Тема 4. Быстрые деревья поиска

Лекция 11. 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 строить рекурсивную процедуру.

```
TreeNode * Add(TreeNode *A, const T &x, int &grow)
{
    if (root == nullptr) {
        A = new TreeNode;
        A->value = x;
        grow = 1;
        return A;
    } else {
        TreeNode *B, *C;
        if (x < A->value) {
            A->left = Add(A->left, x, grow);
            if (grow == 0) {
                grow = 0;
                return A;
            } else {
                switch(A->balance) {
                case 1:
                    A->balance = 0;
                    grow = 0;
                    return A;
                case 0;
                    A->balance = -1;
                    grow = 1;
                    return A;
                case -1:
                    // здесь начинается самое интересное
                    ....
                }
            }
        }
```

```
            if (x > A->value) {
                A->right = Add(A->right, x, grow);
                .... симметричный код ....
            }
            // x должен быть уникальным !!!
        }
    }
```

Иллюстрация. Баланс корня -1. Поворот L1. `B = A->left`
L1 поворот разрешает ситуации с балансом B -1 и 0

```
---------
B = A->left;
switch (B->balance) {
case -1:
   A->left = B->right; B->right = A;
   A->balance = B->balance = 0;
   grow = 0;
   return B;
case  0;
   A->left = B->right; B->right = A;
   A->balance = -1;  B->balance = 1;
   grow = 1;
   return B;
case  1:
   .....
}
```

Иллюстрация. Баланс корня -1. Баланс B есть 1. Поворот L2. Балансы C.
L2 поворот разрешает ситуацию с балансом B 1
Таблица добавления в левое поддерево

балансы							
старые			новые			изм. длины	перестройка
A	B	C	A	B	C		
1			0			0	
0			-1			1	
-1	-1		0	0		0	L1
-1	0		-1	1		1	L1
-1	1	0	0	0	0	0	L2
-1	1	1	0	-1	0	0	L2
-1	1	-1	1	0	0	0	L2

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

```
---------
B = A->left;
```

```
C = B->right;
switch (B->balance) {
case -1:
   A->left = B->right; B->right = A;
   A->balance = B->balance = 0;
   grow = 0;
   return B;
case  0;
   A->left = B->right; B->right = A;
   A->balance = -1;  B->balance = 1;
   grow = 1;
   return B;
case  1:
   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;
}
```

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:

```
TreeNode * Remove(TreeNode * root, const T & x, int &grow)
{
    TreeNode *p;
    if (root == nullptr) return nullptr;
    if (x > root->value) {
        root->right = Remove(root->right, x, grow);
              // могла сократиться длина
              // if (grow < 0) BalanceRight(...)
        if (root->right) root->right->parent = root;
    } else if (x < root->value) {
        root->left = Remove(root->left, x, grow);
              // могла сократиться длина   grow < 0 ?
              // if (grow < 0) BalanceLeft(...)
        if (root->left) root->left->parent = root;
    } else {
        if (root->right == nullptr) {
            p = root->left;
```

```
            delete root;         // длина заведомо сократилась  grow = -1
            return p;
        }
        if (root->left == nullptr) {
            p = root->right;
            delete root;         // длина заведомо сократилась  grow = -1
            return p;
        }
        p = SearchRightmost (root->left);
        root->value = p->value;
        root->left = Remove(root->left, root->value, grow);
               // if (grow < 0) BalanceLeft(...)
        if (root->left) root->left->parent = root;
    }
    return root;
}
```

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.