Двоичные Б-деревья (ДБД) m=1

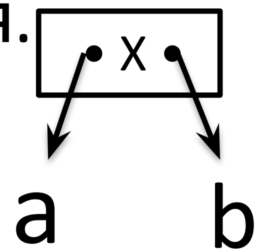
Б-деревья первого порядка
не имеет смысла использовать для представления больших множеств данных, требующих вторичной памяти.

Кроме неэффективного обращения к внешнему носителю, приблизительно половина страниц будут содержать только один элемент.

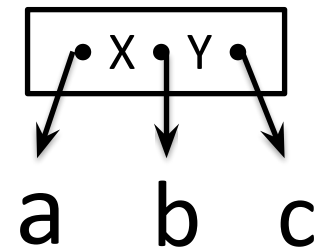
Поэтому забудем о внешней памяти и вновь займемся построением деревьев поиска, находящихся в оперативной памяти.

<u>Определение</u>. ***Двоичное Б-дерево*** состоит из страниц с одним или двумя элементами, страница содержит две или три ссылки на поддеревья.

<u>Пример:</u>

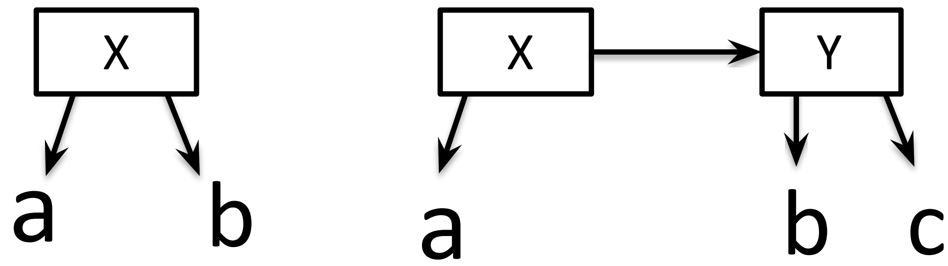


или



Так как имеем дело с оперативной памятью, то необходимо её эффективно использовать. Поэтому *представление страницы в виде массива уже не подходит.*

Решение – ***динамическое размещение на основе списочной структуры***. Страница - список из одного или двух элементов.



Так как каждая страница может иметь не более **трех** потомков (содержать не более трех ссылок), то попытаемся *объединить* *ссылки на потомков и ссылки внутри страницы* (вертикальные и горизонтальные).

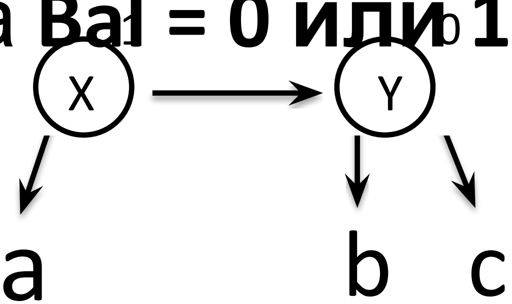
Тогда **страницы Б-дерева теряют свою целостность**. Элементы начинают играть роль вершин в двоичном дереве.

Однако,

1. Необходимо делать различия между горизонтальными и вертикальными ссылками;
2. Необходимо следить, чтобы все листья были на одном уровне.

Введем логические переменные **HR** и **VR** – горизонтальный и вертикальный рост дерева.

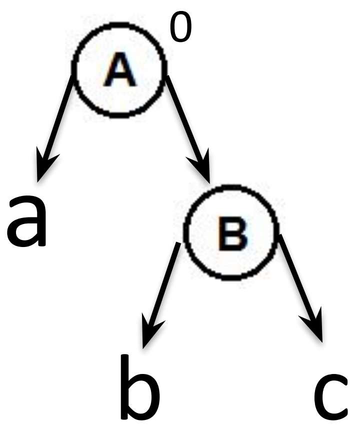
Показатель баланса **Bal = 0 или 1**

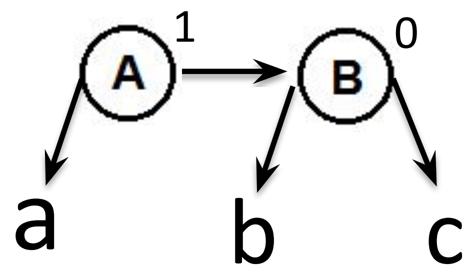


Bal помещаем в структуру дерева,

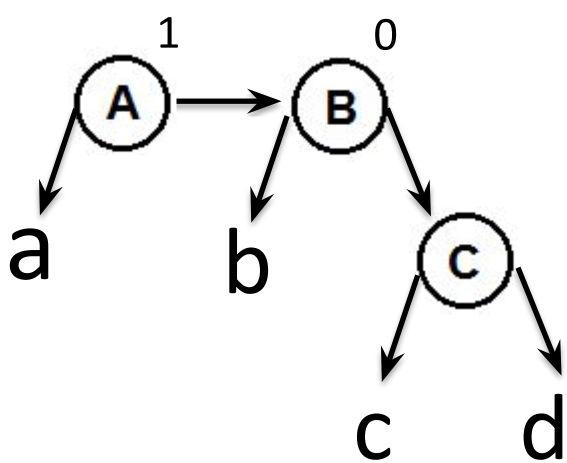
переменные **HR** и **VR** – глобальные.

Рассмотрим **добавление вершины в ДБД**.
Различают 4 возможных ситуации, возникающие при росте левых и правых поддеревьев

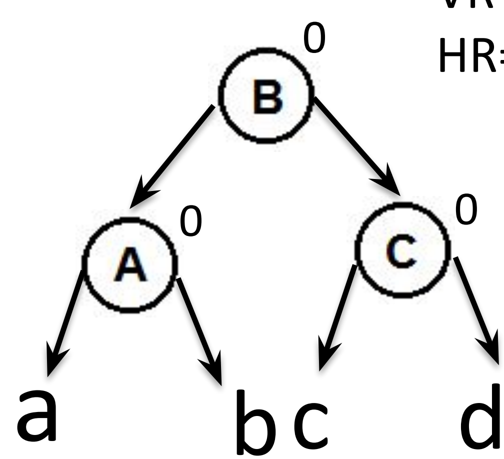


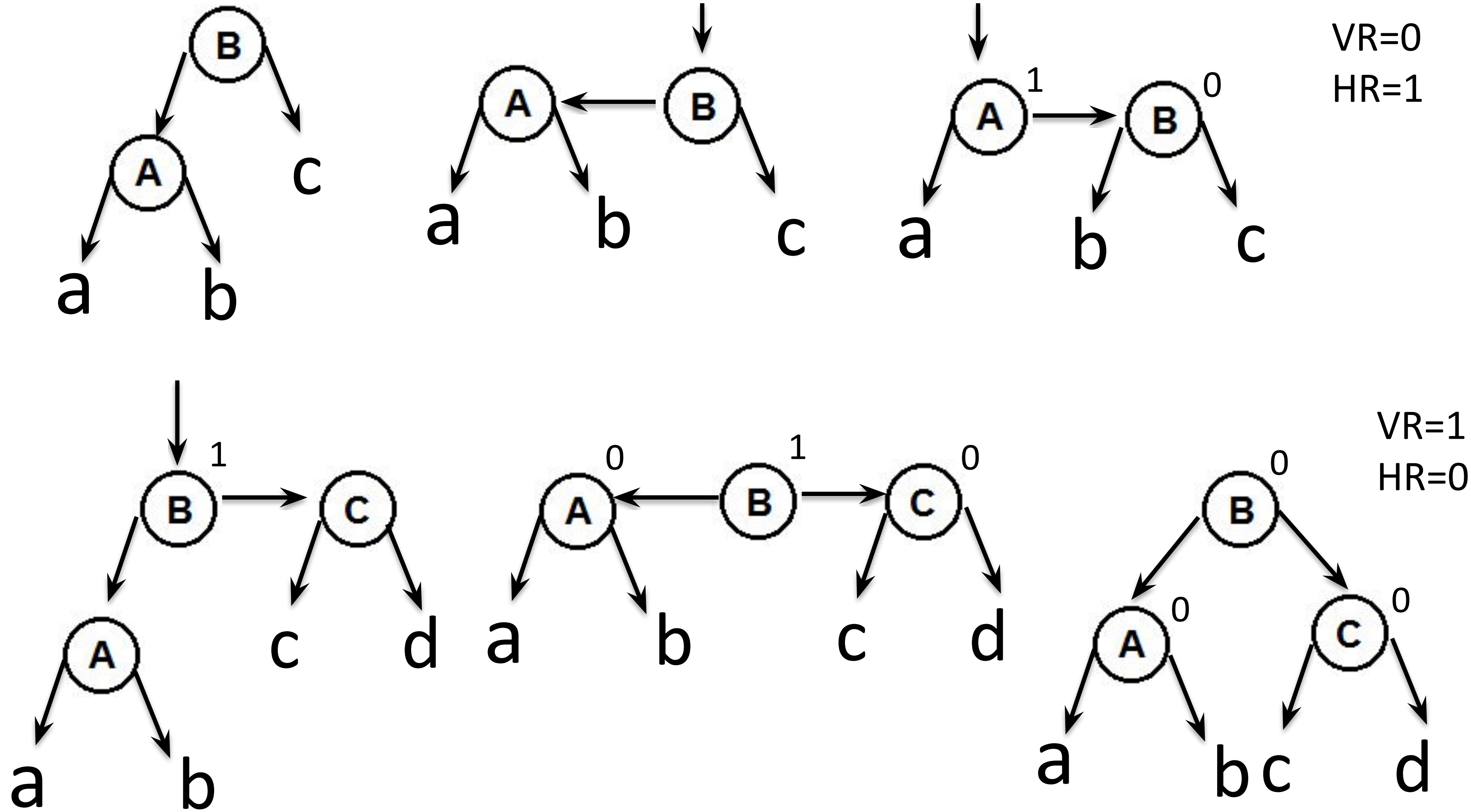


VR=0
HR=1



VR=1
HR=0





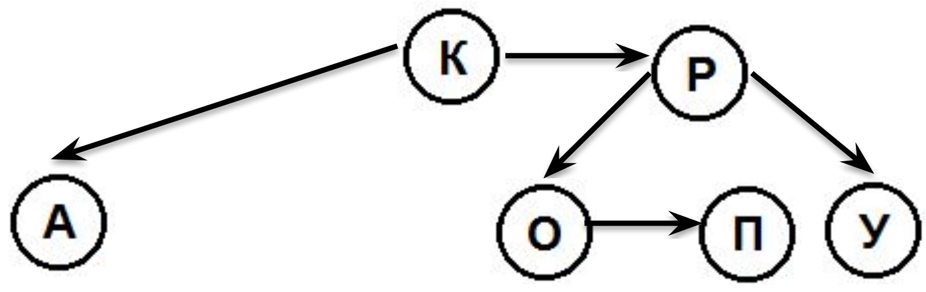
B
A
a
b
c
A
B
a
b
c
A 1
B 0
a
b
c
VR=0
HR=1
B 1
C
A
a
b
c
d
A 0
B 1
C 0
a
b
c
d
B 0
A 0
C 0
a
b
c
d
VR=1
HR=0

Алгоритм построения ДБД

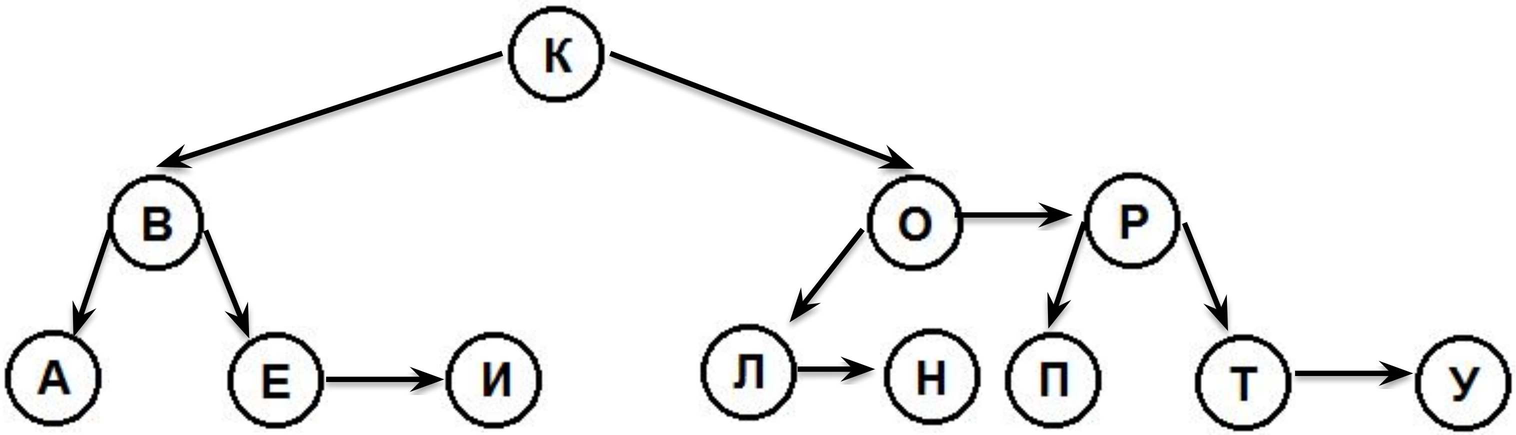
```
VR=1 HR=1
B2INSERT(D, Vertex *&p)
IF ( p=NULL )  <память по адресу p> ,   p-->Data = D,
                p-->Left = p-->Right = NULL,  p-->Bal = 0, VR = 1
ELSE IF ( p-->Data > D)  B2INSERT(D, p-->Left)
          IF ( VR=1 )
             IF (p-->Bal=0) q=p-->Left, p-->Left=q-->Right, q-->Right=p,
                              p=q, q-->Bal=1, VR=0, HR=1
             ELSE   p-->Bal=0, VR=1, HR=0
             FI
          ELSE   HR=0
          FI
```

```
ELSE IF (p-->Data<D) B2INSERT(D, p-->Right)
            IF (VR=1) p-->Bal=1, HR=1, VR=0
            ELSE  IF (HR=1)
                      IF(p-->Bal=1) q=p-->Right,  p-->Bal=0,
                                       q-->Bal=0, p-->Right=q-->Left,
                      q-->Left=p,  p=q, VR=1, HR=0
                      ELSE   HR=0
                      FI
                  FI
            FI
        FI
FI
```

К У Р А П О В Е Л Н И Т



К У Р А П О В Е Л Н И Т



Очевидно, что двоичные Б-деревья представляют собой **альтернативу критерию АВЛ-сбалансированности.**

Отметим некоторые отличия:

АВЛ-деревья – подмножество всех двоичных деревьев.

Следовательно, **класс ДБД шире**, а их длина пути поиска в среднем больше, чем у АВЛ-деревьев.

Высота двоичного Б-дерева:

$$h \le \frac{\log_2(n+1)-1}{\log_2(m+1)} + 1$$

При m=1:

$$h \le \frac{\log_2(n+1)-1}{\log_2(1+1)} + 1 = \log_2(n+1)$$

Длина пути ДБД может в два раза превышать высоту:

$$\mathbf{L \le 2 * \boldsymbol{log_2(n+1)}}$$

Для сравнения, **в плохом АВЛ-дереве**:

$$\mathbf{L \le 1{,}44 * \boldsymbol{log_2(n+2)}}$$

Однако, **при построении ДБД реже приходится переставлять вершины** (повороты выполняются лишь в двух случаях).

Поэтому ДБД предпочтительней, когда **чаще добавляются вершины**,

АВЛ-деревья предпочтительнее, когда **чаще производится поиск элементов**.

Кроме того, существует зависимость от особенностей реализации, поэтому

вопрос о применении того или иного типа деревьев **следует решать индивидуально** для каждой конкретной задачи.