**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

**Отчёт**

**по лабораторной работе №2**

**по дисциплине «Алгоритмы и структуры данных».**

**Тема: Самобалансирующие двоичные деревья поиска**.

|  |  |  |
| --- | --- | --- |
| Студентка гр. 3354 |  | Чикарёва М.Д. |
| Преподаватель |  | Пестерев Д. О. |

**Санкт-Петербург**

**202****4**

**Цель лабораторной работы**: реализация самобалансирующихся деревьев поиска и экспериментальная проверка оценок высоты данных деревьев.

**Представленные в работе самобалансирующие двоичные деревья поиска**: бинарное дерево поиска, АВЛ-дерево и красно-чёрное дерево.

**Теоретическая часть**

*Бинарное дерево поиска* — это структура данных, используемая для организации и хранения данных в отсортированном виде, состоит из узлов, каждый из которых содержит значение-ключ и ссылки на левый и правый потомки. Узел, от которого исходит все дерево — корень дерева, крайние узлы дерева, не имеющие ни одного потомка — листовые элементы. Каждый узел в двоичном дереве поиска имеет не более двух дочерних элементов, причём левый дочерний элемент содержит значения, меньшие, чем родительский узел, а правый дочерний элемент содержит значения, большие, чем родительский узел. Такая иерархическая структура обеспечивает эффективный поиск, вставку и удаление данных, хранящихся в дереве.

Поиск узла по значению зависит от высоты дерева, теоретическая оценка — .

**АВЛ-дерево**.

1. Определение.

АВЛ-дерево прежде всего является бинарным деревом поиска, реализующее то самое общее условие баланса — узлы хранят значение высоты дерева, корнем которого они являются. Это значит, что для поиска нужного ключа в АВЛ-дереве можно использовать стандартный алгоритм. Будем считать, что все ключи в дереве целочисленны и не повторяются.

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. Доказано, что этого свойства достаточно для того, чтобы высота дерева логарифмически зависела от числа его узлов: высота h АВЛ-дерева с n ключами лежит в диапазоне от log2(n+1) до 1.44log2(n+2) − 0.328. Так как основные операции над двоичными деревьями поиска (поиск, вставка и удаление узлов) линейно зависят от его высоты, то получаем гарантированную логарифмическую зависимость времени работы этих алгоритмов от числа ключей, хранимых в дереве

Коэффициент баланса AVL-дерева

- Коэффициент баланса известен как разница между высотой левого и правого поддерева.

- Коэффициент баланса (узел) = высота (узел->слева) – высота (узел->справа)

- Допустимые значения BF: –1, 0 и +1.

- Значение –1 указывает, что правое поддерево содержит одно лишнее, т. е. дерево тяжелое справа.

- Значение +1 указывает, что левое поддерево содержит одно лишнее, т. е. дерево тяжелое слева.

- Значение 0 показывает, что дерево содержит равные узлы с каждой стороны, т. е. дерево идеально сбалансировано.

1. Описание алгоритма вставки/удаления с последующей балансировкой.

- **Вставка**. Операция вставки почти такая же, как в простых бинарных деревьях поиска. После каждой вставки мы балансируем высоту дерева. Операция вставки занимает O(log n) худшей временной сложности.

1. Проходим от начала дерева вниз, на каждом шаге сравнивая

значение нового узла с текущими.

1. Доходим до конца какого-либо поддерева и делаем новый

узел правым или левым его потомком в зависимости от значения.

1. Проверяем соотношение длин поддеревьев и, если нужно,

проводим балансировку. Алгоритм для балансирования может спускаться вниз из начального узла или подниматься вверх от свежедобавленного, по ходу движения пересчитывать разницу высот и совершать повороты, если где-то обнаружилась разница в два уровня. Балансировка продолжается, пока все значения высот не пересчитаются, а дисбаланс не будет устранён.

Если BF(узел) = +2 и BF(узел -> левый дочерний элемент) = +1, выполняем вращение LL.

Если BF(узел) = -2 и BF(узел -> правый дочерний элемент) = 1, выполняем вращение RR.

Если BF(узел) = -2 и BF(узел -> правый дочерний элемент) = +1, выполняем вращение RL.

Если BF(узел) = +2 и BF(узел -> левый дочерний элемент) = -1, выполняем вращение LR.

- **Удаление.** Мы удаляем, используя ту же логику, что и в простых двоичных деревьях поиска. После удаления мы при необходимости реструктурируем дерево, чтобы сохранить его сбалансированную высоту.

1. Находим удаляемый элемент.
2. Переходим в его правого потомка, если он есть, и находим там минимальный узел.
3. Минимальный узел не удаляется безвозвратно, а подставляется на место удаляемого — происходит замена ссылок.
4. Слева к минимальному узлу присоединяется левый потомок удаляемого узла, справа — то, что осталось от правого потомка после вычленения минимального.
5. Проводим балансировку.

Так как вставка рекурсивная, балансировка выполняется начиная с последнего открытого экземпляра функции вставки и заканчивая первым.

Возможны два случая:

1. **удаление из правого поддерева.**

Если BF(узел) = +2 и BF(узел -> левый дочерний элемент) = +1, выполняем вращение LL.

Если BF(узел) = +2 и BF(узел -> левый дочерний элемент) = -1, выполняем вращение LR.

Если BF(узел) = +2 и BF(узел -> левый дочерний элемент) = 0, выполняем вращение LL.

1. **удаление из левого поддерева.**

Если BF(узел) = -2 и BF(узел -> правый дочерний элемент) = -1, выполняем вращение RR.

Если BF(узел) = -2 и BF(узел -> правый дочерний элемент) = +1, выполняем вращение RL.

Если BF(узел) = -2 и BF(узел -> правый дочерний элемент) = 0, выполняем вращение RR.

1. Оценка высоты дерева.

Пусть высота поддерева с корнем x -> h(x), высота поддерева T -> h(T).

Mh - мин. число вершин АВЛ-дерева высоты h. Тогда mh+2 = mh+1 + mh + 1. Докажем равенство mh = Fh+2 - 1 по мат.индукции:

База индукции: m1 = F3 - 1 - верно, m1 = 1, F3 = 2.

Гипотеза: mh = Fh+2 - 1 - верно.

Тогда: mh+1 = mh + mh-1 + 1 = Fh+2 - 1 + Fh+1 - 1 + 1 = Fh+3 - 1.

Таким образом, mh = Fh+2 - 1 - доказано.

Fh = Ω(φh), φ = -> n ≥ φh.

logφn ≥ h -> высота АВЛ-дерева из N вершин - O(logn).

**Красно-чёрное дерево.**

1. Определение.

Красно-чёрное дерево - это самобалансирующееся дерево поиска, где каждый узел имеет дополнительный атрибут: цвет, который может быть как красным, так и черным, таким образом, каждая вершина хранит один дополнительный бит - её цвет.

Основная цель этих деревьев - поддерживать баланс во время вставок и удалений, обеспечивая эффективный поиск данных и манипулирование ими.2.  ¦¤»© «¨±² (nil)|·¥°­»©.

Красно-чёрное дерево обладает следующими свойствами:

- Каждая вершина либо красная, либо чёрная. (1)

- Корень дерева как правило чёрный. (2)

- У красных вершин не может быть красных потомков (нет двух последовательных красных вершин на любом пути), то есть оба ребёнка чёрные. (3)

- Каждый путь, идущий вниз от корня к листьям, содержит одинаковое количество чёрных вершин. (4)

- Все листья (NIL) чёрные. (5)

Благодаря этим ограничениям путь от корня до самого дальнего листа не более чем вдвое длиннее, чем до самого ближнего, и дерево примерно сбалансировано. Операции вставки, удаления и поиска требуют в худшем случае времени, пропорционального длине дерева, что позволяет красно-чёрным деревьям в худшем случае быть более эффективными, чем обычные двоичные деревья поиска.

1. Описание алгоритма вставки/удаления с последующей балансировкой.

- **Вставка.**

1. Вставляем узел и красим его в красный цвет.
2. Смотрим на предка и проверяем, не нарушается ли красно-чёрное свойство.
3. Если необходимо, перекрашиваем узел и производим поворот, чтобы сбалансировать дерево.

Буквой N будем обозначать текущий узел (окрашенный красным). Сначала это новый узел, который вставляется, но эта процедура может применяться рекурсивно к другим узлам.

P будем обозначать предка N, через G обозначим дедушку N, а U будем обозначать дядю (узел, имеющий общего родителя с узлом P).

Случай 1: Текущий узел N в корне дерева, тогда он перекрашивается в чёрный, чтобы оставить верным (2), так как это действие добавляет один чёрный узел в каждый путь, то (4) не нарушается.

Случай 2: Предок P текущего узла чёрный, т.е. (3) не нарушается. В этом случае дерево остаётся корректным. (4) тоже не нарушается, так как N красный.

Случай 3:  Если и родитель P, и дядя U — красные, то они оба могут быть перекрашены в чёрный, и дедушка G станет красным (для сохранения (4)). Теперь у текущего красного узла N чёрный родитель. Так как любой путь через родителя или дядю должен проходить через дедушку, число чёрных узлов в этих путях не изменится. Однако, дедушка G теперь может нарушить (2) или (3) (может быть нарушено, так как родитель G может быть красным). Чтобы это исправить, вся процедура рекурсивно выполняется на G из случая 1.

Случай 4: Родитель P является красным, но дядя U — чёрный. Также, текущий узел N — правый потомок P, а P в свою очередь — левый потомок своего предка G. В этом случае может быть произведен поворот дерева, который меняет роли текущего узла N и его предка P. Тогда, для бывшего родительского узла P в обновленной структуре используем случай 5, потому что (3) все ещё нарушено. Вращение приводит к тому, что некоторые пути проходят через узел N, чего не было до этого. Это также приводит к тому, что некоторые пути не проходят через узел P. Однако, оба эти узла являются красными, так что (4) не нарушается при вращении. Однако (3) всё ещё нарушается, но теперь задача сводится к Случаю 5 .

Случай 5: Родитель P является красным, но дядя U — чёрный, текущий узел N — левый потомок P и P — левый потомок G. В этом случае выполняется поворот дерева на G. В результате получается дерево, в котором бывший родитель P теперь является родителем и текущего узла N и бывшего дедушки G. Известно, что G — чёрный, так как его бывший потомок P не мог бы в противном случае быть красным (без нарушения (3)). Тогда цвета P и G меняются и в результате дерево удовлетворяет (3). (4) также остается верным, так как все пути, которые проходят через любой из этих трех узлов, ранее проходили через G, поэтому теперь они все проходят через P. В каждом случае, из этих трёх узлов только один окрашен в чёрный.

- **Удаление.** Так как при удалении красной вершины свойства дерева не нарушаются, то восстановление балансировки потребуется только при удалении чёрной.

1. Рассматриваем ребёнка удалённой вершины.
2. Если брат этого ребёнка красный, то делаем вращение вокруг ребра между отцом и братом, тогда брат станет родителем отца.
3. Перекрашиваем его в чёрный, а отца - в красный цвет, сохраняя таким образом чёрную вершину дерева.
4. Если брат текущей вершины был чёрным, то красим брата в красный и рассматриваем далее отца вершины. Делаем его чёрным, это не повлияет на кол-во чёрных узлов на путях, проходящих через удалённую вершину, но добавит один к числу чёрных узлов на путях, проходящих через неё, восстанавливая тем самым влияние удалённого чёрного узла.

После вставки или удаления требуется операция перекраски, требующая O(log n) или O(1) смен цветов и не более чем трёх поворотов дерева (для вставки — не более двух).

1. Оценка высоты дерева.

Красно-чёрное дерево с n внутренними вершинами имеет высоту не больше 2lg(n+1). Докажем это.

Поддерево с корнем в x содержит по меньшей мере 2bh(x) - 1 внутренних вершин, докажем проведением индукции от листьев к корню. Для листьев чёрная высота равна 0, поддерево содержит не менее 2bh(x) - 1 = 20 - 1 = 0 внутренних вершин.

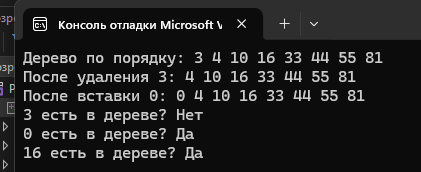
Теперь пусть вершина x не является листом и имеет чёрную высоту k -> оба её ребёнка имеют чёрную высоту не меньше k-1 (красный будет иметь k, чёрный - k-1).

По предположению индукции левое и правое поддеревья вершины x содержат не менее 2k-1 - 1 + 2k-1 - 1 + 1 = 2k - 1 внутренних вершин. Высота дерева h. Из-за свойства (4), чёрная высота дерева не меньше h/2. -> n ≥ 2h/2 - 1 -> lg(n+1) ≥ h/2 -> h ≤ 2lg(n+1).

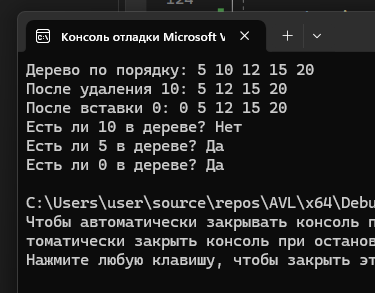
**Практическая часть**

1. Реализация бинарного дерева поиска, АВЛ-дерева и красно-чёрного дерева.

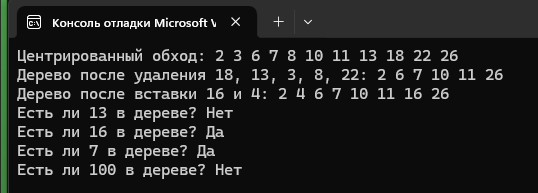
**Бинарное дерево поиска.**



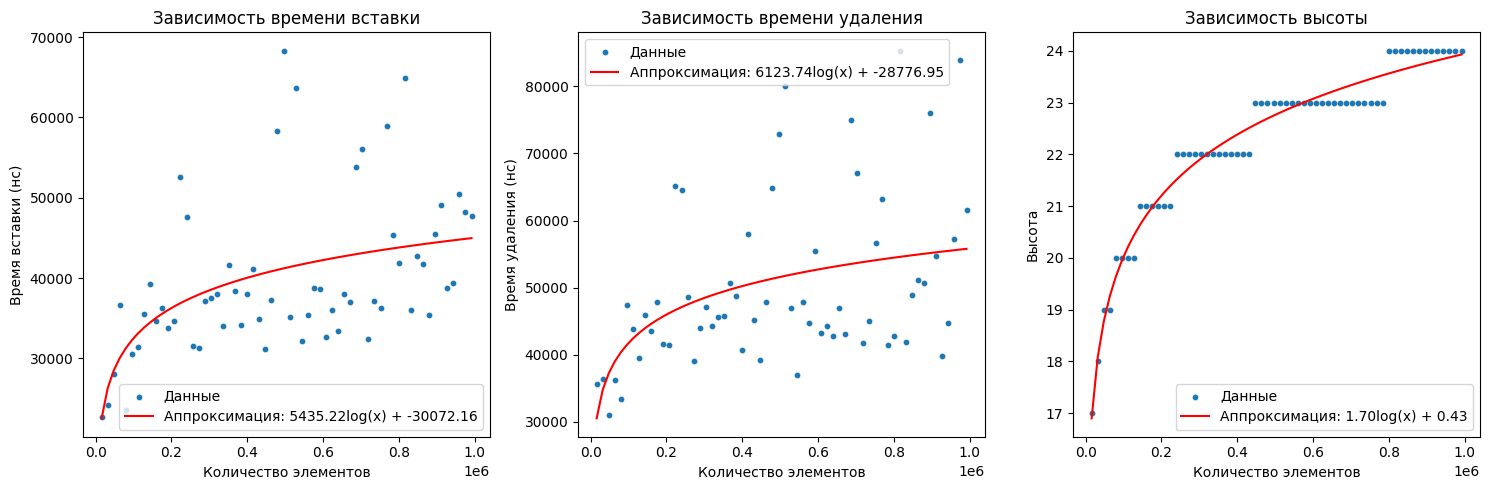
**АВЛ-дерево.**

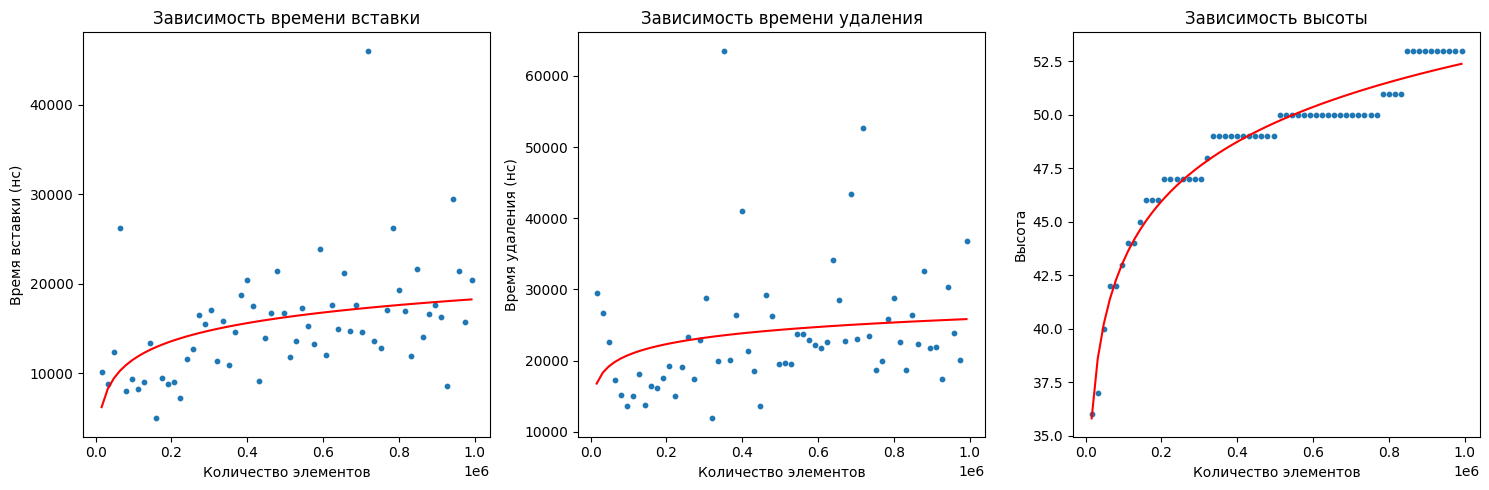


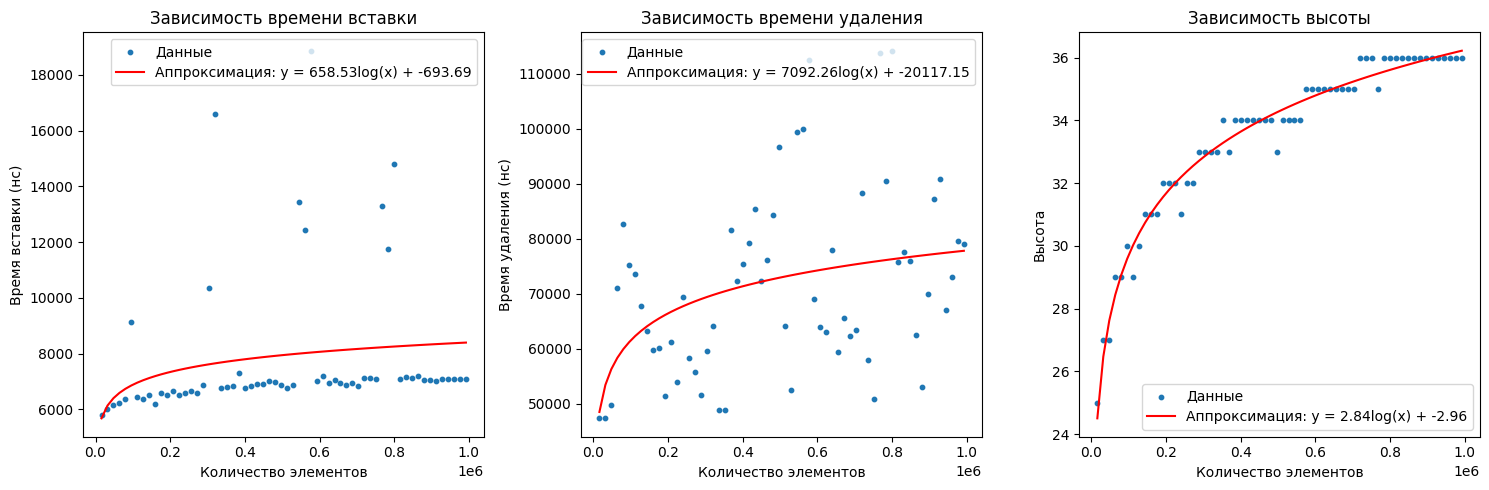
**Красно-чёрное дерево.**



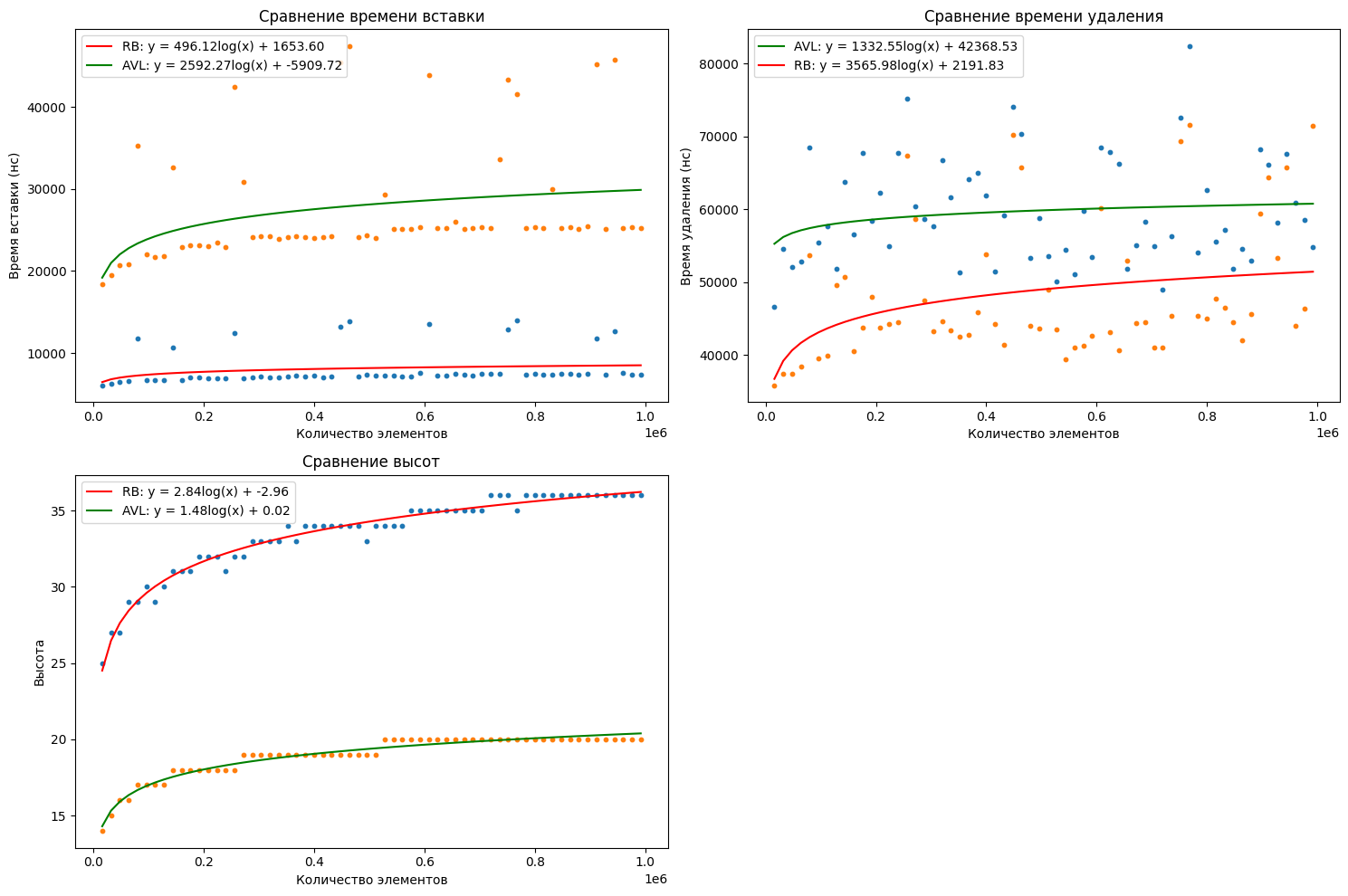
1. Графики.

- Графики для бинарного дерева поиска. Зависимость высоты дерева поиска от количества ключей, при условии, что значение ключа - случайная величина, распределенная равномерно.

- Графики для АВЛ-дерева. Зависимость от количества ключей, при условии, что значени ключей монотонно возрастают.

- Графики для чёрно-красного дерева. Зависимость от количества ключей, при условии, что значени ключей монотонно возрастают.

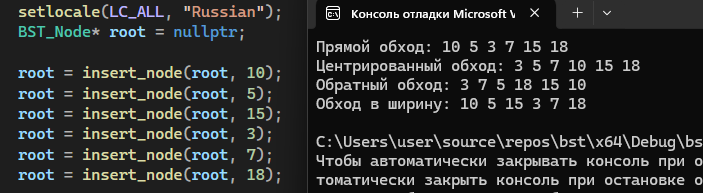
- Графики сравнения авл и чёрно-красного деревьев.



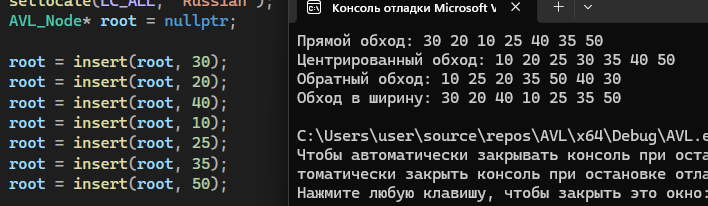
Видно, что чёрно-красное дерево работает эффективнее АВЛ, высоты близки к теоретическим значениям.

1. Реализация обходов в глубину (preorder(прямой), inorder(центрированный) и postorder(обратный), а также обхода в ширину. Коды этих функций будут одинаковыми для всех деревьев.

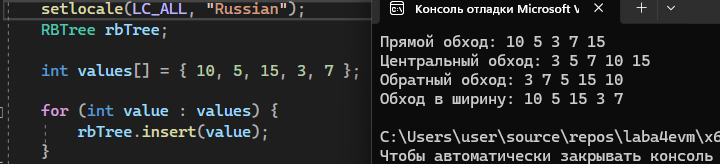
- Бинарное дерево поиска.



- АВЛ-дерево.



- Красно-чёрное дерево.



**Ссылка на Github с исходным кодом**

<https://github.com/essslish/alg2.git>

**Раздел со всем кодом**

**#bst.cpp**

#include <iostream>

using namespace std;

struct BST\_Node {

int data;

BST\_Node\* left;

BST\_Node\* right;

};

BST\_Node\* new\_one(int data) {

BST\_Node\* newNode = new BST\_Node();

newNode->data = data;

newNode->left = newNode->right = nullptr;

return newNode;

}

BST\_Node\* insert\_node(BST\_Node\* root, int data) {

if (root == nullptr) {

return new\_one(data);

}

if (data < root->data) {

root->left = insert\_node(root->left, data);

}

else if (data > root->data) {

root->right = insert\_node(root->right, data);

}

return root;

}

void in\_order(BST\_Node\* root) {

if (root != nullptr) {

in\_order(root->left);

cout << root->data << " ";

in\_order(root->right);

}

}

BST\_Node\* search\_node(BST\_Node\* root, int key) {

if (root == nullptr || root->data == key) {

return root;

}

if (root->data < key) {

return search\_node(root->right, key);

}

return search\_node(root->left, key);

}

BST\_Node\* min\_val(BST\_Node\* node) {

BST\_Node\* cur = node;

while (cur && cur->left != nullptr) {

cur = cur->left;

}

return cur;

}

BST\_Node\* delete\_n(BST\_Node\* root, int data) {

if (root == nullptr)

return root;

if (data < root->data) {

root->left = delete\_n(root->left, data);

}

else if (data > root->data) {

root->right = delete\_n(root->right, data);

}

else {

if (root->left == nullptr) {

BST\_Node\* temp = root->right;

delete root;

return temp;

}

else if (root->right == nullptr) {

BST\_Node\* temp = root->right;

delete root;

return temp;

}

BST\_Node\* temp = min\_val(root->right);

root->data = temp->data;

root->right = delete\_n(root->right, temp->data);

}

return root;

}

int main()

{

setlocale(LC\_ALL, "Russian");

BST\_Node\* root = nullptr;

root = insert\_node(root, 44);

root = insert\_node(root, 33);

root = insert\_node(root, 10);

root = insert\_node(root, 81);

root = insert\_node(root, 4);

root = insert\_node(root, 16);

root = insert\_node(root, 3);

root = insert\_node(root, 55);

cout << "Дерево по порядку:";

in\_order(root);

cout << endl;

root = delete\_n(root, 3);

cout << "После удаления 3: ";

in\_order(root);

cout << endl;

root = insert\_node(root, 0);

cout << "После вставки 0: ";

in\_order(root);

cout << endl;

cout << "3 есть в дереве? " << (search\_node(root, 3) ? "Да" : "Нет") << endl;

cout << "0 есть в дереве? " << (search\_node(root, 0) ? "Да" : "Нет") << endl;

cout << "16 есть в дереве? " << (search\_node(root, 16) ? "Да" : "Нет") << endl;

return 0;

}

**#AVL.cpp**

#include <iostream>

using namespace std;

struct AVL\_Node {

int key;

size\_t height;

AVL\_Node\* left;

AVL\_Node\* right;

AVL\_Node(int k) { key = k; left = right = nullptr; height = 1; }

};

size\_t height(AVL\_Node\* p) {

return p ? p->height : 0;

}

int balance\_factor(AVL\_Node\* p) {

return height(p->right) - height(p->left);

}

void fix\_height(AVL\_Node\* p) {

size\_t h\_left = height(p->left);

size\_t h\_right = height(p->right);

p->height = (h\_left > h\_right ? h\_left : h\_right) + 1;

}

AVL\_Node\* rotate\_left(AVL\_Node\* q) {

AVL\_Node\* p = q->right;

q->right = p->left;

p->left = q;

fix\_height(q);

fix\_height(p);

return p;

}

AVL\_Node\* rotate\_right(AVL\_Node\* p) {

AVL\_Node\* q = p->left;

p->left = q->right;

q->right = p;

fix\_height(p);

fix\_height(q);

return q;

}

AVL\_Node\* balance(AVL\_Node\* p) {

fix\_height(p);

if (balance\_factor(p) == 2) {

if (balance\_factor(p->right) < 0)

p->right = rotate\_right(p->right);

return rotate\_left(p);

}

if (balance\_factor(p) == -2) {

if (balance\_factor(p->left) > 0)

p->left = rotate\_left(p->left);

return rotate\_right(p);

}

return p;

}

AVL\_Node\* insert(AVL\_Node\* p, int k) {

if (p == nullptr) return new AVL\_Node(k);

if (k < p->key) {

p->left = insert(p->left, k);

}

else {

p->right = insert(p->right, k);

}

return balance(p);

}

AVL\_Node\* find\_min(AVL\_Node\* p) {

return p->left ? find\_min(p->left) : p;

}

AVL\_Node\* remove\_min(AVL\_Node\* p) {

if (p->left == nullptr)

return p->right;

p->left = remove\_min(p->left);

return balance(p);

}

AVL\_Node\* remove(AVL\_Node\* p, int k) {

if (p == nullptr) return 0;

if (k < p->key) {

p->left = remove(p->left, k);

}

else if (k > p->key) {

p->right = remove(p->right, k);

}

else {

AVL\_Node\* left = p->left;

AVL\_Node\* right = p->right;

delete p;

if (right == nullptr) return left;

AVL\_Node\* min = find\_min(right);

min->right = remove\_min(right);

min->left = left;

return balance(min);

}

return balance(p);

}

void in\_order(AVL\_Node\* p) {

if (p != nullptr) {

in\_order(p->left);

cout << p->key << " ";

in\_order(p->right);

}

}

bool search(AVL\_Node\* p, int key) {

if (p == nullptr) return false;

if (key == p->key) return true;

if (key < p->key) return search(p->left, key);

return search(p->right, key);

}

int main()

{

setlocale(LC\_ALL, "Russian");

AVL\_Node\* root = nullptr;

root = insert(root, 10);

root = insert(root, 5);

root = insert(root, 15);

root = insert(root, 20);

root = insert(root, 12);

cout << "Дерево по порядку: ";

in\_order(root);

cout << endl;

root = remove(root, 10);

cout << "После удаления 10: ";

in\_order(root);

cout << endl;

root = insert(root, 0);

cout << "После вставки 0: ";

in\_order(root);

cout << endl;

cout << "Есть ли 10 в дереве?" << (search(root, 10) ? " Да" : " Нет") << endl;

cout << "Есть ли 5 в дереве?" << (search(root, 5) ? " Да" : " Нет") << endl;

cout << "Есть ли 0 в дереве?" << (search(root, 0) ? " Да" : " Нет") << endl;

return 0;

}

**#orders\_for\_trees.cpp (обходы для бинарного дерева)**

void pre\_order(BST\_Node\* node) { //вместо BST\_Node\* заменять на AVL\_Node\*, реализация обходов для RB tree в файле rb.cpp

if (node) {

cout << node->data << " ";

pre\_order(node->left);

pre\_order(node->right);

}

else { return; }

}

void post\_order(BST\_Node\* node) {

if (node) {

post\_order(node->left);

post\_order(node->right);

cout << node->data << " ";

}

else { return; }

}

#define MAX\_Q\_SIZE 100

struct qu {

BST\_Node\* items[MAX\_Q\_SIZE];

int front;

int rear;

qu() :front(0), rear(0) {}

bool is\_empty() {

return front == rear;

}

void en\_qu(BST\_Node\* node) {

if (rear >= MAX\_Q\_SIZE) {

cout << "Очередь полная";

return;

}

items[rear++] = node;

}

BST\_Node\* de\_qu() {

if (is\_empty()) {

return nullptr;

}

return items[front++];

}

};

void level\_order(BST\_Node\* root) {

if (!root) return;

qu que;

que.en\_qu(root);

while (!que.is\_empty()) {

BST\_Node\* cur = que.de\_qu();

cout << cur->data << " ";

if (cur->left) que.en\_qu(cur->left);

if (cur->right) que.en\_qu(cur->right);

}

}

void free\_tree(BST\_Node\* tree) {

if (tree != NULL) {

free\_tree(tree->left);

free\_tree(tree->right);

delete tree;

}

}

**#rb.cpp**

#include <iostream>

enum Color { RED, BLACK };

using namespace std;

struct rb\_node {

int data;

Color color;

rb\_node\* left, \* right, \* parent;

rb\_node(int value) : data(value), color(RED), left(nullptr), right(nullptr), parent(nullptr) {}

};

struct rb\_tree {

rb\_node\* root;

rb\_tree() : root(nullptr) {}

void insert(int value) {

rb\_node\* newNode = new rb\_node(value);

root = bst\_ins(root, newNode);

fix\_ins(newNode);

}

rb\_node\* bst\_ins(rb\_node\* root, rb\_node\* newNode) {

if (root == nullptr) return newNode;

if (newNode->data < root->data) {

root->left = bst\_ins(root->left, newNode);

root->left->parent = root;

}

else {

root->right = bst\_ins(root->right, newNode);

root->right->parent = root;

}

return root;

}

void fix\_ins(rb\_node\* node) {

while (node != root && node->parent->color == RED) {

if (node->parent == node->parent->parent->left) {

rb\_node\* uncle = node->parent->parent->right;

if (uncle && uncle->color == RED) {

node->parent->color = BLACK;

uncle->color = BLACK;

node->parent->parent->color = RED;

node = node->parent->parent;

}

else {

if (node == node->parent->right) {

node = node->parent;

rot\_left(node);

}

node->parent->color = BLACK;

node->parent->parent->color = RED;

rot\_right(node->parent->parent);

}

}

else {

rb\_node\* uncle = node->parent->parent->left;

if (uncle && uncle->color == RED) {

node->parent->color = BLACK;

uncle->color = BLACK;

node->parent->parent->color = RED;

node = node->parent->parent;

}

else {

if (node == node->parent->left) {

node = node->parent;

rot\_right(node);

}

node->parent->color = BLACK;

node->parent->parent->color = RED;

rot\_left(node->parent->parent);

}

}

}

root->color = BLACK;

}

void rot\_left(rb\_node\* node) {

rb\_node\* y = node->right;

node->right = y->left;

if (y->left != nullptr) y->left->parent = node;

y->parent = node->parent;

if (node->parent == nullptr) {

root = y;

}

else if (node == node->parent->left) {

node->parent->left = y;

}

else {

node->parent->right = y;

}

y->left = node;

node->parent = y;

}

void rot\_right(rb\_node\* node) {

rb\_node\* y = node->left;

node->left = y->right;

if (y->right != nullptr) y->right->parent = node;

y->parent = node->parent;

if (node->parent == nullptr) {

root = y;

}

else if (node == node->parent->right) {

node->parent->right = y;

}

else {

node->parent->left = y;

}

y->right = node;

node->parent = y;

}

rb\_node\* search(rb\_node\* root, int value) {

if (root == nullptr || root->data == value) {

return root;

}

if (value < root->data) {

return search(root->left, value);

}

return search(root->right, value);

}

bool search\_val(int num) {

return search(root, num) != nullptr;

}

void in\_order() {

in\_order\_help(root);

cout << endl;

}

void in\_order\_help(rb\_node\* node) {

if (node == nullptr) return;

in\_order\_help(node->left);

cout << node->data << " ";

in\_order\_help(node->right);

}

void pre\_order(rb\_node\* node) {

if (node == nullptr) return;

cout << node->data << " ";

pre\_order(node->left);

pre\_order(node->right);

}

void post\_order(rb\_node\* node) {

if (node == nullptr) return;

post\_order(node->left);

post\_order(node->right);

cout << node->data << " ";

}

void level\_order() {

if (root == nullptr) return;

rb\_node\* queue[100];

int front = 0;

int rear = 0;

queue[rear++] = root;

while (front < rear) {

rb\_node\* node = queue[front++];

cout << node->data << " ";

if (node->left != nullptr) {

queue[rear++] = node->left;

}

if (node->right != nullptr) {

queue[rear++] = node->right;

}

}

cout << endl;

}

void del\_val(int value) {

rb\_node\* node\_del = search(root, value);

if (node\_del) {

deleteNode(node\_del);

}

}

void deleteNode(rb\_node\* node) {

rb\_node\* y = node, \* x;

Color or\_col = y->color;

if (node->left == nullptr) {

x = node->right;

rep(node, node->right);

}

else if (node->right == nullptr) {

x = node->left;

rep(node, node->left);

}

else {

y = minimum(node->right);

or\_col = y->color;

x = y->right;

if (y->parent == node) {

if (x) x->parent = y;

}

else {

rep(y, y->right);

y->right = node->right;

if (y->right) y->right->parent = y;

}

rep(node, y);

y->left = node->left;

if (y->left) y->left->parent = y;

y->color = node->color;

}

delete node;

if (or\_col == BLACK) {

if (x != nullptr) {

fix\_del(x);

}

}

}

void rep(rb\_node\* u, rb\_node\* v) {

if (u->parent == nullptr) {

root = v;

}

else if (u == u->parent->left) {

u->parent->left = v;

}

else {

u->parent->right = v;

}

if (v != nullptr) {

v->parent = u->parent;

}

}

rb\_node\* minimum(rb\_node\* node) {

while (node->left != nullptr) {

node = node->left;

}

return node;

}

void fix\_del(rb\_node\* x) {

while (x != root && (x == nullptr || x->color == BLACK)) {

if (x == x->parent->left) {

rb\_node\* w = x->parent->right;

if (w->color == RED) {

w->color = BLACK;

x->parent->color = RED;

rot\_left(x->parent);

w = x->parent->right;

}

if ((w->left == nullptr || w->left->color == BLACK) &&

(w->right == nullptr || w->right->color == BLACK)) {

w->color = RED;

x = x->parent;

}

else {

if (w->right == nullptr || w->right->color == BLACK) {

if (w->left != nullptr) w->left->color = BLACK;

w->color = RED;

rot\_right(w);

w = x->parent->right;

}

w->color = x->parent->color;

x->parent->color = BLACK;

if (w->right != nullptr) w->right->color = BLACK;

rot\_left(x->parent);

x = root;

}

}

else {

rb\_node\* w = x->parent->left;

if (w->color == RED) {

w->color = BLACK;

x->parent->color = RED;

rot\_right(x->parent);

w = x->parent->left;

}

if ((w->right == nullptr || w->right->color == BLACK) &&

(w->left == nullptr || w->left->color == BLACK)) {

w->color = RED;

x = x->parent;

}

else {

if (w->left == nullptr || w->left->color == BLACK) {

if (w->right != nullptr) w->right->color = BLACK;

w->color = RED;

rot\_left(w);

w = x->parent->left;

}

w->color = x->parent->color;

x->parent->color = BLACK;

if (w->left != nullptr) w->left->color = BLACK;

rot\_right(x->parent);

x = root;

}

}

}

if (x != nullptr) x->color = BLACK;

}

};

int main() {

setlocale(LC\_ALL, "Russian");

rb\_tree tree;

tree.insert(7);

tree.insert(3);

tree.insert(18);

tree.insert(10);

tree.insert(22);

tree.insert(8);

tree.insert(11);

tree.insert(26);

tree.insert(2);

tree.insert(6);

tree.insert(13);

cout << "Центрированный обход: ";

tree.in\_order();

tree.del\_val(18);

tree.del\_val(13);

tree.del\_val(3);

tree.del\_val(8);

tree.del\_val(22);

cout << "Дерево после удаления 18, 13, 3, 8, 22: ";

tree.in\_order();

tree.insert(16);

tree.insert(4);

cout << "Дерево после вставки 16 и 4: ";

tree.in\_order();

cout << "Есть ли 13 в дереве?" << (tree.search\_val(13) ? " Да" : " Нет") << endl;

cout << "Есть ли 16 в дереве?" << (tree.search\_val(16) ? " Да" : " Нет") << endl;

cout << "Есть ли 7 в дереве?" << (tree.search\_val(7) ? " Да" : " Нет") << endl;

cout << "Есть ли 100 в дереве?" << (tree.search\_val(100) ? " Да" : " Нет") << endl;

return 0;

}