3.8. ДВОИЧНЫЕ ДЕРЕВЬЯ

Деревья – один из способов организации данных в динамической памяти с целью быстрого поиска.

3.8.1. Основные определения

Определение (рекурсивное)

1. Одиночная вершина есть двоичное дерево.

2. Двоичное дерево – это вершина (V), соединенная с (возможно, пустыми) левым (T_L) и правым (T_R) двоичными деревьями.

Пример двоичного дерева

Кружочками обозначены вершины дерева, стрелками - связи между вершинами.

Каждая вершина дерева может содержать какую-либо информацию.

Начальная вершина называется *корнем*.

Оконечные вершины, не имеющие поддеревьев, называются *листьями*.

Словарь

- tree [три] – дерево
- root [рут] – корень
- vertex [вётэкс] – вершина
- right [райт] – правый
- left [лэфт] – левый

3.8.2. Некоторые свойства деревьев

Свойство 1:

Максимальное число вершин в двоичном дереве высоты h равно

$$n_{max}(h) = 2^h - 1$$

Доказательство:

на первом уровне $1 = 2^0$ вершин

на втором уровне $2 = 2^1$ вершин

на третьем уровне $4 = 2^2$ вершин

на h уровне 2^{h-1} вершин

$$n_{max} = 1 + 2 + \ldots + 2^{h-1} = 2^h - 1$$

Свойство 2:

Минимально возможная высота двоичного дерева с n вершинами равна

$$h_{min}(n) = \lceil \log(n+1) \rceil$$

Доказательство:

Из свойства 1 имеем $h = \log (n_{max} + 1)$

Определение

Двоичное дерево называют идеально сбалансированным (ИСД), если для каждой его вершины размеры левого и правого поддеревьев отличаются не более чем на 1.

ИСД сбалансировано по количеству вершин.

Пример

Свойство 3:

Высота ИСД с n вершинами минимальна.

$$h_{\text{исд}(n)} = h_{min}(n) = \lceil \log(n+1) \rceil$$

3.8.3. Представление деревьев в памяти компьютера

Каждая вершина содержит данные и указатели на вершину слева и справа. В качестве заголовка для дерева используем переменную Root, указывающую на корень.

Структура вершины дерева

```
struct  Vertex
    { int Data;
    Vertex * Left;
    Vertex * Right;
    } ;
Vertex * Root;
```

Графическое представление

3.8.4. Основные операции с деревьями

Существует много работ, которые можно выполнять с деревьями.

Например, посадка, подкормка, подстрижка, полив, окучивание и т.п.

Распространенная задача – выполнение некоторой **определенной операции** с каждой вершиной дерева.

Для этого необходимо «посетить» все вершины дерева, или, как обычно говорят, сделать **обход дерева**.

Основные операции с деревьями

Определение. *Обход дерева* – выполнение некоторой операции с каждой его вершиной.

Существуют

три основных *порядка обхода дерева*:

1. Сверху вниз (↓): корень, левое поддерево, правое поддерево.
2. Слева направо (→): левое поддерево, корень, правое поддерево.
3. Снизу вверх (↑): левое поддерево, правое поддерево, корень.

Обходы легко программируются с помощью рекурсивных процедур.

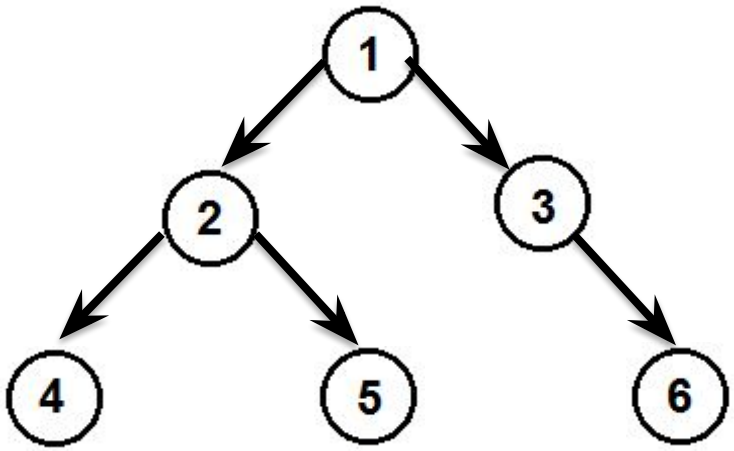
Пример. Процедура обхода дерева сверху вниз.

```
void Obhod1 ( Vertex *p )
    IF ( p!=NULL )
      < печать (p->Data) >
       Obhod1 ( p->Left )
       Obhod1 ( p->Right )
    FI
```

Вызов процедуры: **Obhod1 (Root)**

Чтобы изменить порядок обхода, нужно поменять местами операторы внутри функции.

Пример. Обходы дерева



Корень, левое, правое.

Левое, корень, правое.

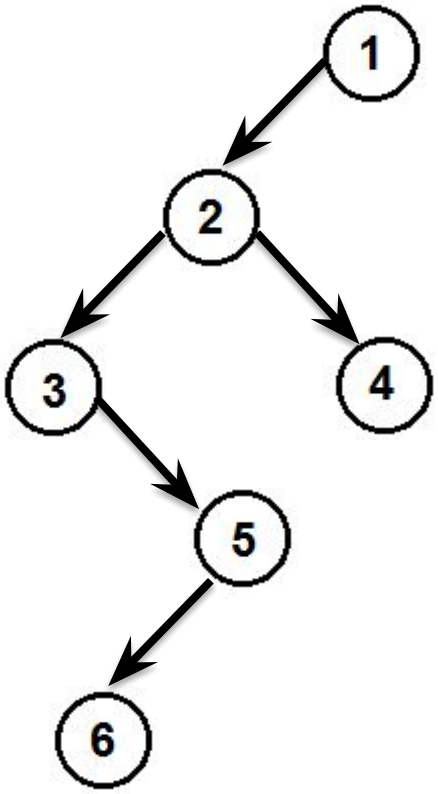
Левое, правое, корень.

(↓): 1 2 4 5 3 6

(→): 4 2 5 1 3 6

(↑): 4 5 2 6 3 1

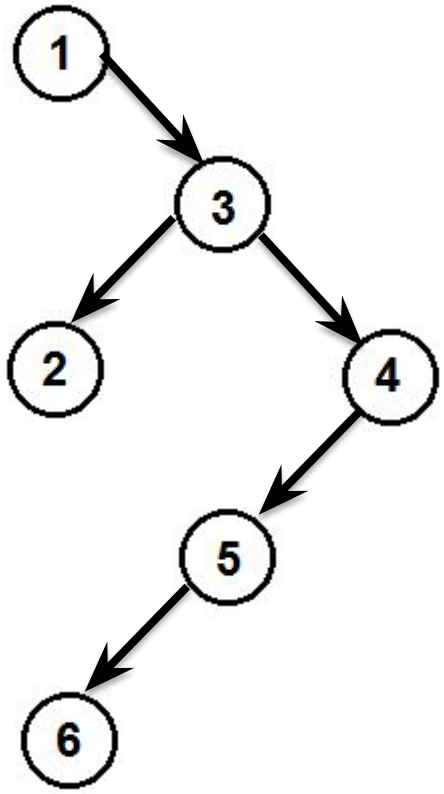
Максимальная глубина рекурсии при обходе = 4



(↓): 1 2 3 5 6 4

(→): 3 6 5 2 4 1

(↑): 6 5 3 4 2 1



(↓): 1 3 2 4 5 6

(→): 1 2 3 6 5 4

(↑): 2 6 5 4 3 1

3.9. Деревья поиска

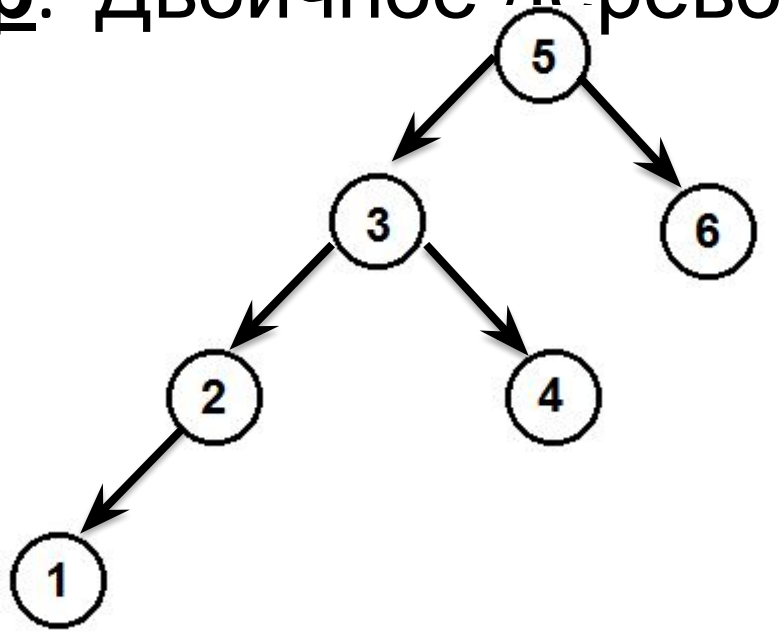
Двоичные деревья часто используются для представления данных, среди которых идет **поиск элементов по уникальному ключу**.

Будем считать, что:

1) часть данных в каждой вершине является ключом поиска;
2) для всех ключей определены операции сравнения (<,>,=);
3) в дереве нет элементов с одинаковыми ключами.

Определение. Двоичное дерево называется деревом поиска, если ключ в каждой его вершине больше ключей в левом поддереве и меньше ключей в правом поддереве.

Пример. Двоичное дерево поиска.



3.9.1. Поиск вершины с ключом X

Начиная с корневой вершины дерева, *сравниваем* **ключ поиска** с данными **в текущей вершине**.

Если **ключ** поиска **меньше**, то переходим в **левое поддерево**, если **ключ** поиска **больше**, то переходим в **правое поддерево**.

Действуем аналогично, **пока не будет найден элемент** с заданным ключом **или листовая вершина** дерева.

Если **достигнута листовая вершина**, то искомого элемента **нет в дереве**.

Поиск вершины с ключом X

Алгоритм на псевдокоде

```
p := Root
DO (p != NULL)
    IF (X < p->Data)  p := p->Left
    ELSE  IF (X > p->Data)  p := p->Right
              ELSE  OD
         FI
    FI
OD
IF (p != NULL) <вершина найдена по адресу p>
ELSE   <вершины нет дереве>
FI
```

Трудоемкость поиска по дереву

Максимальное количество сравнений при поиске: $C_{max} = 2h$

Идеально сбалансированное дерево:

$$C_{max} = 2 \lceil \log(n+1) \rceil$$

Будем считать, что все вершины ищутся одинаково часто.

Тогда **идеально сбалансированное дерево поиска (ИСДП)** обеспечивает **минимальное среднее время поиска:**

$$T = O(\log_2 n)$$

Построение ИСДП

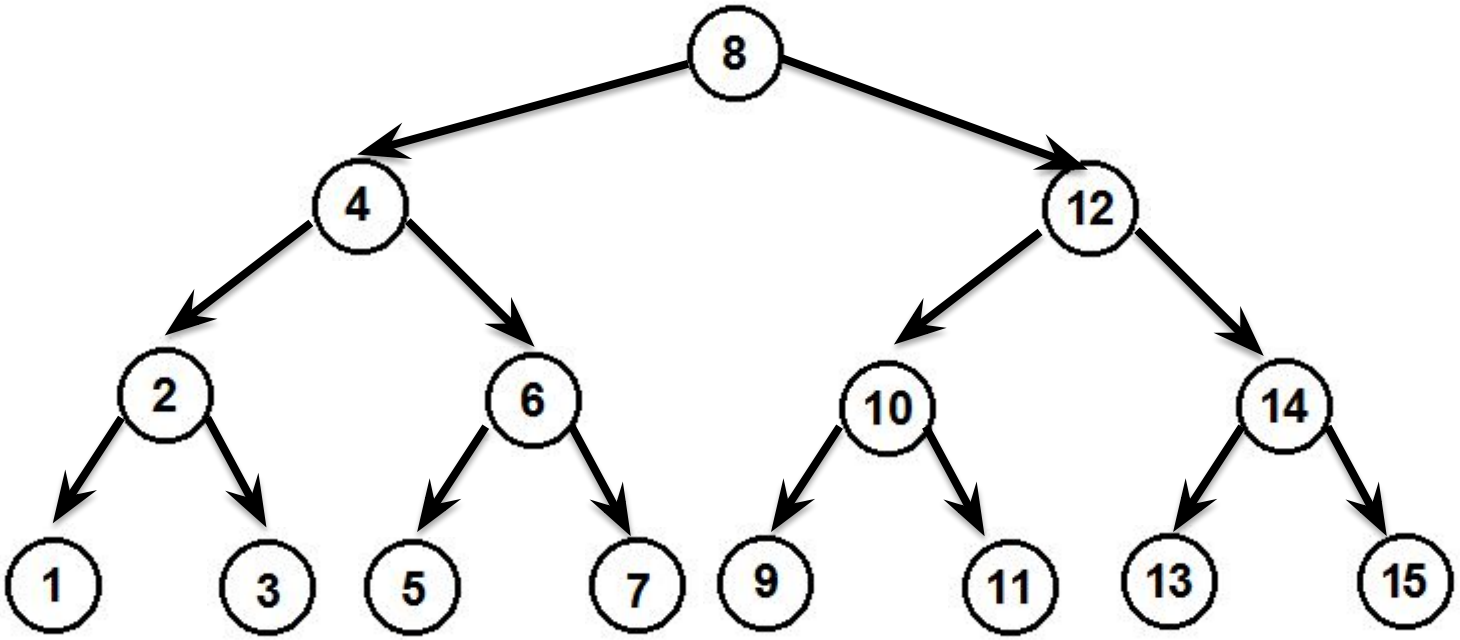
из элементов массива A = (a_1, a_2 ,…, a_n):

1. Отсортировать массив по возрастанию.
2. Построить ИСДП, пользуясь свойством:

   Если дерево идеально сбалансировано, то все его поддеревья тоже идеально сбалансированы.

Идея построения ИСДП: В качестве корня возьмем средний элемент упорядоченного массива, из меньших элементов строим левое поддерево, из больших – правое поддерево.

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15



Построение ИСДП

Алгоритм на псевдокоде

Vertex* ISDP (L,R)

IF (L>R) return NULL;

ELSE m := $\lceil (L+R)/2 \rceil$

<выделение памяти по адресу p>

p->Data := A[m]

p->Left := **ISDP** (L, m-1)

p->Right := **ISDP** (m+1, R)

return **p**

FI

В реальности **количество** элементов данных ***заранее неизвестно*** и они **поступают** последовательно ***в произвольном порядке***.

Требуется строить деревья поиска путем ***добавления новых вершин***, так же необходимо предусмотреть ***удаление вершин***.

Все операции могут ***чередоваться с поиском*** и должны выполняться как можно **быстрее**.

Решение этих задач мы будем

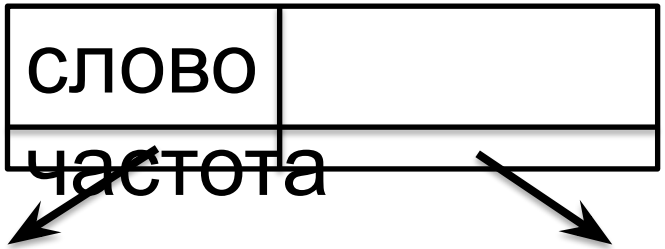
Случайные деревья поиска.

Все **преимущества деревьев** реализуются именно тогда, когда ***меняется их структура*** в ходе выполнения программы.
Рассмотрим случай, когда дерево **только растет**.
Пример – построение словаря частот встречаемости слов в тексте.

Каждое слово надо **искать** в дереве. Если его нет, то слово ***добавляется*** с частотой, равной ***1***. Если слово найдено в дереве, то ***увеличиваем*** частоту на ***1***.

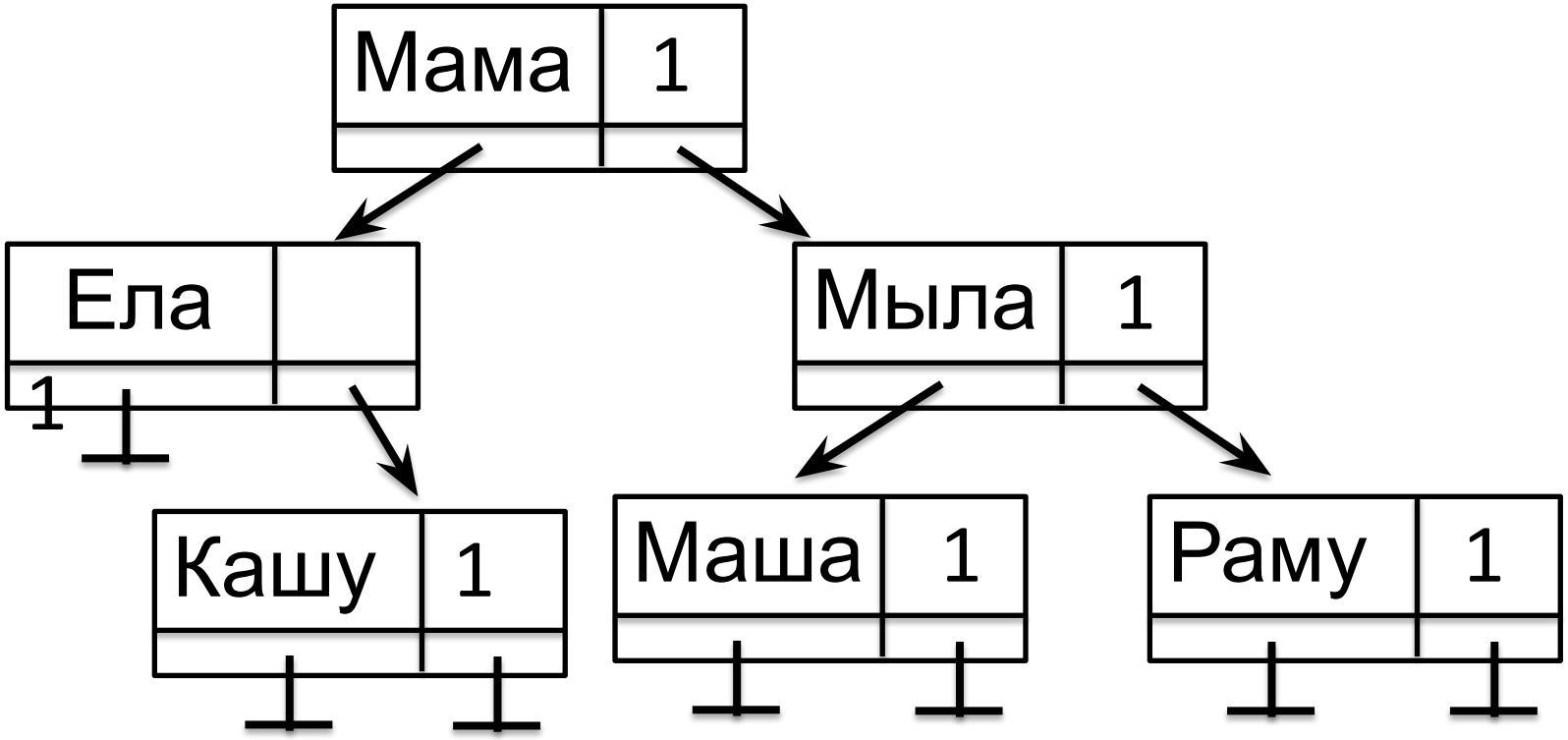
Эту задачу часто называют **поиском по дереву с включением**.

Форма дерева определяется ***случайным порядком поступления элементов***.



Пример:

Мама мыла раму, Маша ела кашу.



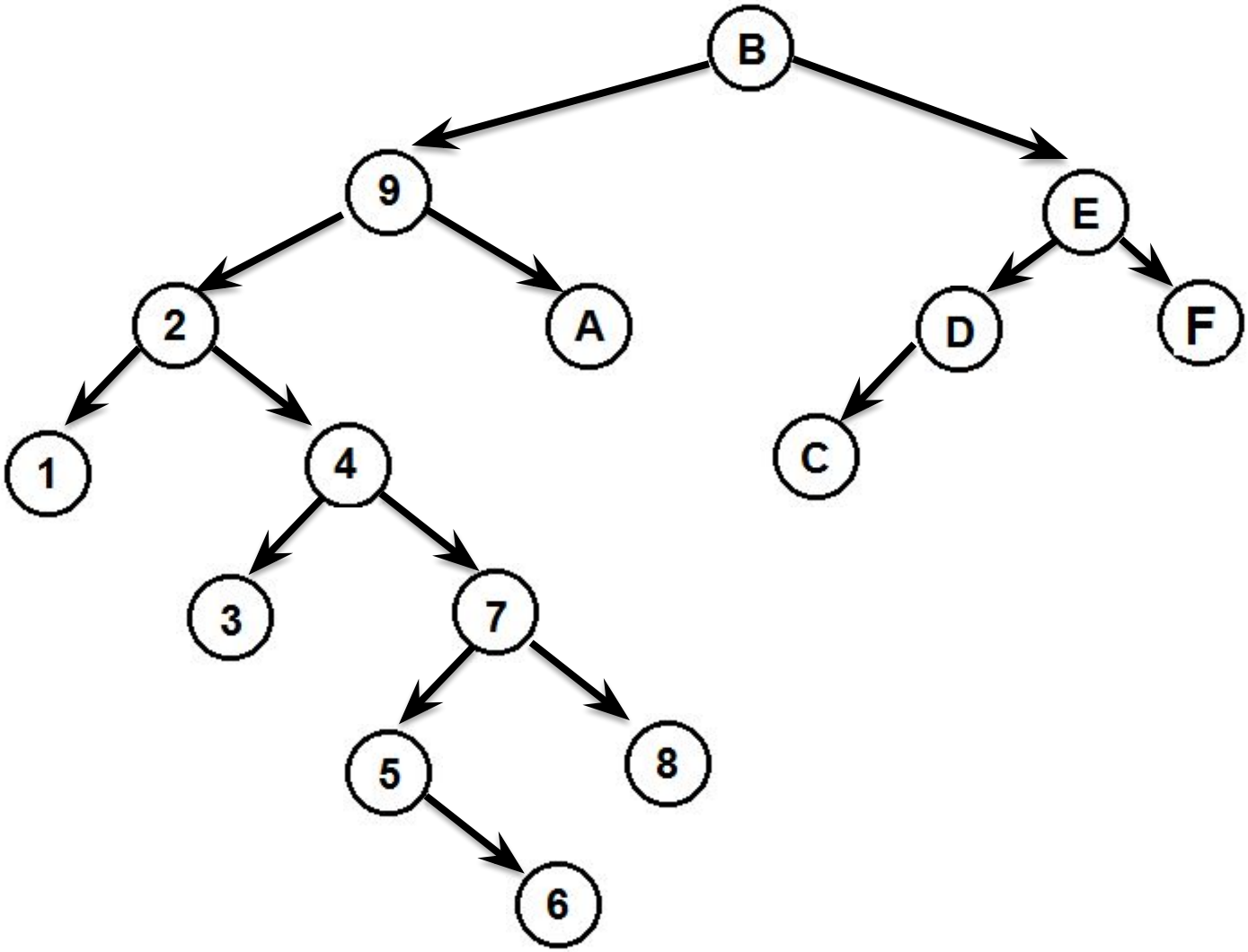
Построение СДП

Идея: построение выполняется путем ***добавления новых вершин в дерево***.

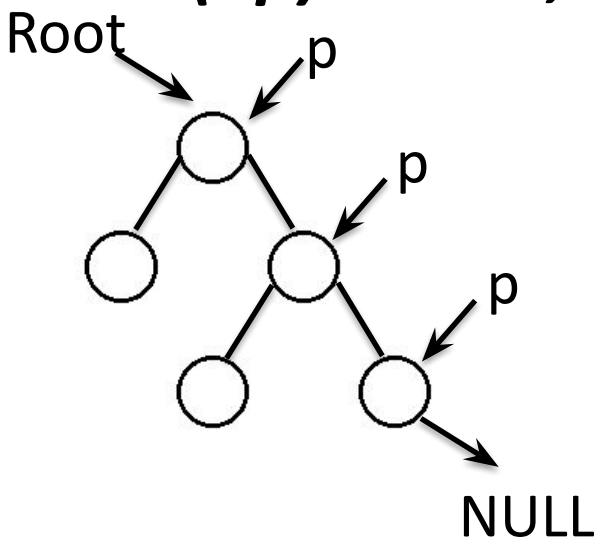
Если ***дерево пустое***, то **создать вершину** (распределить память) и записать в неё данные. Указатели **Left** и **Right** обнуляются.

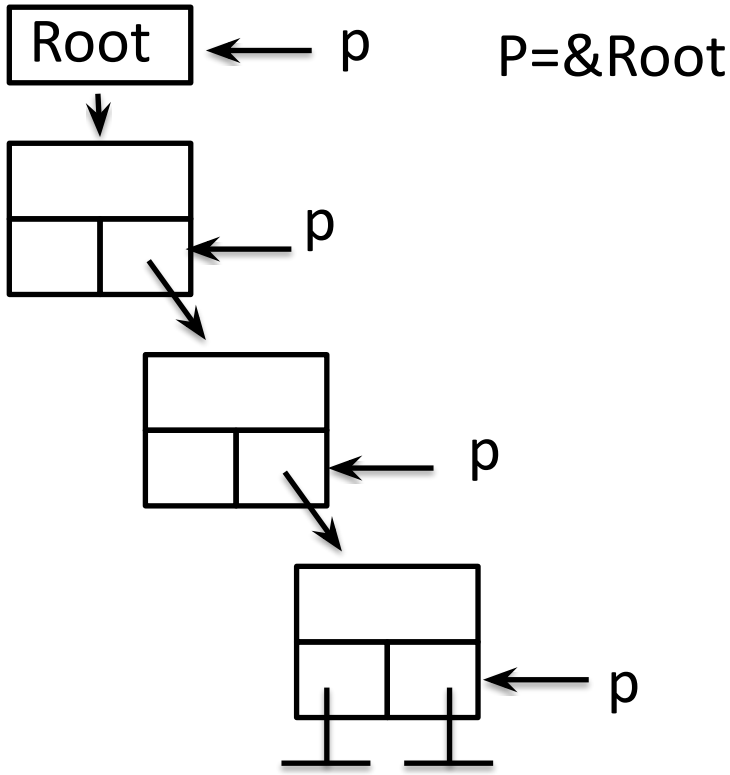
Если ***дерево не пустое***, то вершина **добавляется к левому или правому поддереву** в зависимости от соотношения с данными текущей вершины.

B 9 2 4 1 7 E F A D C 3 5 8 6



При создании новой вершины **нужно изменить** значение указателя на неё, поэтому нам нужен ***указатель на указатель (*****двойная косвенность*****)***: ***Vertex**p;*** Обращение к данным ***(*p)->Data;***





Обозначения: ***Root*** - корень, ***D*** – данные,
p - указатель на указатель

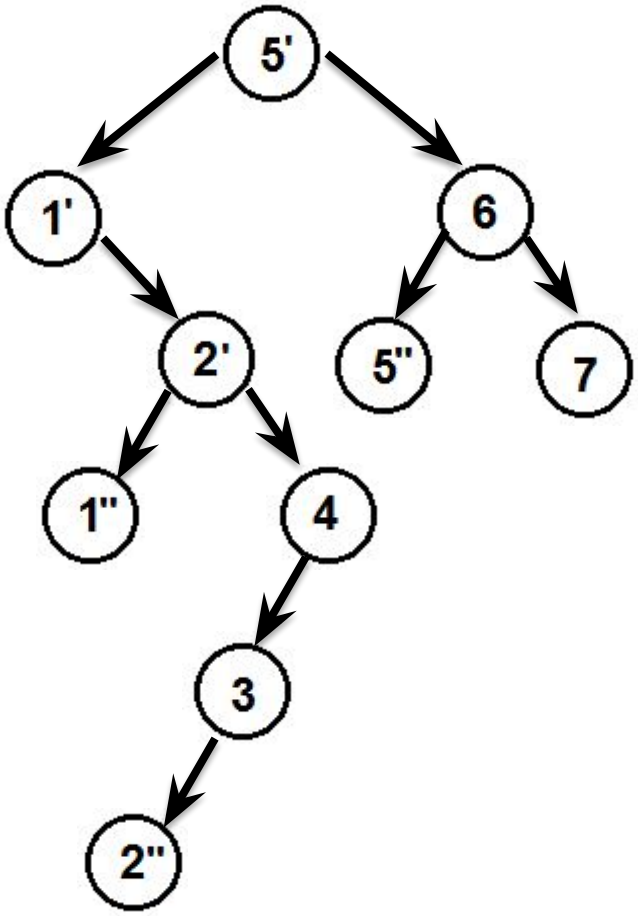
Добавить (данные **D** в дерево с корнем **Root**)

```
p=&Root
DO(*p!=NULL)     // поиск элемента
      IF (D<(*p)->Data)   p=&((*p)->Left)
      ELSE  IF (D>(*p)->Data)   p=&((*p)->Right)
            ELSE  OD  {данные уже есть в дереве}
         FI
      FI
OD
IF (*p=NULL)
    память(*p), (*p)->Data=D;
    (*p)->Left=NULL; (*p)->Right=NULL;
FI
```

Хотя назначение этого алгоритма - **поиск с включением**, его можно использовать ***и для сортировки.***

Если мы хотим **сортировать данные с помощью двоичного дерева**, то одинаковые элементы нужно добавлять **вправо** для сохранения устойчивости сортировки.

5’ 1’ 2’ 4 3 2” 1” 6 5” 7

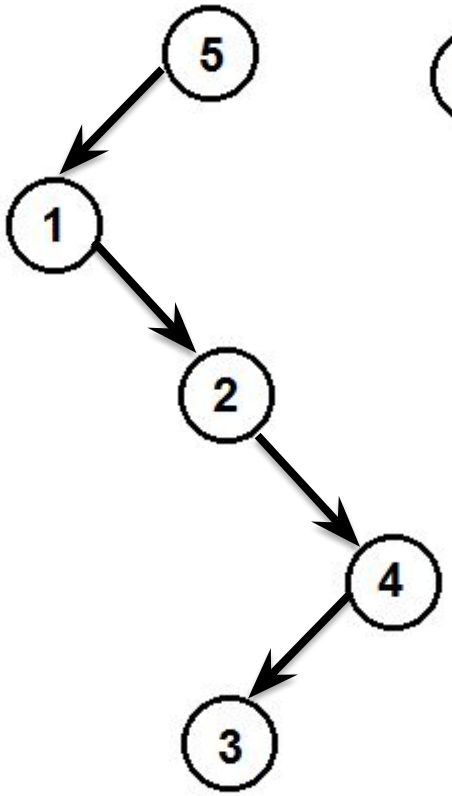


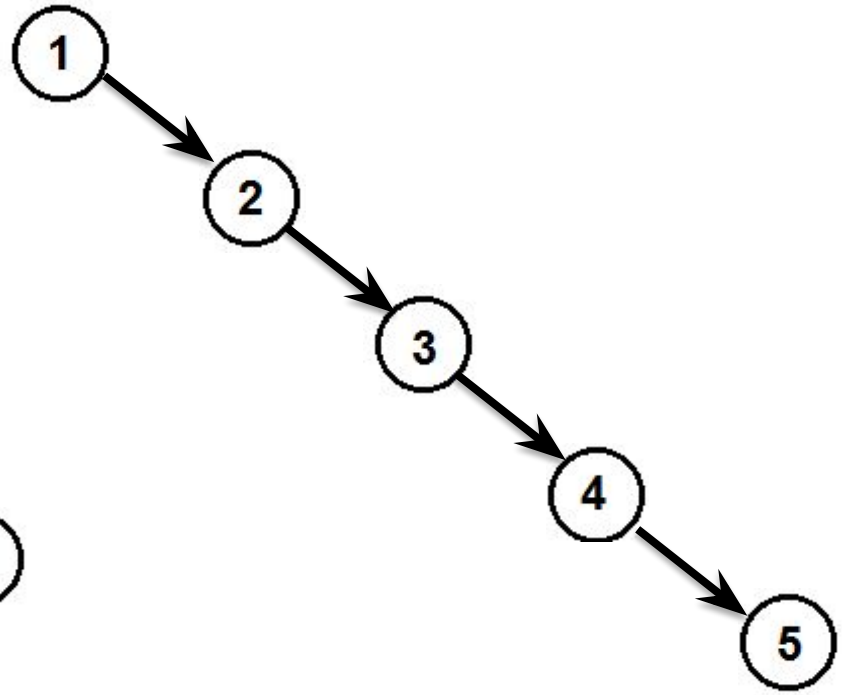
1’ 1” 2’ 2” 3 4 5’ 5” 6 7

Случайное дерево быстро строится, но его **недостаток**: оно может слишком вытянуться, ***в худшем случае выродиться в список***.

1 2 3 4 5

5 1 2 4 3





Максимальная высота дерева: $h_{max} = n$

Средняя высота дерева:

$$h_{ср.} = \frac{\text{сумма длин путей к каждой вершине}}{\text{количество вершин}}$$

$$h_{ср.(СДП)} = \frac{1+2+2+3+3+3+3+4+4+4+5+5+6+6}{15} = \frac{58}{15} = \mathbf{3,86}$$

$$h_{ср.(ИСДП)} = \frac{1+2+2+3*4+4*8}{15} = \frac{49}{15} = \mathbf{3,28}$$

Н. Вирт доказал:

$h_{ср.(ИСДП)} = \log(n)$ при $n \to \infty$

$h_{ср.(СДП)} = 2*\ln(n)$ при $n \to \infty$

$$\lim \frac{hcp.(СДП)}{hcp.(ИСДП)} = \frac{\log(n)}{2*\ln(n)} = 2*\ln 2 = 1,386$$

т. е. **средняя высота СДП хуже ИСДП на 39%**

Рекурсивная процедура добавления в случайное дерево поиска

Добавить рекурсивно (D, Vertex***&**p)

IF (p=NULL)

память (p), p->Data=D,

p->Left=NULL, p->Right=NULL

ELSE IF (D< p->Data)

Добавить рекурсивно(D, p->Left)

ELSE IF (D> p->Data)

Добавить рекурсивно(D, p->Right)

ELSE <Вершина есть в дереве>

FI

Вызов процедуры:

Добавить рекурсивно (D, root)