Санкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материалов и транспорта
Высшая школа машиностроения

Курсовая работа

по дисциплине «Объектно-ориентированное программирование»

Вариант 16

Тема: алгоритм оценки выражений AVL-дерево

Выполнил
Студент Прокофьев А.А.
гр. 3331506/90401 *(подпись)*
Работу принял
Ананьевский М.С.
(подпись)

Санкт-Петербург
2022 г.

Оглавление

1. Введение

Бинарное дерево — это иерархическая структура данных, в которой каждый узел имеет значение (оно же является в данном случае и ключом) и ссылки на левого и правого потомка (см. рис. 1). Узел, находящийся на самом верхнем уровне (не являющийся чьим-либо потомком) называется корнем. Узлы, не имеющие потомков (оба потомка которых равны NULL) называются листьями.

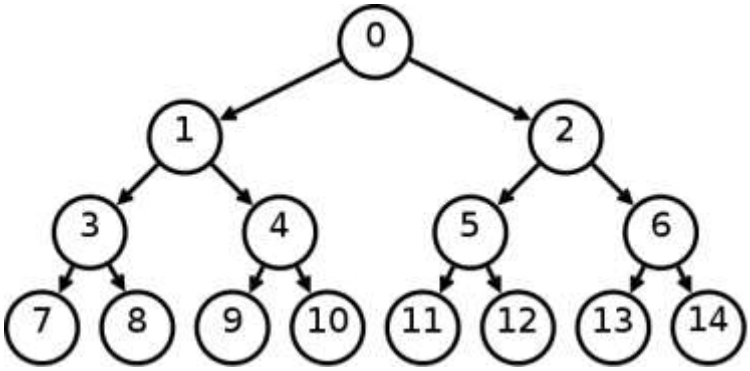

Рисунок 1 – Бинарное дерево

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 (см. рис. 2). То есть, данные в бинарном дереве поиска хранятся в отсортированном виде. При каждой операции вставки нового или удаления существующего узла отсортированный порядок дерева сохраняется. При поиске элемента сравнивается искомое значение с корнем. Если искомое больше корня, то поиск продолжается в правом потомке корня, если меньше, то в левом, если равно, то значение найдено и поиск прекращается.

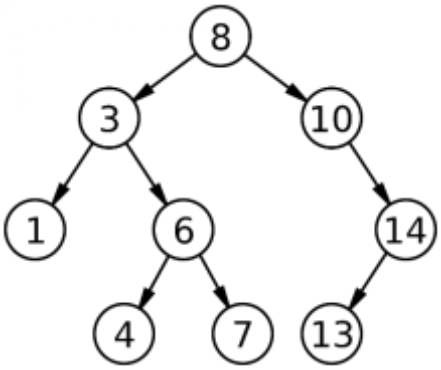

Рисунок 2 – Бинарное дерево поиска

Сбалансированное бинарное дерево поиска — это бинарное дерево поиска с логарифмической высотой (см. рис. 3). Данное определение скорее идейное, чем строгое. Строгое определение оперирует разницей глубины самого глубокого и самого неглубокого листа (в AVL-деревьях) или отношением глубины самого глубокого и самого неглубокого листа (в красно-черных деревьях). В сбалансированном бинарном дереве поиска операции поиска, вставки и удаления выполняются за логарифмическое время (так как путь к любому листу от корня не более логарифма). В вырожденном случае несбалансированного бинарного дерева поиска, например, когда в пустое дерево вставлялась отсортированная последовательность, дерево превратится в линейный список (см. рис. 3), и операции поиска, вставки и удаления будут выполняться за линейное время. Поэтому балансировка дерева крайне важна. Технически балансировка осуществляется поворотами частей дерева при вставке нового элемента, если вставка данного элемента нарушила условие сбалансированности.

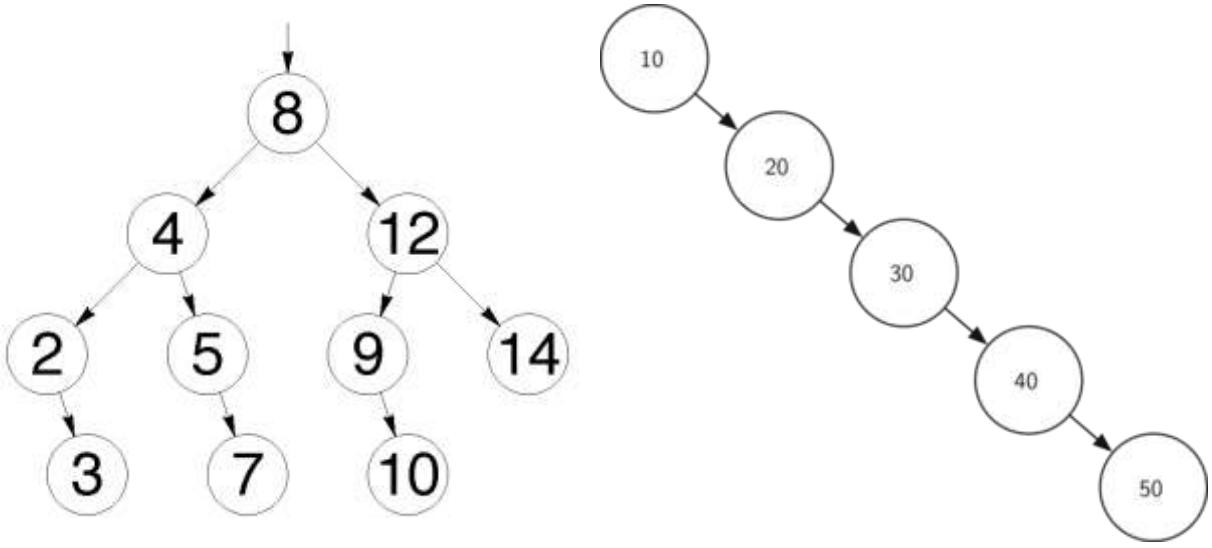

Рисунок 3 – Сбалансированное и несбалансированное бинарное дерево поиска

Сбалансированное бинарное дерево поиска применяется, когда необходимо осуществлять быстрый поиск элементов, чередующийся со вставками новых элементов и удалениями существующих. В случае, если набор элементов, хранящийся в структуре данных фиксирован и нет новых

вставок и удалений, то массив предпочтительнее. Потому что поиск можно осуществлять алгоритмом бинарного поиска за то же логарифмическое время, но отсутствуют дополнительные издержки по хранению и использованию указателей. Например, c++ ассоциативные контейнеры *set* и *map* представляют собой сбалансированное бинарное дерево поиска.

2. Описание алгоритма

Особенностью AVL-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 $\log_2(n + 1)$ до $1.44 \log_2(n + 2) - 0.328$. А 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. Если вставка или удаление элемента приводит к нарушению сбалансированности дерева, то необходимо выполнить его балансировку.

Традиционно, узлы АВЛ-дерева хранят не высоту, а разницу высот правого и левого поддеревьев (так называемый *balance factor*), которая может принимать только три значения -1, 0 и 1. Однако, заметим, что эта разница все равно хранится в переменной, размер которой равен минимум одному байту (если не придумывать каких-то хитрых схем «эффективной» упаковки таких величин). Вспомним, что высота $h < 1.44 \log_2(n + 2)$, это значит, например, что при $n=10^9$ (один миллиард ключей, больше 10 гигабайт памяти под хранение узлов) высота дерева не превысит величины h=44, которая с успехом помещается в тот же один байт памяти, что и *balance factor*. Таким образом, хранение высот с одной стороны не увеличивает объем памяти, отводимой под

узлы дерева, а с другой стороны существенно упрощает реализацию некоторых операций.

Балансировка. В процессе добавления или удаления узлов в АВЛ-дереве возможно возникновение ситуации, когда *balance factor* некоторых узлов оказывается равными 2 или -2, т.е. возникает *расбалансировка* поддерева. Для выправления ситуации применяются повороты вокруг тех или иных узлов дерева. Простой поворот вправо (влево) производит трансформацию дерева, показанную на рис. 4.

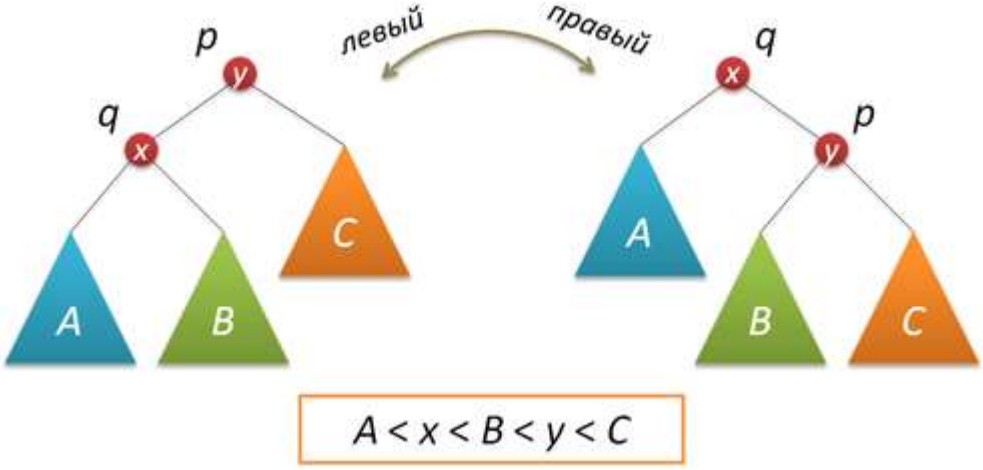

Рисунок 4 – Поворот вокруг узла дерева

Рассмотрим теперь ситуацию дисбаланса, когда высота правого поддерева узла p на 2 больше высоты левого поддерева (обратный случай является симметричным и реализуется аналогично) см. рис. 5. Пусть q — правый дочерний узел узла p, а s — левый дочерний узел узла q.

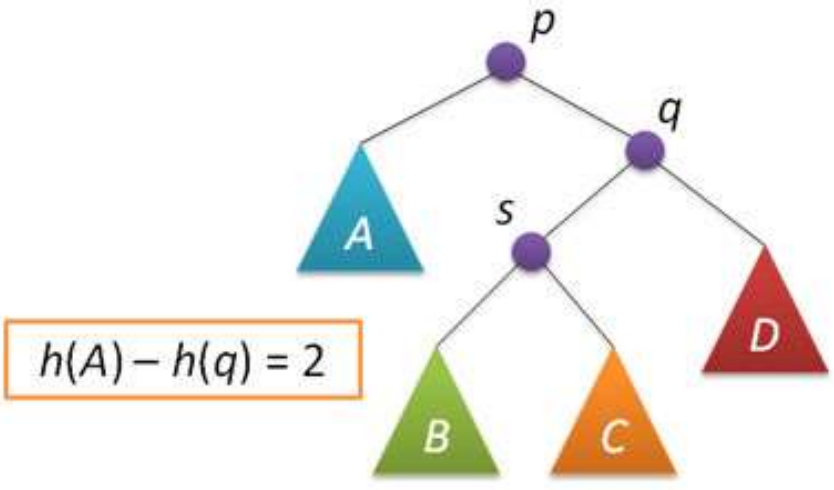

Рисунок 5 – Дисбаланс

Анализ возможных случаев в рамках данной ситуации показывает, что для исправления разбалансировки в узле p достаточно выполнить либо простой поворот влево вокруг p, либо так называемый *большой поворот* влево вокруг того же p. Простой поворот выполняется при условии, что высота левого поддерева узла q больше высоты его правого поддерева: h(s)≤h(D) (см. рис 6).

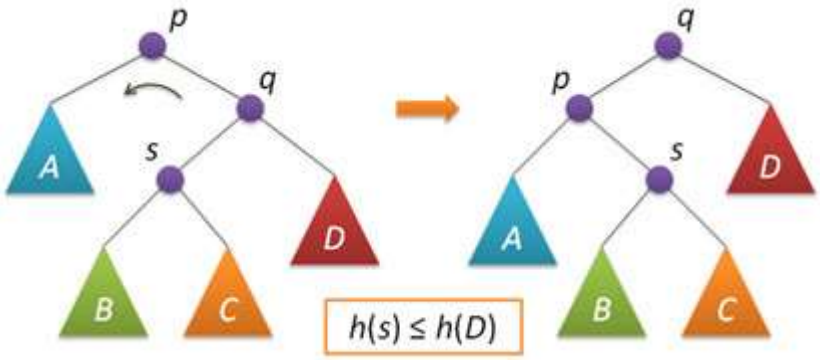

Рисунок 6 – Простой поворот

Большой поворот применяется при условии h(s)>h(D) и сводится в данном случае к двум простым — сначала правый поворот вокруг q и затем левый вокруг p (см. рис. 7).

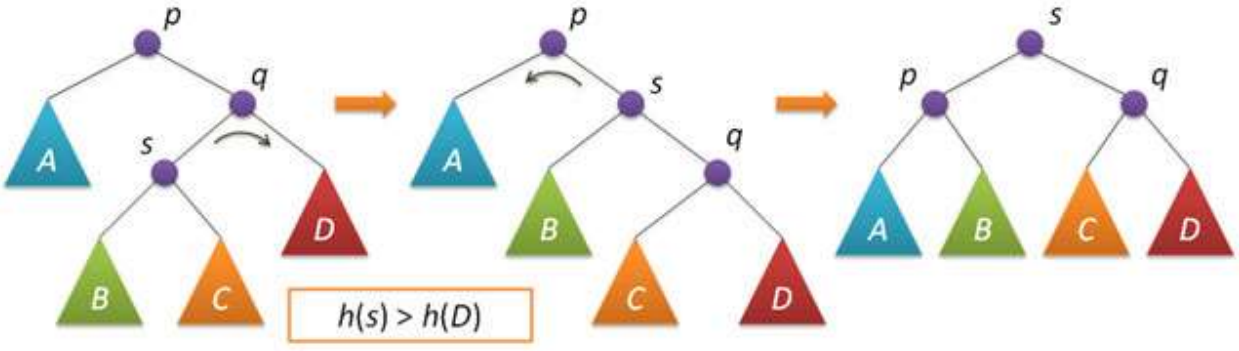

Рисунок 7 – Большой поворот

Вставка ключей. Вставка нового ключа в АВЛ-дерево выполняется, по большому счету, так же, как это делается в простых деревьях поиска: спускаемся вниз по дереву, выбирая правое или левое направление движения в зависимости от результата сравнения ключа в текущем узле и вставляемого ключа. Единственное отличие заключается в том, что при возвращении из рекурсии (т.е. после того, как ключ вставлен либо в правое, либо в левое поддерево, и это дерево сбалансировано) выполняется балансировка текущего

узла. Строго доказывается, что возникающий при такой вставке дисбаланс в любом узле по пути движения не превышает двух, а значит применение вышеописанной функции балансировки является корректным.

Удаление ключей. Находим узел p с заданным ключом k (если не находим, то делать ничего не надо), в правом поддереве находим узел min с наименьшим ключом и заменяем удаляемый узел p на найденный узел min (см. рис. 8).

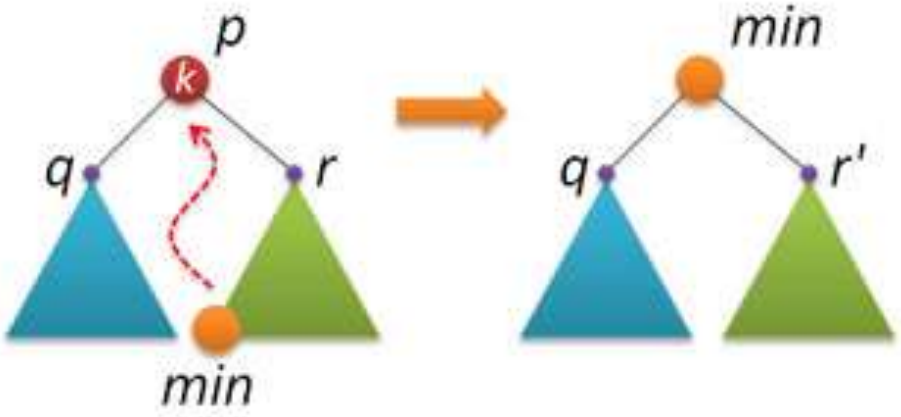

Рисунок 8 – Удаление ключей

При реализации возникает несколько нюансов. Прежде всего, если у найденный узел p не имеет правого поддерева, то по свойству АВЛ-дерева слева у этого узла может быть только один единственный дочерний узел (дерево высоты 1), либо узел p вообще лист. В обоих этих случаях надо просто удалить узел p и вернуть в качестве результата указатель на левый дочерний узел узла p.

Поиск. Поиск выполняется так же как в деревьях поиска. Для каждого узла сравниваем значение его ключа с искомым ключом. Если ключи одинаковы, то функция возвращает текущий узел, в противном случае функция вызывается рекурсивно для левого или правого поддерева. Сложность по времени O(log(n)) по памяти O(n).

3. Исследование алгоритма

Сложность алгоритма AVL-дерева в различных компонентах работы в O-символике представлена на рис. 9.

	В среднем	В худшем случае
Расход памяти	O(n)	O(n)
Поиск	O(log n)	O(log n)
Вставка	O(log n)	O(log n)
Удаление	O(log n)	O(log n)

Рисунок 9 – Сложность алгоритма AVL-дерева

Чтобы проверить соответствие реализованного алгоритма теоретическим оценкам для высоты АВЛ-деревьев, был построен график, представленный на рис. 10. Генерировался массив из случайно расположенных чисел от 1 до 10000, далее эти числа последовательно вставлялись в изначально пустое АВЛ-дерево и измерялась высота дерева после каждой вставки. Полученные результаты были усреднены по 1000 расчетам. На следующем графике показана зависимость от n средней высоты (красная линия); минимальной высоты (зеленая линия); максимальной высоты (синяя линия). Кроме того, показаны верхняя и нижняя теоретические оценки.

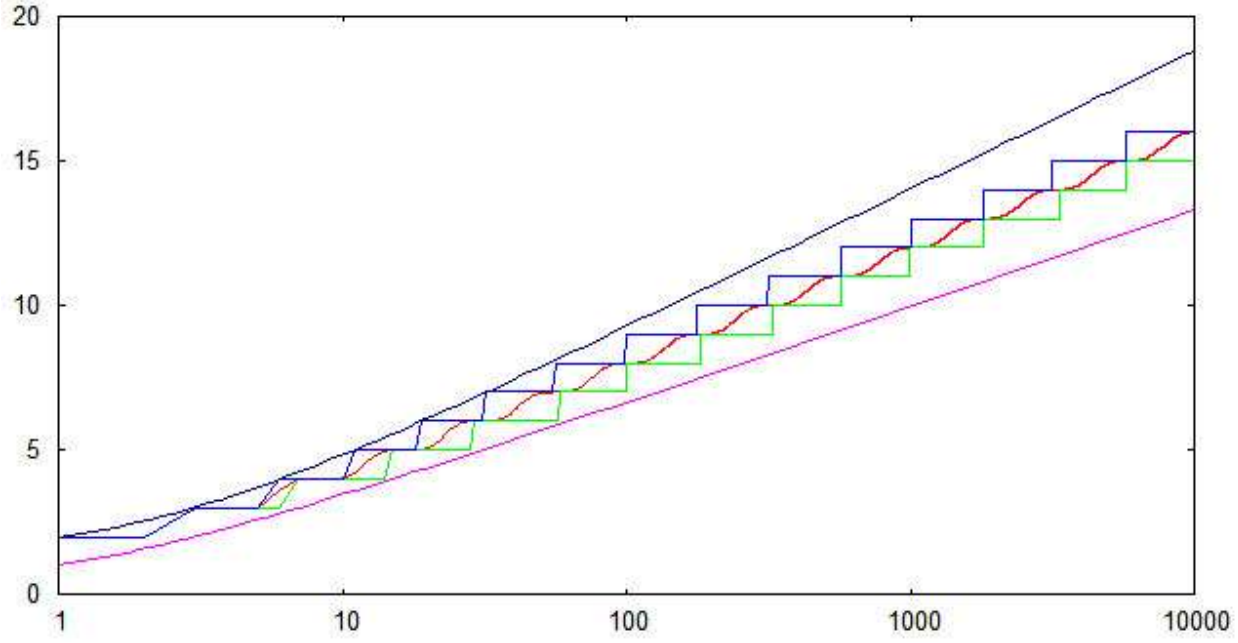

Рисунок 10 – Зависимость высоты дерева от количества элементов

Протестируем время выполнения программы с помощью функции *clock*. Результаты представлены на графиках. Рис. 11 соответствует результатам

работы программы при вводе чисел сгенерированных с помощью функции rand.

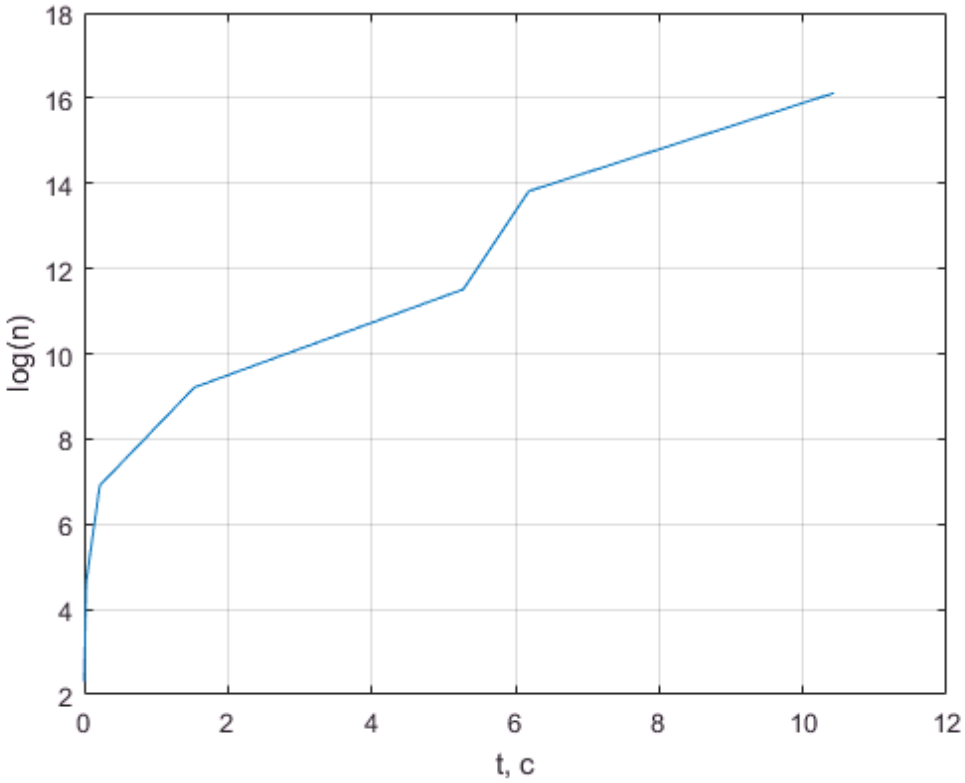

Рисунок 11 – Результат работы алгоритма при последовательном увеличении чисел

Данные, используемые в графике с рис. 11, приведены в табл. 1.

Таблица 1 – Данные для построения графика

Время в секундах	№ опыта	Количество узлов						
		10	100	1000	10000	10^5	10^6	10^7
	1	0.004	0.034	0.255	1.732	5.376	6.066	10.529
	2	0.003	0.024	0.200	1.450	5.210	6.005	10.433
	3	0.002	0.020	0.226	1.444	5.265	5.972	10.388
	4	0.006	0.035	0.194	1.591	5.253	6.269	10.403
	5	0.001	0.023	0.217	1.455	5.281	6.623	10.393
	медиана	0.0032	0.0272	0.2184	1.5344	5.277	6.187	10.4292

Исходя из графика можно сделать вывод, что скорость и время выполнения кода соответствуют теоретическим выкладкам. Реализация кода представлена в приложении 1.

4. Заключение

В работе был рассмотрен алгоритм AVL-дерева, позволяющий осуществлять быстрый поиск элементов.

AVL-деревья обеспечивают более быстрый поиск, чем *Red Black Trees*, потому что они более строго сбалансированы. Красно-чёрные деревья используются в большинстве языковых библиотек, таких как *map, multimap, multiset* в C ++, тогда как деревья AVL используются в базах данных, где требуется более быстрый поиск. Красно-черные деревья более общего назначения. Они относительно хорошо справляются с добавлением, удалением и поиском, но деревья AVL имеют более быстрый поиск за счет более медленного добавления / удаления.

5. Список литературы

1. АВЛ-деревья, Хабр [сайт] URL: https://habr.com/ru/post/150732/.
2. А.В. Никитин, Т.Н. Ничушкина Иерархические структуры данных: бинарные деревья – 2018.
3. О.В. Сенюкова, Сбалансированные деревья поиска – 2014, стр. 17-36.

Приложение 1

Полный код реализации алгоритма

```
#ifndef AVL_TREE_H
#define AVL_TREE_H

// An AVL tree node
class Node {
public:
    int key;
    int height;
    Node * left;
    Node * right;

public:
    int get_height(Node *N);            // Get the height of the tree
    int max(int a, int b);              // Get maximum of two numbers
    int getBalance(Node *N);            // Get balance factor of node N
    void preOrder(Node *root);          /* Print preliminary traversal of the
tree.
                                        The function also prints height of
every node.*/
    Node* newNode(int key);             /* Allocates a new node with the given
                                        key and NULL left and right
pointers.*/
    Node *rightRotate(Node *y);         // Right rotate subtree rooted with y.
    Node *leftRotate(Node *x);          // A utility function to left rotate
subtree rooted with x.
    Node* insert(Node* node, int key); /* Recursive function to insert a key
in the subtree rooted
                                        with node and returns the new root of
the subtree.*/
};

#endif //AVL_TREE_H
#include "avl_tree.h"
#include<bits/stdc++.h>

int Node:: get_height(Node *N) {
    if (N == NULL) {
        return 0;
    }
    return N->height;
}

int Node::max(int a, int b) {
    return (a > b)? a : b;
}

Node* Node::newNode(int key) {
    Node* node = new Node();
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1; // The new node is initially added at the end (at leaf)
    return(node);
}

Node *Node::rightRotate(Node *y) {
    Node *x = y->left;
    Node *T2 = x->right;
```

```
    // Rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(get_height(y->left),
                    get_height(y->right)) + 1;
    x->height = max(get_height(x->left),
                    get_height(x->right)) + 1;

    // Return new root
    return x;
}

Node *Node::leftRotate(Node *x) {
    Node *y = x->right;
    Node *T2 = y->left;

    // Rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(get_height(x->left),
                    get_height(x->right)) + 1;
    y->height = max(get_height(y->left),
                    get_height(y->right)) + 1;

    // Return new root
    return y;
}

int Node::getBalance(Node *N) {
    if (N == NULL)
        return 0;
    return get_height(N->left) - get_height(N->right);
}

Node* Node::insert(Node* node, int key) {
    /* 1. Insertion */
    if (node == NULL) {
        return (newNode(key));
    }
    if (key < node->key) {
        node->left = insert(node->left, key);
    } else if (key > node->key) {
        node->right = insert(node->right, key);
    } else {    // Equal keys are not allowed in binary search trees
        return node;
    }
    /* 2. Update height of this previous node */
    node->height = 1 + max(get_height(node->left),
                           get_height(node->right));

    /* 3. Get the balance factor of this previous
        node to check whether this node became unbalanced */
    int balance = getBalance(node);

    /* If this node becomes unbalanced, then there are 4 cases:
       1. Left Left Case*/
    if (balance > 1 && key < node->left->key) {
        return rightRotate(node);
    }
```

```
    // 2. Right Right Case
    if (balance < -1 && key > node->right->key) {
        return leftRotate(node);
    }
    // 3. Left Right Case
    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
    // 4. Right Left Case
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    /* If node is balanced return the (unchanged) node pointer */
    return node;
}

void Node::preOrder(Node *root) {
    if(root != NULL) {
        std::cout << root->key << " ";
        preOrder(root->left);
        preOrder(root->right);
    }
}
#include "avl_tree.h"
#include<bits/stdc++.h>

int main() {
    Node New_Node;
    Node *root = NULL;

    /* Building tree by adding values */
    root = New_Node.insert(root, 10);
    root = New_Node.insert(root, 20);
    root = New_Node.insert(root, 30);
    root = New_Node.insert(root, 40);
    root = New_Node.insert(root, 50);
    root = New_Node.insert(root, 25);
    root = New_Node.insert(root, 60);
    root = New_Node.insert(root, 70);
    root = New_Node.insert(root, 70);
    root = New_Node.insert(root, 45);
    root = New_Node.insert(root, 75);
    root = New_Node.insert(root, 55);
    root = New_Node.insert(root, 90);
    root = New_Node.insert(root, 15);
    root = New_Node.insert(root, 80);
    root = New_Node.insert(root, 5);
    root = New_Node.insert(root, 65);
    /* The AVL Tree structure will be look like:
                  _____50_____
                 /            \
               30             70
             /    \         /    \
           20      40      60     80
          /   \      \    /  \   /  \
        10   25      45  55  65 75  90
       /  \
      5   15
    */
    std::cout << "The preliminary traversal of the "
```

```
                "constructed AVL tree looks like: \n";
    New_Node.preOrder(root);

    return 0;
}
```