Лекция 17 AVL-деревья

17.1. АВЛ -деревья.

17.1.1. В АВЛ-деревьях (Адельсон-Вельский, Ландис) оценка сложности не лучше, чем в совершенном дереве, но не хуже, чем в деревьях Фибоначчи для всех операций: поиск, исключение, занесение. *АВЛ-деревом* (подравненным деревом) называется такое двоичное дерево, когда для любой его вершины высоты левого и правого поддерева отличаются не более, чем на 1.

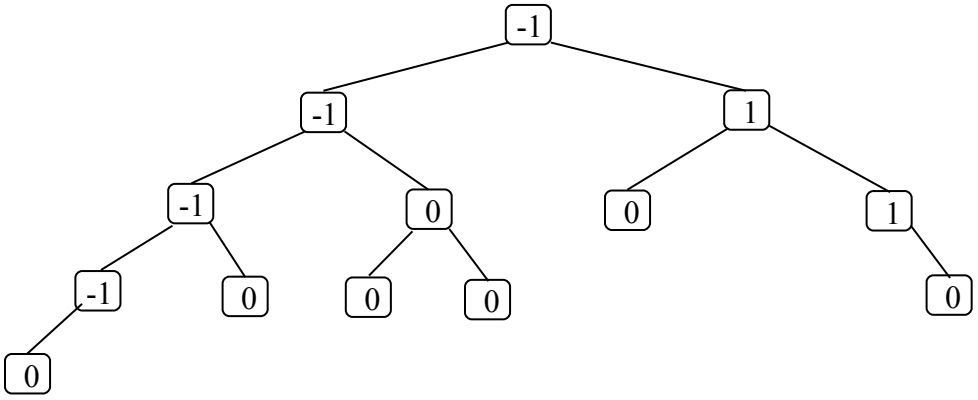


Рис. 1. Пример АВЛ-дерева.

17.1.2 ***Пример АВЛ-дерева*** показан на рисунке 1. В узлах дерева записаны значения показателя сбалансированности (*balance Factor*), определяемого по формуле:

balance Factor = *height*(*right subtree*) – *height*(*left subtree*)

Показатель баланса может иметь одно из трех значений

–1: Высота левого поддерева на 1 больше высоты правого поддерева.

0: Высоты обоих поддеревьев одинаковы.

+1: Высота правого поддерева на 1 больше высоты левого поддерева.

У совершенного дерева *все* узлы имеют показатель баланса 0, т.е. это самое «хорошее» АВЛ-дерево, а у дерева Фибоначчи *все* узлы имеют показатель баланса +1 (либо –1), т.е. это самое «плохое» АВЛ-дерево.

Числа Леонарда (количество узлов в дереве Фибоначчи):

$N_0 = 0$; $N_1 = 1$; $N_h = N_{(h-1)} + 1 + N_{(h-2)}$

17.1.3. Типичная структура узла АВЛ-дерева:

```
typedef int KeyType;
struct AvlNode;
typedef struct AvlNode *Position;
typedef struct AvlNode *AvlTree;
struct AvlNode {
    KeyType   Key;      //ключ
    AvlTree   Left;     //левое поддерево
    AvlTree   Right;    //правое поддерево
    Int       Balance;      //показатель баланса
```

```
        Int       Hight;    //высота поддерева
};
```

17.1.4 Базовые операции над АВЛ-деревьями.

```
AvlTree MakeEmpty(AvlTree T);      //удалить дерево
Position Find(KeyType X, AvlTree T);      //поиск по ключу
Position FindMin(AvlTree T);       //минимальный ключ
Position FindMax(AvlTree T);       //максимальный ключ
AvlTree Insert(KeyType X, AvlTree T);   //вставить узел
AvlTree Delete(KeyType X, AvlTree T);   //исключить узел
```

17.2. ***Реализация простейших базовых операций***:

17.2.1. Удалить дерево:

```
AvlTree MakeEmpty(AvlTree T){
     if(T != NULL) {
          MakeEmpty(T->Left);
          MakeEmpty(T->Right);
          free(T);
     }
     return NULL;
}
```

17.2.2. Поиск по ключу:

```
Position Search(KeyType X, AvlTree T){
     if(T == NULL) return NULL;
     if(X < T->Element) return Find(X, T->Left);
     else
          if(X > T->Element) return Find(X, T->Right);
          else return T;
}
```

17.2.3. Минимальный и максимальный ключи:

```
Position FindMin(AvlTree T){
     if(T == NULL) return NULL;
     else
          if(T->Left == NULL) return T;
          else return FindMin(T->Left);
}


Position FindMax(AvlTree T){
     if(T != NULL)
          while(T->Right != NULL)T = T->Right;
          return T;
}
```

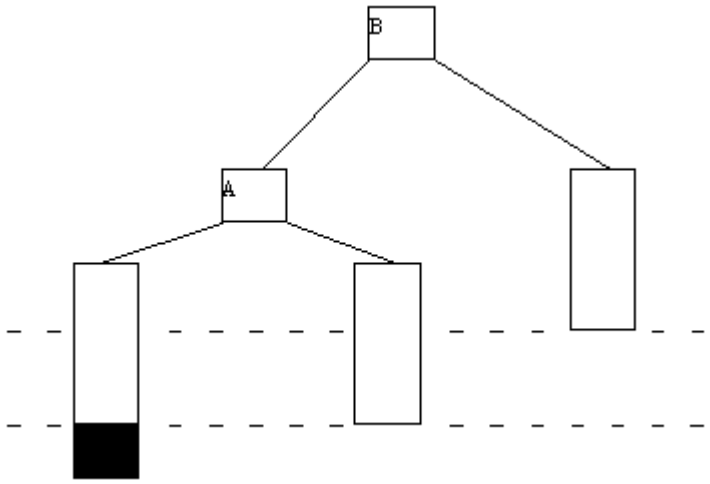
17.3. ***Включение узла в АВЛ-дерево***

17.3.1. ***Поддержка балансировки АВЛ-дерева при выполнении операции включения ключей.*** Пусть рассматриваемое дерево состоит из корневой вершины *r* и левого (*L*) и

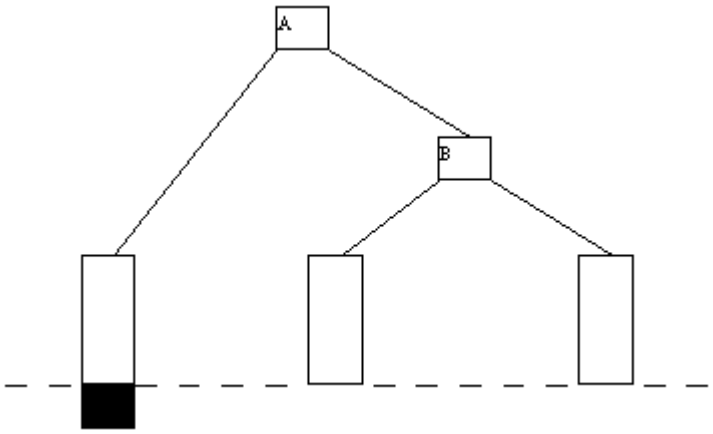
правого (*R*) поддеревьев. Будем обозначать через *hl* высоту *L*, а через *hr* - высоту *R*. Для определенности будем считать, что новый ключ включается в поддерево *L*. Если высота *L* не изменяется, то не изменяются и соотношения между высотой *L* и *R*, и свойства АВЛ-дерева сохраняются. Если же при включении в поддерево *L* высота этого поддерева увеличивается на 1, то возможны следующие три случая:

- если *hl* = *hr*, то после добавления вершины *L* и *R* станут разной высоты, но свойство сбалансированности сохранится;
- если *hl* < *hr*, то после добавления новой вершины *L* и *R* станут равной высоты, т.е. сбалансированность общего дерева даже улучшится;
- если *hl* > *hr*, то после включения ключа критерий сбалансированности нарушится, и *потребуется перестройка дерева*.

Рассмотрим две разные ситуации: (1) новая вершина добавляется к поддереву *L*, (2) новая вершина добавляется к поддереву *R*. Правила восстановления балансировки показаны на рисунках 2 и 3.

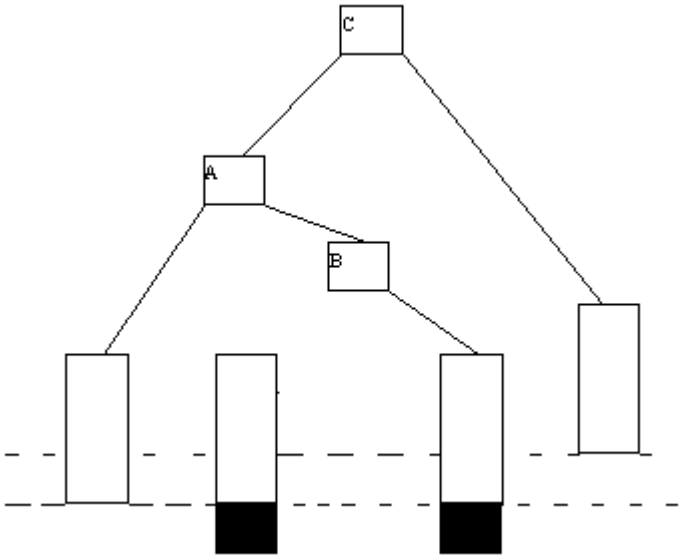


(a) Исходное состояние ситуации (1): в результате добавления поддерево с корнем в узле *B* разбалансировалось: разность высот его левого и правого поддеревьев стала равной –2 .

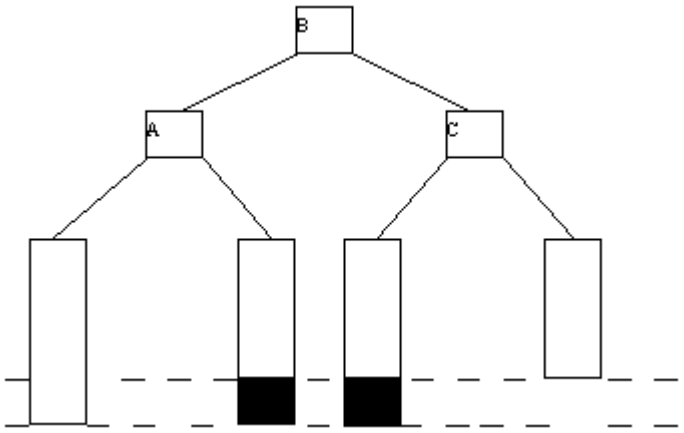


(b) Преобразование, разрешающее ситуацию (1) (Случай *RR* – однократный поворот): Делаем узел *A* корневым узлом поддерева, в результате правое поддерева с корнам в узле *B* «опускается» и разность высот становится равной –1 . Это преобразование подобно применению ассоциативного закона: $(\alpha\beta)\gamma = \alpha(\beta\gamma)$.

Рис. 2. Случай *RR* (симметричный случай *LL*: $\alpha(\beta\gamma) = (\alpha\beta)\gamma$).



(a) Исходное состояние ситуации (2): в результате добавления элемента в левое поддерево узла *B* поддерево с корнем в узле *C* разбалансировалось: разность высот его левого и правого поддеревьев стала равной 2.



(b) Преобразование, разрешающее ситуацию (2) (Случай *LR* – двукратный поворот). Нужно вытянуть узел *B* на самый верх, чтобы его поддеревья поднялись. Для этого сначала делаем левый поворот, меняя местами поддеревья с корневыми узлами *A* и *B*, а потом – правый поворот, меняя местами поддеревья с корневыми узлами *B* и *C*.
Рис. 3. Случай *LR* (симметричный случай *RL*).

В обоих случаях в дереве требуется изменить несколько ссылок (соответствующие функции приводятся ниже). Кроме того, новые деревья имеют высоту $h + 2$, т.е. точно равную высоте, которая была до вставки. Следовательно, остаток поддерева корневым узлом *A* (если таковой имеется) всегда остается сбалансированным.
На рисунках 4 и 5 представлены примеры, иллюстрирующие балансирование АВЛ-дерева в обоих рассмотренных случаях.

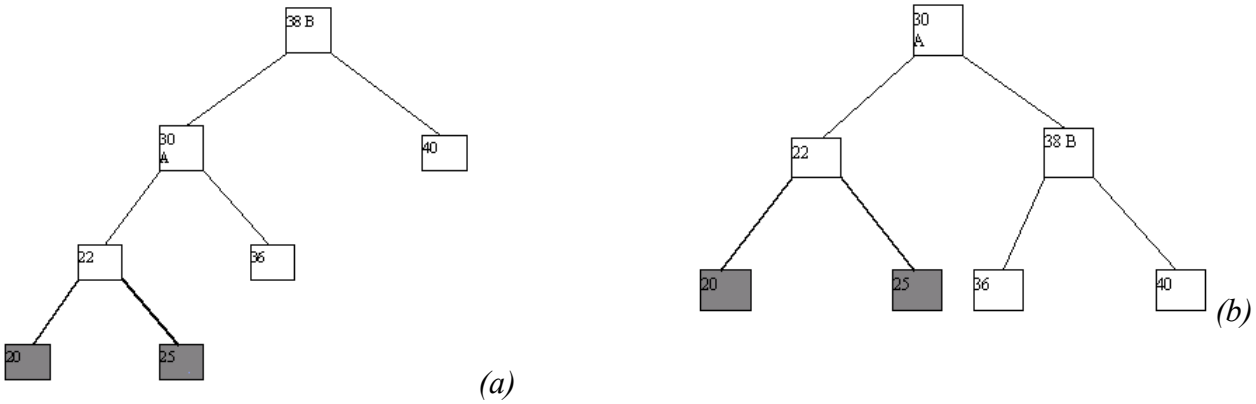


Рис. 4. Пример балансировки АВЛ-дерева. (Случай *RR* – однократный поворот)

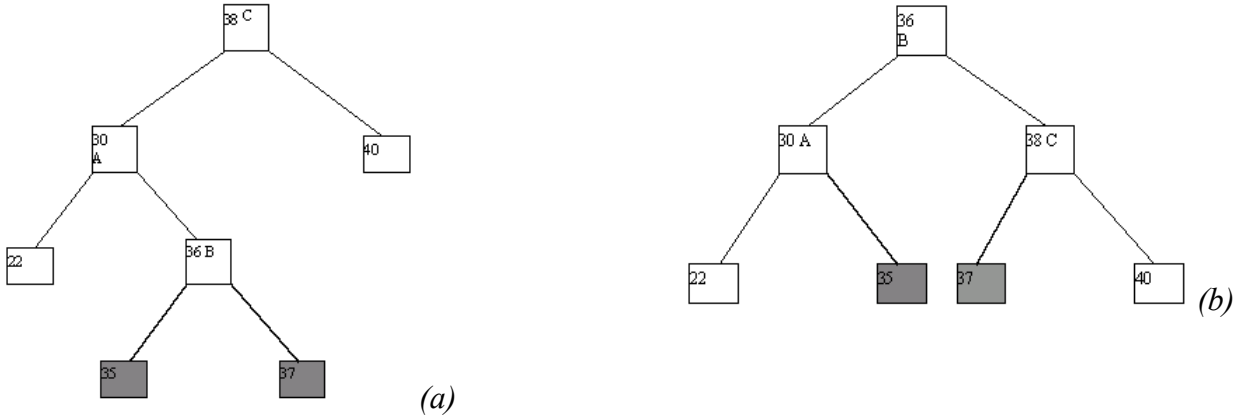


Рис. 5. Пример балансировки АВЛ-дерева. (Случай *LR* – двукратный поворот)

Рисунки хорошо проясняют суть преобразования.

Есть простой признак того, когда следует применить один поворот *LL*, а когда – два поворота *RR* и *RL*. Если $height(r \rightarrow l) \le height(r \rightarrow r)$, то следует выполнить *LL*.

17.4. ***Построение АВЛ-дерева***

17.4.1. 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;
      }
```

17.4.2. Однократные повороты

17.4.2.1. 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 */
}
```

17.4.2.2. 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 */
}
```

17.4.3. Двойные повороты

17.4.3.1. *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);
}
```

17.4.3.1. *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);
    }
```

17.4.4. 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;
        }
```

17.4.5. ***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)

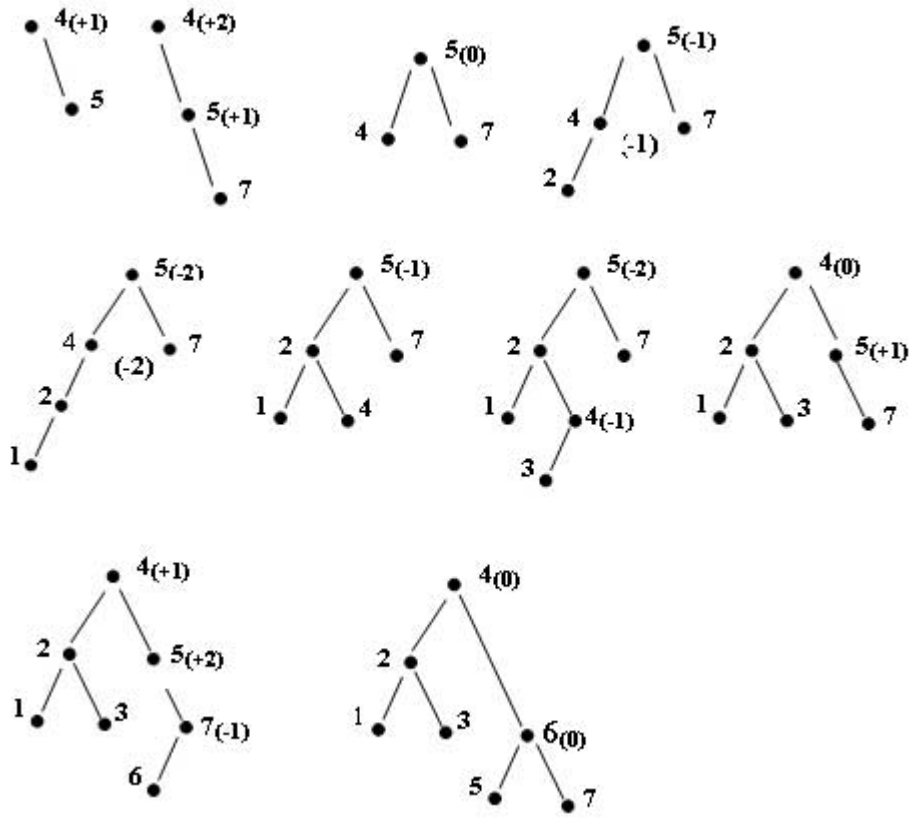


Рис.6. Построение АВЛ-дерева

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

17.5.1. Объявление функции:

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

17.5.2. 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)`

17.5.3. При исключении узла, имеющего двух детей возможен случай, не возникающий при вставлении узлов: дерево, подвешенное к узлу *rr*, имеет высоту $h + 1$ (при вставлении 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*.

17.6. ***Оценки сложности***

На прошлой лекции 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		