

Тема 5. Быстрые деревья поиска

AVL дерево

Обычное дерево поиска может вырождаться в список и соответственно трудоемкость работы может стать $O(N)$. Однако любой набор различных элементов можно организовать в дерево с логарифмической глубиной.

Проблема: можно ли организовать работу с деревом (поиск, добавление, удаление) с сохранением логарифмической сложности в любой момент времени?

Одно из решений — AVL дерево (Адельсон-Вельский и Ландис).

Идея — поддержка сбалансированности по глубине (длине) путем “поворотов” (иллюстрация).

Определение. Глубина (длина) дерева — это наибольшая длина его ветвей от корня.

Определение. Идеально сбалансированное дерево — это такое, в котором длины всех ветвей от корня отличаются не более чем на 1.

Очевидно, длина идеально сбалансированного дерева с N узлами есть $\sim \log_2 N$. Но поддерживать идеальную сбалансированность при добавлении и удалении не удается со сложностью $O(\log_2 N)$.

Определение. Сбалансированное AVL дерево — это такое, в котором в любой вершине длины левого и правого поддеревьев отличаются не более чем на 1.

Пример. Идеально сбалансированное и AVL дерево — это разные вещи!

Определение. Баланс вершины есть разность длин правого и левого поддеревьев.

Пример. Дерево Фибоначчи — баланс всех внутренних вершин равен 1.

Утверждение. Длина дерева Фибоначчи с N вершинами есть $O(\log_2 N)$.

Доказательство. Пусть $N(k)$ есть количество вершин дерева Фибоначчи глубины k . Имеем рекуррентное соотношение

$$N(k) = N(k-2) + N(k-1) + 1$$

$$N(0) = 0, \quad N(1) = 1$$

Будем искать решением этого рекуррентного соотношения в виде

$$N(k) = a^k - 1$$

Действительно,

$$a^k - 1 = a^{k-2} - 1 + a^{k-1} - 1 + 1$$

или

$$a^2 - a - 1 = 0, \quad a_1, a_2 = \frac{1 \pm \sqrt{5}}{2}$$

Умножение на константу не меняет решения, т.е. $C_1 a_1^k + C_2 a_2^k - 1$ — тоже решение. Константы находятся из начальных условий:

$$C_1 + C_2 - 1 = 0$$

$$C_1 a_1 + C_2 a_2 - 1 = 1$$

Поскольку главным членом в элементах последовательности $N(k)$ является $C_1 a_1^k$, то при больших k можно считать, что $N(k) \sim C_1 a_1^k$, откуда получается, что

$$k \sim \log_2 N / \log_2 a_1 - \log_2 C_1 / \log_2 a_1 < \log_2 N / \log_2 a_1 < 1.5 \log_2 N$$

Утверждение. Произвольное AVL дерево длины k имеет не меньше узлов, чем дерево Фибоначчи длины k .

Доказательство легко получается по индукции.

Пусть утверждение верно для всех AVL деревьев длины от 0 до $k-1$. Рассмотрим корень произвольного AVL дерева длины k . Пусть это дерево имеет M узлов. Поддеревья этого корня имеют длины M_1 и M_2 не меньше $k-1$ и $k-2$. По предположению индукции имеем

$$M_1 \geq N(k-1), \quad M_2 \geq N(k-2) \implies M = M_1 + M_2 + 1 \geq N(k-1) + N(k-2) + 1 = N(k)$$

Утверждение. Длина произвольного AVL дерева с N узлами не превосходит $1.5 \log_2 N$.

Доказательство. Идея. Сопоставим дереву глубины k с N узлами точку на плоскости с координатами (k, N) . По предыдущему утверждению все точки для AVL деревьев будут лежать не ниже кривой $N = N(k)$ (для дерева Фибоначчи). Кстати, эти точки будут лежать не выше кривой $N = 2^k - 1$. Таким образом, для данного N точки всех AVL деревьев будут образовывать горизонтальный отрезок между этими кривыми, т.е. максимальное k реализуется для дерева Фибоначчи.

Итак, AVL дерево имеет логарифмическую длину относительно числа узлов, и теперь надо обеспечить трудоемкость добавления и удаления с таким деревом в границах $O(\log N)$.

AVL дерево, добавление и удаление элементов

Для узла AVL дерева примем структуру

```
template <class T>
struct TreeNode
{
    T value;
    TreeNode *left, *right;
    int balance; // разность между длинами поддеревьев
    TreeNode() { left = right = nullptr; }
}
```

Добавление.

Будем реализовывать добавление на основе рекурсивной процедуры. Основная идея состоит в том, что добавляем элемент как в обычное дерево поиска, но если при этом нарушается баланс в корне какого либо поддерева, то пытаемся исправить ситуацию при помощи так называемого поворота вокруг некоторой вершины поддерева.

Иллюстрация: поворот вокруг указанной вершины (левый или правый).

Поворот позволяет нам переместить узлы из одного поддерева в другое и таким образом попробовать выровнять балансы в данном поддереве.

Для определенности рассмотрим случай добавления в левое поддерево.

Иллюстрация: дерево до и после добавления в левое поддерево.

При добавлении левое поддерево может сохранить свою длину, и в этом случае больше нам ничего делать не надо. Если же длина меняется (увеличивается на 1), то нам, возможно, потребуется перестройка. Какую перестройку и как выполнять зависит от того какие балансы были зафиксированы в исходном дереве.

Итак, мы добавили элемент в левое поддерево. Рекурсивная процедура добавления выполнила свою работу в этом поддереве и привела его корректному AVL состоянию. При этом, возможно, корень поддерева изменился, но процедура вернула нам указатель на этот корень. Таким образом, заголовок процедуры добавления может иметь такой (упрощенный) вид

```
TreeNode * Add(TreeNode *r, const T &x, int &grow)
```

где r — корень поддерева до добавления, x — добавляемое значение, $grow$ — величина прироста поддерева после добавления (0 или 1), возвращаемое значение — корень поддерева после добавления,

Рассмотрим последовательно все возможные случаи, считая, что исходное дерево имело длину k и левое поддерево увеличилось при добавлении. Пусть A — корень текущего поддерева, B — корень левого поддерева (т.е. левый потомок A).

1. Баланс A есть 1 или 0. Тогда перестройка не нужна. Только балансы A изменяются соответственно на 0 и -1.

2. Баланс A есть -1. Перестройка однократным правым поворотом в A . Необходимо рассмотреть балансы B . Проблема разрешается для балансов B равных -1 и 0.

3. Баланс A есть -1, баланс B есть 1. Рассматриваем C — правый потомок вершины B и его поддерева. выполняем двойной поворот (левый в B и правый в A). Проблема разрешается, расстановка балансов зависит от балансов вершины C .

Иллюстрация: повороты и балансы для случаев 1, 2, 3.

Результаты этих действий можно свести в общую таблицу.

Таблица добавления в левое поддерево

балансы						изм. длины	перестройка
старые			новые				
A	B	C	A	B	C		
1			0			0	
0			-1			1	

-1	-1		0	0		0	L1 однократный поворот
-1	0		-1	1		1	L1

-1	1	0	0	0	0	0	L2 двойной поворот
-1	1	1	0	-1	0	0	L2
-1	1	-1	1	0	0	0	L2

Теперь, опираясь на эту табличку будем строить рекурсивную процедуру.

```
TreeNode * Add(TreeNode *A, const T &x, int &grow)
{
    if (root == nullptr) { // пустое дерево
        A = new TreeNode;
        A->value = x;
        grow = 1;
        return A;
    }
}
```

```
} else {
    TreeNode *B, *C;
    if (x < A->value) { // добавление налево
        A->left = Add(A->left, x, grow); // добавили
        if (grow == 0) { // длина не увеличилась
            grow = 0;
            return A;
        } else { // длина увеличилась
            switch(A->balance) {
                case 1: // наш случай 1
                    A->balance = 0;
                    grow = 0;
                    return A;
                case 0; // тоже наш случай 1
                    A->balance = -1;
                    grow = 1;
                    return A;
                case -1: // наш случай 2
                    // здесь начинается самое интересное
                    ....
            }
        }
    }
    if (x > A->value) { // добавление направо
        A->right = Add(A->right, x, grow);
        .... симметричный код ....
    }
    // x должен быть уникальным !!!
}

// вот и оно --- то самое интересное
B = A->left;
switch (B->balance) {
case -1: // разрешение случая 2
    A->left = B->right; B->right = A;
    A->balance = B->balance = 0;
    grow = 0;
    return B;
case 0; // разрешение случая 2
    A->left = B->right; B->right = A;
    A->balance = -1; B->balance = 1;
    grow = 1;
    return B;
case 1: // наш случай 3
    ....
}
```

```

// и теперь случай 3
switch (B->balance) {
    ....
case 1:
    C = B->right;
    B->right = C->left; C->left = B;
    A->left = C->right; C->right = A;
    switch(C->balance) {
        case 0: A->balance = B->balance = C->balance = 0; break;
        case 1: A->balance = C->balance = 0; B->balance = -1; break;
        case -1: A->balance = 1; B->balance = C->balance = 0; break;
    }
    grow = 0;
    return C;
}

```

Из таблицы следует, что выполняется не более двух перестроек для каждого добавления.

Если используется ссылка на родителя, то ее восстанавливаем стандартным образом

```

    if (A->left) A->left->parent = A;
    if (A->right) A->right->parent = A;
\end{verbatim}

```

И аналогично для других вершин при поворотах.

{\bf Удаление.}

Тот же принцип, что и в простом дереве поиска.

Если у элемента не более одного потомка, то удаляем, а если потомков 2, то подменяем значение и удаляем подменный элемент.

Т.е. необходимость балансировки возникает при реальном удалении элемента (длина дерева может сократиться).

прежняя процедура:

```

\begin{verbatim}
TreeNode * Remove(TreeNode * A, const T & x, int &grow)
{
    TreeNode *p;
    if (A == nullptr) return nullptr;
    if (x > A->value) {
        A->right = Remove(A->right, x, grow);
        // могла сократиться длина
        // if (grow < 0) BalanceRight(...)
    } else if (x < A->value) {

```

```

    A->left = Remove(A->left, x, grow);
        // могла сократиться длина  grow < 0 ?
        // if (grow < 0) BalanceLeft(...)
} else {
    if (A->right == nullptr) {
        p = A->left;
        delete root;          // длина заведомо сократилась
        grow = -1
        return p;
    }
    if (A->left == nullptr) {
        p = A->right;
        delete root;          // длина заведомо сократилась
        grow = -1
        return p;
    }
    p = SearchRightmost (A->left);
    A->value = p->value;
    A->left = Remove(A->left, A->value, grow);
    // if (grow < 0) BalanceLeft(...)
}
return A; // если раньше не вернули другой корень :)
}

```

Рассмотрим удаление слева

```

TreeNode * Remove (TreeNode *A, T &x, int grow);

TreeNode * BalanceLeft (TreeNode *A, int grow);
// возвращает указатель на корень перестроенного дерева

```

Иллюстрация. Перестройка R1

Иллюстрация. Перестройка R2

Сводная таблица балансов при перестройке правого поддерева при удалении слева

балансы							
старые			новые			изм. длины	перестройка
A	B	C	A	B	C		
-1			0			-1	
0			-1			0	
1	1		0	0		-1	R1
1	0		1	-1		0	R1
1	-1	0	0	0	0	-1	R2
1	-1	-1	0	1	0	-1	R2
1	-1	1	-1	0	0	-1	R2

Из таблицы следует, что перестройки могут выполняться по цепочке вплоть до корня.

Симметрично, если задействованы другие стороны.