поиск подходящего места для вставки ключа, с той разницей, что перед спуском в поддерево проверяется, достаточность количества ключей (т.е. ⩾t⩾t) в нем, а также возможность провести удаление, не нарушив структуры B-дерева. Таким образом, удаление аналогично вставке, и его проведение не потребует последующего восстановления структуры B-дерева. Если поддерево, выбранное поиском для спуска, содержит минимальное количество ключей t−1t−1, производится либо перемещение, либо слияние. Удаление из листа и из внутреннего узла рассмотрено, а также операции слияния поддеревьев и перемещения ключей при удалении ключа рассмотрены ниже. Для удаления требуется время O(tlogtn)O(tlogt⁡n) и O(h)O(h) дисковых операций.

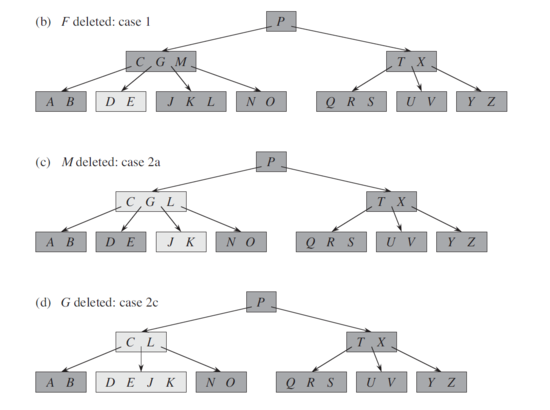
#### Удаление ключа из листа

Если удаление происходит из листа, смотрим на количество ключей в нем. Если ключей больше t−1t−1, то просто удаляем ключ.

[](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A4%D0%B0%D0%B9%D0%BB:B3dell.PNG) В противном случае, если существует соседний лист с тем же родителем, который содержит больше t−1t−1 ключа, выберем ключ-разделитель из соседа разделяющий оставшиеся ключи соседа и ключи исходного узла (то есть не больше всех из одной группы и не меньше всех из другой). Обозначим этот ключ как k1k1. Выберем другой ключ из родительского узла, разделяющий исходный узел и его соседа, который был выбран ранее. Этот ключ обозначим k2k2. Удалим из исходного узла ключ, который нужно было удалить, спустим в этот узел k2k2, а вместо k2k2 в родительском узле поставим k1k1. Если все соседи содержат по t−1t−1 ключу, то [объединяем](https://neerc.ifmo.ru/wiki/index.php?title=B-%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE" \l ".D0.A1.D0.BB.D0.B8.D1.8F.D0.BD.D0.B8.D0.B5" \o "B-дерево) узел с каким-либо из соседей, удаляем ключ, и ключ из родительского узла, который был разделителем разделённых соседей, [переместим](https://neerc.ifmo.ru/wiki/index.php?title=B-%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE" \l ".D0.9F.D0.B5.D1.80.D0.B5.D0.BC.D0.B5.D1.89.D0.B5.D0.BD.D0.B8.D0.B5_.D0.BA.D0.BB.D1.8E.D1.87.D0.B0" \o "B-дерево) в новый узел.

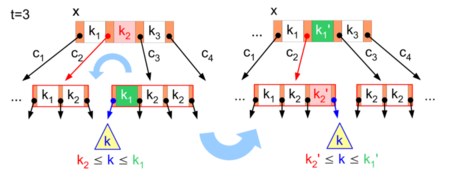
#### Удаление ключа из внутреннего узла

Рассмотрим удаление из внутреннего узла. Имеется внутренний узел xx и ключ, который нужно удалить, kk. Если дочерний узел, предшествующий ключу kk, содержит больше t−1t−1 ключа, то находим k1k1 – предшественника kk в поддереве этого узла. Удаляем его. Заменяем kk в исходном узле на k1k1. Проделываем аналогичную работу, если дочерний узел, следующий за ключом kk, имеет больше t−1t−1 ключа. Если оба (следующий и предшествующий дочерние узлы) имеют по t−1t−1 ключу, то [объединяем](https://neerc.ifmo.ru/wiki/index.php?title=B-%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE" \l ".D0.A1.D0.BB.D0.B8.D1.8F.D0.BD.D0.B8.D0.B5" \o "B-дерево) этих детей, [переносим](https://neerc.ifmo.ru/wiki/index.php?title=B-%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE" \l ".D0.9F.D0.B5.D1.80.D0.B5.D0.BC.D0.B5.D1.89.D0.B5.D0.BD.D0.B8.D0.B5_.D0.BA.D0.BB.D1.8E.D1.87.D0.B0" \o "B-дерево) в них kk, а далее удаляем kk из нового узла. Если [сливаются](https://neerc.ifmo.ru/wiki/index.php?title=B-%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE" \l ".D0.A1.D0.BB.D0.B8.D1.8F.D0.BD.D0.B8.D0.B5" \o "B-дерево) 22 последних потомка корня – то они становятся корнем, а предыдущий корень освобождается.

[](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A4%D0%B0%D0%B9%D0%BB:B3delin.png)

#### Перемещение ключа

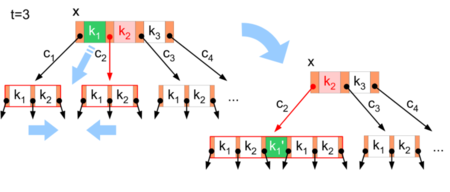
Если выбранное для нисходящего прохода поддерево содержит минимальное количеcтво ключей t−1t−1, и предшествующие и следующие узлы-братья имеют по меньшей мере tt ключей, то ключ перемещается в выбранный узел. Поиск выбрал для спуска x.c2x.c2 (x.k1<kdelete<x.k2x.k1<kdelete<x.k2). Этот узел имеет лишь t−1t−1 ключ (красная стрелка). Так как следующий брат x.c3x.c3 содержит достаточное количество ключей, самый маленький ключ x.c3.k1x.c3.k1 может перемещаться оттуда в родительский узел, чтобы переместить, в свою очередь, ключ x.k2x.k2 как дополнительный ключ в выбранный для спуска узел. Левое поддерево x.c3.k1x.c3.k1 — новое правое поддерево перемещённого ключа x.k2x.k2.

[](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A4%D0%B0%D0%B9%D0%BB:BTMv.png) Легко убедиться в том, что эти повороты поддерживают структуру B-дерева: для всех ключей kk на отложенном поддереве до и после перенесения выполняется условие x.k2⩽k⩽x.c3.k1x.k2⩽k⩽x.c3.k1. Симметричная операция может производиться для перенесения ключа из предшествующего брата.

#### Слияние

Ниже будет рассмотрено слияние узлов при удалении ключей, то есть слияние узлов равной степени и высоты. Для произвольных же слияний потребуется приведение сливаемых деревьев к одной степени и высоте.

Итак, если выбранное для спуска поддерево x.c2x.c2 и предшествующий и следующий узел-брат содержит минимальное количество ключей, то перемещение не возможно. На иллюстрации приводится слияние выбранного поддерева с предшествующим или следующим братом для такого случая. Для этого откладывается ключ из родительского узла xx, который разделяет ключи на два сливаемых узла, в то время средний ключ перемещается в слитый узел. Ссылки на слитые дочерние узлы заменяются ссылкой на новый узел.

[](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A4%D0%B0%D0%B9%D0%BB:BTMg.png) Так как алгоритм гарантирует, что узел, в который будет совершаться спуск, содержит по меньшей мере tt ключей вместо требуемых условиями B-дерева t−1t−1 ключей, родительский узел xx содержит достаточное количество ключей, чтобы выделить ключ для слияния. Это условие может быть нарушено, только в том случае, если два ребенка корня сливаются, так как поиск начинается с этого узла. По условиям B-дерева у корня должен быть как минимум один ключ, если дерево не пусто. При слиянии двух последних детей корня последний ключ перемещается во вновь возникшего единственного ребёнка, что приводит к пустому корневому узлу в не пустом дереве. В этом случае пустой узел корня удаляется и заменяется на единственного ребенка.

## Часова складність пошуку

…

Часова складність операцій пошуку, видалення, додавання, редагування

O(logn)

## Програмна реалізація

### Вихідний код

using System;

using System.IO;

namespace la2

{

class BTreeNode

{

public struct kv

{

public int key;

public string value;

}

public kv[] keys; // Ключи узла

public int MinDeg; // Минимальная степень узла B-дерева

public BTreeNode[] children; // дочерний узел

public int num; // Количество ключей узла

public bool isLeaf; // Истина, если это листовой узел

public static int chislo\_proidenih\_vuzliv;

// Конструктор

public BTreeNode(int deg, bool isLeaf)

{

this.MinDeg = deg;

this.isLeaf = isLeaf;

this.keys = new kv[2 \* this.MinDeg - 1]; // Узел имеет не более 2 \* MinDeg-1 ключей

this.children = new BTreeNode[2 \* this.MinDeg];

this.num = 0;

}

//Находим индекс первой позиции, равный или больший, чем ключ

public int findKey(int key)

{

int idx = 0;

// Условия выхода из цикла: 1.idx == num, то есть сканировать все

// 2.idx <num, т.е. найти ключ или больше ключа

while (idx < num && keys[idx].key < key)

++idx;

return idx;

}

public void remove(int key)

{

int idx = findKey(key);

if (idx < num && keys[idx].key == key)

{ // Найди ключ

if (isLeaf) // ключ находится в листовом узле

removeFromLeaf(idx);

else // ключ отсутствует в листовом узле

removeFromNonLeaf(idx);

}

else

{

if (isLeaf)

{ // Если узел является листовым узлом, то этот узел не входит в B-дерево

Console.WriteLine($"The key {key} is does not exist in the tree");

return;

}

// В противном случае удаляемый ключ существует в поддереве с корнем в этом узле

// Этот флаг указывает, существует ли ключ в поддереве с корнем в последнем дочернем узле узла

// Когда idx равно num, сравнивается весь узел, и флаг равен true

bool flag = idx == num;

if (children[idx].num < MinDeg) // Когда дочерний узел узла не заполнен, сначала заполняем его

fill(idx);

// Если последний дочерний узел был объединен, то он должен был быть объединен с предыдущим дочерним узлом, поэтому мы рекурсивно переходим к (idx-1) -ому дочернему узлу.

// В противном случае мы рекурсивно переходим к (idx) -ому дочернему узлу, который теперь имеет как минимум ключи наименьшей степени

if (flag && idx > num)

children[idx - 1].remove(key);

else

children[idx].remove(key);

}

}

public void removeFromLeaf(int idx)

{

// возвращаемся из idx

for (int i = idx + 1; i < num; ++i)

keys[i - 1] = keys[i];

num--;

}

public void removeFromNonLeaf(int idx)

{

int key = keys[idx].key;

// Если поддерево перед ключом (children [idx]) имеет не менее t ключей

// Затем находим предшественника key'pred 'в поддереве с корнем в children [idx]

// Заменить ключ на'pred ', рекурсивно удалить пред в дочерних [idx]

if (children[idx].num >= MinDeg)

{

kv pred = getPred(idx);

keys[idx].key = pred.key;

keys[idx].value = pred.value;

children[idx].remove(pred.key);

}

// Если у детей [idx] меньше ключей, чем у MinDeg, проверяем дочерние элементы [idx + 1]

// Если дочерние элементы [idx + 1] имеют хотя бы ключи MinDeg, в поддереве с корнем дочерние элементы [idx + 1]

// Находим преемника ключа 'ucc' для рекурсивного удаления succ в дочерних элементах [idx + 1]

else if (children[idx + 1].num >= MinDeg)

{

kv succ = getSucc(idx);

keys[idx].key = succ.key;

keys[idx].value = succ.value;

children[idx + 1].remove(succ.key);

}

else

{

// Если ключи children [idx] и children [idx + 1] меньше MinDeg

// затем объединяем ключ и дочерние элементы [idx + 1] в дочерние элементы [idx]

// Теперь children [idx] содержит ключ 2t-1

// Освобождаем дочерние элементы [idx + 1], рекурсивно удаляем ключ в children [idx]

merge(idx);

children[idx].remove(key);

}

}

public kv getPred(int idx)

{ // Узел-предшественник должен найти крайний правый узел из левого поддерева

// Продолжаем двигаться к крайнему правому узлу, пока не достигнем листового узла

BTreeNode cur = children[idx];

kv rez;

while (!cur.isLeaf)

cur = cur.children[cur.num];

rez.key = cur.keys[cur.num - 1].key;

rez.value = cur.keys[cur.num - 1].value;

return rez;

}

public kv getSucc(int idx)

{ // Узел-преемник находится от правого поддерева к левому

// Продолжаем перемещать крайний левый узел от дочерних [idx + 1], пока не достигнем конечного узла

BTreeNode cur = children[idx + 1];

kv rez;

while (!cur.isLeaf)

cur = cur.children[0];

rez.key = cur.keys[0].key;

rez.value = cur.keys[0].value;

return rez;

}

// Заполняем дочерние элементы [idx], у которых меньше ключей MinDeg

public void fill(int idx)

{

// Если предыдущий дочерний узел имеет несколько ключей MinDeg-1, заимствовать из них

if (idx != 0 && children[idx - 1].num >= MinDeg)

borrowFromPrev(idx);

// Последний дочерний узел имеет несколько ключей MinDeg-1, заимствовать от них

else if (idx != num && children[idx + 1].num >= MinDeg)

borrowFromNext(idx);

else

{

// объединить потомков [idx] и его брата

// Если children [idx] - последний дочерний узел

// затем объединить его с предыдущим дочерним узлом, иначе объединить его со следующим братом

if (idx != num)

merge(idx);

else

merge(idx - 1);

}

}

// Заимствуем ключ у потомков [idx-1] и вставляем его в потомки [idx]

public void borrowFromPrev(int idx)

{

BTreeNode child = children[idx];

BTreeNode sibling = children[idx - 1];

// Последний ключ из дочерних [idx-1] переходит к родительскому узлу

// ключ [idx-1] из недополнения родительского узла вставляется как первый ключ в дочерних [idx]

// Следовательно, sibling уменьшается на единицу, а children увеличивается на единицу

for (int i = child.num - 1; i >= 0; --i) // дети [idx] продвигаются вперед

child.keys[i + 1] = child.keys[i];

if (!child.isLeaf)

{ // Если дочерний узел [idx] не является листовым, переместите его дочерний узел назад

for (int i = child.num; i >= 0; --i)

child.children[i + 1] = child.children[i];

}

// Устанавливаем первый ключ дочернего узла на ключи текущего узла [idx-1]

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

if (!child.isLeaf) // Устанавливаем последний дочерний узел в качестве первого дочернего узла дочерних элементов [idx]

child.children[0] = sibling.children[sibling.num];

// Перемещаем последний ключ брата к последнему из текущего узла

keys[idx - 1] = sibling.keys[sibling.num - 1];

child.num += 1;

sibling.num -= 1;

}

// Симметричный с заимствованиемFromPrev

public void borrowFromNext(int idx)

{

BTreeNode child = children[idx];

BTreeNode sibling = children[idx + 1];

child.keys[child.num] = keys[idx];

if (!child.isLeaf)

child.children[child.num + 1] = sibling.children[0];

keys[idx] = sibling.keys[0];

for (int i = 1; i < sibling.num; ++i)

sibling.keys[i - 1] = sibling.keys[i];

if (!sibling.isLeaf)

{

for (int i = 1; i <= sibling.num; ++i)

sibling.children[i - 1] = sibling.children[i];

}

child.num += 1;

sibling.num -= 1;

}

// объединить childre [idx + 1] в childre [idx]

public void merge(int idx)

{

BTreeNode child = children[idx];

BTreeNode sibling = children[idx + 1];

// Вставляем последний ключ текущего узла в позицию MinDeg-1 дочернего узла

child.keys[MinDeg - 1] = keys[idx];

// ключи: children [idx + 1] скопированы в children [idx]

for (int i = 0; i < sibling.num; ++i)

child.keys[i + MinDeg] = sibling.keys[i];

// children: children [idx + 1] скопированы в children [idx]

if (!child.isLeaf)

{

for (int i = 0; i <= sibling.num; ++i)

child.children[i + MinDeg] = sibling.children[i];

}

// Перемещаем клавиши вперед, а не зазор, вызванный перемещением ключей [idx] к дочерним [idx]

for (int i = idx + 1; i < num; ++i)

keys[i - 1] = keys[i];

// Перемещаем соответствующий дочерний узел вперед

for (int i = idx + 2; i <= num; ++i)

children[i - 1] = children[i];

child.num += sibling.num + 1;

num--;

}

public void insertNotFull(int key, string value)

{

int i = num - 1; // Инициализируем i индексом самого правого значения

if (isLeaf)

{ // Когда это листовой узел

// Находим, куда нужно вставить новый ключ

while (i >= 0 && keys[i].key > key)

{

keys[i + 1] = keys[i]; // клавиши возвращаются

i--;

}

keys[i + 1].key = key;

keys[i + 1].value = value;

num = num + 1;

}

else

{

// Находим позицию дочернего узла, который нужно вставить

while (i >= 0 && keys[i].key > key)

i--;

if (children[i + 1].num == 2 \* MinDeg - 1)

{ // Когда дочерний узел заполнен

splitChild(i + 1, children[i + 1]);

// После разделения ключ в середине дочернего узла перемещается вверх, а дочерний узел разделяется на два

if (keys[i + 1].key < key)

i++;

}

children[i + 1].insertNotFull(key, value);

}

}

public void splitChild(int i, BTreeNode y)

{

// Сначала создаем узел, содержащий ключи MinDeg-1 y

BTreeNode z = new BTreeNode(y.MinDeg, y.isLeaf);

z.num = MinDeg - 1;

// Передаем все атрибуты y в z

for (int j = 0; j < MinDeg - 1; j++)

z.keys[j] = y.keys[j + MinDeg];

if (!y.isLeaf)

{

for (int j = 0; j < MinDeg; j++)

z.children[j] = y.children[j + MinDeg];

}

y.num = MinDeg - 1;

// Вставляем новый дочерний узел в дочерний узел

for (int j = num; j >= i + 1; j--)

children[j + 1] = children[j];

children[i + 1] = z;

// Перемещаем ключ по y к этому узлу

for (int j = num - 1; j >= i; j--)

keys[j + 1] = keys[j];

keys[i] = y.keys[MinDeg - 1];

num = num + 1;

}

public void traverse()

{

int i;

for (i = 0; i < num; i++)

{

if (!isLeaf)

children[i].traverse();

Console.Write($" {keys[i].key}-{keys[i].value}");

}

if (!isLeaf)

{

children[i].traverse();

}

}

public string tree\_to\_string()

{

string s="";

int i;

for (i = 0; i < num; i++)

{

if (!isLeaf)

s+=children[i].tree\_to\_string();

s+= keys[i].key + "@" + keys[i].value + "@";

}

if (!isLeaf)

{

s+=children[i].tree\_to\_string();

}

return s;

}

public BTreeNode search(int key)

{

int i = 0;

while (i < num && key > keys[i].key)

i++;

if (keys[i].key == key)

return this;

if (isLeaf)

return null;

return children[i].search(key);

}

public string poshuk(int key)

{

chislo\_proidenih\_vuzliv++;

int left = -1;

int right = num;

while (left < right - 1)

{

int mid = (left + right) / 2;

if (keys[mid].key < key)

{

left = mid;

}

else

{

right = mid;

}

}

if (right < keys.Length)

if (keys[right].key == key) return keys[right].value;

if (isLeaf) return key + " не знайдено";

else return children[right].poshuk(key);

}

}

class BTree

{

public BTreeNode root;

int MinDeg;

// Конструктор

public BTree(int deg)

{

this.root = null;

this.MinDeg = deg;

}

public void traverse()

{

if (root != null)

{

root.traverse();

}

}

// Функция для поиска ключа

public BTreeNode search(int key)

{

BTreeNode.chislo\_proidenih\_vuzliv = 0;

return root == null ? null : root.search(key);

}

public void insert(int key, string value)

{

if (root == null)

{

root = new BTreeNode(MinDeg, true);

root.keys[0].key = key;

root.keys[0].value = value;

root.num = 1;

}

else

{

// Когда корневой узел заполнится, дерево станет выше

if (root.num == 2 \* MinDeg - 1)

{

BTreeNode s = new BTreeNode(MinDeg, false);

// Старый корневой узел становится дочерним узлом нового корневого узла

s.children[0] = root;

// Отделяем старый корневой узел и даем ключ новому узлу

s.splitChild(0, root);

// Новый корневой узел имеет 2 дочерних узла, переместите туда старый корневой узел

int i = 0;

if (s.keys[0].key < key)

i++;

s.children[i].insertNotFull(key,value);

root = s;

}

else

root.insertNotFull(key,value);

}

}

public void remove(int key)

{

if (root == null)

{

Console.WriteLine("The tree is empty");

return;

}

root.remove(key);

if (root.num == 0)

{ // Если у корневого узла 0 ключей

// Если у него есть дочерний узел, используйте его первый дочерний узел как новый корневой узел,

// В противном случае установите корневой узел в ноль

if (root.isLeaf)

root = null;

else

root = root.children[0];

}

}

public string poshuk(int pkey)

{

BTreeNode.chislo\_proidenih\_vuzliv = 0;

if (root != null) return root.poshuk(pkey);

else return "не знайдено";

}

public void redakt(int pkey, string val)

{

if (search(pkey) != null)

{

remove(pkey);

insert(pkey, val);

Console.WriteLine($"Змiнено ключ {pkey}: value = {poshuk(pkey)}");

}

else Console.WriteLine($"key={pkey} not found! ");

}

public void save()

{

if (root != null)

{

string fname = "data.txt";

using (StreamWriter sw = new StreamWriter(fname, false, System.Text.Encoding.Default))

{

sw.WriteLine(root.tree\_to\_string());

}

}

}

public void load()

{

string fname = "data.txt";

string s = "";

using (StreamReader sr = new StreamReader(fname))

{

s=sr.ReadToEnd();

string[] dataS = s.Split("@");

root = null;

for (int i = 0; i < dataS.Length-1; i += 2) insert(Int32.Parse(dataS[i]), dataS[i + 1]);

}

}

}

class Program

{

static void Main(string[] args)

{

BTree t = new BTree(25); // A B-Tree with minium degree 2

var rand = new Random();

for (int i = 0; i < 10000; i++) t.insert(i, "v" + i);

for (int i = 0; i < 10; i++) {

int k = rand.Next(10000);

Console.WriteLine("Пошук ключа "+k + " Знайшов значенння: " + t.poshuk(k) + " Число пройдених вузлiв: " + BTreeNode.chislo\_proidenih\_vuzliv);

}

}

}

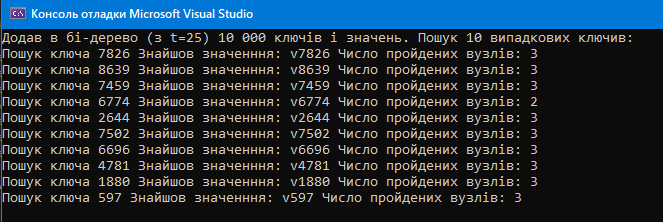
}

### Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми для додавання і пошуку запису.

Рисунок 3.1 –Додавання запису

Рисунок 3.2 – Пошук запису



## Тестування алгоритму

### Часові характеристики оцінювання

В таблиці 3.1 наведено кількість порівнянь для **10**-15 спроб пошуку запису по ключу.

Таблиця 3.1 – Число порівнянь при спробі пошуку запису по ключу

|  |  |
| --- | --- |
| Номер спроби пошуку | Число порівнянь |
| 1 |  |
| 2 |  |
| 3 |  |
| 4 |  |
| … |  |
|  |  |

Висновок

В рамках лабораторної роботи я написав програму для роботи з бі деревом. Для Т=25 з бінарним пошуком. Написав можливість зберігати дерево в файлі і його зчитування. Заповним дерево 10 000 ключами і значеннями. Здійснив 10 пошуків і зафіксував число пройдених вузлів. Середнє число пройдених вузлів виявилось 2.9. Вивожу дерево обходом по лівому краю. Складність пошуку O(logn).

Критерії оцінювання

За умови здачі лабораторної роботи до 15.10.2021 включно максимальний бал дорівнює – 5. Після 15.10.2021 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

* псевдокод алгоритму – 20%;
* аналіз часової складності – 5%;
* програмна реалізація алгоритму – 60%;
* тестування алгоритму – 10%;
* висновок – 5%.

+1 додатковий бал можна отримати за реалізацію графічного інтерфейсу.

+1 додатковий бал можна отримати за реалізацію графічного зображення структури ключів.