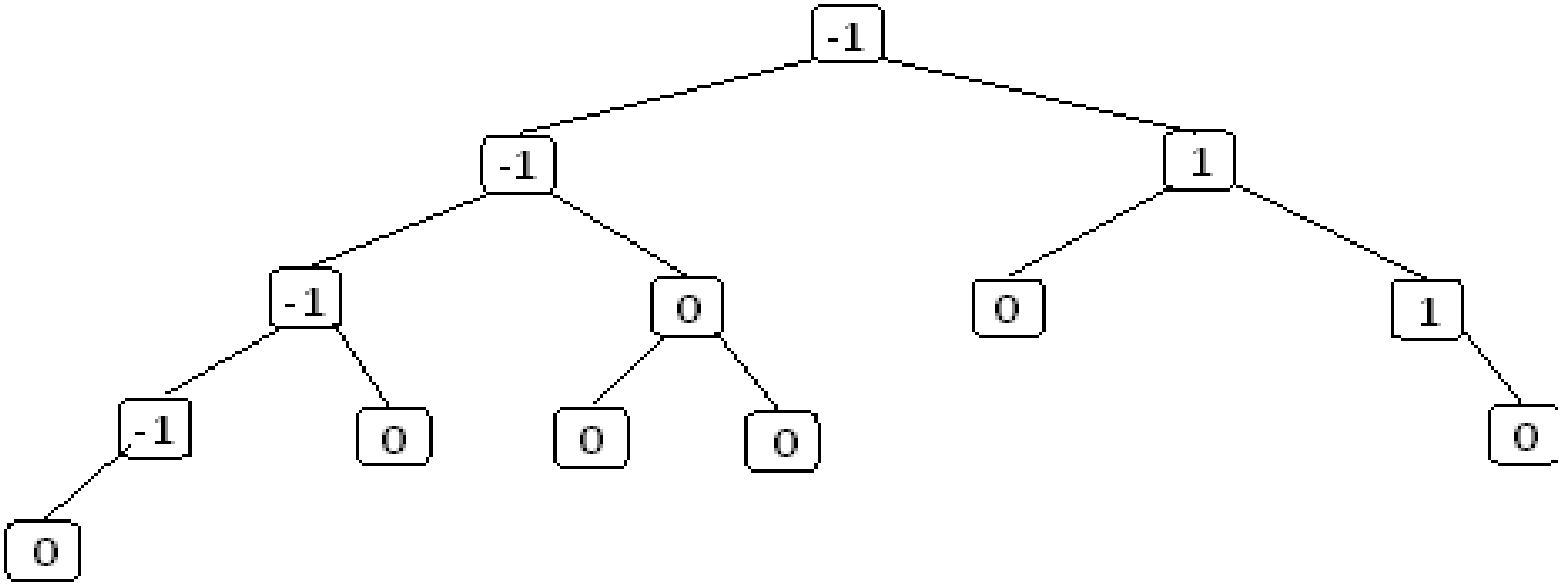
Курс «Алгоритмы и алгоритмические языки»

Лекция 19

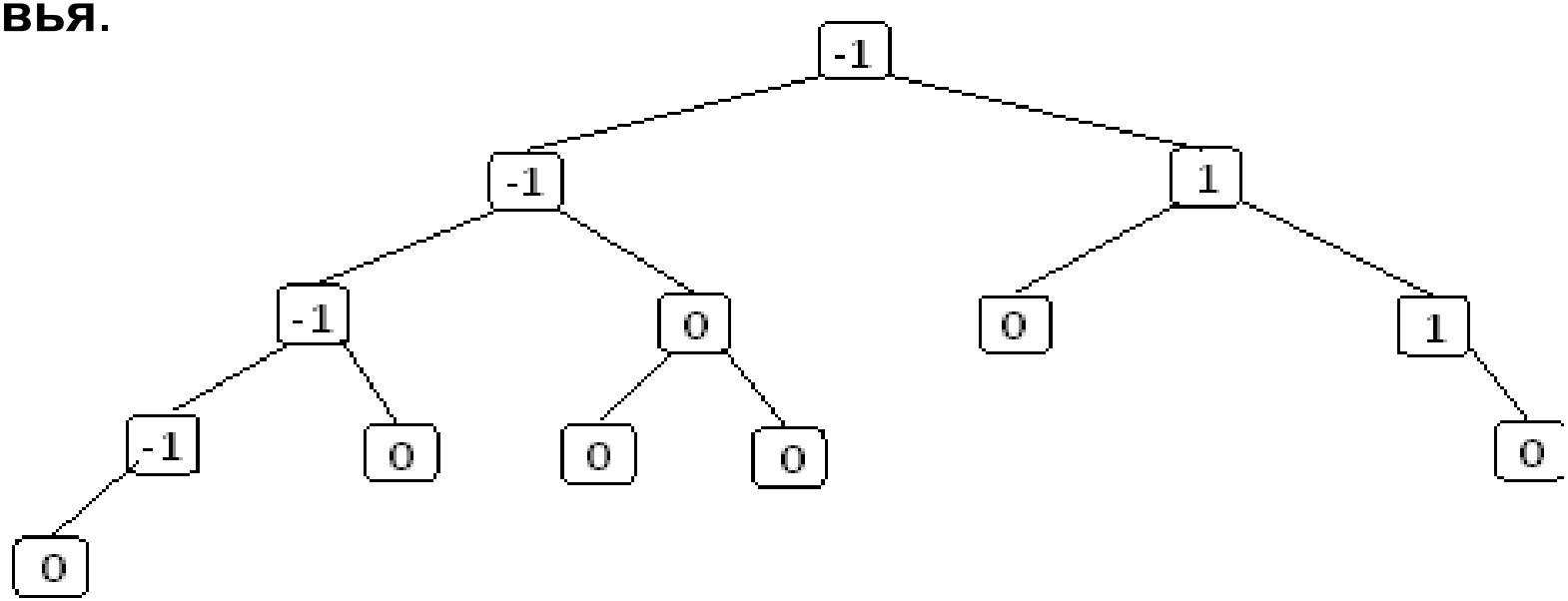
АВЛ -деревья.

- 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.



Пример АВЛ-дерева.

АВЛ -деревья.



- В узлах дерева записаны значения *показателя сбалансированности* (*balance Factor*), определяемого по формуле:

  $$balance\ Factor = height(right\ subtree) - height(left\ subtree)$$

  Показатель сбалансированности 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 (это самое «хорошее» АВЛ-дерево) а у дерева Фибоначчи *все* узлы имеют показатель баланса +1 (либо –1) (это самое «плохое» АВЛ-дерево).

АВЛ -деревья.

- Типичная структура узла АВЛ-дерева:

```
typedef int KeyType;
struct AvlNode;
typedef struct AvlNode *Position;
typedef struct AvlNode *AvlTree;
struct AvlNode {
  KeyType   Element;     //ключ
  AvlTree   Left;        //левое поддерево
  AvlTree   Right;       //правое поддерево
  int       Balance;     //показатель баланса
  int       Height;      //высота поддерева
};
```

Включение узла в АВЛ-дерево.

- ***Поддержка балансировки АВЛ-дерева при выполнении операции включения ключей.***

  Рассматриваемое дерево состоит из корневой вершины r и левого (L) и правого (R) поддеревьев, имеющих высоты h_L и h_R соответственно.

  Для определенности будем считать, что новый ключ включается в поддерево L.

- h_L **не изменяется** $\Rightarrow$ не изменяются соотношения между h_L и h_R $\Rightarrow$ свойства АВЛ-дерева сохраняются.

- h_L **увеличивается на 1** $\Rightarrow$ возможны три случая:

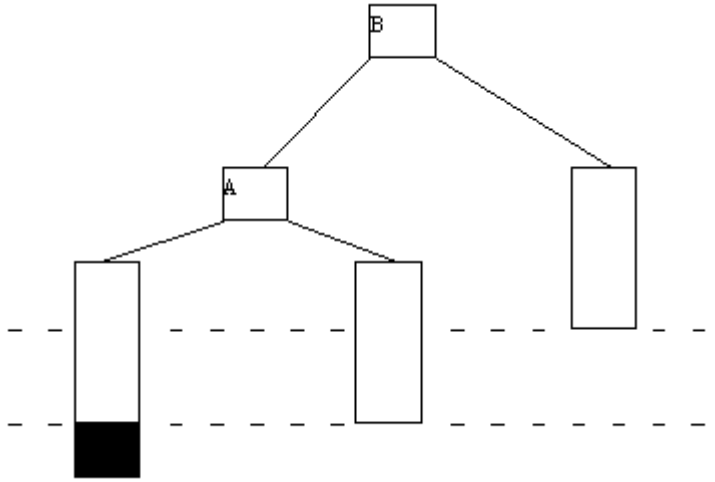
  (1) $h_L = h_R \Rightarrow$ после добавления вершины L и R станут разной высоты, но свойство сбалансированности сохранится

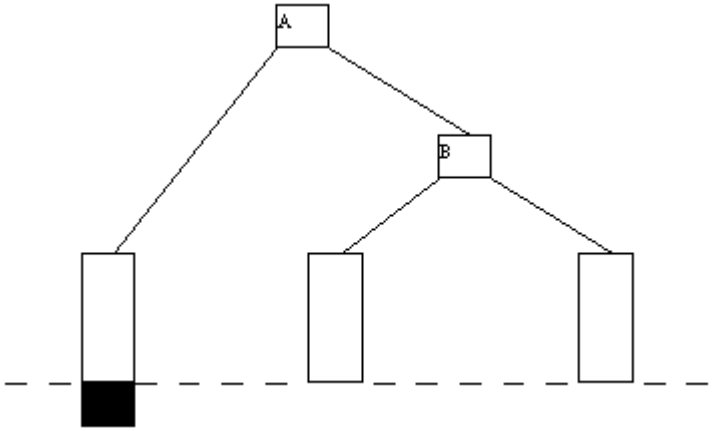
  (2) $h_L < h_R \Rightarrow$ после добавления новой вершины L и R станут равной высоты, т.е. сбалансированность общего дерева даже улучшится

  (3) $h_L > h_R \Rightarrow$ после включения ключа сбалансированность нарушится, и *потребуется перестройка дерева*.

Включение узла в АВЛ-дерево.

- (3а) **Новая вершина добавляется к левому поддереву поддерева** L. В результате поддерево с корнем в узле B разбалансировалось: разность высот его левого и правого поддеревьев стала равной –2 .



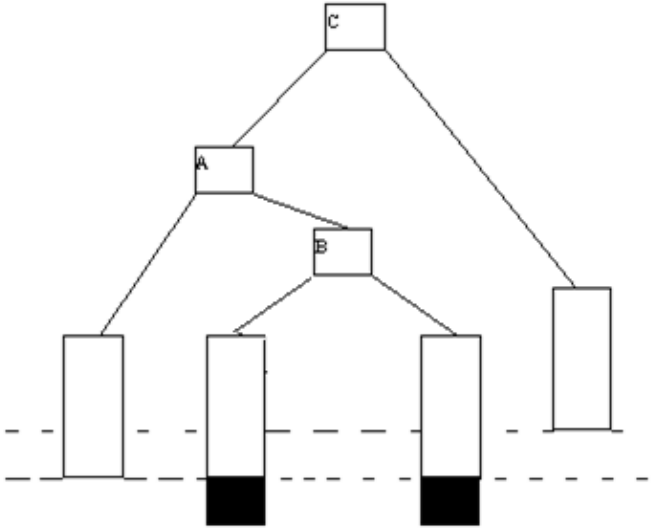


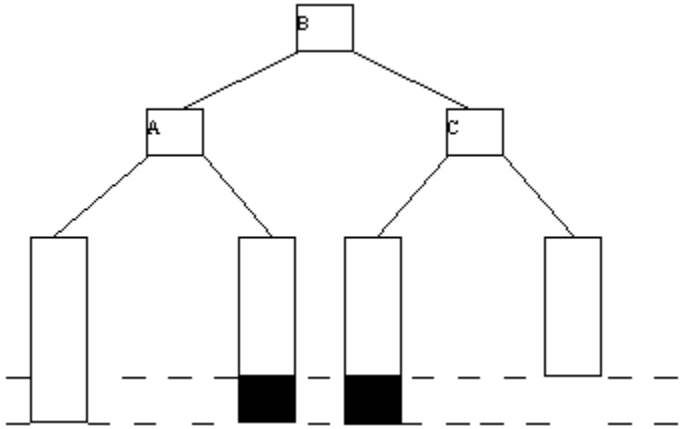
Преобразование, разрешающее ситуацию (3а) (однократный поворот $\boldsymbol{RR}$):

Делаем узел A корневым узлом поддерева, в результате правое поддерева с корнам в узле B «опускается» и разность высот становится равной –1

Включение узла в АВЛ-дерево.

- (3b) **Новая вершина добавляется к левому поддереву поддерева** L. В результате поддерево с корнем в C разбалансировалось: разность высот его левого и правого поддеревьев стала равной - 2.





Преобразование, разрешающее ситуацию (3b) (двукратный поворот LR):

«Вытягиваем» 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.

Построение АВЛ-дерева.

- Высота поддерева с корнем в узле P.

```
static int Height (Position P) {
      if (P == NULL)
        return -1;
      return P->Height;
}
```

- Выбор более длинного поддерева

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

Построение АВЛ-дерева.

- Однократные повороты
  - Между узлом и его левым сыном

Функция SingleRotateWithLeft вызывается только в том случае, когда у узла K2 есть левый сын. Функция выполняет поворот между узлом (K2) и его левым сыном, 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 */
}
```

Построение АВЛ-дерева.

- Однократные повороты
  - Между узлом и его правым сыном

Эта функция вызывается только в том случае, когда у узла K1 есть правый сын. Функция выполняет поворот между узлом (K1) и его правым сыном, 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 */
}
```

Построение АВЛ-дерева.

- Двойные повороты
  - LR- поворот

```
   Эта функция вызывается только тогда, когда
   у узла K3 есть левый сын, а у левого сына
   K3 есть правый сын. 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);
}
```

Построение АВЛ-дерева.

- Двойные повороты
  - *RL*- поворот

```
Эта функция вызывается только в том случае, когда у
узла K1 есть правый сын, а у правого сына узла K1
есть левый сын. 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);
}
```

Построение АВЛ-дерева.

- Вставить новый узел (все)

```
AvlTree Insert (KeyType X,
                AvlTree T) {
  if (T == NULL) {
    /* создание дерева с одним узлом */
    T = (AvlTree) malloc (sizeof
                 (struct AvlNode));
    if (!T)
      abort();
    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);
      }
  }
  else 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;
}
```

Построение АВЛ-дерева.

- Вставить новый узел

```
AvlTree Insert (KeyType X,
                AvlTree T) {
  if (T == NULL) {
    /* создание дерева с одним узлом */
    T = (AvlTree) malloc (sizeof
                 (struct AvlNode));
    if (!T)
      abort();
    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) {
```

- Вставить новый узел

```
if (X < T->Left->Element)
      T = SingleRotateWithLeft (T);
    else
      T = DoubleRotateWithLeft (T);
    }
  }
  else 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;
}
```

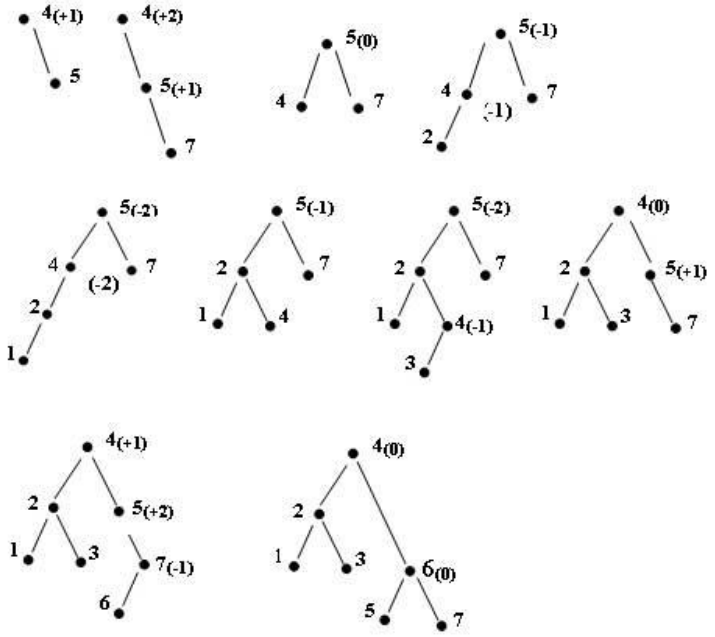
Построение АВЛ-дерева.

- ***Пример***

Пусть на «вход» функции **`Insert()`** последовательно поступают целые числа 4,5,7,2,1,3,6.

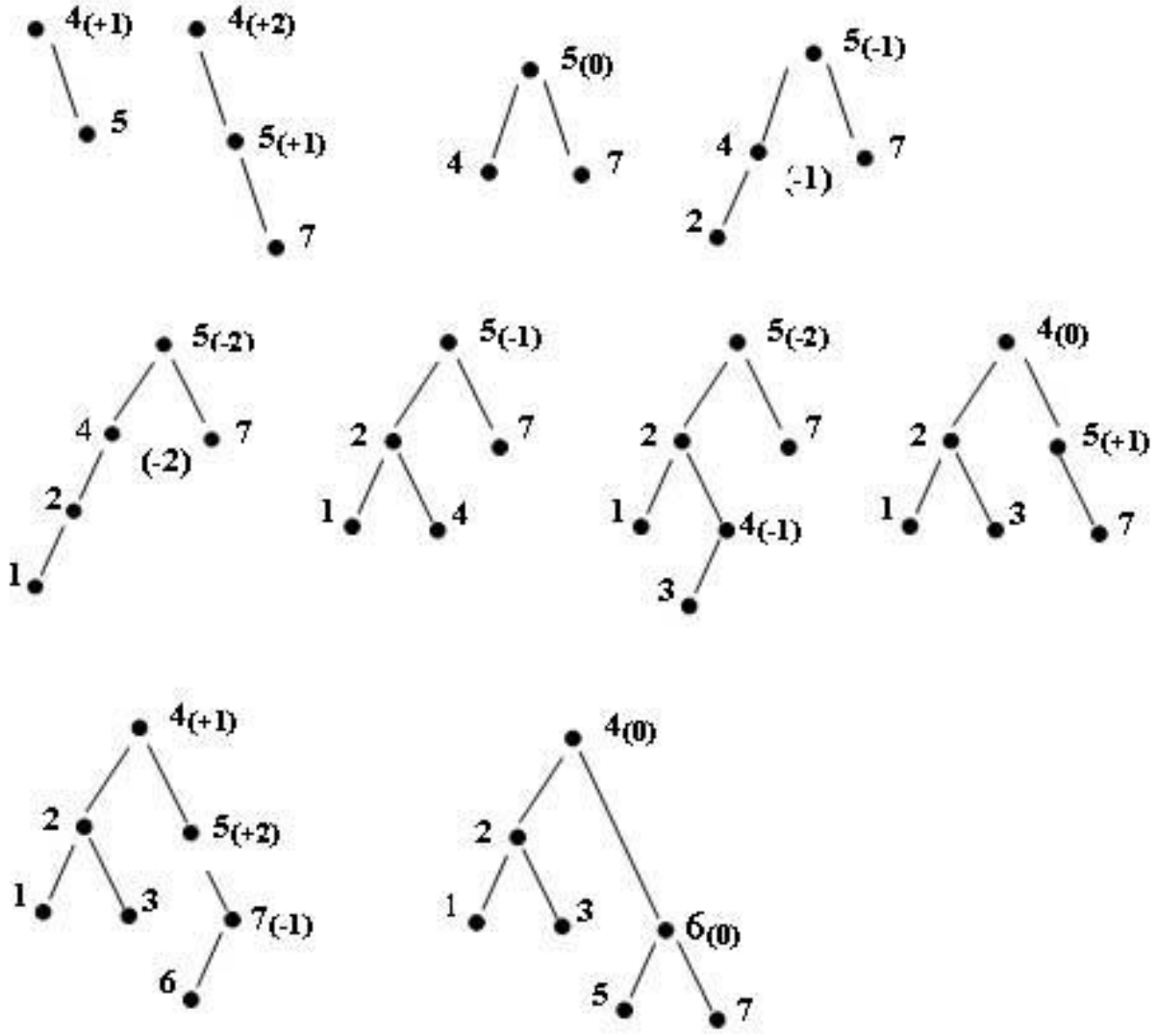
Изобразим процесс «роста» АВЛ-дерева (в скобках для части вершин указан показатель сбалансированности):

Числа в примере подобраны так, чтобы обеспечить как можно больше поворотов при минимальном числе включений.



Построение АВЛ-дерева.

- ***Пример построения АВЛ-дерева.***



Исключение узла из АВЛ-дерева

- Объявление функции:

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

- Исключение узла из АВЛ-дерева требует балансировки дерева.

  Иными словами в конец функции, выполняющей исключение узла, необходимо добавить вызовы функций:

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

- 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.

Оценка сложности

- Ранее были получены оценки высоты

(1) самого «хорошего» АВЛ-дерева, содержащего m узлов (*полностью сбалансированное дерево*)

$$h = O(\log_2(m+1))$$

(2) самого «плохого» АВЛ-дерева, содержащего m узлов (*дерево Фибоначчи*)

$$h \le 1.44 \cdot \log_2(m+1) - 0.32$$

Следовательно, для «среднего» АВЛ-дерева, содержащего m узлов оценка высоты будет где-то посередине:

$$\log_2(m+1) \le h \le 1.44 \cdot \log_2(m+1) - 0.32$$