МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ «ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

Кафедра САПР

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Алгоритмы и структуры данных»
тема «Самобалансирующиеся двоичные деревья»

Студент гр. 2301 ______________________ Комиссаров П.Е.

Преподаватель: ______________________ Пестерев Д.О.

Санкт-Петербург

2023

1. Постановка задачи

1. Реализовать двоичное дерево поиска
2. Красно-черное дерево
3. АВЛ-дерево.
4. Сравнить высоты деревьев на случайном наборе входных данных, распределенных случайно.
5. Сравнить временные затраты на балансировку для красно-черного и АВЛ-дерева при удалении элементов, при вставке элементов.
6. Отдельно реализовать функции обхода по дереву: в ширину, в глубину: прямой (preorder), обратный (postorder), симметричный (inorder).

2. Описание алгоритмов

Бинарное дерево — это иерархическая структура данных, в которой каждый узел имеет значение (ключ) и ссылки на левого и правого потомка. Узел, находящийся на самом верхнем уровне (не являющийся чьим либо потомком) называется корнем. Узлы, не имеющие потомков называются листьями.

1. Двоичное дерево поиска

Бинарное дерево поиска — это бинарное дерево, обладающее дополнительными свойствами: значение левого потомка меньше значения родителя, а значение правого потомка больше значения родителя для каждого узла дерева. То есть, данные в бинарном дереве поиска хранятся в отсортированном виде.

Код сортировки:

- **Нода:** Состоит из ключа, левого, правого детей

```
struct Node
{
	int key;
	Node* l = nullptr, * r = nullptr;
	Node(int key) {
		this->key = key;
	}
} *root = nullptr;
```

- **Поиск по ключу:** если элемент есть, то выводится соответствующее сообщение

```
bool findKey(Node* n, int key) const {
	while ((n != nullptr) and (key != n->key)) {
		if (key > n->key)
			n = n->r;
		else if (key < n->key)
			n = n->l;
		if (n == nullptr)
			return false;
		if (key == n->key)

		return true;
	}
}
```

- **Вставка:** идем по дереву, находим нужную позицию для вставки

```
Node* insert(Node* n, int key) {

        if (!n) {
                return new Node(key);

        }
        if (n->key < key) n->r=insert(n->r, key);
        else if (n->key > key) n->l=insert(n->l, key);
        return n;
}
```

- **Обход в глубину:**
  - **Преордер:** выводим с корня и доходим до левого и тд

```
void preorderPrint(Node *n) const {
        if (n == nullptr)
                return;
        cout << n->key << endl;
        preorderPrint(n->l);
        preorderPrint(n->r);
}
```

  - **Постордер:** доходим до крайнего левого, потом правого и выводим

```
void postorderPrint(Node* n) const {
        if (n == nullptr)                       //если дошли до конца возврат
                return;

        postorderPrint(n->l);    //идем по левой до макс
        postorderPrint(n->r);    //идем по правой до макс
        cout << n->key << endl;  //выводим значение

}
```

  - **Симметричный:** выводим дерево по возрастанию ключей

```
void inorderPrint(Node* n) const {
        if (n == nullptr)
                return;
        inorderPrint(n->l);
        cout << n->key << endl;
        inorderPrint(n->r);
}
```

- **По ширине:** выводится по уровням

```
void widthPrint(Node* root) {
vector<Node*> top;                   //вектор вершин (ссылки)
vector<Node*> tops_;         //промежуточный вектор
if (!root) {
        return;
}
top.push_back(root);         //присваиваем первую вершину
while (!top.empty()) {       //пока есть вершины
        for (int i = 0; i < top.size(); i++) {
//проходим по всем вершинам уровня
                cout << top[i]->key << " ";      //выводим ключ вершин
                if (top[i]->l)
//формируем веришны след уровня
                        tops_.push_back(top[i]->l);
                if (top[i]->r)
                        tops_.push_back(top[i]->r);
        }
        top.clear();
        top = tops_;         //вектор вершин переходит на след уровень
        tops_.clear();
        cout << endl;
}
}
```

- **Удаление эл по ключу**:
  1. Если 1 ребенок, то удаляем эл., на его место ставим его ребенка
  2. Если нет детей, удаляем
  3. Если 2 ребенка, то находим самый левый эл в правом поддереве от удаляемого эл.

```
Node* removeMin(Node* n) {
        if (n->l == 0)
                return n->r;
        n->l = removeMin(n->l);
        //return n;
}
Node* remove(Node* n, int key) {
        if (!n) return 0;
        if (key < n->key) n->l = remove(n->l, key);
        else if (key > n->key)n->r = remove(n->r, key);
        else
        {
                Node* left = n->l;
                Node* right = n->r;
                delete n;
                if (!right) return left;
                Node* min = findMin(right);
                min->r = removeMin(right);
                min->l = left;
                return min;
        }
        return n;
}
```

- **Удаление дерева:**

```
void destroy(Node* n) {
        if (n==nullptr) return;
        destroy(n->l);
        destroy(n->r);
        delete n;
}
```

Плюсы и минусы реализации

Плюсы:

1. Простота реализации.
2. Операции вставки, удаления элемента и поиска выполняются за O(log n) в среднем случае.
3. Возможность эффективного обхода дерева в порядке возрастания/убывания.

Минусы:

1. Несбалансированное дерево может привести к деградации производительности до O(n).
2. Неспецифическая структура дерева может привести к неэффективности операций вставки, удаления и поиска в худшем случае.

Анализ реализации:

Кол-во **$n*10^4$**	**высота**
1	29
2	32
3	35
4	34
5	34
6	42
7	35
8	35
9	39
10	39

2. АВЛ - дерево

АВЛ-дерево — двоичное дерево поиска, ключи которого удовлетворяют стандартному свойству. Особенностью АВЛ-дерева является то, что оно является сбалансированным в следующем смысле: для любого узла дерева высота его правого поддерева отличается от высоты левого поддерева не более чем на единицу.

Теоретическая функция:

N-min(h) = F_{h+3} - 1
БИ: N-min(1) = F_4 - 1 = 3 - 1 = 2 верно
n=1
ШИ: N - min(h+1) = N - min(h) + N-min(h-1) +1=F_{h+3} - 1 + F_{h+2} - 1 + 1 = **F_{h+4} - 1**
N - min(h) = F_{h+3} -1 = $\phi^{h+3}/\sqrt{5}$ - 1
Fn ~ $\phi^n/\sqrt{5}$
h_{avl} <= $\log_\phi N$

Код сортировки:

- **Нода:** Состоит из ключа, высоты, левого, правого детей

```
struct Node
{
	 int height;
	int key;
	Node* l = nullptr, * r = nullptr;
	Node(int key) {
		this->key = key;
		this->height = 1;

	}
} *root = nullptr;
```

- **Поиск по ключу:** если элемент есть, то выводится соответствующее сообщение

```
bool findKey(Node* n, int key) const {
	while ((n != nullptr) and (key != n->key)) {
		if (key > n->key)
			n = n->r;
		else if (key < n->key)
			n = n->l;
		if (n == nullptr)
			return false;
		if (key == n->key)
		return true;
	}
}
```

- **Вставка:** идем по дереву, находим нужную позицию для вставки, делаем балансировку

```
Node* insert(Node* n, int key) { //вставка по ключу
        if (!n) {
                return new Node(key);

        }

        if (key < n->key)
                n->l=insert(n->l, key);
        else
                n->r=insert(n->r, key);
         return balance(n);
}
```

- **Обход в глубину:**
  - **Преордер:** выводим с корня и доходим до левого и тд

```
Node* insert(Node* n, int key) {

        if (!n) {
                return new Node(key);

        }
        if (n->key < key) n->r=insert(n->r, key);
        else if (n->key > key) n->l=insert(n->l, key);
        return n;
}
```

  - **Постордер:** доходим до крайнего левого, потом правого и выводим

```
void postorderPrint(Node* n) const {
        if (n == nullptr)
                return;

        postorderPrint(n->l);
        postorderPrint(n->r);
        cout << n->key << endl;

}
```

  - **Симметричный:** выводим дерево по возрастанию ключей

```
void inorderPrint(Node* n) const {
        if (n == nullptr)
                return;
        inorderPrint(n->l);
        cout << n->key << endl;
        inorderPrint(n->r);
}
```

- **По ширине:** выводится по уровням

```
void widthPrint(Node* root) {
vector<Node*> top;
vector<Node*> tops_;
if (!root) {
        return;
}
top.push_back(root);
while (!top.empty()) {
        for (int i = 0; i < top.size(); i++) {
                cout << top[i]->key << " ";
                        tops_.push_back(top[i]->l);
                if (top[i]->r)
                        tops_.push_back(top[i]->r);
        }
        top.clear();
        top = tops_;
        tops_.clear();
        cout << endl;
        }
}
```

- **Удаление эл по ключу**:
  1. Если 1 ребенок, то удаляем эл., на его место ставим его ребенка
  2. Если нет детей, удаляем
  3. Если 2 ребенка, то находим самый левый эл в правом поддереве от удаляемого эл.
  4. Рекурсивная балансировка

```
Node* remove(Node* n, int key) {
        if (!n) return 0;
        if (key < n->key) n->l = remove(n->l, key);
        else if (key > n->key)n->r = remove(n->r, key);
        else
        {
                Node* left = n->l;
                Node* right = n->r;
                delete n;
                if (!right) return left;
                Node* min = findMin(right);
                min->r = removeMin(right);
                min->l = left;
                return balance(min);
        }
        return balance(n);
}
```

- **Удаление дерева:**

```
void destroy(Node* n) {
        if (n==nullptr) return;
        destroy(n->l);
        destroy(n->r);
        delete n;
}
```

- ## Балансировка

```
Node* balance(Node* n) {
	updateHeight(n);
	int balance = bFactor(n);
	if (balance == -2) {
		if (bFactor(n->l) >0) leftRotate(n->l);
		return rightRotate(n);
	}
	else if (balance == 2) {
		if (bFactor(n->r) <0) rightRotate(n->r);
		return leftRotate(n);
	}
	return n;
}
```

- ## Поиск высоты дерева

```
int hight(Node* root) {
	vector<Node*> top;
	vector<Node*> tops_;
	if (!root) {
		return 0;
	}
	int hight = 1;
	top.push_back(root);
	while (!top.empty()) {
		for (int i = 0; i < top.size(); i++) {
			if (top[i]->l)
				tops_.push_back(top[i]->l);
			if (top[i]->r)
				tops_.push_back(top[i]->r);
		}
		hight++;
		top.clear();
		top = tops_;
		tops_.clear();
	}
	return hight;
}
```

- ## Вспомогательные функции

Обновление высоты родителя: берет макс высоту ребенка + 1

```
void updateHeight(Node* n) {
	n->height = max(getHeight(n->l), getHeight(n->r))+1;
}
```

Вычисление баланс фактора: из высоты правого ребенка вычитаем высоту левого

```
int bFactor(Node* n) {
	return (getHeight(n->r) - getHeight(n->l));
}
```

Вывод высоты элемента

```
int getHeight(Node* n) {
	if (!n) return 0;
	else return n->height;
}
```

Обмен ключами элементов (таким способом исключаем момент переназначения корня при поворотах)

```
void swap(Node* a, Node* b) {
        int temp = a->key;
        a->key = b->key;
        b->key = temp;
}
```

Левый и правый повороты:

```
Node* rightRotate(Node* n) {
        swap(n, n->l);
        Node* temp = n->r;
        n->r = n->l;
        n->l = n->r->l;
        n->r->l = n->r->r;
        n->r->r = temp;
        updateHeight(n->r);
        updateHeight(n);
        return n;

}
Node* leftRotate(Node* n) {
        swap(n, n->r);
        Node* temp = n->l;
        n->l = n->r;
        n->r = n->l->r;
        n->l->r = n->l->l;
        n->l->l = temp;
        updateHeight(n->l);
        updateHeight(n);
return n;
}
```

Описание балансировки АВЛ-дерева:

1. Когда балансировка дерева ломается то:
   1.1. Если ломается из-за вставки, то балансировку делаем один раз. Так как добавленную высоту от вставки балансируем, приводя дерево в исходное состояние
   1.2. Если ломается из-за удаления, то рекурсивно повторяем балансировку. Так как итоговая высота поддерева может уменьшиться на 1
2. Находим поломку в балансировке (с помощью функции bfactor- из высоты правого ребенка вычитаем левого)
   2.1. Если bfactor=-2, то разбалансировка в левом поддереве
   2.2. Если bfactor=2, то разбалансировка в правом поддереве
3. Балансируем путем поворотов (малые)
   3.1. Если балансируем левое поддерево, применяем правый поворот
   3.2. Если балансируем правое поддерево, применяем левый поворот
4. Балансируем с помощью больших поворотов
   4.1. Если bfactor для разбалансированного поддерева=1, то делаем большой поворот

Плюсы и минусы реализации

Плюсы:

1. Гарантированная сбалансированность, что обеспечивает выполнение операций вставки, удаления и поиска за O(log n).
2. Поддерживает различные операции вставки/удаления/поиска без потери эффективности.
3. Используется во многих библиотеках и базах данных благодаря своей эффективности и предсказуемости.

Минусы:

1. Сложности в реализации и поддержании сбалансированности дерева.
2. Требует дополнительное хранение информации о сбалансированности в каждом узле, что требует больше памяти.
3. Некоторые операции, такие как вставка и удаление, могут потребовать поворотов и перебалансировки дерева, что может замедлить производительность.

Анализ реализации:

Кол-во	***Время (в нс)***		
n*10^4	**вствка**	**удаление**	**высота**
1	1839	1970	16
2	2103	2018	17
3	2069	2116	18
4	2170	2164	18
5	2292	2323	19
6	2385	2316	19
7	2436	2333	19
8	2509	2460	20
9	2638	2468	20
10	2819	2510	20

3. Красно-черное дерево

Красно-чёрное дерево — двоичное дерево поиска, в котором баланс осуществляется на основе "цвета" узла дерева, который принимает только два значения: "красный" и "чёрный"

При этом все листья дерева являются фиктивными и не содержат данных, но относятся к дереву и являются чёрными

Для экономии памяти фиктивные листья сделаны одним общим фиктивным листом

Также для кч дерева выполняются следующие свойства:

1. Каждый узел промаркирован красным или чёрным цветом
2. Корень и конечные узлы (листья) дерева — чёрные
3. У красного узла родительский узел — чёрный
4. Все простые пути из любого узла x до листьев содержат одинаковое количество чёрных узлов
5. Чёрный узел может иметь чёрного родителя

Теоретическая функция:

Если в дереве хранится n ключей, то глубина дерева $h<=2\log_2(n+1)$

Док-во:

x- произвольная вершина, а bh(x)- черная глубина вершины x, то в поддереве x лежит $>=2^{bh(x)}-1$ ключ

Докажем индукцией по h(x) :

База индукции: $h(x)=0$, x-лист, фиктивная вершина, нет ключа

$2^{bh(x)}-1= 2^0-1=0$

Переход: пусть L-левое поддерево X, R- правое поддерево X

$bh(L)>=bh(x)-1, bh(R)=bh(x)-1$ (т.к. R/L вершина могут быть красными)

По предположению индукции в $L>=2^{bh(L)}-1$, а в $R>=2^{bh(R)}-1$

Суммарно в x: $>=1+ 2^{bh(L)}-1 + R>=2^{bh(R)}-1=2^{bh(x)-1}+2^{bh(x)-1}-1=2^{bh(x)}-1$

Пусть h-высота всего дерева, то $bh(root)>=h/2$

$n>=2^{h/2}-1 \qquad n+1>=2^{h/2} \qquad h/2<=\log_2(n+1) \qquad h<=\log_2(n+1)$

Код сортировки:

- **Нода:** Состоит из ключа, цвета, левого, правого детей. Цвет добавляемой ноды всегда красный, она указывает на фиктивную вершину

```
struct Node
{
        int key;
        boolean color;
        Node* l = nullptr, * r = nullptr, * p = nullptr;
        Node() {
                Node* l = nullptr,
                        * r = nullptr,
                        * p = nullptr;
                this->color = black;
        }

}
*root = nil;
Node* nil = new Node();
void createNode(Node* n, int key) {
        n->p = nil;
        n->l = nil;
        n->r = nil;
        n->key = key;
        n->color = red;
}
```

- **Поиск по ключу:** если элемент есть, то выводится соответствующее сообщение

```
bool findKey(Node* n, int key) const {
        while ((n != nullptr) and (key != n->key)) {
                if (key > n->key)
                        n = n->r;
                else if (key < n->key)
                        n = n->l;
                if (n == nullptr)
                        return false;
                if (key == n->key)

                return true;
        }
}
```

- **Вставка:** идем по дереву, находим нужную позицию для вставки, делаем балансировку

```
void insert(Node* n, int key) {   //вставка
        Node* currentNode = n;
        Node* parent = nil;
        while (nodeExist(currentNode)) {  //пока ребенок существует
                parent = currentNode;     //идем по списку
                if (key < currentNode->key) currentNode = currentNode->l;
                else currentNode = currentNode->r;
        }
        Node* newNode = new Node();       //создаем ноду
        createNode(newNode, key);
        newNode->p = parent;      //присваиваем родителя для ноды
        if (newNode->p == nil) { root = newNode; }
//несли родителя нет, то корень
```

```
else if (key < parent->key) parent->l = newNode;
else parent->r = newNode;
balanceTreeInsert(newNode);      //балансируем дерево
```

Обход в глубину:

- **Преордер:** выводим с корня и доходим до левого и тд

```
void preorderPrint(Node* n) {
	if (!nodeExist(n))
		return;
	cout << n->key << endl;
	preorderPrint(n->l);
	preorderPrint(n->r);
}
```

- **Постордер:** доходим до крайнего левого, потом правого и выводим

```
void postorderPrint(Node* n)  {
	if (!nodeExist(n))
		return;
	postorderPrint(n->l);    //идем по левой до макс
	postorderPrint(n->r);    //идем по правой до макс
	cout << n->key << endl;  //выводим значение
}
```

- **Симметричный:** выводим дерево по возрастанию ключей

```
void inorderPrint(Node* n)  {

	if (!nodeExist(n))
		return;
	inorderPrint(n->l);
	cout << n->key<< endl;
	inorderPrint(n->r);
}
```

- **По ширине:** выводится по уровням

```
void widthPrint(Node* root) {
	vector<Node*> top;
	vector<Node*> tops_;
	if (root == nullptr) {
		return;
	}
	top.push_back(root);
	while (!top.empty()) {
		for (int i = 0; i < top.size(); i++) {
			cout << "\t"<<top[i]->key << "|";
			if (top[i]->color) cout << "red  ";
			else cout << "black  ";
			if (top[i]->l!=nil)
				tops_.push_back(top[i]->l);
			if (top[i]->r != nil)
				tops_.push_back(top[i]->r);
		}
		top.clear();
		top = tops_;
		tops_.clear();
		cout << endl;
	}
```

```
}
```

- **Удаление эл по ключу**:
    1. Если 1 ребенок, то удаляем эл., на его место ставим его ребенка
    5. Если нет детей, удаляем
    6. Если 2 ребенка, то находим самый левый эл в правом поддереве от удаляемого эл.
    7. Делаем нужные балансировки

```
void remove(Node* n, int key) {
Node* removeNode;          //удаляемая нода
Node* child;               //ребенок удаляемого
Node* minRight;

removeNode = findKeyLast(root, key);

if (removeNode == nil) {
        cout << "Такого ключа нет" << endl;
        return;
}

bool removeNodeColor = removeNode->color; //какого цвета ноду удаляем

if (removeNode->l == nil) {                          //если один правый ребенок
        child = removeNode->r;
//меняем местами ребенка и удаляемый
        rbTransplant(removeNode, child);
}
else if (removeNode->r == nil) {//если один левый ребенок
        child = removeNode->l;
        rbTransplant(removeNode, child);
}
else {  //если 2 ребенка
        minRight = findMin(removeNode->r);
        removeNodeColor = minRight-
>color;//раз удаляем другой, запоминаем его цвет

        rbTransplant(removeNode, minRight);
//меняем местами минимальный и удаляемый
        minRight->l = removeNode->l;
//связываем узел с другими после перестановки
        minRight->l->p = minRight;
        minRight->color = removeNode->color;
        child = removeNode;
}
delete removeNode;
if (removeNodeColor == black) {           //если удаляемый был черным
        balanceTreeRemove(child);//балансируем

//(child это тот, кто пришел на место удаляемого)
}
}
```

Балансировка после вставки

Описание балансировки:

1. При вставке большое значение имеет дядя удаляемого элемента
2. Дерево может быть разбалансировано в 5 случаях

   2.1 Если добавили в корень, то просто перекрашиваем его в черный

   2.2 Меняем цвет родителя на черный, если дядя красный, то меняем цвет дяди (тем самым bh равна)

   Чтобы вернуть bh высоту у прадеда, нужно дедушку перекрасить в красный и рекурсивно запуститься от него(т.к. неизвестно, какого цвета прадед)

   А если прадед будет корнем, то его цвет менять не нужно

   2.3 Если дядя черный, то выполняем либо большой, либо малый поворот (меняем цвет у некоторых элементов)

```
void balanceTreeInsert(Node* n) {
        Node* uncle;
        while (n->p->color == red) {
                if (n->p == n->p->p->l) {        //если родитель слева
                        uncle = n->p->p->r;              //находим дядю
                        if (uncle->color == red) {//если дядя красный
                                uncle-
>color = black;//меняем цвет дяди, родителя
                                n->p->color = black;
                                n->p->p->color = red;//дедушка красный
                                n = n->p->p;//переходим наверх(к дедушке)

                        }
                        else {
                                if (n == n->p->r) {      //если сын справа
                                        leftRotate(n->p);

                                }
                                n->p->color = black;//цвет родителя меняем
                                n->p->p->color = red;
                                n = n->p->p;
                                rightRotate(n);
                                return;
                        }
                }
                else {
                        uncle = n->p->p->l;
        //то же самое для левого дяди (поворот в другую сторону)
                        if (uncle->color == red) {//если дядя красный
                                uncle-
>color = black;//меняем цвет дяди, родителя
                                n->p->color = black;
                                n->p->p->color = red;//дедушка красный
                                n = n->p->p;//переходим наверх(к дедушке)
                                balanceTreeInsert(n);
                        }
                        else {
                                if (n == n->p->l) {      //если сын справа
                                        rightRotate(n->p);
```

```
                                }
                                n->p->color = black;//цвет родителя меняем
                                n->p->p->color = red;
                                n = n->p->p;
                                leftRotate(n);
                                return;
                        }
                }
        }
        root->color = black;
}
```

Балансировка после удаления

Описание балансировки:

1. При удалении большое значение имеет брат удаляемого элемента
2. Если удаляем красный, то просто удаляем
3. Если удаляем черный с одним ребенком, то удаляем, а ребенка перекрашиваем в черный (по условию дерева ребенок был красный)
4. При удалении элемента с черными детьми есть много случаев, там нужно рассматривать брата удаляемого элемента

```
void balanceTreeRemove(Node* n) {
        /*В этой функции входное n это то, что пришло на место удаленного элемента*/

                Node* brother;

                while (n != root && n->color == black) {
                        if (n == n->p->l) {
        //если был удален левый элемент
                                brother = n->p->r;              //брат справа
                                if (brother->color == red) {
        //цвет брата красный
                                        brother->color = black;
                                        n->p->color = red;
                                        leftRotateDelete(n->p);
                                        brother = n->p->r;
                                }

                                if (brother->l->color == black && brother->r-
>color == black) {
                                        brother->color = red;
        //если дети брата черные
                                        n = n->p;
                                }
                                else {  //если у брата есть красный сын
                                        if (brother->r-
>color == black) {//если сын слева
                                                brother->l->color = black;
                                                brother->color = red;
                                                rightRotateDelete(brother);
        //большой поворот
                                                brother = n->p->r;
                                        }
                                        brother->color = n->p->color;
                                        n->p->color = black;
                                        brother->r->color = black;
```

```
					leftRotateDelete(n->p);
					n = root;
				}
			}
			else {	//если удалили правый элемент
				brother = n->p->l;
				if (brother->color == red) {
	//если брат красный
					brother->color = black;
					n->p->color = red;
					rightRotateDelete(n->p);
					brother = n->p->l;
				}

				if (brother->r->color == black && brother->r-
>color == black) {
					brother->color = red;
	//если дети брата черные
					n = n->p;
				}
				else {	//если есть красный сын
					if (brother->l-
>color == black) {//если красный справа
						brother->r->color = black;
						brother->color = red;
						leftRotateDelete(brother);
	//большой поворот
						brother = n->p->l;
					}
					brother->color = n->p->color;
					n->p->color = black;
					brother->l->color = black;
					rightRotateDelete(n->p);
					n = root;
				}
			}
		}
		n->color = black;		//перекрашиваем корень
	}
```

- **Поиск высоты дерева**

```
	int hight(Node* root) {
		vector<Node*> top;
		vector<Node*> tops_;
		if (!root) {
			return 0;
		}
		int hight = 1;
		top.push_back(root);
		while (!top.empty()) {
			for (int i = 0; i < top.size(); i++) {
				if (top[i]->l)
					tops_.push_back(top[i]->l);
				if (top[i]->r)
					tops_.push_back(top[i]->r);
			}
			hight++;
			top.clear();
			top = tops_;
			tops_.clear();
		}
		return hight;
	}
```

- ## Вспомогательные функции

Проверка существования ноды(не листа)

```
bool nodeExist(Node* n) {        //существует ли нода
        return ((n != nil) and (n != nullptr));
}
```

Создание ноды для вставки (указывает на лист, цвет красный)

```
void createNode(Node* n, int key) {
        n->p = nullptr;
        n->l = nil;
        n->r = nil;
        n->key = key;
        n->color = red;
}
```

Поиск минимального элемента

```
Node* findMin(Node* n) {
        if (n->l!=nil) return findMin(n->l);
        else return n;
}
```

Левый и правый повороты:

```
void rightRotate(Node* n) {
        swap(n, n->l);
        Node* temp = n->r;
        n->r = n->l;
        n->l = n->r->l;
        n->l->p = n;
        n->r->l = n->r->r;
        n->r->r = temp;
        n->r->r->p = n->r;
}
void leftRotate(Node* n) {
        swap(n, n->r);
        Node* temp = n->l;
        n->l = n->r;
        n->r = n->l->r;
        n->r->p = n;
        n->l->r = n->l->l;
        n->l->l = temp;
        n->l->l->p = n->l;
}
```

Перестановка нод метсами:

```
void transportNode(Node* toNode, Node* fromNode) {
        if (toNode == root) root = fromNode;
        else if (toNode == toNode->p->l) toNode->p->l = fromNode;
        else toNode->p->r = fromNode;
        fromNode->p = toNode->p;
}
```

Плюсы и минусы реализации

Плюсы:

1. Гарантированная сбалансированность, что обеспечивает выполнение операций вставки, удаления и поиска за O(log n).
2. Операции добавления/удаления элемента эффективны и быстры (в больших деревьях быстрее, чем в AVL-деревьях).
3. Используется во многих реализациях языков программирования и баз данных.

Минусы:

1. Сложности в реализации и поддержании сбалансированности дерева.
2. Требует дополнительное хранение информации о цвете узлов, что требует больше памяти.
3. Операции поиска в худшем случае могут быть немного медленнее, чем в AVL-дереве.

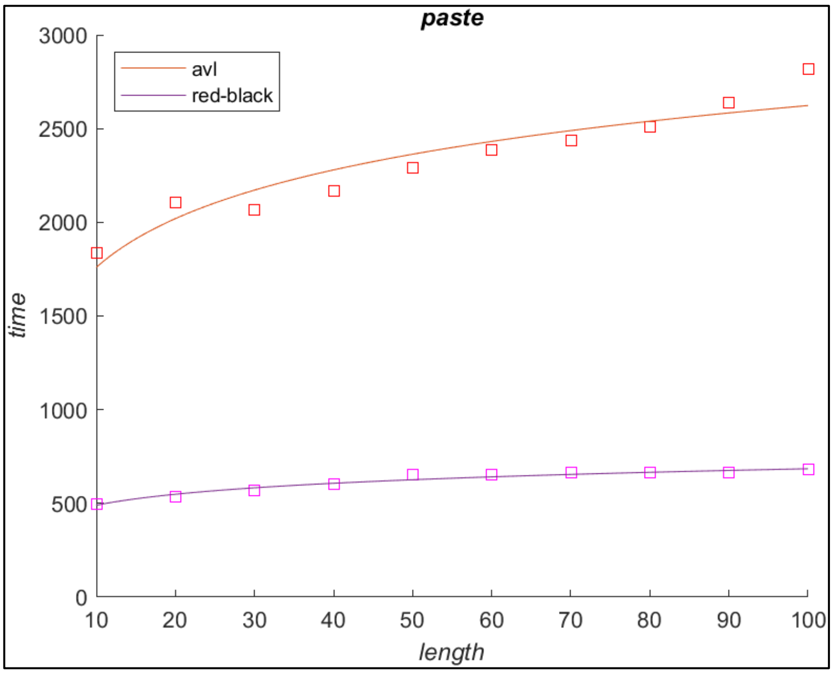
Анализ реализации:

Кол-во	***Время (в нс)***		
n*10^4	**вствка**	**удаление**	**высота**
1	499	532	17
2	536	589	19
3	567	624	20
4	602	625	20
5	653	656	21
6	652	661	21
7	662	654	22
8	662	658	22
9	665	680	23
10	680	670	23

Анализ бинарных деревьев:

Сравнение вставки случайного элемента в АВЛ и КЧ деревья. Данные о времени вставки являются средним значением от вставки случайных элементов length/2 раз

Кол-во	***Время (в нс)***	
n*10^4	**AVL**	**КЧ**
1	1839	499
2	2103	536
3	2069	567
4	2170	602
5	2292	653
6	2385	652
7	2436	662
8	2509	662
9	2638	665
10	2819	680

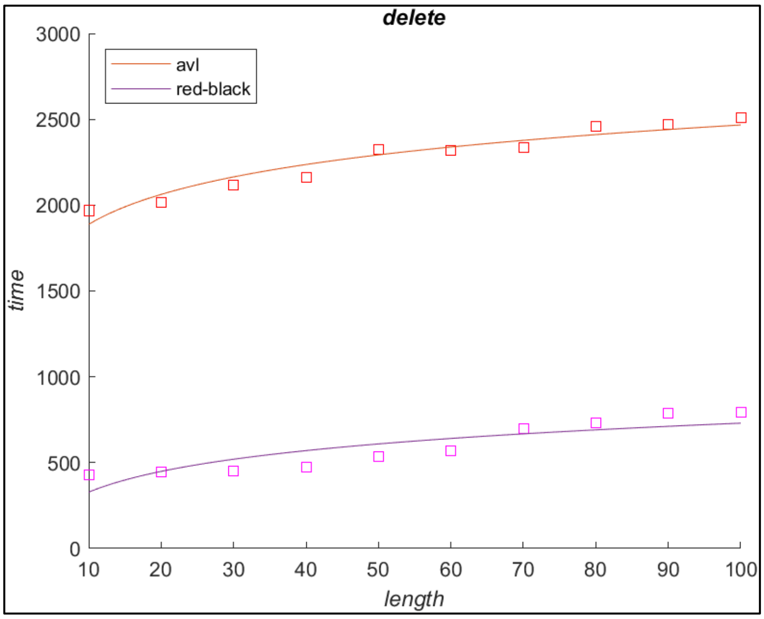


Аппроксимированные функции:

Дерево:	**Уравнение:**	**Цвет графика:**
АВЛ-дерево	***374.9*log(n)+896.4***	Красный
КЧ дерево	***84.4*log(n)+295.8***	Пурпурный

Сравнение удаления случайного элемента в АВЛ и КЧ деревья. Данные о времени удаления являются средним значением от удаления случайных элементов length/2 раз

Кол-во	***Время (в нс)***	
n*10^4	**AVL**	**КЧ**
1	1970	532
2	2018	589
3	2116	624
4	2164	625
5	2323	656
6	2316	661
7	2333	654
8	2460	658
9	2468	680
10	2510	670

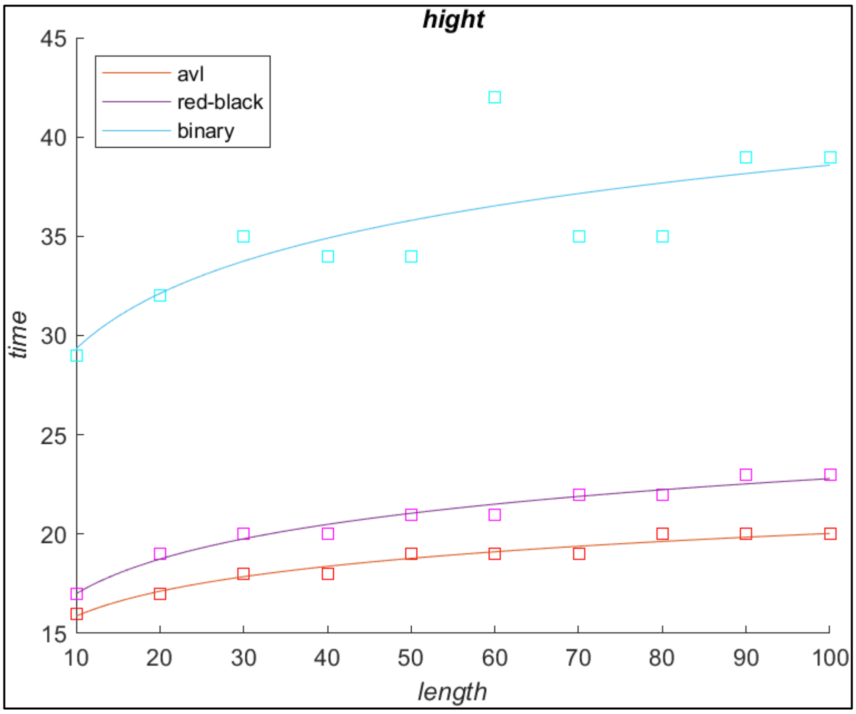


Аппроксимированные функции:

Дерево:	**Уравнение:**	**Цвет графика:**
АВЛ-дерево	***251.3*log(n)+1309.4***	Красный
КЧ дерево	***174.6*log(n)+-73.8***	Пурпурный

Сравнение высот для АВЛ, КЧ, Бинарного деревьев. Высота бралась от вставки случайных элементов

Кол-во			
n*10^4	**АВЛ**	**КЧ**	**Бинарное**
1	16	17	29
2	17	19	32
3	18	20	35
4	18	20	34
5	19	21	34
6	19	21	42
7	19	22	35
8	20	22	35
9	20	23	39
10	20	23	39



Аппроксимированные функции:

Дерево:	**Уравнение:**	**Цвет графика:**
АВЛ-дерево	***1.8*log(n)+11.7***	Красный
КЧ дерево	***2.5*log(n)+11.2***	Пурпурный
Бинарное дерево	***4*log(n)+20***	Голубой

Вывод

Экспериментальные данные подтверждают теоретические материалы:

1. Красно черное дерево работает быстрее, чем АВЛ дерево, в плане удаления и вставки элементов
2. В АВЛ дереве высота более сбалансированная, чем в Красно Черном. Это дает преимущество в поиске элементов для больших высот (на небольших высотах разница незаметна)
3. Самый долгий поиск будет в бинарном дереве, так как в худшем случае может получиться связный список со сложностью поиска элемента O(n).

Ссылка на репозиторий
https://github.com/pavlichek121/2301KomissarovPElr3.git