Лекция 18 AVL-деревья (окончание)

18.1. *Построение АВЛ-дерева*

18.1.1. Высота поддерева с корнем в узле *P*.

```
static int Height(Position P){ //static, чтобы можно было
                              //использовать в рекурсивных функциях
      if(P == NULL) return -1;
      else return P->Height;
}
```

Выбор более длинного поддерева

```
static int Max( int Lhs, int Rhs ){
      return Lhs > Rhs ? Lhs : Rhs;
      }
```

18.1.2. Однократные повороты

18.1.2.1. Между узлом и его левым ребенком

```
/* Эту функцию можно вызывать только в том случае, когда у */
/* узла K2 есть левый ребенок. Функция выполняет поворот */
/* между узлом (K2) и его левым ребенком, корректирует */
/* высоты поддеревьев, после чего возвращает новый корень */
static Position SingleRotateWithLeft(Position K2) {
        Position K1;
        /* выполнение поворота */
        K1 = K2->Left;
        K2->Left = K1->Right;
        K1->Right = K2;
        /* корректировка высот переставленных узлов */
        K2->Height = Max(Height(K2->Left), Height(K2->Right))+1;
        K1->Height = Max(Height(K1->Left), K2->Height) + 1;
        return K1; /* новый корень */
}
```

18.1.2.2. Между узлом и его правым ребенком

```
/* Эту функцию можно вызывать только в том случае, когда у */
/* узла K1 есть правый ребенок. Функция выполняет поворот */
/* между узлом (K1) и его правым ребенком, корректирует */
/* высоты поддеревьев, после чего возвращает новый корень */
static Position SingleRotateWithRight(Position K1){
        Position K2;
        K2 = K1->Right;
        K1->Right = K2->Left;
        K2->Left = K1;
        K1->Height = Max(Height(K1->Left), Height(K1->Right))+1;
        K2->Height = Max(Height(K2->Right), K1->Height) + 1;
        return K2; /* новый корень */
}
```

18.1.3. Двойные повороты

18.1.3.1. *LR*- поворот

```
/* Эту функцию можно вызывать только тогда, когда */
/* у узла K3 есть левый ребенок, а у левого ребенка */
/* K3 есть правый ребенок. Функция выполняет двойной */
/* поворот LR, корректирует высоты поддеревьев, после */
/* чего возвращает новый корень */
static Position DoubleRotateWithLeft(Position K3){
        /* Поворот между K1 и K2 */
```

```
        K3->Left = SingleRotateWithRight(K3->Left);
        /* Поворот между K3 и K2 */
        return SingleRotateWithLeft(K3);
}
```

18.1.3.1. *RL*- поворот

```
/* Эту функцию можно вызывать только в том случае, когда у */
/* узла K1 есть правый ребенок, а у правого ребенка узла K1 */
/* есть левый ребенок. Функция выполняет двойной поворот */
/* RL, корректирует  высоты поддеревьев, после чего */
/* возвращает новый корень */
static Position DoubleRotateWithRight(Position K1){
        /* Поворот между K3 и K2 */
        K1->Right = SingleRotateWithLeft(K1->Right);
        /* Поворот между K1 и K2 */
        return SingleRotateWithRight(K1);
}
```

18.1.4. Вставить новый узел

```
AvlTree Insert(ElementType X, AvlTree T){
        if(T == NULL){
              /* создание дерева с одним узлом */
              T = malloc(sizeof(struct AvlNode));
              if(T == NULL)
                    FatalError("Out of space!");
              else {
                    T->Element = X; T->Height = 0;
                    T->Left = T->Right = NULL;
              }
        }
        else
              if(X < T->Element){
                    T->Left = Insert(X, T->Left);
                    if(Height(T->Left) - Height(T->Right) == 2)
                    if(X < T->Left->Element)
                          T = SingleRotateWithLeft(T);
              else
                    T = DoubleRotateWithLeft(T);
              }
        if(X > T->Element){
              T->Right = Insert(X, T->Right);
              if(Height(T->Right) - Height(T->Left) == 2)
              if(X > T->Right->Element)
                    T = SingleRotateWithRight(T);
              else
                    T = DoubleRotateWithRight(T);
        }
        /* Иначе X уже в дереве и ничего не нужно делать; */

        T->Height = Max(Height(T->Left), Height(T->Right)) + 1;
        return T;
        }
```

18.1.5. ***Пример построения АВЛ-дерева.***

Пусть на «вход» функции Search() последовательно поступают целые числа 4,5,7,2,1,3,6. Изобразим процесс «роста» АВЛ-дерева (в скобках для части вершин указан показатель сбалансированности) [1,с.256]:

Это тщательно подобранный пример, показывающий ситуацию: как можно больше поворотов при минимальном числе включений. (рис. 6)

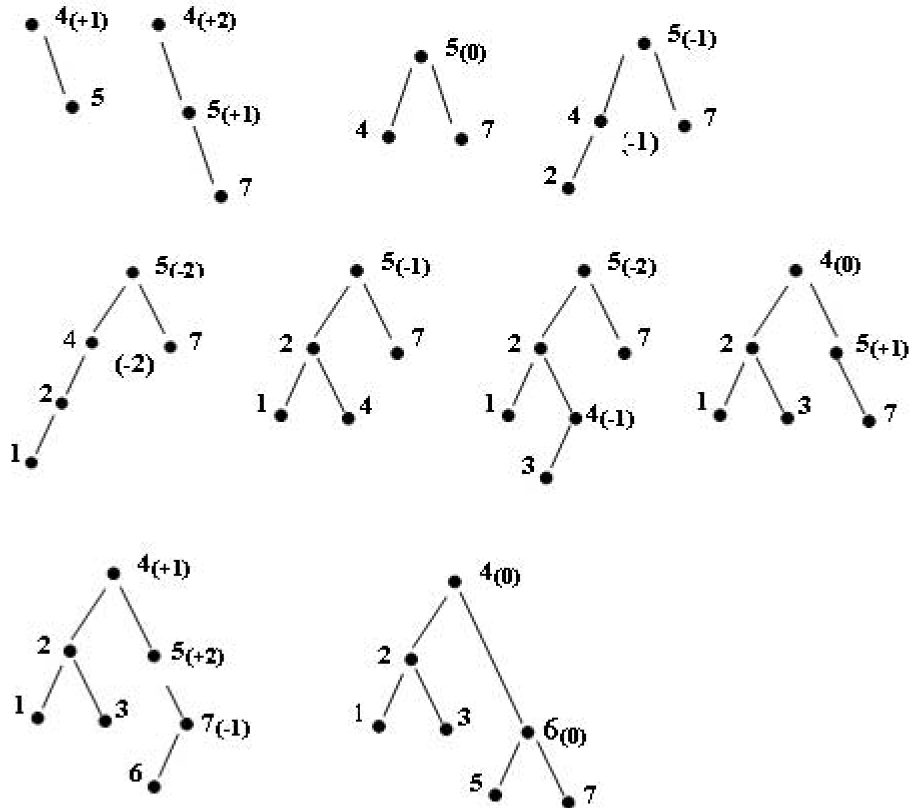

Рис.6. Построение АВЛ-дерева

18.2. ***Исключение узла из АВЛ-дерева***

18.2.1. Объявление функции:

```
AvlTree Delete( ElementType X, AvlTree T ){
        }
```

18.2.2. Исключение узла из АВЛ-дерева отличается от исключения узла из двоичного дерева (рассматривалось на прошлой лекции) необходимостью балансировки дерева после исключения узла. Иными словами в конец функции, выполняющей исключение узла необходимо добавить вызовы функций:

SingleRotateWithRight(T), SingleRotateWithLeft(T), DoubleRotateWithRight(T) и **DoubleRotateWithLeft(T)**

18.2.3. При исключении узла, имеющего двух детей возможен случай, не возникающий при вставке узлов: дерево, подвешенное к узлу *rr*, имеет высоту $h + 1$ (при вставке узлов такой случай возникнуть не может, так как в этом случае может увеличиться высота только одного поддерева). В этом случае необходим дополнительный поворот относительно узла *r*.

18.3. ***Оценки сложности***

18.3.1. На лекции 16 были получены оценки высоты для самого «хорошего» АВЛ-дерева, содержащего m узлов – $h = O(\log_2 m)$ (*полностью сбалансированное дерево*) и самого «плохого» АВЛ-дерева, содержащего m узлов – $h \le 1.44 \cdot \log_2(m+1) - 0.32$ (*дерево Фибоначчи*). Следовательно, для «среднего» АВЛ-дерева, содержащего m узлов оценка высоты будет $\log_2(m+1) \le h \le 1.44 \cdot \log_2(m+1) - 0.32$.

m	$\log_2 m$	m	$\log_2 m$
1	0	65,536	16
16	4	1,048,476	20
256	8	16,778,616	24
4,096	12		