Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Российский химико-технологический университет имени Д.И. Менделеева»

Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 3

Выполнил студент группы КС-30 (Ноль Эльвира Гарриевна)

Ссылка на репозиторий: (https://github.com/Elviranng/Nol-Elvira-KS-30/tree/master/lab3.sem2)

Приняли: Пысин Максим Дмитриевич

Краснов Дмитрий Олегович

Дата сдачи: (05.06.21)

Оглавление

[Описание задачи. 2](#_Toc69330426)

[Описание модели. 3](#_Toc69330427)

[Выполнение задачи. 4](#_Toc69330428)

[Заключение. 13](#_Toc69330429)

# Описание задачи.

В рамках лабораторной работы необходимо изучить и реализовать структуру – AVL дерево.

AVL дерево: Класс, операцию вставки и удаления элемента с последующей балансировкой(операция удаления и вставки вызывает семеричные операции балансировки, поэтому в отличии от красно черного дерево реализовав балансировку для вставки, ее можно применить и для удаления) и операцию поиска.

Для всех трех структур провести тестирование сложности операций вставки и поиска.

Для этого использовать серию тестов:

Под отдельным тестом понимается проведение 30 запусков тестирования для массива, состоящего из N элементов, где N меняется от теста к тесту на 50000, первый тест при это проводиться при N=50000, последний N=500000

В каждом тесте выполнять 20 запусков генерирования и заполнения массива.

Массив заполнять числами с плавающей запятой двойной точности сгенерированных в диапазоне от -1 до 1.

5 из 20 запусков выполнять для предварительно отсортированного массива (сортировку можно взять из стандартной библиотеки), и выделять их отдельно при анализе результатов.

Для каждого запуска проводить 100 операций поиска, замеряя время, которое тратиться на эту операцию, по итогу получая среднее значение времени поиска для каждого из запусков. (Искать случайно сгенерированное число)

Для каждого запуска проводить 100 операций вставки в текущее дерево, замеряя время, которое тратиться на эту операцию (замерять нужно именно отдельную от первичного заполнения операцию вставки), по итогу должно получиться среднее время вставки для каждого запуска (вставлять случайно сгенерированное число)

Аналогичное тестирование можно провести для операции удаления, но это уже исключительно дополнительное задание.

По результатам проведенных тестов необходимо построить графики зависимости затраченного на операцию (вставка, поиск) времени от количества элементов сгенерированного дерева для лучшего, худшего и среднего случаев.

Для каждого из вариантов асимптотическая сложность операций равно O(h), где h это высота дерева, так:

Для АВЛ h ~ log(N), то и асимптотическая сложность будет O(c\*log(N)) Необходимо взять график худшего случая и построить график O(c \* f(N)), подобрав такую константу c, что бы полученный график был ненамного выше графика худшего случая начиная с первой рассматриваемой точки в N=50000

# Описание модели.

Двоичное дерево – это иерархическая структура, в которой каждый узел содержит не более чем двух потомков.

Для каждого узла, тот узел, который стоит выше по иерархии для него называют родительским узлом, а те узлы, что стоят ниже, для которых этот узел является родительским, называются правым и левым наследниками.

Двоичное дерево поиска – это двоичное дерево придерживающееся 2х правил, согласно которым левые потомки всегда меньше или равны текущего элемента, а правые больше или равны текущего элемента.

Поисковые деревья — это решения так называемой «словарной проблемы». Предположим, что имеется большое количество ключей, каждый из которых имеет значение. В немецко–английском словаре немецкое слово является ключевым, а английские слова являются значением, которое вы ищете. Аналогично ведет себя телефонная книга с именем и адресом в качестве ключа, а номер телефона — в качестве искомого значения.

Этот подход получил в информатике название «двоичный поиск». Она воссоздана очевидным образом с помощью очень известного метода поиска «двоичный поиск в массиве». Их поведение оптимально с точки зрения информации, а именно логарифмически.

**АВЛ-дерево** — сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на 1.

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 1.44 log2(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. Напомним, что рандомизированные деревья поиска обеспечивают сбалансированность только в вероятностном смысле: вероятность получения сильно несбалансированного дерева при больших n хотя и является пренебрежимо малой, но остается не равной нулю.

# Выполнение задачи.

Полный код программы можно посмотреть в репозитории. Здесь же, будут подробно рассмотрены основные части программы.

Сперва, необходимо было создать шаблонный класс, чтобы поддерживать работу с разными типами данных. Как раз с этого момента и начинается реализация AVL-дерева.

// Узел

    class node{

    public:

        T key;

        int height;

        node\* left;

        node\* right;

        node(T k){

            height = 1;

            key = k;

            left = nullptr;

            right = nullptr;

        }

    };

Узлом дерева является объект внутреннего класса node, который обладает свойствами: значение, высота, указатель на левый потомок-узел и указателем на правый потомок-узел. Конструктор узла принимает в себя значение, которое будет храниться в узле.

Все методы класса AVLTree разделены на публичные и приватные. Так как эта реализация дерева содержит рекурсивные методы, то такое разделение имеет смысл – приватные методы способны рекурсивно вызывать сами себя для поиска/вставки/удаления из дерева, а публичные методы доступны извне класса, но в свою очередь вызывают приватные рекурсивные методы.

В классе дерева имеются два приватных свойства:

private:

    node\* root = nullptr;  // Главный узел - корень

    int amount = 0;  // Кол-во элементов дерева

Указатель на узел и кол-во узлов дерева соответственно. Они нужны для работы некоторых методов.

Описание класса начну с приватных (внутренних) методов.

**Получение высоты дерева:**

// Метод, возвращающий высоту узла

    int height(node\* curr\_node){

        if (curr\_node == nullptr)

            return 0;

        return curr\_node->height;

    }

Данный метод получает указатель на узел и возвращает оттуда значение его высоты. Можно использовать на поддеревьях, само собой. Если узел пуст, то возвращает 0.

**Правый поворот:**

// Правый поворот

    node\* rightRotation(node\* curr\_node){

        node\* new\_curr\_node = curr\_node->left;

        curr\_node->left = new\_curr\_node->right;

        new\_curr\_node->right = curr\_node;

        curr\_node->height = 1 + max(height(curr\_node->left), height(curr\_node->right));

        new\_curr\_node->height = 1 + max(height(new\_curr\_node->left), height(new\_curr\_node->right));

        return new\_curr\_node;

    }

Метод принимает на вход указатель на узел, относительно которого надо совершить правый поворот, после совершает алгоритм поворота и возвращает обновленный узел.

**Левый поворот:**

// Левый поворот

    node\* leftRotation(node\* curr\_node){

        node\* new\_curr\_node = curr\_node->right;

        curr\_node->right = new\_curr\_node->left;

        new\_curr\_node->left = curr\_node;

        curr\_node->height = 1 + max(height(curr\_node->left), height(curr\_node->right));

        new\_curr\_node->height = 1 + max(height(new\_curr\_node->left), height(new\_curr\_node->right));

        return new\_curr\_node;

    }

Метод принимает на вход указатель на узел, относительно которого надо совершить левый поворот, после совершает алгоритм поворота и возвращает обновленный узел.

**Вывод дерева в консоль:**

// Внутрений метод вывода в консоль элементов (рекурсивный)

    void inorderPrivate(node\* curr\_node){

        if (curr\_node == nullptr)

            return;

        inorderPrivate(curr\_node->left);

        cout << curr\_node->key << " ";

        inorderPrivate(curr\_node->right);

    }

Рекурсивно проходим по всем узлам дерева по порядку и выводим их все в том же порядке в консоль. Удобно применять, чтобы проверить какие значения хранятся в дереве.

**Вставка в дерево:**

// Внутрений метод вставки (рекурсивный)

    node\* insertPrivate(node\* curr\_node, T x){

        // Проверка на то, дошли ли до места для вставки узла:

        if (curr\_node == nullptr){

            node\* temp = new node(x);

            amount++;

            return temp;

        }

        if (x < curr\_node->key)

            curr\_node->left = insertPrivate(curr\_node->left, x);

        else if (x > curr\_node->key)

            curr\_node->right = insertPrivate(curr\_node->right, x);

Один из 3-х основных методов – вставка элемента в дерево. Принимает в себя указатель на узел и значение, которое надо вставить. Метод довольно большой, поэтому рассмотрим его по частям. В первой части происходит проверка на то, нашли ли мы пустой (переданный в метод) узел. Если это так, это означает, что мы дошли до места вставки нового элемента, то есть, можно вставлять сюда новый узел. Если же мы ещё не добрались до пустого узла, то сравниваем значения дочерних узлов с текущим значением и уходим дальше от корня (рекурсивно).

// После вставки идет балансировка:

        curr\_node->height = 1 + max(height(curr\_node->left), height(curr\_node->right));

        int bal = height(curr\_node->left) - height(curr\_node->right);

        if (bal > 1){

            if (x < curr\_node->left->key){

                return rightRotation(curr\_node);

            }

            else{

                curr\_node->left = leftRotation(curr\_node->left);

                return rightRotation(curr\_node);

            }

        }

        else if (bal < -1){

            if (x > curr\_node->right->key){

                return leftRotation(curr\_node);

            }

            else{

                curr\_node->right = rightRotation(curr\_node->right);

                return leftRotation(curr\_node);

            }

        }

        return curr\_node;

    }

Как только мы вставили элемент, необходимо отбалансировать наше дерево, так как оно AVL, то есть, само балансируемое. Обновляем данные о высоте нашего узла. Если баланс дерева вышел за границы [-1;1], то начинаем процесс балансировки. Рассматриваем 4 возможных случая, и выполняем соответствующий алгоритм. Если баланс больше 1, то используем правый поворот, иначе – левый.

После всех операций возвращаем новый узел.

**Удаление из дерева:**

// Внутрений метод удаления узла (рекурсивный)

    node\* removePrivate(node\* curr\_node, T x){

        // Проверяем не уткнулись ли в пустой узел

        if (curr\_node == nullptr)

            return nullptr;

        if (x < curr\_node->key){

            curr\_node->left = removePrivate(curr\_node->left, x);

        }

        else if (x > curr\_node->key){

            curr\_node->right = removePrivate(curr\_node->right, x);

        }

Ещё один основной метод, который используется для рекурсивного удаления из дерева. Принимает те же параметры, что и предыдущий, и возвращает то же самое. Он ещё больше, чем метод вставки, поэтому снова разобьем его на части.

Во-первых, сначала проверяется не дошли ли мы до пустого узла (в случае, если элемента с таким значением нету в дереве). Если всё нормально, то сравниваем искомый ключ с ключами дочерних узлов, чтобы дальше рекурсивно искать нужный узел.

else{

            // Если нашли нужный узел, то удаляем значения из него:

            node\* r = curr\_node->right;

            if (curr\_node->right == nullptr){

                node\* l = curr\_node->left;

                delete (curr\_node);

                curr\_node = l;

                amount--;

            }

            else if (curr\_node->left == nullptr){

                delete (curr\_node);

                curr\_node = r;

                amount--;

            }

            else{

                while (r->left != nullptr)

                    r = r->left;

                curr\_node->key = r->key;

                curr\_node->right = removePrivate(curr\_node->right, r->key);

            }

        }

        if (curr\_node == nullptr){

            return curr\_node;

        }

Если ни одно из условий не выполнилось, значит текущий элемент – искомый, в этом случае начинаем само удаление. В зависимости от того, какие потомки были у удаленного узла, ставим этого потомка вместо данного узла. Если же у узла оба потомка имеются, то находим самый дальний левый потомок у правого потомка удаляемого узла, и удаляем его, перемещая его ключ в удаляемый узел.

Если новый узел после удаления является пустым, то выходим из метода (т.к. нечего балансировать), иначе – балансируем.

// Балансировка после удаления:

        curr\_node->height = 1 + max(height(curr\_node->left), height(curr\_node->right));

        int bal = height(curr\_node->left) - height(curr\_node->right);

        if (bal > 1){

            if (height(curr\_node->left) >= height(curr\_node->right)){

                return rightRotation(curr\_node);

            }

            else{

                curr\_node->left = leftRotation(curr\_node->left);

                return rightRotation(curr\_node);

            }

        }

        else if (bal < -1){

            if (height(curr\_node->right) >= height(curr\_node->left)){

                return leftRotation(curr\_node);

            }

            else{

                curr\_node->right = rightRotation(curr\_node->right);

                return leftRotation(curr\_node);

            }

        }

        return curr\_node;

    }

Аналогичная балансировка, как и после вставки, т.к. это AVL дерево. В итоге, возвращаем новый узел.

**Поиск в дереве:**

// Внутрений метод поиска узла (рекурсивный)

    node\* findPrivate(node\* curr\_node, T x){

        if (curr\_node == nullptr)

            return nullptr;

        T k = curr\_node->key;

        if (k == x)

            return curr\_node;

        if (k > x)

            return findPrivate(curr\_node->left, x);

        if (k < x)

            return findPrivate(curr\_node->right, x);

    }

Метод поиска по дереву так же выполнен рекурсивно. Здесь просто пробегаемся по дереву, пока не найдем нужный элемент (рекурсивно). Если нашли искомый элемент, то возвращаем его. Соответственно, если не нашли, то вернем нулевой указатель.

Помимо всех выше указанных приватных методов, так же имеются публичные для работы с деревом. Они вызывают рекурсивные приватные методы, и принимают в себя по одному (или не принимают) аргументу:

// Метод подсчета элементов

    int count(){

        return amount;

    }

    // Метод добавления узла

    void insert(T x){

        root = insertPrivate(root, x);

    }

    // Метод удаления узла

    void remove(T x){

        root = removePrivate(root, x);

    }

    // Метод поиска в дереве

    node\* find(T x){

        return findPrivate(root, x);

    }

    // Метод вывода дерева в консоль по порядку

    void inorder(){

        inorderPrivate(root);

        cout << endl;

    }

    // Метод получения узла-корня

    node\* getRoot(){

        return root;

    }

Так же класс АВЛ дерева имеет конструктор по умолчанию и деструктор, который рекурсивно удаляет все элементы, пока дерево не станет пустым. Утечек памяти не допускается:

// Конструктор по-умолчанию

    AVLTree(){}

    // Деструктор

    ~AVLTree(){

        // Удаляем корень, пока корень не станет пустым

        while(this->root){

            remove(this->root->key);

        }

    }

Так же в лабораторной работе имеются тесты данной структуры, которые замеряют скорость работы методов вставки и поиска данной реализации AVL дерева. Их результаты приведены далее.

Данные по времени на вставку 100 элементов (миллисекунд)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **50000** | **100000** | **150000** | **200000** | **250000** | **300000** | **350000** | **400000** | **450000** | **500000** |
| 0,157667 | 0,241613 | 0,227074 | 0,266274 | 0,264775 | 0,326902 | 0,303981 | 0,314032 | 0,329361 | 0,305316 |
| 0,165887 | 0,2228 | 0,247988 | 0,248333 | 0,26661 | 0,282109 | 0,301044 | 0,276283 | 0,319196 | 0,307015 |
| 0,156976 | 0,195722 | 0,240845 | 0,261524 | 0,270294 | 0,325012 | 0,283452 | 0,298933 | 0,319496 | 0,318449 |
| 0,158386 | 0,198704 | 0,249826 | 0,237481 | 0,273093 | 0,286259 | 0,29998 | 0,340502 | 0,316081 | 0,306908 |
| 0,167029 | 0,21062 | 0,249389 | 0,240645 | 0,274768 | 0,297415 | 0,291575 | 0,314331 | 0,301109 | 0,324598 |
| 0,11168 | 0,144007 | 0,166604 | 0,192368 | 0,184359 | 0,231089 | 0,220199 | 0,189977 | 0,209341 | 0,205765 |
| 0,121804 | 0,136587 | 0,169128 | 0,25764 | 0,181321 | 0,196107 | 0,207505 | 0,199363 | 0,218735 | 0,216927 |
| 0,112726 | 0,159892 | 0,161428 | 0,157736 | 0,180801 | 0,172323 | 0,227509 | 0,199137 | 0,229741 | 0,192884 |
| 0,127083 | 0,148075 | 0,167623 | 0,166074 | 0,193529 | 0,185354 | 0,191179 | 0,201913 | 0,19672 | 0,239429 |
| 0,139014 | 0,169036 | 0,166006 | 0,163124 | 0,182395 | 0,191148 | 0,21414 | 0,220352 | 0,206229 | 0,224977 |
| 0,138042 | 0,133269 | 0,216644 | 0,158425 | 0,166156 | 0,178199 | 0,190165 | 0,200598 | 0,20642 | 0,214094 |
| 0,107765 | 0,153566 | 0,184476 | 0,177151 | 0,188177 | 0,213216 | 0,1985 | 0,206665 | 0,202291 | 0,21874 |
| 0,145892 | 0,148966 | 0,14838 | 0,205821 | 0,186741 | 0,1743 | 0,20391 | 0,193064 | 0,203088 | 0,21539 |
| 0,111662 | 0,156629 | 0,154686 | 0,164976 | 0,188734 | 0,188444 | 0,190954 | 0,204244 | 0,213743 | 0,222676 |
| 0,104586 | 0,137907 | 0,157816 | 0,163735 | 0,175433 | 0,184229 | 0,197808 | 0,220834 | 0,208847 | 0,235945 |
| 0,106809 | 0,148171 | 0,157146 | 0,166743 | 0,194633 | 0,178578 | 0,221159 | 0,206728 | 0,211939 | 0,215973 |
| 0,112236 | 0,137785 | 0,161854 | 0,159542 | 0,182983 | 0,204509 | 0,211657 | 0,203178 | 0,234607 | 0,22903 |
| 0,120667 | 0,144898 | 0,151118 | 0,179491 | 0,182996 | 0,195592 | 0,210934 | 0,188954 | 0,221897 | 0,211152 |
| 0,115481 | 0,142972 | 0,160283 | 0,178865 | 0,179053 | 0,183974 | 0,18995 | 0,218808 | 0,20472 | 0,239628 |
| 0,110296 | 0,152327 | 0,169521 | 0,19036 | 0,256847 | 0,180906 | 0,186745 | 0,206957 | 0,212193 | 0,226604 |

Данные по времени на поиск 100 элементов (миллисекунд)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **50000** | **100000** | **150000** | **200000** | **250000** | **300000** | **350000** | **400000** | **450000** | **500000** |
| 0,137119 | 0,201784 | 0,207082 | 0,213983 | 0,233701 | 0,23354 | 0,244964 | 0,273497 | 0,247296 | 0,258212 |
| 0,142214 | 0,194093 | 0,209236 | 0,235888 | 0,229346 | 0,270831 | 0,242849 | 0,24293 | 0,239333 | 0,270447 |
| 0,129744 | 0,179802 | 0,201155 | 0,221399 | 0,232723 | 0,247898 | 0,250474 | 0,250293 | 0,239315 | 0,295557 |
| 0,134811 | 0,192975 | 0,204485 | 0,201821 | 0,231103 | 0,256063 | 0,245997 | 0,249831 | 0,269335 | 0,266465 |
| 0,148487 | 0,181252 | 0,194736 | 0,218558 | 0,23377 | 0,247141 | 0,245293 | 0,255262 | 0,2458 | 0,255301 |
| 0,073488 | 0,085185 | 0,1257 | 0,120571 | 0,10538 | 0,142358 | 0,131013 | 0,135339 | 0,131144 | 0,140206 |
| 0,067199 | 0,078825 | 0,107131 | 0,155648 | 0,102033 | 0,127023 | 0,121193 | 0,118597 | 0,135933 | 0,138748 |
| 0,085273 | 0,084469 | 0,094673 | 0,114586 | 0,108186 | 0,119841 | 0,137289 | 0,125359 | 0,165158 | 0,120389 |
| 0,066731 | 0,100435 | 0,100481 | 0,09876 | 0,108421 | 0,118508 | 0,125624 | 0,138301 | 0,124022 | 0,150318 |
| 0,159386 | 0,109579 | 0,098488 | 0,116105 | 0,127889 | 0,122209 | 0,12964 | 0,127589 | 0,125441 | 0,141976 |
| 0,065194 | 0,08533 | 0,118807 | 0,106459 | 0,103954 | 0,123379 | 0,125444 | 0,125378 | 0,124804 | 0,147631 |
| 0,064425 | 0,08686 | 0,098715 | 0,103902 | 0,115882 | 0,126815 | 0,118372 | 0,132137 | 0,133381 | 0,150495 |
| 0,063921 | 0,099683 | 0,096958 | 0,126203 | 0,122749 | 0,115578 | 0,121133 | 0,1344 | 0,138364 | 0,128417 |
| 0,098296 | 0,081776 | 0,092423 | 0,113977 | 0,123683 | 0,125103 | 0,132323 | 0,117099 | 0,136243 | 0,129826 |
| 0,074449 | 0,081381 | 0,100082 | 0,100421 | 0,113064 | 0,117701 | 0,122669 | 0,14988 | 0,150399 | 0,146075 |
| 0,073031 | 0,09502 | 0,104408 | 0,106396 | 0,121572 | 0,120088 | 0,131528 | 0,135279 | 0,123337 | 0,145152 |
| 0,063829 | 0,090017 | 0,099214 | 0,106646 | 0,106517 | 0,129528 | 0,124081 | 0,125398 | 0,134068 | 0,14487 |
| 0,06324 | 0,080689 | 0,110035 | 0,100432 | 0,106411 | 0,113256 | 0,131649 | 0,120478 | 0,130213 | 0,135345 |
| 0,063321 | 0,080852 | 0,098361 | 0,103935 | 0,103497 | 0,126005 | 0,1164 | 0,127854 | 0,132408 | 0,148412 |
| 0,067166 | 0,085571 | 0,094528 | 0,13362 | 0,149809 | 0,119346 | 0,125392 | 0,1195 | 0,131392 | 0,132262 |

Так же, приведены графики лучшего, среднего, худшего случаев и график c\*log(N) для обоих операций:

C = 0,061

C = 0,052

# Заключение.

В ходе данной лабораторной работы была реализована структура AVL-дерева, изучена сама структура деревьев (в том числе само балансируемых), а также изучены алгоритмы вставки, поиска и удаления элементов из АВЛ деревьев.

По результатам замеров времени на операции вставки и поиска можно сделать вывод о том, что операция поиска проходит быстрее, чем операция вставки, а сама структура дерева позволяет быстро и эффективно работать с данными, которые в нем хранятся.

Нужно отметить, что результаты замеров времени были найдены при запуске программы на онлайн компиляторе, так как при работе на моем компьютере почти все 20 раундов каждого теста показывали нулевые значения по времени. Такой результат получался из-за быстродействия самой структуры и алгоритмов, которые с ней работают.