

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

Курсовая работа

Дисциплина: Программирование на языках высокого уровня

Тема: AVL-дерево (удаление узла)

Студент группы 3331506/80401

В.С. Редров

Преподаватель

Е.М. Кузнецова

«__» _____ 2021 г.

Санкт-Петербург

2021 г

Оглавление

Оглавление	2
Введение.....	3
Принцип работы	4
Оценка скорости и памяти	6
Применение алгоритма.....	7
Список литературы	8

Введение

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

Дерево АВЛ названо в честь двух его советских разработчиков, Георгия Адельсона-Вельского и Евгения Ландиса, которые опубликовали его в своей статье 1962 года «Алгоритм организации информации».

В дереве АВЛ высота двух дочерних поддеревьев любого узла отличается не более чем на единицу; если в любой момент они отличаются более чем на единицу, выполняется перебалансировка для восстановления этого свойства. Поиск, вставка и удаление занимают время $O(\log n)$ как в среднем, так и в худшем случаях, где n - количество узлов в дереве до операции. Вставки и удаления могут потребовать перебалансировки дерева путем одного или нескольких вращений дерева.

Деревья АВЛ часто сравнивают с красно-черными деревьями, потому что оба поддерживают один и тот же набор операций и занимают $O(\log n)$ в для основных операций. Для приложений с интенсивным поиском деревья АВЛ быстрее, чем красно-черные деревья, потому что они более строго сбалансированы. Подобно красно-черным деревьям, деревья АВЛ сбалансированы по высоте.

В рамках данной курсовой работы будет рассмотрена реализация АВЛ дерева и алгоритм удаления узла на языке программирования C++ с использованием методов объектно-ориентированного программирования.

Принцип работы

Алгоритм был реализован при помощи языка программирования C++. Узел дерева представлен классом Node, полями которой являются значение ключа в узле, полезные данные узла, высота дерева, указатель на объект Node для левой и правой ветви.

Отсутствие узлов слева или справа будем обнаруживать при помощи нулевого указателя в поле left и right соответственно. Для правильной работы программы необходимо реализовать следующую функцию:

Функция `bfactor` возвращает разницу между высотой правой и левой ветви. По свойству AVL дерева он может принимать значения -1, 0, 1. При добавлении и удалении узлов может возникать ситуация, когда это условие нарушится. Для этого в программе предусмотрена функция балансировки дерева.

В качестве вспомогательных функций также выступают функция `height`, которая возвращает значение высоты поддерева, и `update_height`, которая обновляет значение высоты поддерева.

Балансировка узлов может быть осуществлена с помощью двух типов поворота дерева: большого и малого.

Малый левый поворот показан на рисунке 1. Применяется, когда `bfactor` узла «a» равен 2 и `bfactor` узла «b» больше либо равен нулю. Ее суть заключается в том, что корневым узлом становится узел «b», его левый потомок, становится правым потомком узла «a», а левым потомком узла «b» становится узел «a».

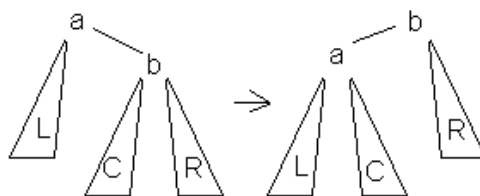


Рисунок 1 – Малый левый поворот

Большой левый поворот показан на рисунке 2. Применяется, когда bfactor узла «a» равен 2 и bfactor узла «b» меньше либо равен 0. Суть этого поворота заключается в том, что корневым узлом становится узел «с», его левый потомок становится правым потомком узла «a», и правый потомок становится левым потомком узла «b». Сами узлы «a» и «b» становятся левым и правым потомками узла «с» соответственно.

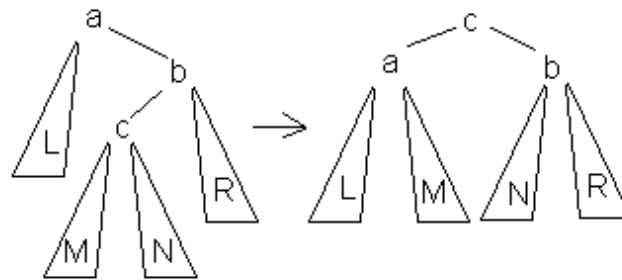


Рисунок 2 – Большой левый поворот

Малый правый и большой правый повороты представлены на рисунках 3 и 4 и являются зеркальным отражением левых поворотов.

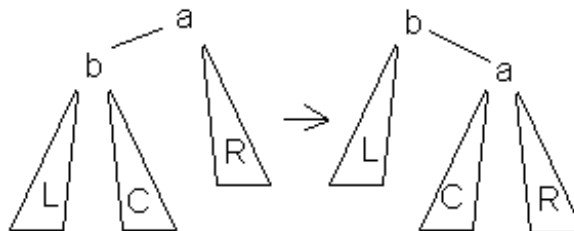


Рисунок 3 – Малый правый поворот

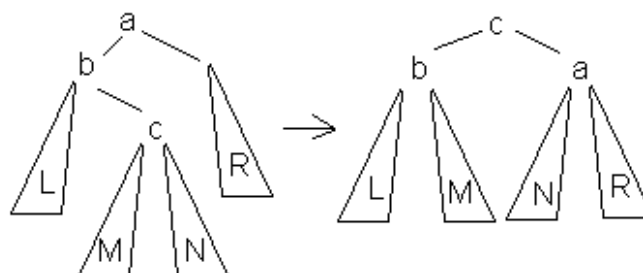


Рисунок 4 – Большой правый поворот

Функция балансировки основана на проверке всех этих условий и возвращает тот же узел, который был подан на вход, но со сбалансированными ветвями.

Функция удаления узла основана на рекурсивном алгоритме, который сначала идет вглубь дерева и ищет необходимый узел, затем, если он лист, то удаляет его и вызывает балансировку для каждого родителя, поднимаясь по рекурсии. Если узел – не лист, то функция находит в поддереве наибольшей длины ближайший по значению ключа элемент и заменяет удаляемый элемент на заменяющий, вызвав функцию удаления для заменяющего элемента. После удаления вызывается функция балансировки для каждого родителя, на каждом этапе возврата из рекурсии. Функция возвращает логический тип данных: false – в случае, если элемент с указанным ключом не найден, true – в случае удачного удаления.

Оценка скорости и памяти

Самая затратная операция в удалении – поиск узлов. В представленном алгоритме в худшем случае осуществляется 3 поиска – удаляемого узла, заменяющего узла и узла-родителя для удаляемого элемента. Эта операция занимает в худшем случае $O(\log(N))$ операций. Также сама функция удаления вызывается максимум два раза (для изначального узла и для заменяющего). Все это дает оценку сложности порядка $O(\log(N))$.

График скорости работы алгоритма удаления узла от количества узлов изображен на рисунке 5

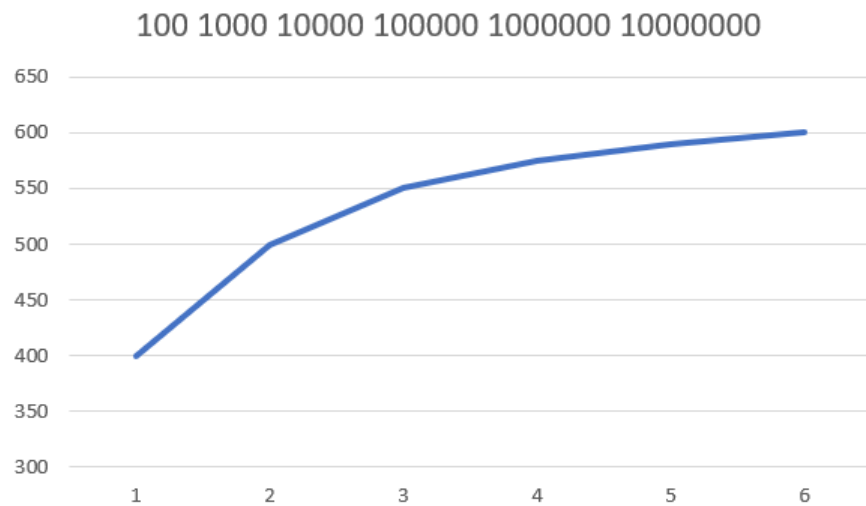


Рисунок 5 – График скорости работы алгоритма от количества узлов

Расход памяти $O(N)$.

Применение алгоритма

АВЛ-деревья могут быть применены для упорядоченного хранения элементов, вставки, поиска и удаления за время порядка $O(\log(N))$, что требуется, например, для баз данных.

Список литературы

1. Адельсон-Вельский Г. М., Ландис Е. М. Один алгоритм организации информации: Доклады АН СССР, 1962, с. 263—266.
2. Thomas H. Cormen Introduction to algorithms: учебное пособие / Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: Massachusetts Institute of Technology, 2009, с. 266 – 338.
3. АВЛ – дерево // [Wikipedia]. URL:
<https://ru.wikipedia.org/wiki/АВЛ-дерево>

Приложение – Листинг кода функции удаления ноды

```
bool Tree::del(Node* root, const int key){
    bool status = false;
    if (root == nullptr){
        return false;
    }
    if (root -> key == key){
        // if node to delete is found
        Node* parent = find_parent(this->root, key);
        if (!(root -> left) && !(root -> right)){
            // if it has no children (is leaf)
            if (root != this->root){
                // if it is not root of current tree
                if (parent -> left == root)
                    parent -> left = nullptr;
                else
                    parent -> right = nullptr;
            }
            else
                root = nullptr;
        }
        else{
            // if it has at least one child we should find node, which is going to
            // replace node to delete
            // then we should delete replace node and finally replace node to delete
            // with replace node
            Node* replace_node = find_replace_node(root);
            if (!del(root, replace_node -> key)){
                return false;
            }

            replace_node -> left = root -> left;
            replace_node -> right = root -> right;
            if (parent != root){
                if (parent -> left == root)
                    parent -> left = replace_node;
                else
                    parent -> right = replace_node;
            }
            else{
                root = replace_node;
            }
        }
    }

    else if (key > root -> key){
        status = del(root -> right, key);
    }
}
```

```
    else {  
        status = del(root -> left, key);  
    }  
    if (!status){ return false; }  
    balance(root);  
    return true; // balance tree, while going up the recursion  
}
```