Деревья

Дерево – это **нелинейная структура данных**, используемая при представлении иерархических связей, имеющих отношения «один ко многим».

Терминология (взята из ботаники и генеалогии)

Дерево **– это совокупность элементов, называемых *узлами* или *вершинами*, и отношений («родительских») между ними, образующих иерархическую структуру узлов.**

Отношения между узлами дерева (из генеалогии):

верхний узел (вершина) называется ***родителем (предком)***,

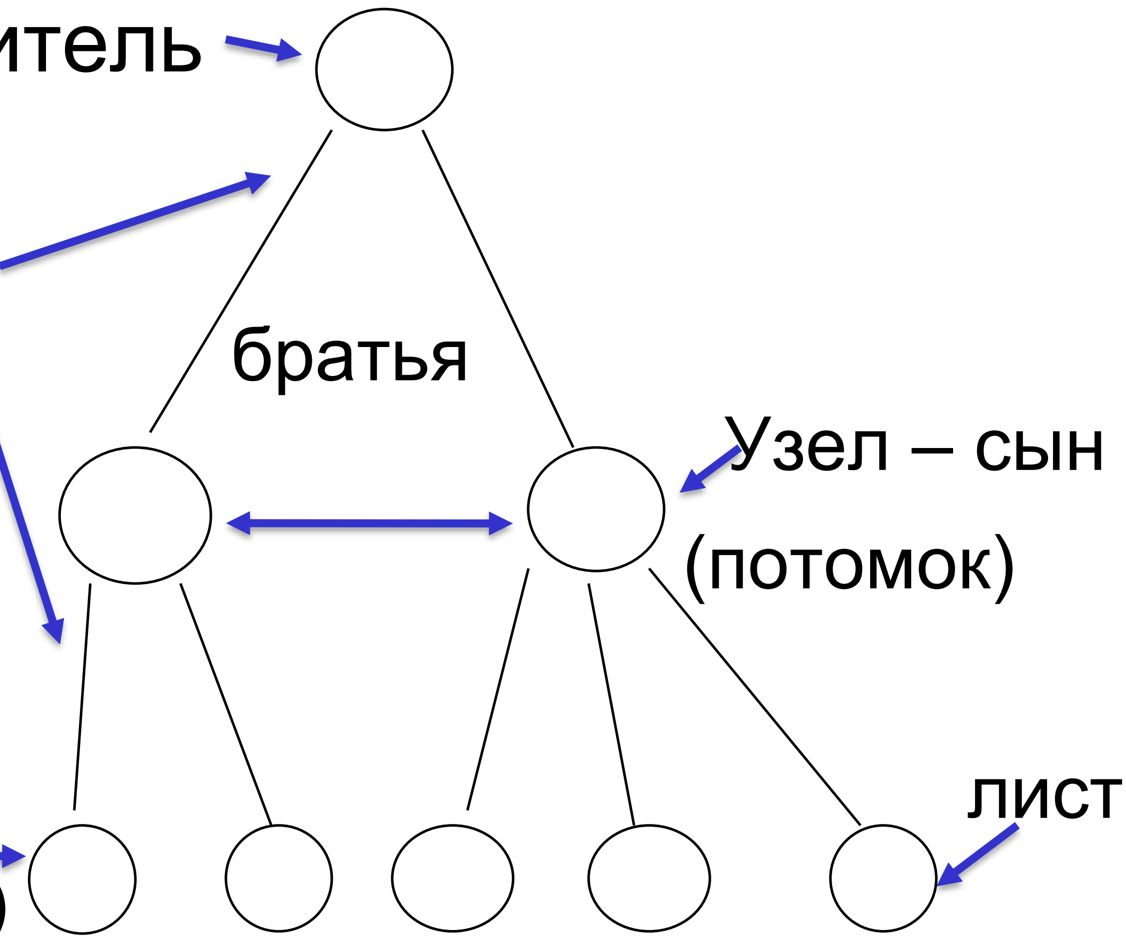
нижний – ***потомком (сыном или дочерней вершиной).***

Определения узлов – из ботаники.

Самая верхняя вершина - ***корень***, а самые нижние вершины – ***листья.*** Вершины, не имеющие потомков, - ***терминальные (***через отношения) или ***листья*** (через определения). Нетерминальные вершины - ***внутренние*****.**

Дерево через отношения и определения

- Корень – родитель
-
- терминальная
- вершина (лист)



Деревья определяются **рекурсивно**,

т. е., ***дерево с базовым типом Т*** – это:

- либо пустая структура (пустое дерево, **один корень)**;
- либо узел типа **Т** с конечным числом древовидных структур этого же типа **Т**, называющихся ***поддеревьями***.

Т.о. - дерево без ветвей с одной вершиной – это ***пустое*** или ***нулевое*** дерево.

Уровни:

Корень дерева лежит на ***нулевом*** уровне.

Максимальный уровень какой-либо вершины дерева - ***глубина*** **(**от корня до узла***)*** или ***высота*** **(**от узла до максимально удаленного листа***).***

Отсюда макс. уровень корня = 0.

Максимальный уровень всех вершин называется ***глубиной дерева.***

Число непосредственных потомков у вершины (узла) дерева называется ***степенью вершины (узла).***

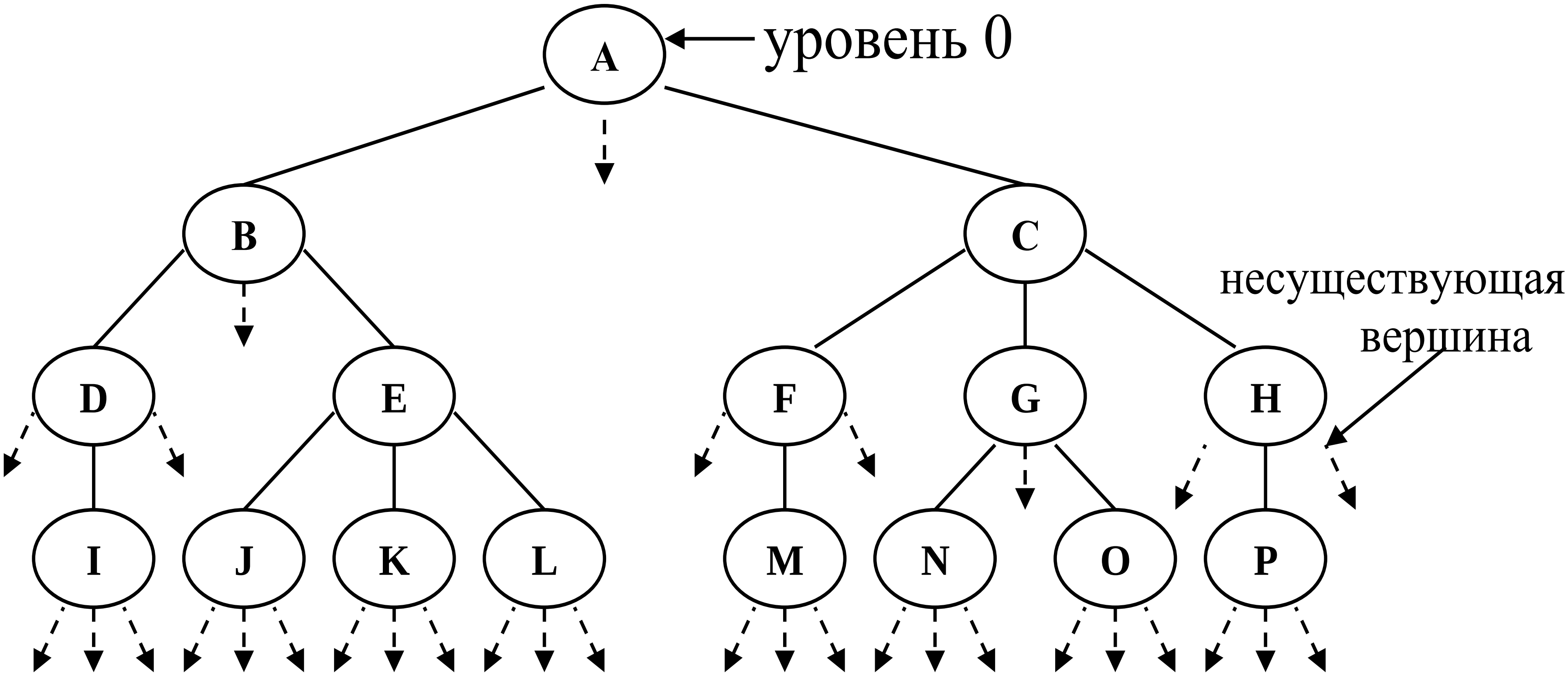
Максимальная степень всех вершин является ***степенью дерева.***

Длина пути

- **Число ветвей от корня к вершине** есть ***длина пути*** к этой вершине.
- Т. о., корень имеет длину пути, равную 0, длина пути его прямых (т. е., связанных с ним одной ветвью) потомков равна 1 и т.д. Вершина на уровне **i** имеет длину пути **i**.
- ***Длина внутреннего пути дерева*** – это **сумма длин путей для каждой его вершины.**

- ***Длина внешнего пути дерева*** – это **сумма длин путей всех специальных вершин, которые дополняют дерево так, чтобы степени всех вершин были равны степени дерева. Д**лина внешн. пути дерева:
- $\sum_{i=2}^{n}$ = **(Vi * max степень *** h**vi) + V1 * степень**,
- где **n** – количество вершин, **Vi – i**-тая вершина,
- h**vi** – глубина **i**-той вершины.

Пример:



Глубина дерева = 3.
Максимальная степень дерева = 3.
Длина внутреннего пути дерева равна 36.
Длина внешнего пути дерева равна 120.

Представление древовидной структуры

а) скобочное (в выражениях):
(A(B(D(I),E(J,K,L)), C(F(O),G(M,N),H(P))))

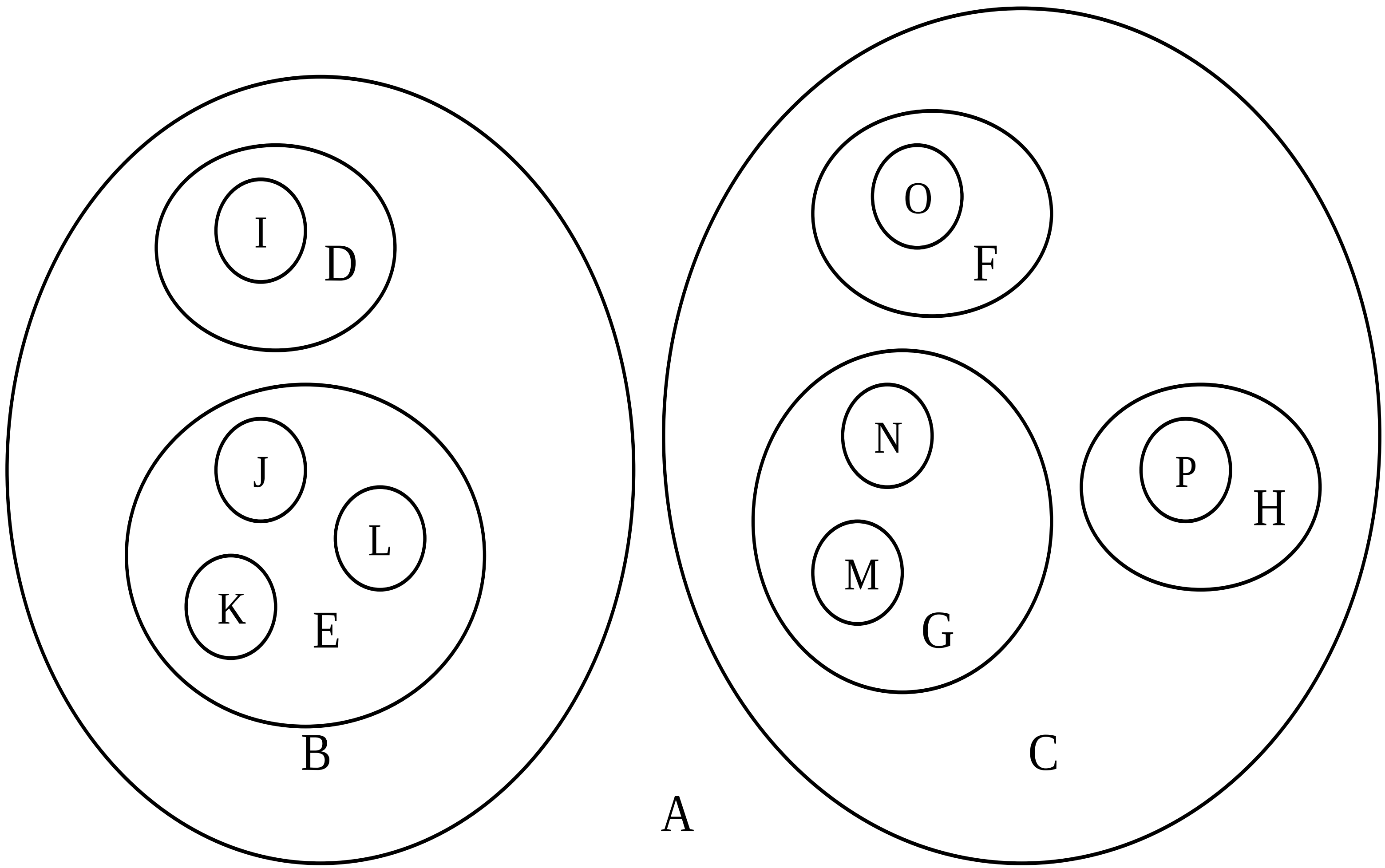
б) в виде вложенных множеств

в) отступами (в программах – структура)

г) в виде графа

Представление деревьев

а) <u>в виде вложенных множеств</u>:



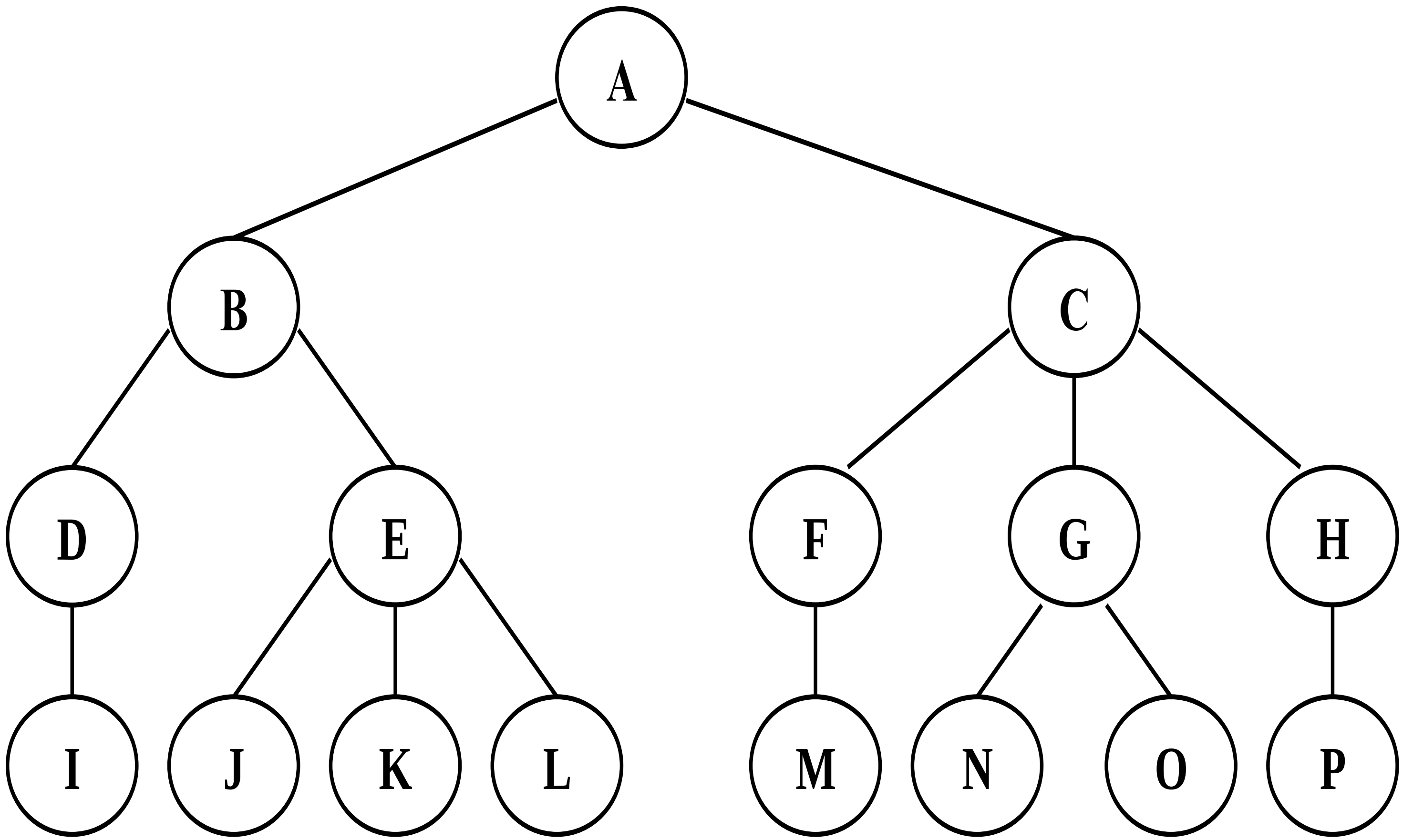
Представление деревьев

в) отступами (в программах – структура)

A			
	B		
		D	
			I
		E	
			J
			K
			L
	C		
		F	
			O
		G	
			M
			N
		H	
			P

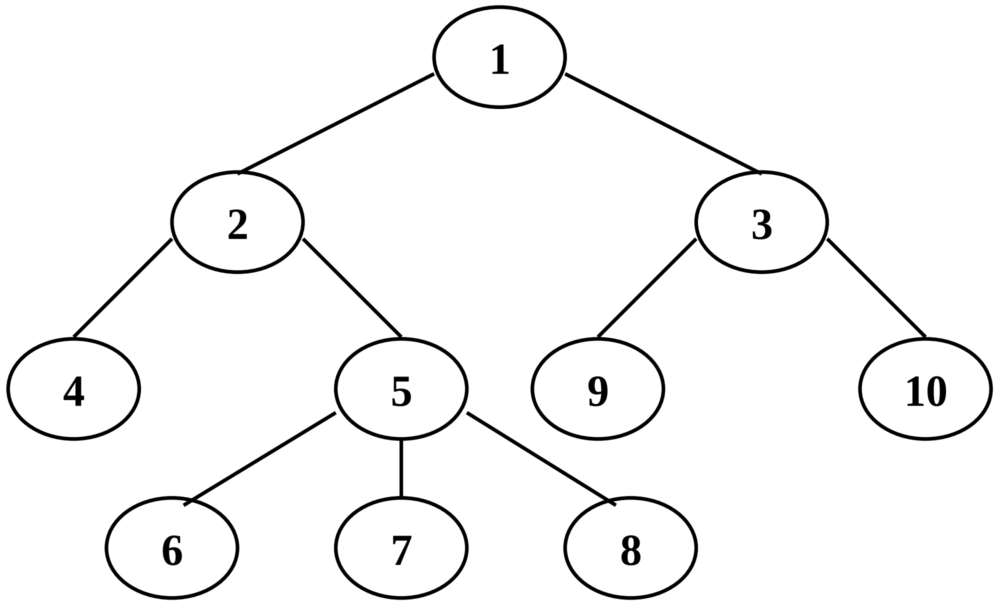
Представление деревьев

г) <u>в виде графа</u>:



Представление деревьев в памяти

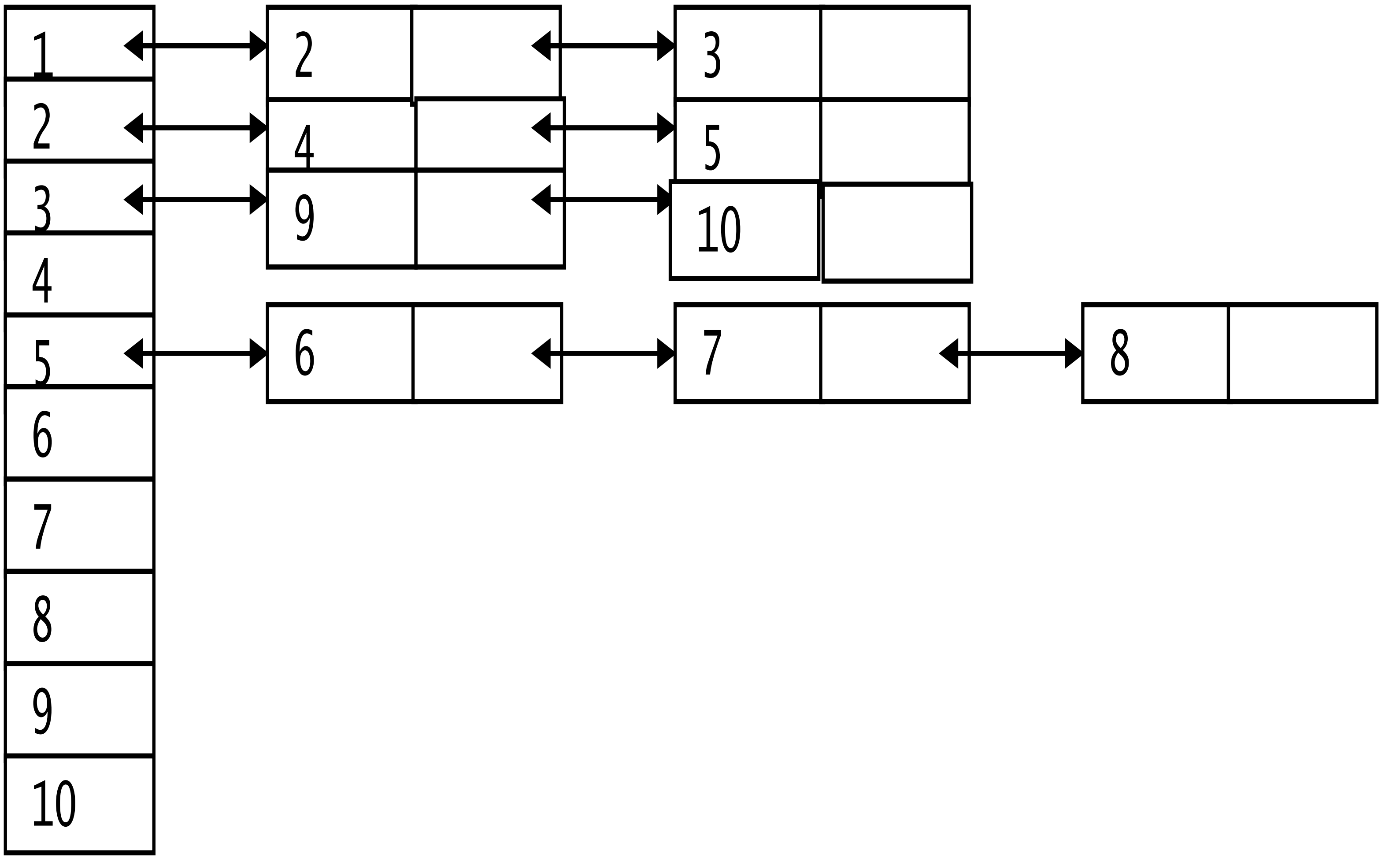
Пример:



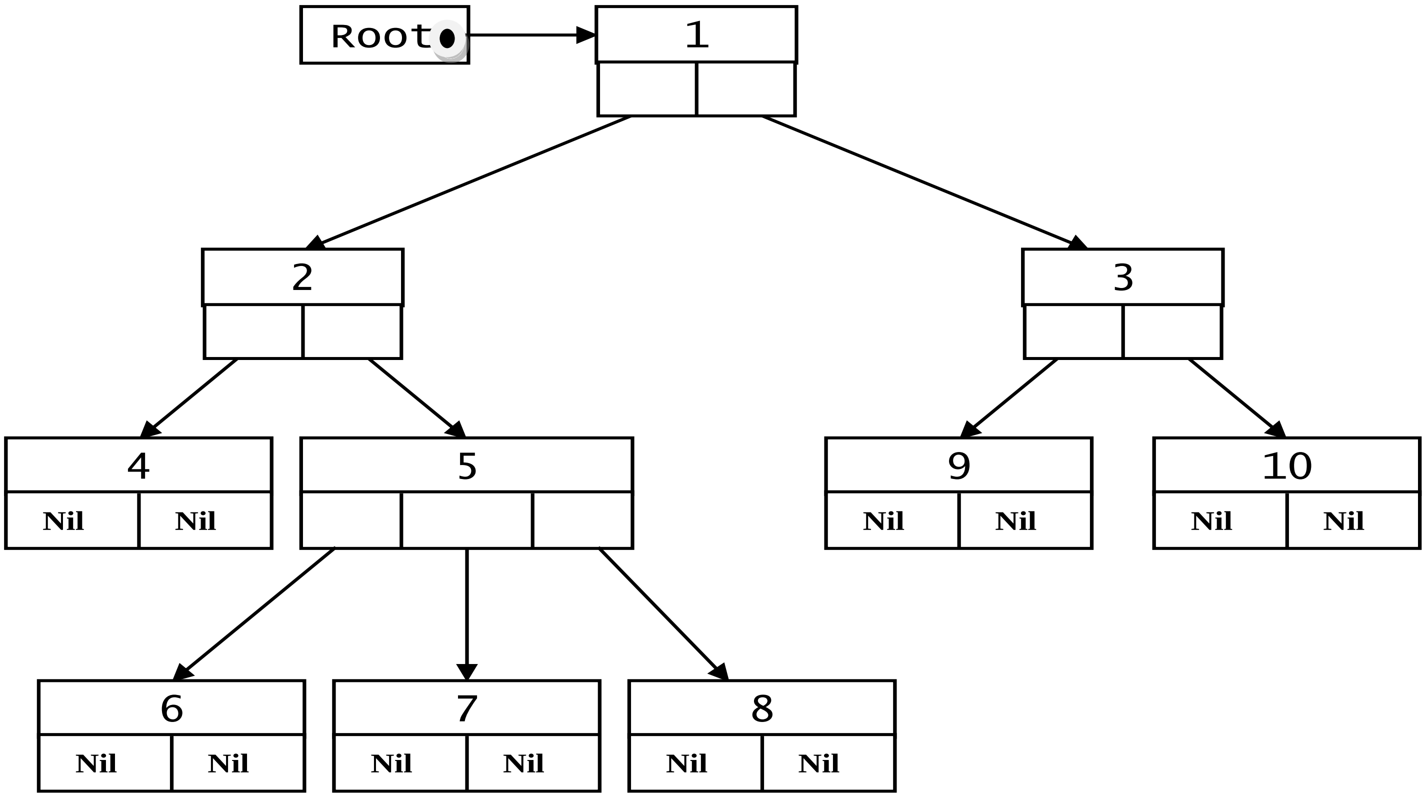
а) в виде курсоров на родителей:

№ верш.	1	2	3	4	5	6	7	8	9	10
	0	1	1	2	2	5	5	5	3	3

б) в виде связного списка сыновей:



в) в виде структуры данных:



Возвращаясь к терминологии:

Упорядоченное дерево – это такое дерево, у которого все ветви, исходящие из одной вершины, **упорядочены.**

Позиционное дерево **–** это корневое дерево, у которого дети любой вершины **помечены номерами от 1 до k.**

К-ичное дерево **–** это дерево, у которого нет вершины **более чем с k – детьми.**

Полное k-ичное дерево **–** это дерево, у которого **все листья** имеют одинаковую глубину**,** а **все внутренние вершины – степень k.**

Тем самым, структура k-ичного дерева полностью определена его высотой.

Т. о., количество листьев (n) у k-ичного дерева высотой k:

$\mathbf{n = k^h}$, **где h – высота, равная соответственно**

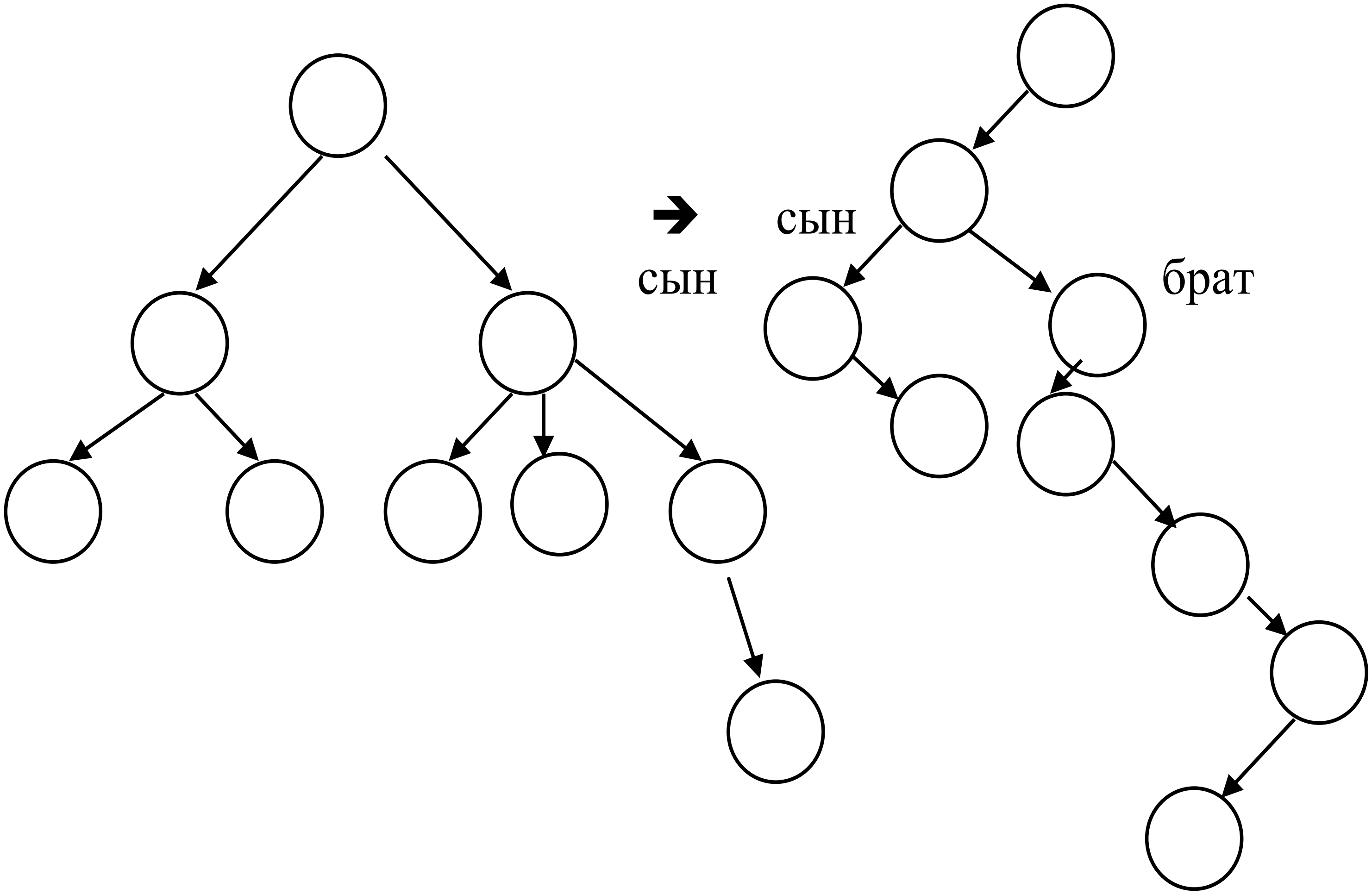
$\mathbf{h = \log_k n}$.

Двоичные (бинарные) деревья

- Если у каждой вершины дерева имеется не более **двух потомков,** то такое дерево называется ***двоичным*** или ***бинарным.***
- Т.е., ***двоичным деревом*** называют **конечный набор элементов, являющихся *узлами (вершинами)* дерева,** такой, что:
- а) **Т – это пустое или нулевое дерево**,
- либо
- б) **Т состоит из корня (вершины) с двумя отдельными двоичными деревьями,** называющимися соответственно ***левым*** и ***правым поддеревьями.***

- **Деревья степени больше 2** называют ***сильно ветвящимися деревьями***.

Пример: Представление троичного дерева в виде двоичного.



Двоичные деревья (2-Д) используются:

- а) для представления алгебраических выражений
- б) в представлении генеалогических деревьев,
- в) в описании турниров и т.п.

- 2-Д является ***полным***, если **все его уровни, кроме последнего, имеют по 2 узла и все нижние узлы имеют хотя бы левого сына**.
- Глубина полного 2-Д с **n** узлами:
- : $\mathbf{D_n = \log_2 n + 1}$,
- например, при $\mathbf{n = 10^6}$, **то** $\mathbf{D_n = 21}$.

Идеально сбалансированное 2-дерево

Идеально сбалансированное дерево (ИДС)– это дерево, у которого **количество вершин** в левом и правом поддеревьях отличается не более, чем на **1**.

Для построения ИСД используется рекурсия. **Алгоритм построения идеально сбалансированного дерева:**

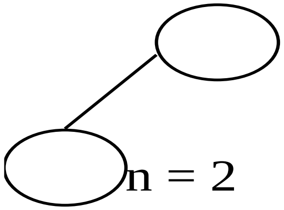
1. Взять одну вершину в качестве корня;
2. Построить левое поддерево с **nl = n div (\) 2** узлами тем же способом; (где **n** – количество всех вершин)
3. Построить правое поддерево с **nr = n – nl – 1** вершинами тем же способом.

Алгоритм построения ИДС

```
Type
     P_Tr = ^Node;            //указатель на вершину
     Node =    Record         // запись (структура)
           Inf             : Type_Inf;
           Left, Right : P_Tr;
                   End;                  // нотация Паскаля
Function Tree (N : Integer): P_tr;  //результат функции – указатель на дерево
   Var  NewNode: P_Tr;
                Nl,Nr: Integer;
 Begin
    If N=0
      Then Tree ← пусто
      Else
         Nl  ← N div 2
         Nr  ← N-Nl-1
         Создать новую вершину (NewNode)
             В поле данных (Inf) ← данные
             Left ← Tree(Nl)
             Rigt ←  Tree(Nr)

    Tree ← NewNode
 End;
```

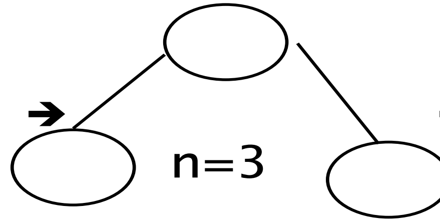
Иллюстрация построения идеально сбалансированного дерева:



n = 1

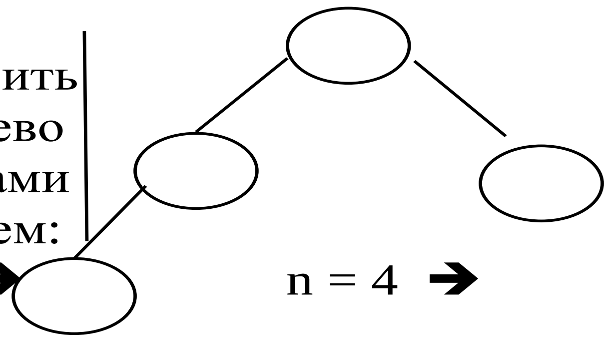
n = 2

сначала строим левое поддерево с n div (\)2 узлами ➔



n=3 ➔

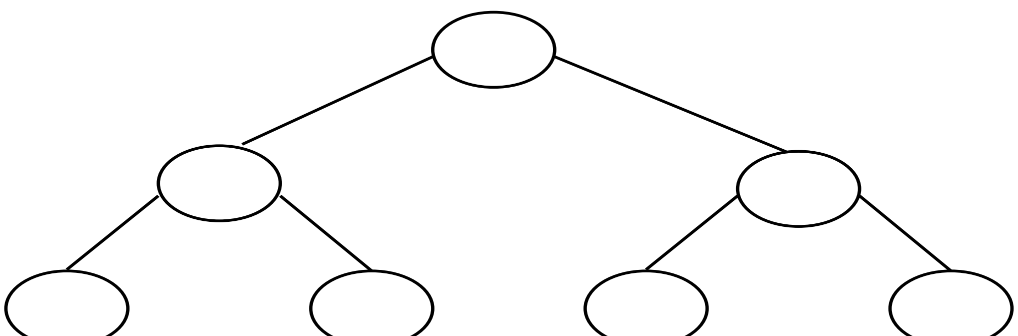
➔ начинаем строить правое поддерево с n – nl – 1 узлами после чего имеем:



n = 4 ➔

процесс повторяется несколько раз,

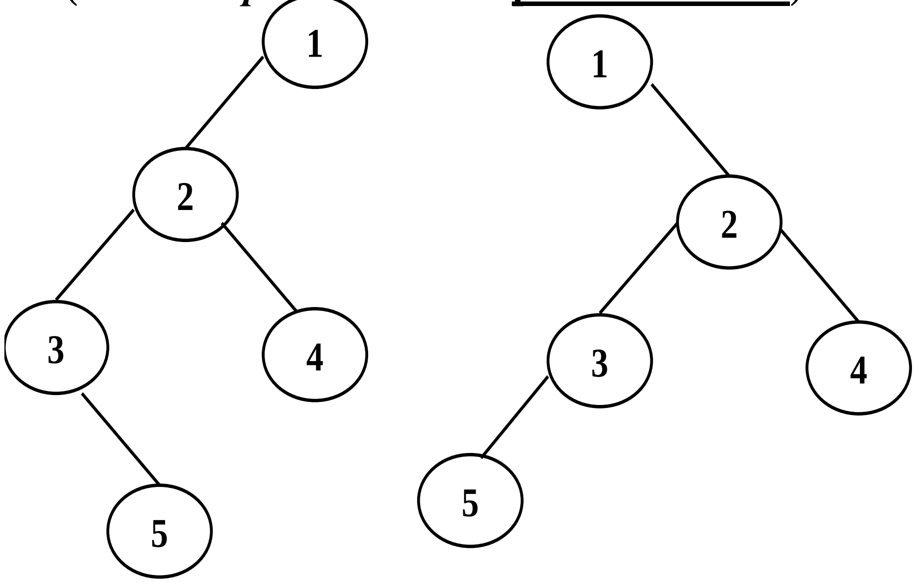
➔ идеально сбалансированное дерево:



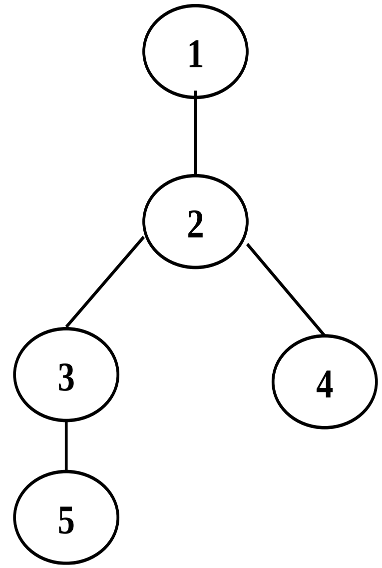
Представление бинарных деревьев

Графическое представление

Два двоичных дерева:
(***левые*** и ***правые*** сыновья **<u>различаются!!!</u>**)



«Обычное»
дерево:



Обозначим:

P(n) – объем памяти, занимаемый представлением двоичного дерева, **n** – количество узлов.

- ***Списочное представление***
- ***Массивы***
- ***Польская запись***

***Списочное представление*:**

структура типа **N**ode**,** элементы которой **l**eft и **r**igth – это левый и правый сыновья и

I – указатель на информацию об узле или сама информация.

```
Struct Node{
    inf  I;
  Node *left;
  Node *rigth;
};
```

В этом случае **P(n) = 3n**, т. к. хранится еще и **nil**, и половина связей не используется.

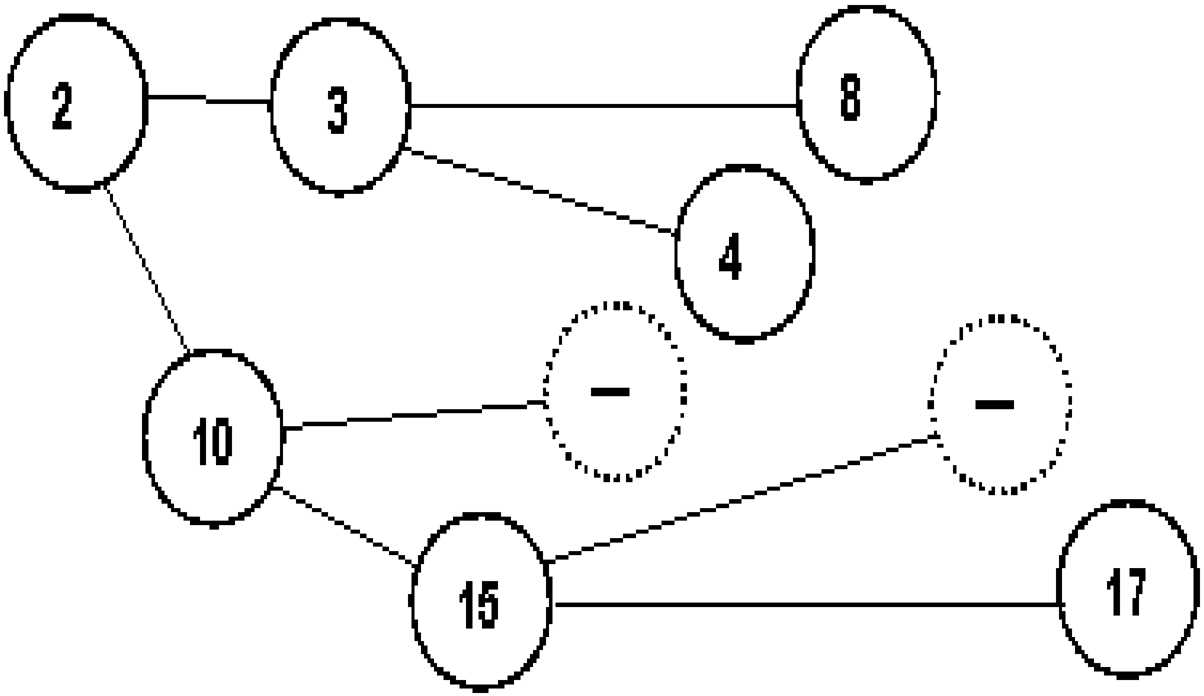
Массивы:

Главным недостатком статического способа представления двоичного дерева является то, что массив имеет фиксированную длину. Размер массива выбирается исходя из максимально возможного количества уровней двоичного дерева, и чем менее полным является дерево, тем менее рационально используется память. Кроме того, недостатком являются большие накладные расходы при изменении структуры дерева (например, при обмене местами двух поддеревьев).

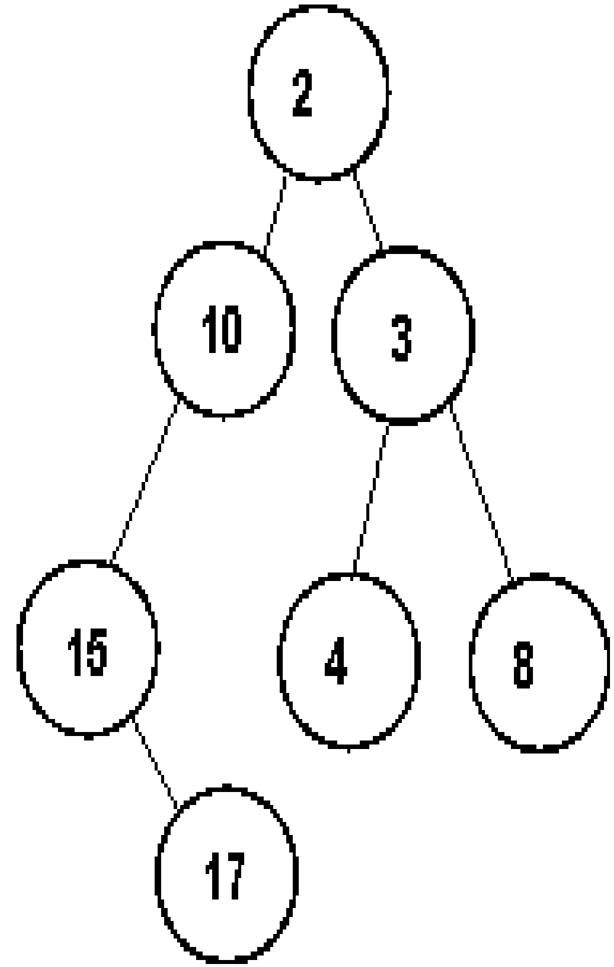
Вершины в массиве располагаются так, что все узлы поддерева данной вершины располагаются вслед за этой вершиной. Вместе с вершиной хранится индексы левого и правого сыновей, т. е, дерево **T** определяется таким образом:

```
T: Array[1..n]   of Record
                    i : info;  //вершина
                   k : 1..n;  //индексы
                      end;
```

Здесь **P(n) = 2n**



2	10	3	15	---	4	8	---	17
1	2	3	4	5	6	7	8	9



Польская запись:

аналогична второму представлению, но вместо связей фиксируется «размеченная степень» 0 – лист, 1 – левая связь, 2 – правая связь, 3 – обе связи, тогда

```
T: Array[1..n]  of Record
                  i : info;
                  d : 0..3;
                  end;
```

И в этом случае **P(n) = 2n**.

Если степень узла имеется в информации об узле, то ее можно не хранить: В наиболее компактный вид хранения, где **P(n) = n**.

Этот вид хранения используется для представления выражений.

Например, **a + b * c** в польской записи будет выглядеть так: **a b c * +.**

Префиксный код бинарного дерева.

Каждой вершине дерева (кроме корня) сопоставляется число. Левому потомку корня приписывается 0, а правому 1, потомкам 00 и 01 соответственно и т.д.

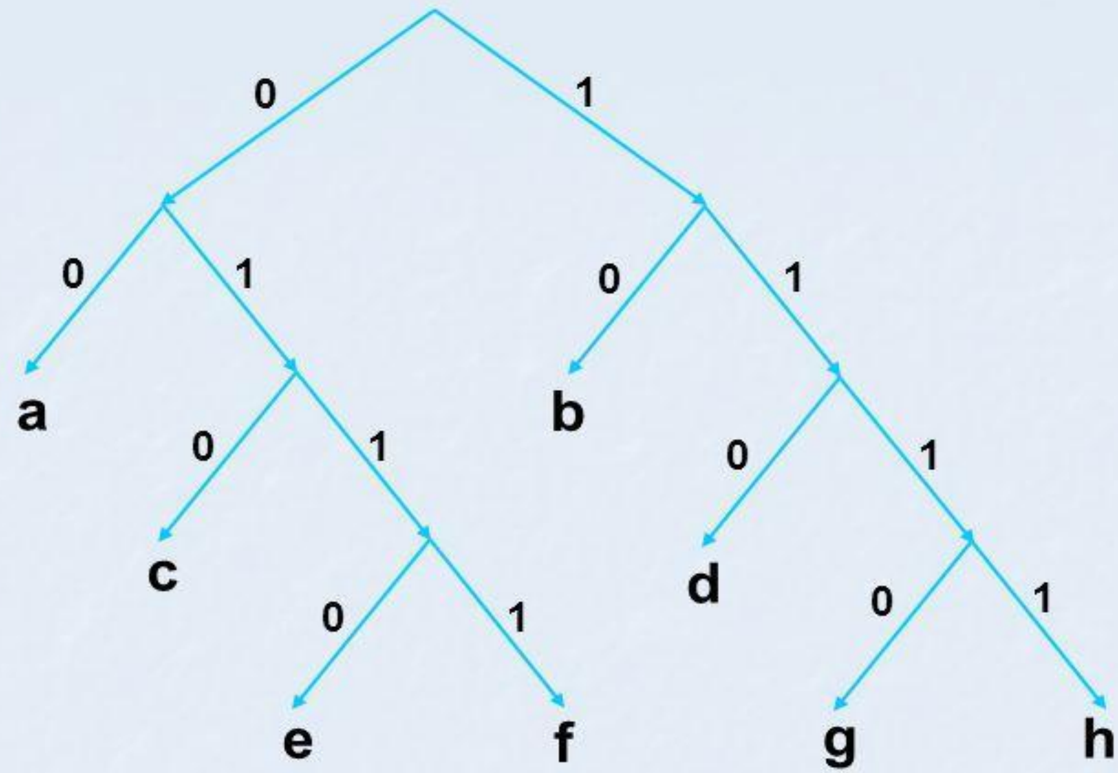
Достаточно хранить в памяти компьютера не все построенные числа, а только расположенные на листах, слева направо.

{000, 010, 011, 10, 110, 1110, 1111} – это и есть префиксный вид бинарного дерева.

Он определяет дерево однозначно. По нему можно восстановить бинарное дерево, т.к. каждое из чисел в перфективном коде позволяет однозначно восстановить весь путь от корня до листа, тем самым все дерево.

Но, для больших деревьев префиксивный код очень объемен. Если у бинарного дерева 10 уровней, то возникает 10-битовые двоичные числа, в количестве до 1024.

a	00
b	10
c	010
d	110
e	0110
f	0111
g	1110
h	1111



Данный код – префиксный, т.к. кодируемые символы располагаются в вершинах, из которых не выходят новые дуги.

Основные операции с двоичными деревьями:

1. Обход (посещение) вершин
2. Поиск по дереву
3. Включение узла в дерево
4. Удаление узла в ДДП

1. Обход (посещение) вершин:

Основной рекурсивный подход для обхода (непустого) бинарного дерева: Начиная с узла N делаем следующее:

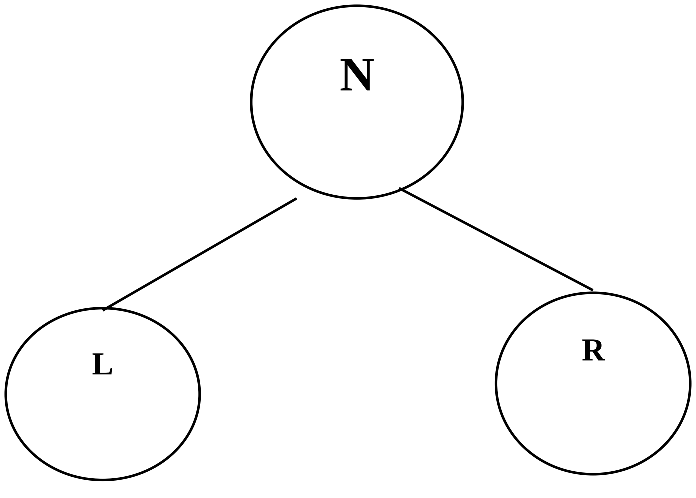
(L) Рекурсивно обходим левое поддерево. Этот шаг завершается при попадании опять в узел N.

(R) Рекурсивно обходим правое поддерево. Этот шаг завершается при попадании опять в узел N.

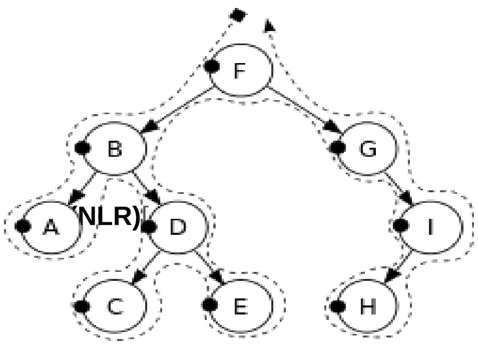
(N) Обрабатываем сам узел N.

Эти шаги могут быть проделаны *в любом порядке*. Если (L) осуществляется перед (R), процесс называется обходом слева направо, в противном случае — обходом справа налево.

– сверху вниз: N,L,R.
– слева направо: L,N,R
– снизу вверх: L,R,N

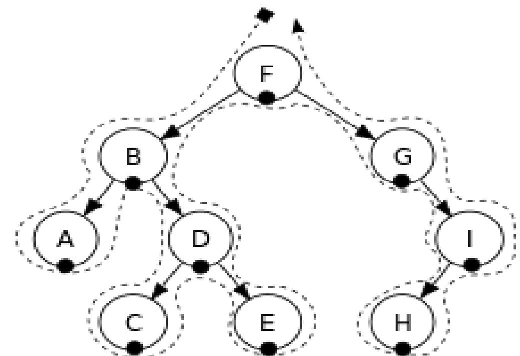


Прямой обход – сверху вниз (NLR)



Прямой обход: F, B, A, D, C, E, G, I, H.
1.Проверяем, не является ли текущий узел пустым или null.
2.Показываем поле данных корня (или текущего узла).
3.Обходим левое поддерево рекурсивно, вызвав функцию прямого обхода.
4.Обходим правое поддерево рекурсивно, вызвав функцию прямого обхода.

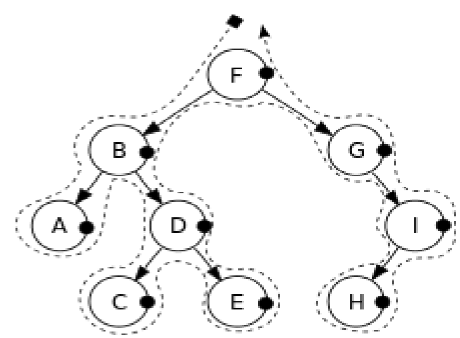
Слева направо - центрированный обход (LNR)[



Центрированный обход: A, B, C, D, E, F, G, H, I.
1.Проверяем, не является ли текущий узел пустым или null.
2.Обходим левое поддерево рекурсивно, вызвав функцию центрированного обхода.
3.Показываем поле данных корня (или текущего узла).
4.Обходим правое поддерево рекурсивно, вызвав функцию центрированного обхода.

В двоичном дереве поиска центрированный обход извлекает данные в отсортированном порядке.[4].

Обратный обход - снизу вверх (LRN)



Обратный порядок: A, C, E, D, B, H, I, G, F.

Проверяем, не является ли текущий узел пустым или null.
2.Обходим левое поддерево рекурсивно, вызвав функцию обратного обхода.
3.Обходим правое поддерево рекурсивно, вызвав функцию обратного обхода.
4.Показываем поле данных корня (или текущего узла).
Последовательность обхода называется секвенциализацией дерева. Последовательность обхода — это список всех посещённых узлов. Ни одна из секвенциализаций согласно прямому, обратному или центрированному порядку не описывает дерево однозначно. Если задано дерево с различными элементами, прямой или обратный обход вместе с центрированным обходом достаточны для описания дерева однозначно. Однако прямой обход вместе с обратным оставляет некоторую неоднозначность в структуре дерева[5].

Если дерево **Т** является нулевым деревом, то в список обхода записывается пустая строка;

Если дерево **Т** состоит из одного узла, то в список обхода записывается этот узел;

Пусть дерево **Т** имеет корень N и поддеревья **Т1** , **Т2** , ... **Тm** , как показано на рисунке

Тогда для различных способов обхода имеем следующее:

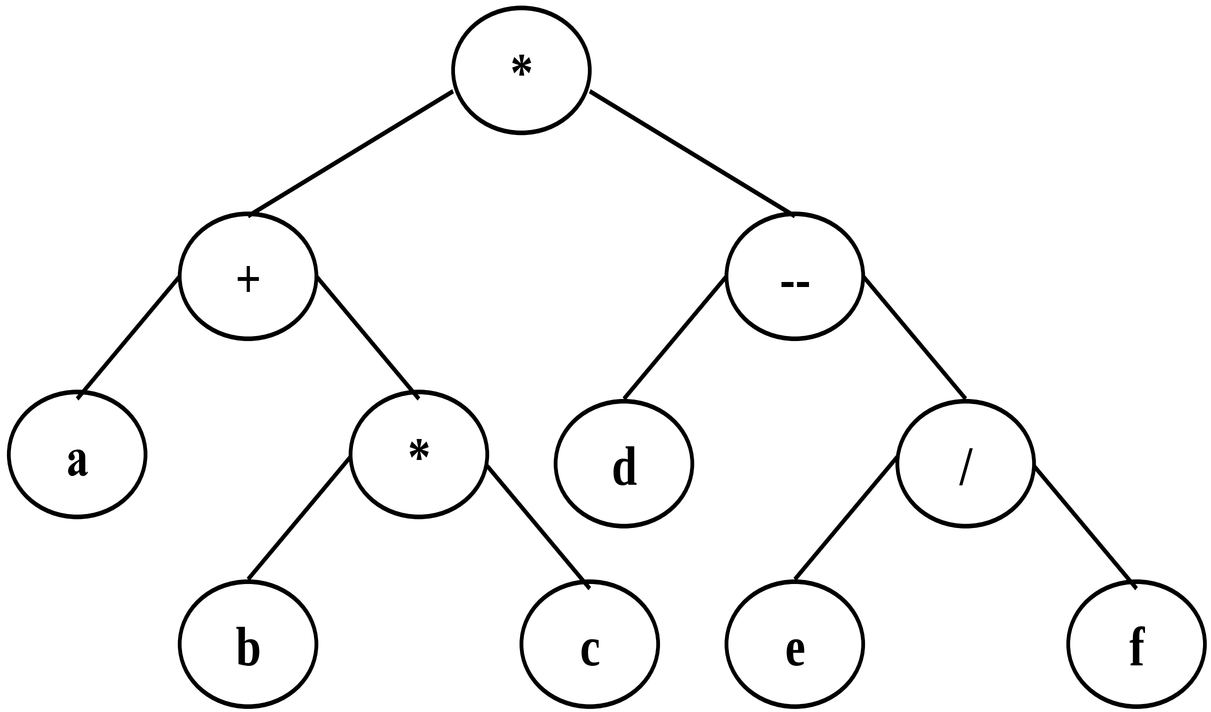
Прямой обход (сверху вниз, префиксный) . Сначала посещается корень **N** , затем в прямом порядке узлы поддерева **Т1** , далее все узлы поддерева **Т2** и т.д. Последними посещаются в прямом порядке узлы поддерева **Тm** .

Обратный обход (снизу вверх, постфиксный) . Сначала посещаются в обратном порядке все узлы поддерева **Т1** , затем в обратном порядке узлы поддеревьев **Т2** … **Тm** , последним посещается корень **N** .

Симметричный обход (**слева направо, инфиксный)**. Сначала в симметричном порядке посещаются все узлы поддерева **Т1** , затем корень **N** , после чего в симметричном порядке все узлы поддеревьев **Т2 … Тm** .

- Например, если представить алгебраическое выражение вида
- (a + b * c) * (d – e / f) в виде дерева:
- то обходя это дерево и выписывая символы, находящиеся в вершинах, получим:
- При обходе дерева ***сверху вниз***: *** + a * b c – d / e f** Эта форма записи называется ***<u>префиксной.</u>***
- При обходе дерева ***слева направо*****:**
- **a + b * c * d – e / f,** т. е., исходное выражение, но без скобок. ***<u>Инфиксная</u>*** форма записи.
- При обходе дерева ***снизу вверх***:
- **a b c * + d e f / – *   -** ***<u>постфиксноая.</u>***

Представление алгебраическое выражение вида
`(a + b * c) * (d – e / f)` в виде дерева:



сверху вниз <u>(префиксная, прямая)</u>: *** + a * b c – d / e f**

слева направо <u>(инфиксная, симметричная)</u>: **a + b * c * d – e / f**

снизу вверх <u>(постфиксная, обратная)</u>: **a b c * + d e f / – ***

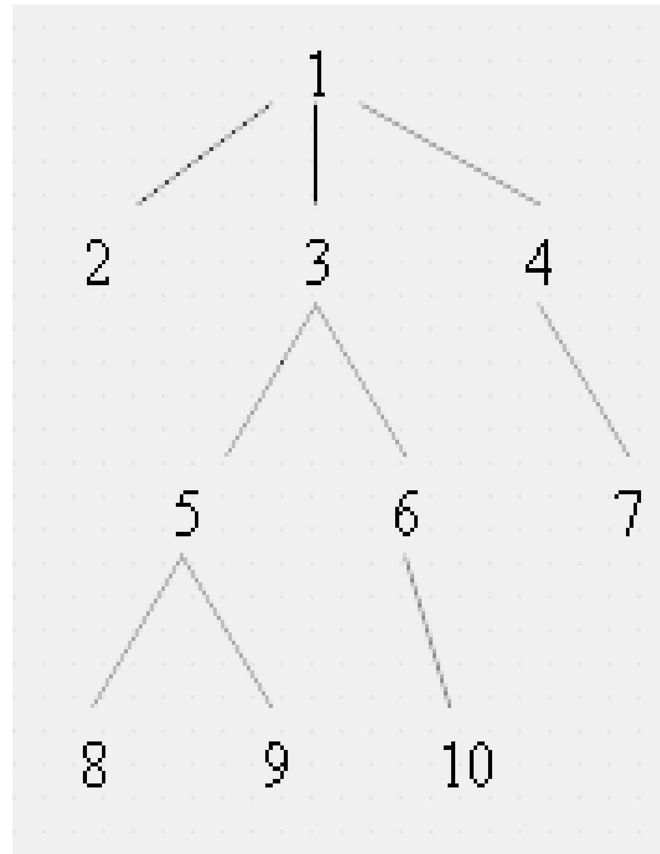
Для деревьев общего вида

Порядок узлов этого дерева в случае **прямого обхода** будет следующим:
1 2 3 5 8 9 6 10 4 7.

Обратный обход даст нам следующий порядок узлов:

2 8 9 5 10 6 3 7 4 1.

При **симметричном обходе** мы получим такую последовательность узлов:
2 1 8 5 9 3 10 6 7 4.



операции в виде рекурсивных процедур:

```
Type
   P_Tr = ^Node;
   Node = Record    Inf         : Type_Inf;
                    Left, Right : P_Tr;
          End;
Procedure Wr_TPref(Q : P_Tr);          void preOrderTravers(Node* root) {
 //префиксный обход                       if (root)
 Begin                                       {
   WriteLn(Q^.Inf)  //печать инф-ции узла      printf("%d ", root->data)
   If Q^.Left <> Nil  Then Wr_TPref(Q^.Left);   preOrderTravers(root->left);
   If Q^.Right <> Nil Then Wr_TPref(Q^.Right);  preOrderTravers(root->left);
 End;                                          }
                                            }
```

Процедура работает в двух режимах. В первом режиме осуществляется обход по направлению к левым потомкам до тех пор, пока не встретится лист, при этом выполняется печать значений вершин, и занесение указателей на них в стек. Во втором режиме осуществляется возврат по пройденному пути с поочередным извлечением указателей из стека до тех пор, пока не встретится вершина, имеющая еще ненапечатанного правого потомка. Тогда процедура переходит в первый режим и исследует новый путь, начиная с этого потомка.

- **Procedure Wr_TInf (Q : P_Tr);** //инфиксный обход
- **Begin**
- **If Q^.Left <> Nil Then Wr_TInf(Q^.Left);**
- **WriteLn(Q^.Inf);**
- **If Q^.Right <> Nil Then Wr_TInf(Q^.Right);**
- **End;**

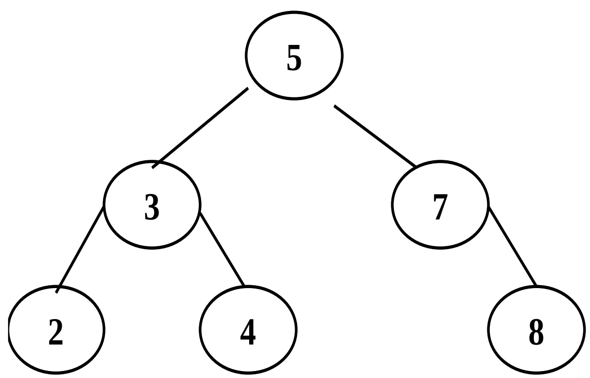
- **Procedure Wr_TPost (Q : P_Tr);** //постфиксный обход
- **Begin**
- **If Q^.Left <> Nil Then Wr_TPost(Q^.Left);**
- **If Q^.Right <> Nil Then Wr_TPostf(Q^.Right);**
  Writeln(Q^.Inf);
- **End;**

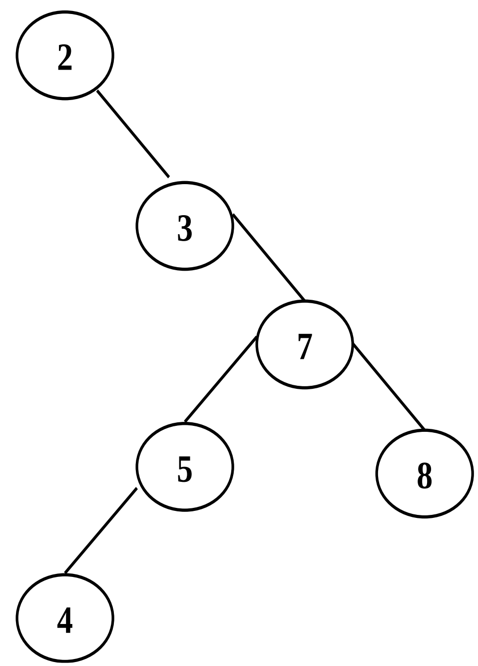
- Следует обратить внимание на то, **что <u>Q – параметр-значение, не ссылка (НЕ указатель)</u> .**

Дерево двоичного поиска (ДДП)

(BST: Bynary Search Tree)

Каждая вершина ДДП (BST) имеет значение, которое больше, чем содержание любой из вершин его левого поддерева и меньше, чем содержание любой из вершин его правого поддерева.





2. Поиск по дереву

Алгоритмы поиска по дереву.

Для реализации алгоритмов поиска по дереву используются **ДДП** (BST) т.е. **дерево, в котором все левые сыновья моложе предка, а все правые – старше.**

Это свойство называется ***характеристическим свойством дерева двоичного поиска*** и выполняется для любого узла, включая корень.

Поиск узла со значением X с помощью рекурсии:

```
• Function Poisk_R (X : Type_Inf; Q : P_Tr) : P_Tr;
•     Begin
•          If (Q = Nil) Or (X = Q^.Inf)     Then Return Q;
           If X < Q^.Inf   Then  Return Poisk_R (X, Q^.Left)
•                            Else Return Poisk_R (X,Q^.Right);
      End;
```

либо с помощью итерации:

```
• Function Poisk_ I(X : Type_Inf; Q : P_Tr) : P_Tr;
•     Begin
•         Poisk_I ← Nil;
•         While Q^.Inf <> X  Do
•              If X < Q^.Inf   Then Q ← Q^.Left
•                                   Else Q ← Q^.Right;
•              Poisk_I ← Q;
      End;
```

Если элемент с ключом не найден, то возвращает значение **Nil**.

3. Включение в дерево

Элемент с ключом X можно включить в дерево, реализовав алгоритм **поиска по дереву с включением.**

Процедура поиска по дереву с включением выглядит так:

```
•     Procedure Search(X : Type_Inf; Var Q : P_Tr);
•         Begin
•          If Q = Nil  Then    Begin
                      New (Q);          //создание узла
•                         Q^.Inf    ← X;
•                         Q^.Left   ← Nil;
•                         Q^.Right ← Nil;
•                              End
•                     Else    Begin
•                        If X < Q^.Inf  Then Search (X, Q^.Left);
•                        If X > Q^.Inf  Then Search (X, Q^.Right);
•                              End;
•         End;
```

Здесь **Q – параметр-переменная (ссылка), а не параметр-значение(!).**

- Алгоритм поиска по дереву с включением применяется и для сортировки дерева.
- При появлении одинаковых ключей, можно сделать: **X ≥ Q^.Inf**.
- При большом количестве одинаковых ключей можно добавить их количество:
- **Type**
- **P_Tr = ^Node;**
- **Node = Record**
- **Inf : Type_Inf;**
- **Count : Word;**
  Left, Right : P_Tr;
- **End;**

4. Удаление узла в ДДП

Удаление не столь тривиально и зависит от расположения в дереве удаляемого элемента.

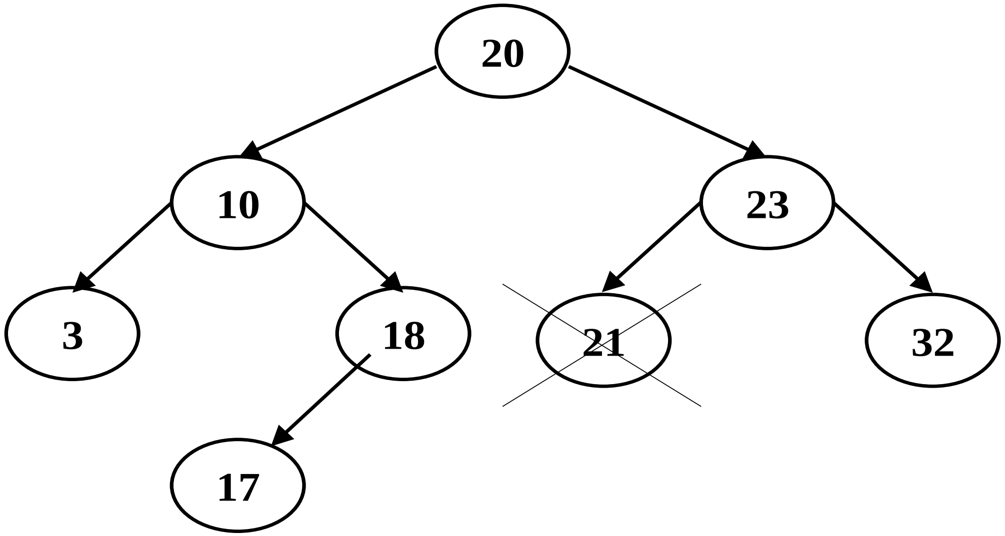
Бывает 3 случая:

1. Элемента с ключом **X** в дереве нет.
2. Элемент с ключом **X** имеет не более одного потомка или является листом (терминальной вершиной).
3. Элемент с ключом **X** имеет двух потомков.

Удаление элементов

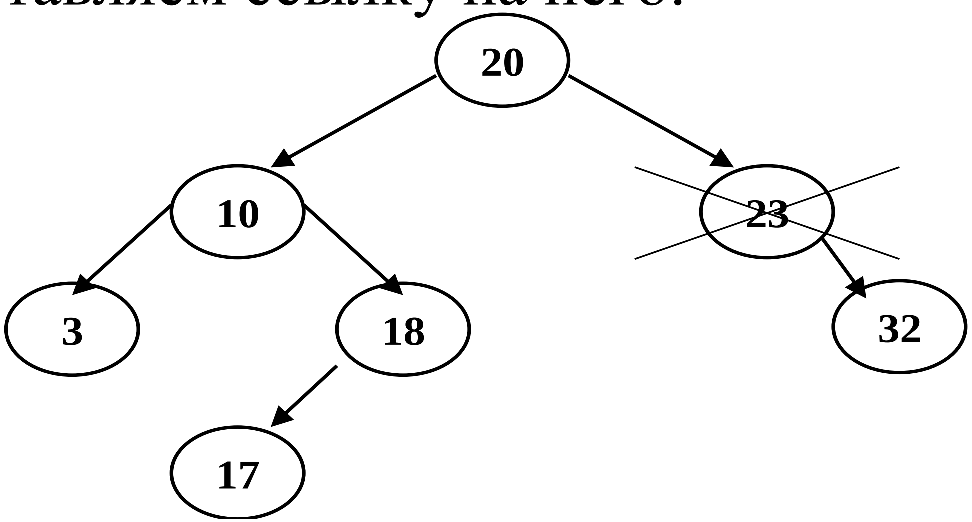
Один потомок или лист:

а) исключаем ссылку на лист:



или

б) переставляем ссылку на него:



В первом случае ничего не делаем - удалять нечего.

Во втором случае, или

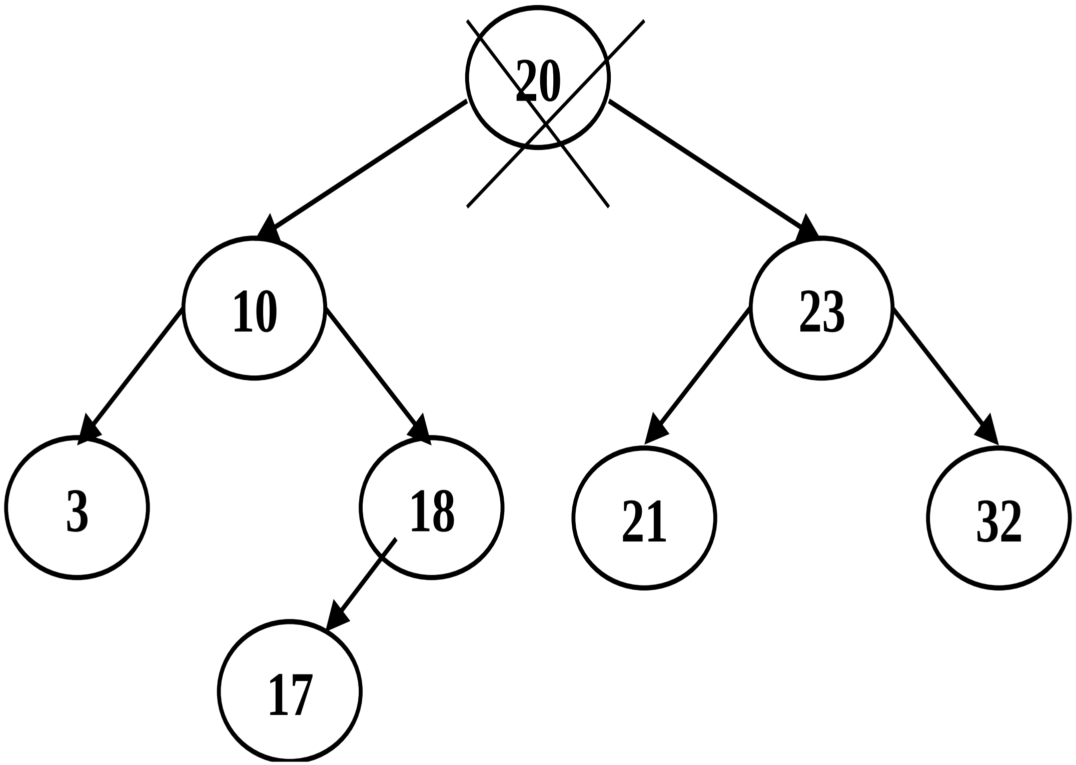
а) исключаем ссылку на лист;

б) или переставляем ссылку на него.

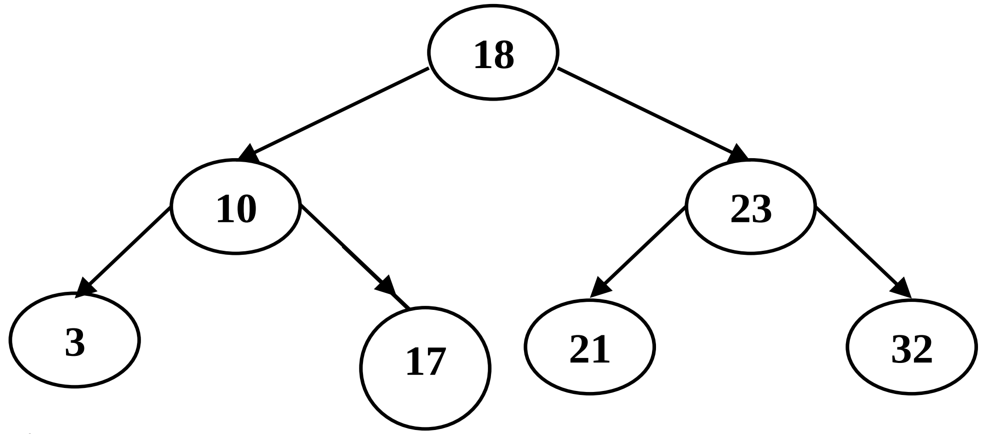
В третьем случае, чтобы не нарушить характеристическое свойство ДДП, необходимо заменить удаляемый элемент или на **самый правый элемент** его **левого поддерева** (т. е., на наибольший элемент среди потомков левого сына),

или на **самый левый элемент** его **правого поддерева** (т. е.. на наименьший элемент среди потомков правого сына), причем, они должны иметь не более одного потомка. :

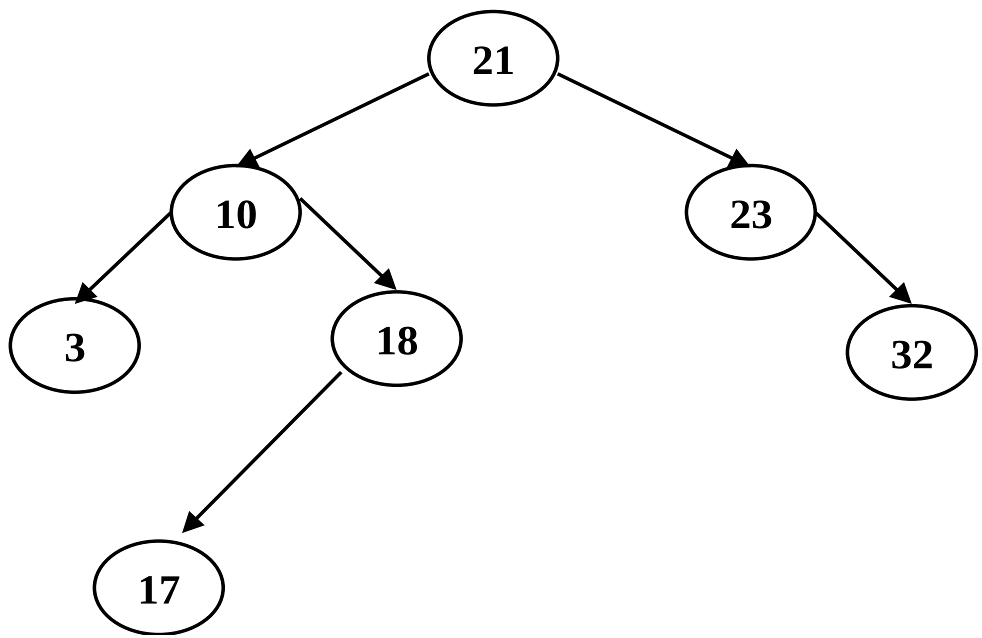
Например, для случая:



удалить элемент с ключом 20 можно либо так:



либо так:



Рекурсивная процедура удаления элемента из дерева включает вспомогательную рекурсивную процедуру, которая вызывается в случае удаления элемента, имеющего двух потомков.

При удалении какого-либо элемента необходимо изменить ссылку на сам элемент в его родителе. Поэтому введем дополнительный указатель для просмотра удаляемых элементов из его родителей.

Алгоритм процедуры удаления:

```
   Type
   P_Tr = ^Node;
   Node = Record
               Key        : Type_Inf;
             Left, Right : P_Tr;
          End;
Procedure Delete_X (X : Type_Inf; Var P : P_Tr);
     Var
          Q : P_Tr;                        //вершина –заменитель

    Procedure Changer (Var R : P_Tr);
       Begin
         If R^.Right <> Nil
               Then Changer(R^.right)
               Else
                   Q^.Key ← R^.Key;
                   Q ← R;
                   R ← R^.Left;          //вых –левый потомок
    End;
```

```
Begin              {исключение эл-та}
    If P = Nil
           Then Writeln('Элемента в дер. нет’)
            Else
               If X < P^.Key
                       Then Delete_X (X, P^.Left)
                        Else
                            If X > P^.Key
                                 Then Delete_X (X, P^.Right)
                                  Else
                                     Begin       //удаление P
                                        Q ← P;
                                         If Q^.Right = Nil
                                             Then P← Q^.Lleft
                                              Else
                                                 If Q^.Left = Nil
                                                       Then   P ← Q^.Right
                                                        Else   Changer (Q^.Left);
      Dispose(P);  // освобождение вершины
End;
```

- Вспомогательная рекурсивная процедура **Changer** начинает работать только в случае 3 (см. выше). Она «спускается» вдоль правой ветви левого поддерева элемента **Q^**, который нужно исключить, и заменяет существующую в **Q^** информацию
- (т. е., ключ) на соответствующее значение из самой правой компоненты **R^** левого поддерева, после чего элемент **R^** можно исключить.

Ассоциативная память

- ***Ассоциативная память (АП):*** каждой ***записи*** (порции) данных ставят в соответствие (ассоциируют**)** ***ключ*** **– значение из некоторого вполне упорядоченного множества.**
- Записи могут иметь произвольную природу и различные размеры. Доступ к данным осуществляется **по значению ключа**, который обычно выбирается простым, компактным и удобным для работы. (см. 2л.р.)

Примеры Ассоциативной памяти

Толковый словарь:

- запись – это словарная статья,
- ключ – заголовок словарной статьи.

Адресная книга:

- запись – адресная информация (адрес, телефон и др.),
- ключ – имя абонента.

Банковские счета:

- запись – финансовая информация, ключ – номер счета.

Способы реализации АП:

- Неупорядоченный массив;
- Упорядоченный массив;
- Двоичное дерево поиска;
- Таблица расстановки (хэш-таблица);

Основные операции с АП:

- Добавление ключа (записи);
- Поиск ключа (записи);
- Удаление ключа (записи);

Сравнение представлений ассоциативной памяти (АП)

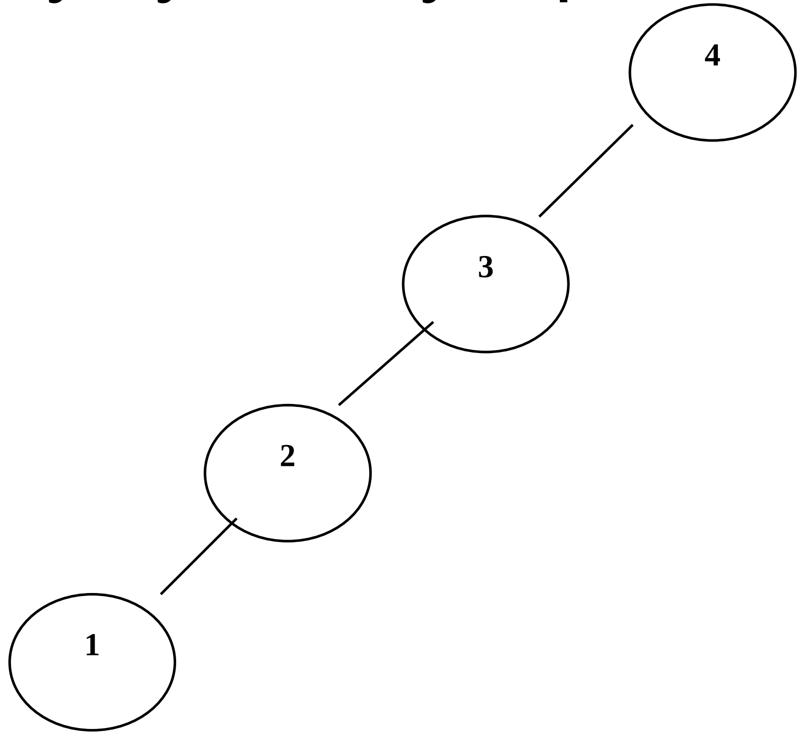
- n – количество элементов
- «Стоимость» операций для различных представлений АП

	Неупорядоченный массив	Упорядоченный массив	Дерево сортировки
Добавление	O(1)	O(n)	$O(\log_2 n)$… O(n)
Поиск	O(n)	$O(\log_2 n)$	$O(\log_2 n)$… O(n)
Удаление	O(n)	O(n)	$O(\log_2 n)$… O(n)

- Эффективность операций сортировки ограничена сверху высотой дерева. Дерево сортировки может расти неравномерно. Например, если при загрузке дерева исходные данные уже упорядочены, то полученное дерево будет право или леволинейным (левосторонним) и будет даже менее эффективным, чем неупорядоченный массив. Т. е., худшими случаями при использовании бинарных деревьев являются такие, когда дерево имеет вид последовательности вершин:

Левостороннее дерево поиска:

Почему хуже неупорядоченного массива?



Сбалансированные деревья поиска

self-balancing binary search tree

Высота поддеревьев любого узла различаются не более чем на заданную константу **k**

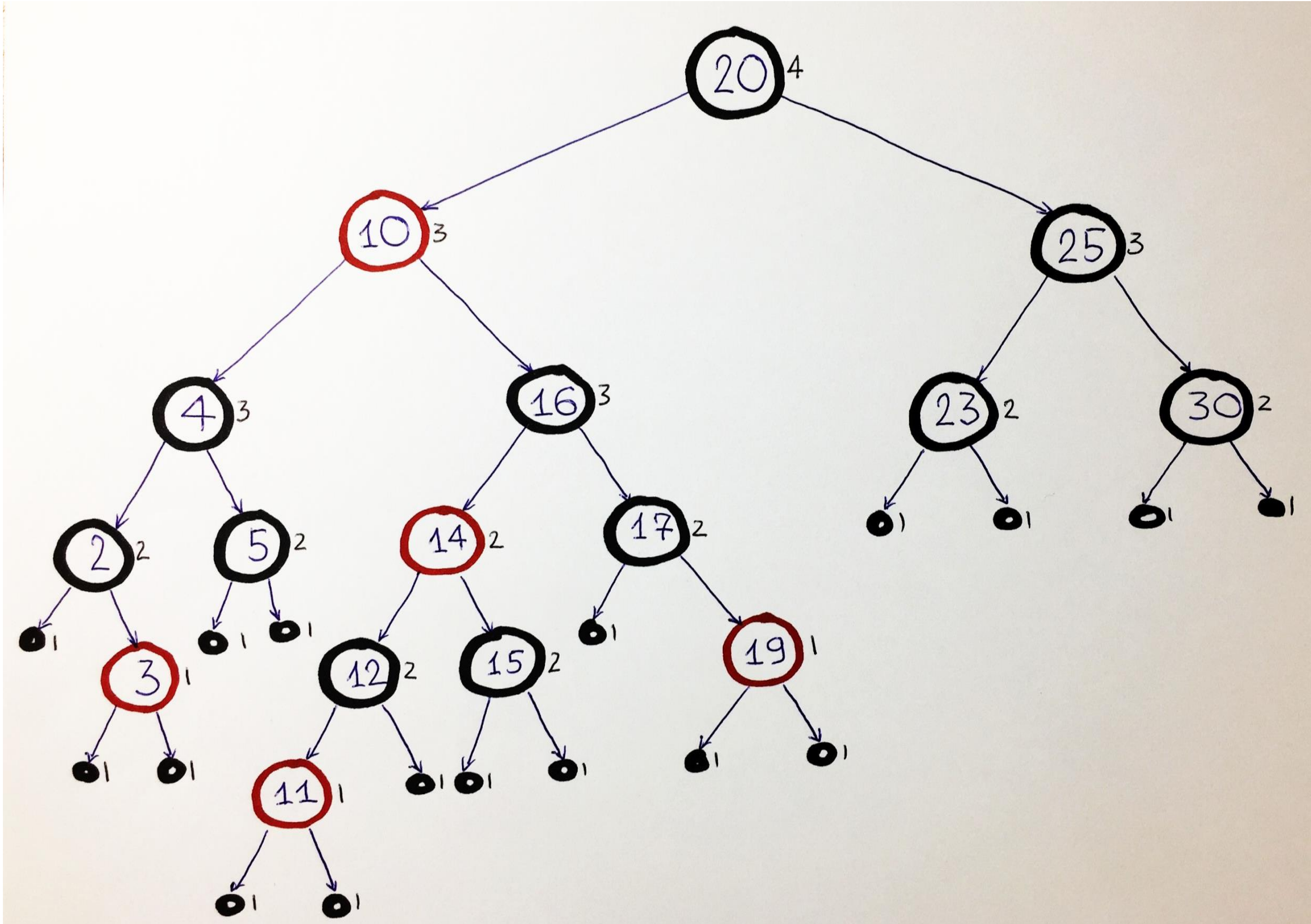
Виды сбалансированных деревьев поиска:

АВЛ-деревья(AVL tree) эффективность временная - O(log n) емкостная O(n)

Красно-черные деревья (Red-black tree)

B-деревья (B-tree)

Красно-черное дерево



КЧ-деревья – это двоичные деревья поиска, каждый узел которых хранит дополнительное поле color, цвет: красный или черный:

```
struct RBNode {
        key_type key;
        struct RBNode *left;
        struct RBNode *right;
        struct RBNode *parent;
        char color; // цвет
        }
```

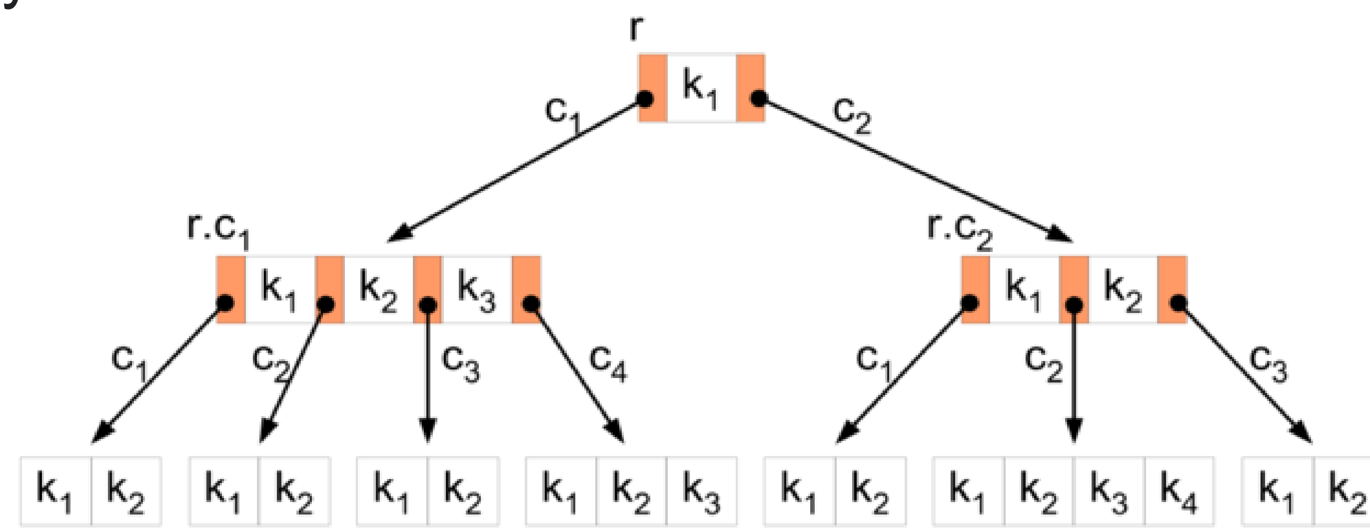
Если left или right равны NULL, то это «указатели» на **фиктивные** листья. Т.о., все узлы – внутренние (нелистовые).

для КЧ-деревьев выполнняются свойства:

1. каждый узел либо красный, либо черный;
2. каждый лист (фиктивный) – черный;
3. если узел красный, то оба его сына – черные
4. любой простой путь от узла-предка до листового узла-потомка содержит одинаковое число чёрных узлов.

(Рудольф Байер (немецк.). Название эта СД получила в статье Леонидаса Гимпаса и Роберта Седжвика 1978 года)

B-дерево - СД, дерево поиска. С точки зрения внешнего логического представления, оно сбалансированное и сильно ветвистое. Часто используется для хранения данных во **внешней** памяти.
Использование B-деревьев впервые было предложено Р. Бэйером в 1970г.
Сбалансированность означает, что длина любых двух путей от корня до листьев различается не более, чем на единицу.

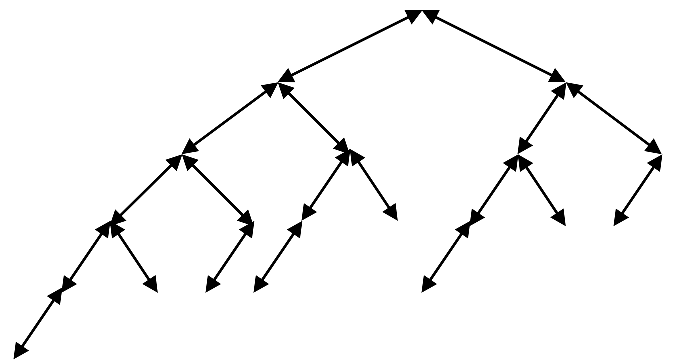


Адельсон-Вельский и Ландис (1962г.), сформулировали критерий сбалансированности двоичного дерева:

дерево называется ***сбалансированным*** **тогда и только тогда, когда высоты двух поддеревьев каждой из его вершин отличаются не более чем на единицу**. Такие деревья называют ***АВЛ-деревьями.***

- (в ***идеально сбалансированном*** дереве не более чем на единицу отличается **<u>число вершин</u>** в левом и правом поддереве. ***Идеально сбалансированное дерево*** является также и ***АВЛ-деревом***).

- Максимально несимметричное АВЛ-дерево:
-
-



- Для каждого узла высота двух поддеревьев отличается не более чем на 1.

Включение в сбалансированные деревья (СБ)

Рассмотрим включение в левое поддерево.

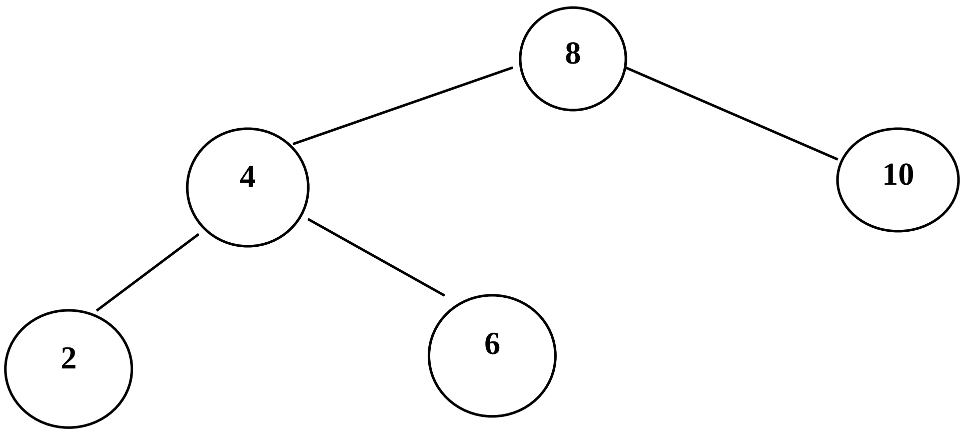
При включении в СБ возможны 3 случая:

- Левое и правое поддеревья становятся не равной высоты, но критерий сбалансированности не нарушается.
- Левое и правое поддерево приобретают равную высоту и т.о. сбалансированность даже улучшается.
- Критерий сбалансированности нарушается, и дерево надо перестраивать

включение в левое поддерево

эффективность временная - O(log n)   емкостная   O(n)

Bal		Высота после включения	Действие	Bal = **hr – hl**
0	h**l** = h**r**	h**l** > h**r**	нет	-1
1	h**l** < h**r**	h**l** = h**r**	нет	0
-1	h**l** > h**r**	h**l** >> h**r**	балансировка	**-2 (!)**

- Рассмотрим эти случаи.
- Допустим, есть дерево:
- 

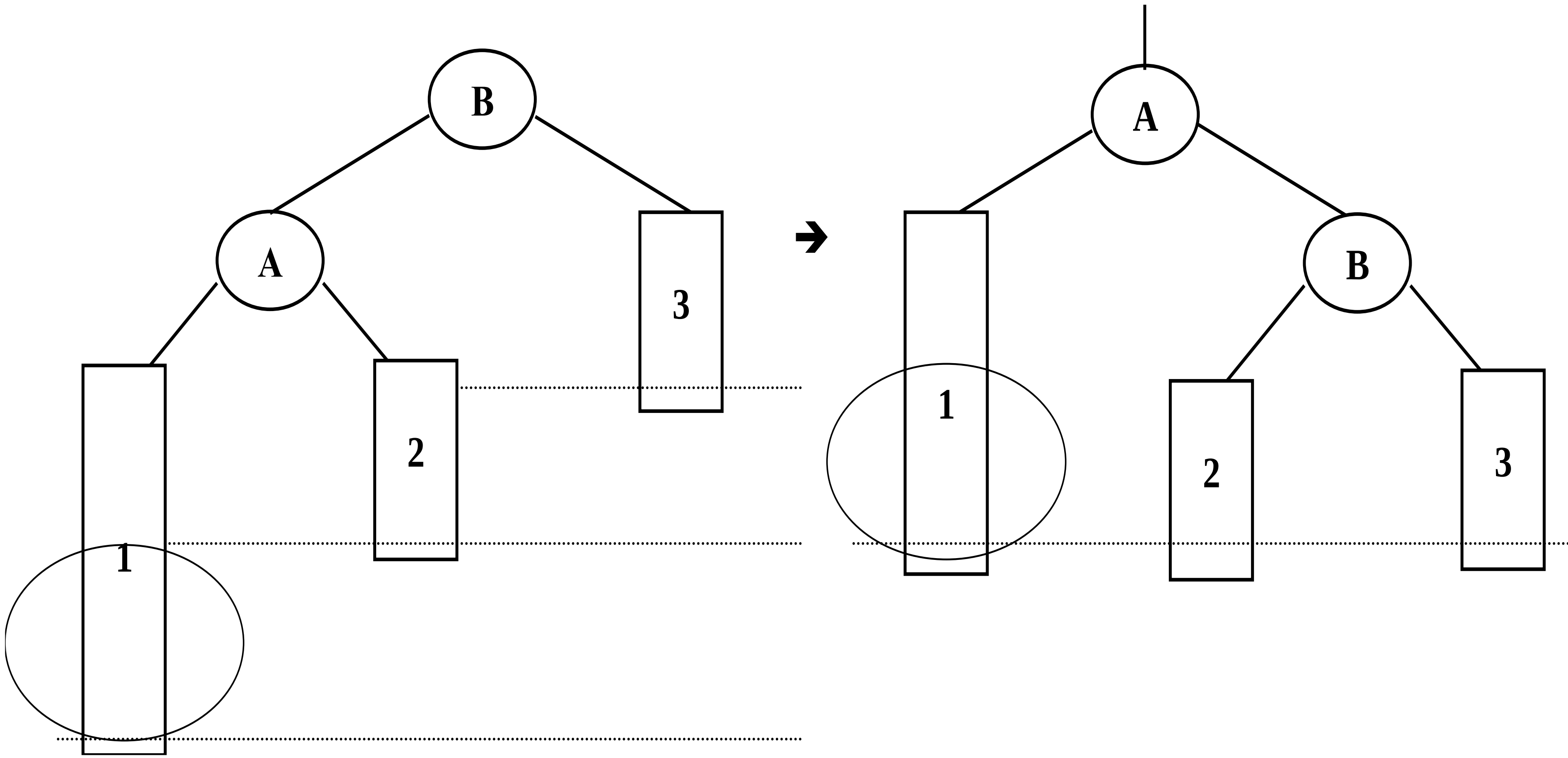
- Не нарушая сбалансированности этого дерева, в него можно включить 9 и 11 вершины. Если же мы включим вершины 1, 3, 5, 7, то тем самым мы нарушим баланс дерева.

Рассмотрим схематично эти случаи.

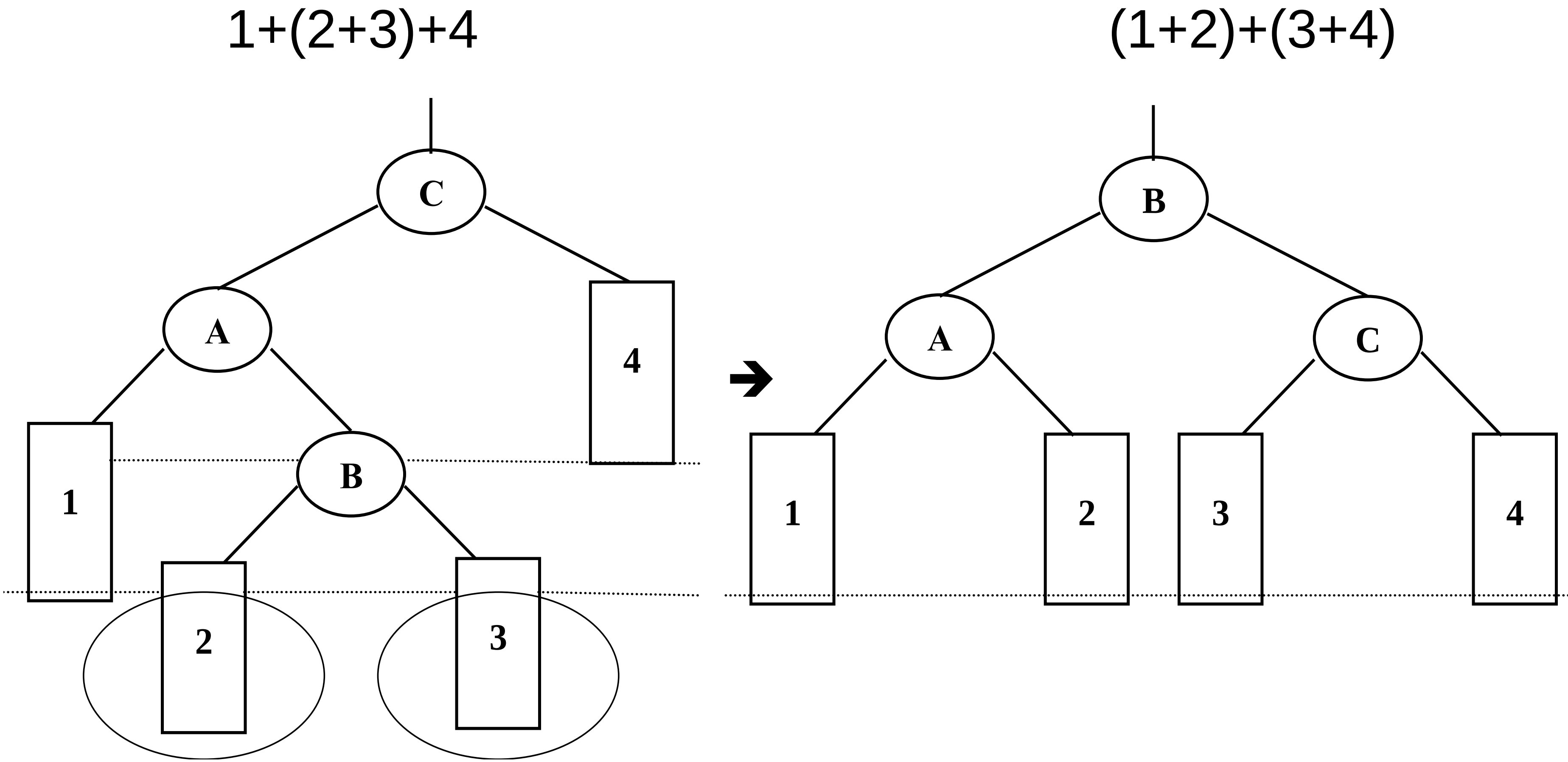
- (Допускается перемещение только по вертикали, а относительно горизонтали расположение показанных вершин и поддеревьев должно оставаться без изменения.)
- Имеются лишь две существенно различные возможности. Остальные могут быть получены симметричным преобразованием этих двух возможностей. **Вариант 1** определяется включением узла 1 или ключа 3, **вариант 2** – включение узла 5 или 7.

Пример1

(1+2)+3 ➔ 1+(2+3)



Пример2



- Алгоритм включения и балансировки существенно зависит от способа хранения информации о сбалансированности дерева. Можно хранить в каждой вершине показатель ее сбалансированности:

- **Type**
- **P_Tr = ^Node;**
- **Node = Record**
- **Key : Integer;**
- **Count : Integer; {количество вершин с одинаковым значением}**
- **Left, Right : P_Tr;**
- **Bal : -1..1; {показатель сбалансированности}**
- **End;**

```
struct node  {
        int key;
        int count;
      node* left;
      node* right;
        int bal;        // -1..1
    };
```

- В дальнейшем сбалансированность будет у нас определяться как разность между высотой правого и высотой левого поддеревьев. Процесс включения вершины практически состоит из трех последовательно выполняемых частей:
- 1. Проход по пути поиска (пока не убедимся, что элемента с таким ключом в дереве нет);
- 2. Включение новой вершины и определение показателя сбалансированности;
- 3. «Отступление» по пути поиска, проверка показателей сбалансированности каждой вершины, и если необходимо, балансировка.

- На каждом шаге необходима информация о высоте дерева **H**. Возьмем **H** типа Boolean, причем **H** для процедуры параметр-переменная, указывающий, что высота дерева увеличилась (H = True).

- Допустим, процесс возвращается из левой ветви к вершине **P^** и ее высота увеличилась.

Тогда возможны следующие ситуации:

- 1. **hl < hr, P^.bal = 1.** В этом случае предыдущая несбалансированность уравновешивается;
- **2. hl = hr, P^.bal = 0** т.е.левое дерево стало перевешивать**.**
- **3. hl > hr, P^.bal = -1** т.е. необходима балансировка.
- Операция по балансировке состоит только из последовательных переприсваиваний ссылок. Фактически ссылки циклически меняются, что приводит к одно- или двукратному повороту двух или трех участвующих в процессе балансировки вершин. Кроме вращения, необходимо должным образом изменять и показатели сбалансированности этих вершин (баланс-фактор).

Типы поворотов:

Одиночный правый поворот

(**RR**-rotation, single right rotation)

Одиночный левый поворот

(**LL**-rotation, single left rotation)

Двойной лево-правый поворот (большой)

(**LR**-rotation, double left-right rotation)

Двойной право-левый поворот (большой)

(**RL**-rotation, double right

```
struct node // структура для представления узлов дерева
 {
    int key;
    unsigned char height;
    node* left;
    node* right;
   node(int k)  { key = k; left = right = 0; height = 1; }
};
```

три вспомогательные функции, связанные с высотой

1-я является оберткой для поля height, она может работать и с нулевыми указателями (с пустыми деревьями):

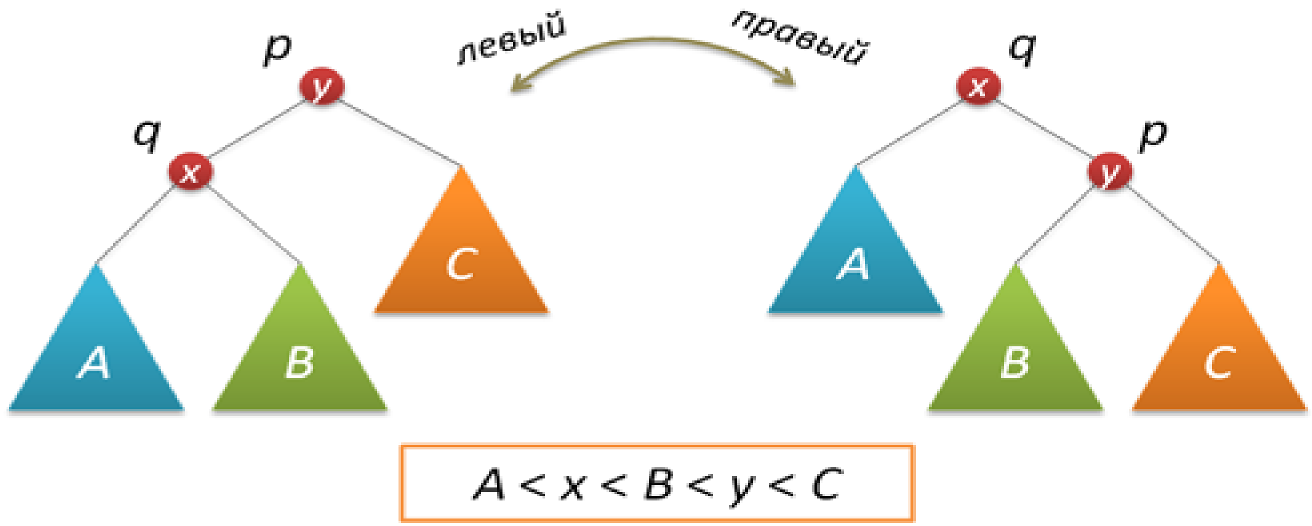
```
unsigned char height(node* p)
 {     return p?p->height:0;    }
```

2-я вычисляет balance factor заданного узла (работает только с ненулевыми указателями):

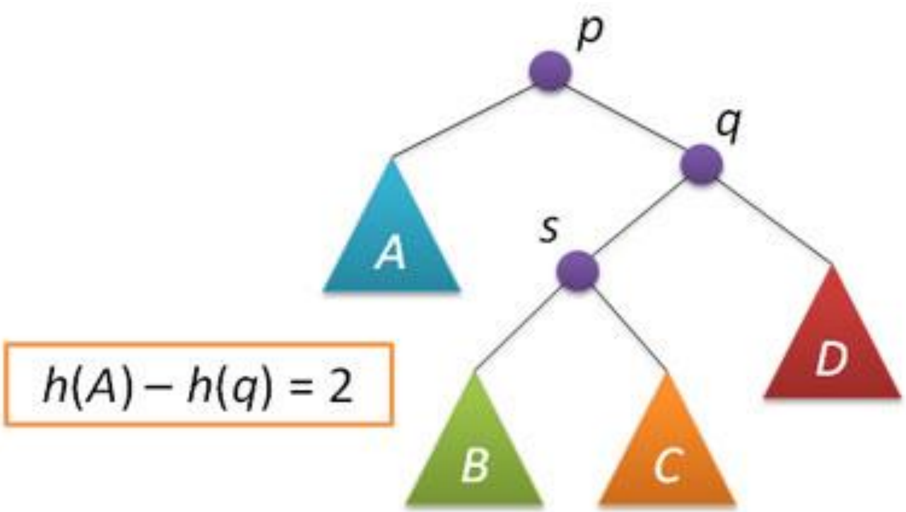
```
int bfactor(node* p)
 {     return height(p->right)-height(p->left);    }
```

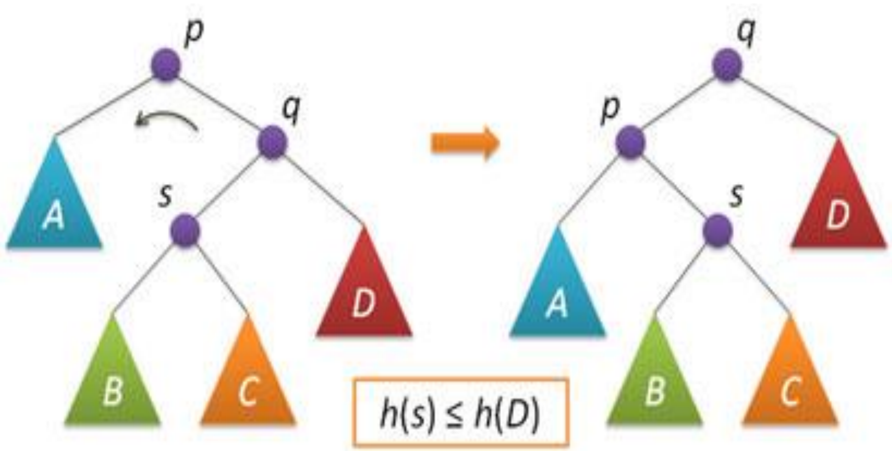
3-я восстанавливает корректное значение поля height заданного узла (при условии, что значения этого поля в правом и левом дочерних узлах являются корректными):

```
void fixheight(node* p)
 {    unsigned char hl = height(p->left);
     unsigned char hr = height(p->right);
     p->height = (hl>hr?hl:hr)+1;
}
```




```
node* rotateright(node* p) // правый поворот вокруг p
    {   node* q = p->left;
        p->left = q->right;
        q->right = p;
         fixheight(p);
         fixheight(q);
         return q;
    }
```

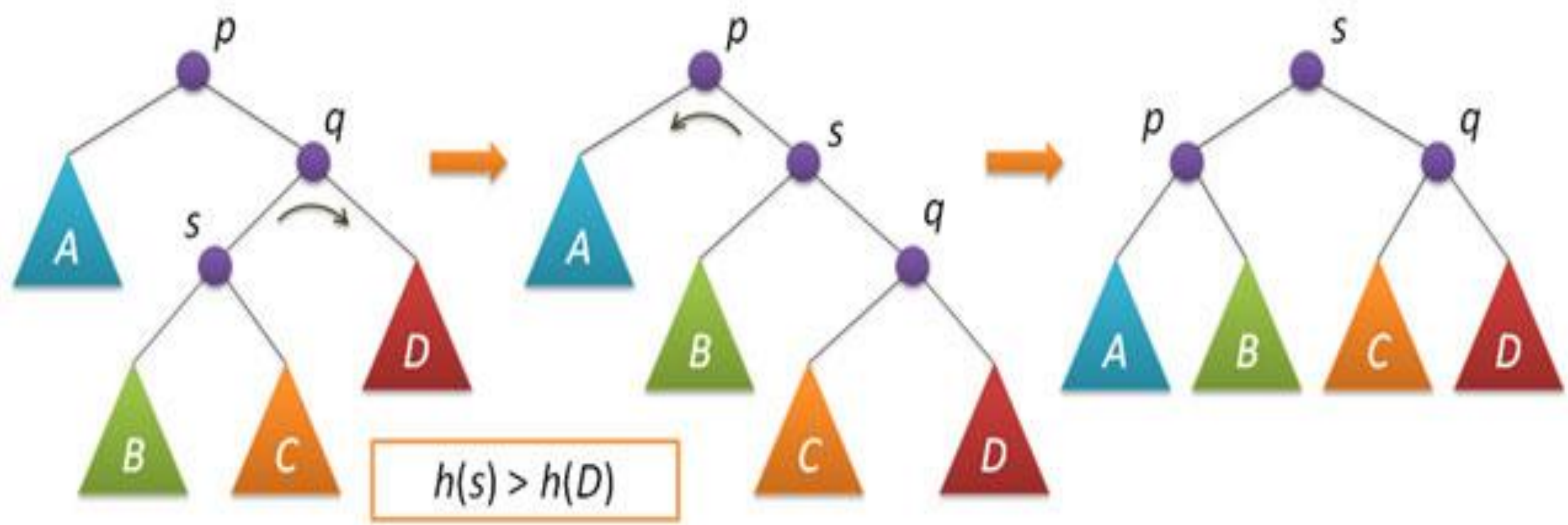




Анализ возможных случаев в рамках данной ситуации показывает, что для исправления расбалансировки в узле p достаточно выполнить либо простой поворот влево вокруг p, либо так называемый *большой поворот* влево вокруг того же p. Простой поворот выполняется при условии, что высота левого поддерева узла q

Большой поворот

Большой поворот применяется при условии h(s)>h(D) и сводится в данном случае к двум простым — сначала правый поворот вокруг q и затем левый вокруг p.



балансировка

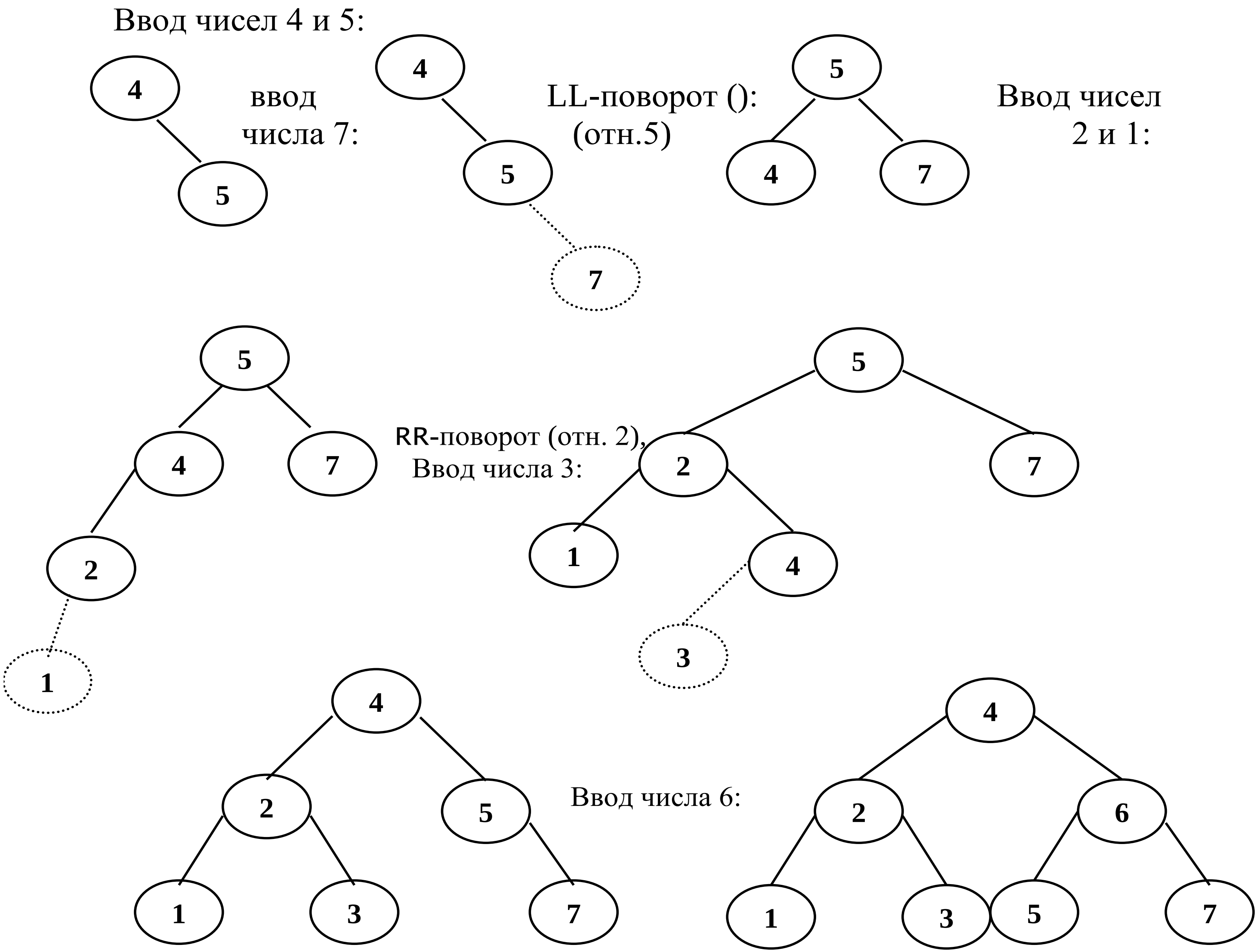
```
node* balance(node* p) // балансировка узла p
{
	fixheight(p);
	if( bfactor(p)==2 )
		{
			if( bfactor(p->right) < 0 )
				p->right = rotateright(p->right);
			return rotateleft(p);
		}
	if( bfactor(p)==-2 )
		{
			if( bfactor(p->left) > 0 )
				p->left = rotateleft(p->left);
			 return rotateright(p);
		}
	return p; // балансировка не нужна
 }
```

- Описанные функции поворотов и балансировки не содержат ни циклов, ни рекурсии, а значит выполняются за постоянное время, не зависящее от размера АВЛ-дерева.
- Из-за условия балансированности высота дерева O(log(N)),
- где N- количество вершин, поэтому добавление элемента требует O(log(N)) операций.

Вставка ключа

Вставка нового ключа в АВЛ-дерево выполняется, так же, как это делается в простых деревьях поиска: спускаемся вниз по дереву, выбирая правое или левое направление движения в зависимости от результата сравнения ключа в текущем узле и вставляемого ключа. Единственное отличие заключается в том, что при возвращении из рекурсии (т.е. после того, как ключ вставлен либо в правое, либо в левое поддерево, и это дерево сбалансировано) выполняется балансировка текущего узла. Строго доказывается, что возникающий при такой вставке дисбаланс в любом узле по пути движения не превышает двух, а значит применение вышеописанной функции балансировки является корректным.

балансировка



Удаление ключей

основана на алгоритме удаления из дерева, т.е., замене удаляемой вершины на самого левого потомка из правого поддерева или на самого правого потомка из левого поддерева, с учетом операции балансировки (тех же поворотов узлов).

При выходе из рекурсии не забываем выполнить балансировку узлов.

- Г. М. Адельсон-Вельский и Е. М. Ландис доказали теорему, согласно которой высота АВЛ-дерева с N внутренними вершинами заключена между $\log_2(N+1)$ и $1.4404*\log_2(N+2)-0.328$, то есть высота АВЛ-дерева никогда не превысит высоту идеально сбалансированного дерева более, чем на 45 %. Для больших N имеет место оценка $1.04*\log_2(N)$. Таким образом, выполнение основных операций 1-3 (слайд 71) требует порядка $\mathbf{\log_2(N)}$ сравнений. Экспериментально выяснено, что одна балансировка приходится на каждые два включения и на каждые пять исключений.

Деревья оптимального поиска

- Построены по вероятности появления ключей или обращения к ключам.
- n - количество вершин
- Pi –вероятность обращения для Ki-й вершины
- Сумма всех вероятностей =1, т.е.

$$\sum_{1}^{n} Pi = 1$$

Например:

Ключи: 1, 2, 3 с вероятностью соотв. 1/7, 2/7, 4/7.

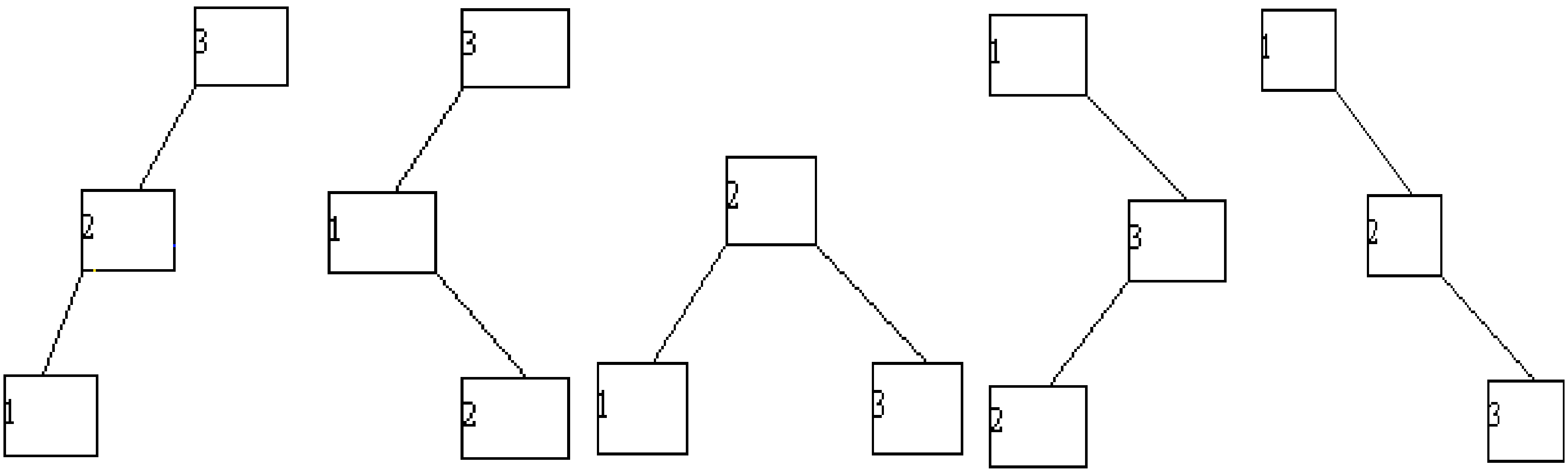
Для построенных деревьев, считая, что корень дерева имеет высоту =1, то:

Взвешенная длина пути = Pi*Hi (1<=I<=n)

- P(a) = 1*4/7 + 2*2/7 + 3*1/7 = 11/7
- P(b) = 1*4/7 + 2*1/7 + 3*2/7 = 12/7
- P(c) = 1*2/7 + 2*1/7 + 2*4/7 = 12/7
- P(d) = 1*1/7 + 2*4/7 + 3*2/7 = 15/7
- P(e) = 1*4/7 + 2*2/7 + 3*4/7 = 17/7

Дерево с минимальной взвешенной длиной пути – а)

Деревья оптимального поиска



(a) (b) (c) (d) (e)

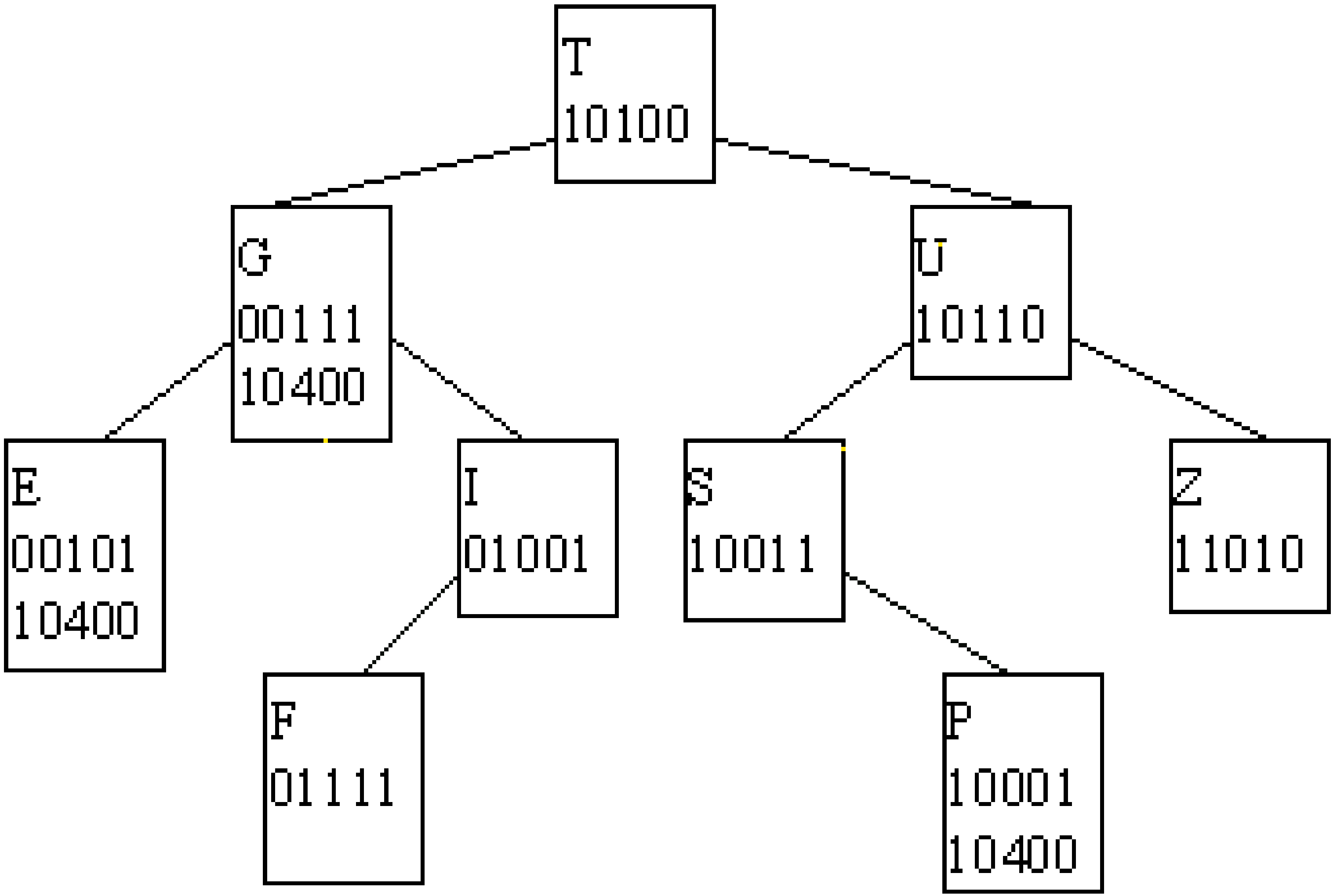
- На практике приходится решать несколько более общую задачу, а именно, при построении дерева учитывать вероятности неудачного поиска, т.е. поиска ключа, не включенного в дерево. В частности, при реализации сканера желательно уметь эффективно распознавать идентификаторы, которые не являются ключевыми словами. Можно считать, что поиск по ключу, отсутствующему в дереве, приводит к обращению к "специальной" вершине, включенной между реальными вершинами с меньшим и большим значениями ключа соответственно. Если известна вероятность **qj** обращения к специальной **j**-той вершине, то к общей средней взвешенной длине пути дерева необходимо добавить сумму **qj*ej** для всех специальных вершин, где **ej** - высота **j**-той специальной вершины.

- При построении дерева оптимального поиска вместо значений **pi** и **qj** обычно используют полученные статистически значения числа обращений к соответствующим вершинам.
- Т.О.избегается вычисление вероятностей по измеренным частотам, а еще имеется выигрыш от использования целых чисел для построения оптимального дерева.
- Процедура построения дерева оптимального поиска достаточно сложна и опирается на тот факт, что любое поддерево дерева оптимального поиска также обладает свойством оптимальности. Поэтому известный алгоритм строит дерево "снизу-вверх", т.е. от листьев к корню.
- Сложность этого алгоритма и расходы по памяти составляют $O(n^2)$. Имеется эвристический алгоритм, дающий дерево, близкое к оптимальному, со сложностью $O(n*\log n)$ и расходами памяти - $O(n)$.

Деревья цифрового поиска

- Методы цифрового поиска достаточно громоздки и плохо иллюстрируются. Поэтому кратко остановимся на наиболее простом механизме - бинарном дереве цифрового поиска. Как и в деревьях, рассмотренных ранее, в каждой вершине такого дерева хранится полный ключ, но переход по левой или правой ветви происходит не путем сравнения ключа-аргумента со значением ключа, хранящегося в вершине, а на основе значения очередного **бита** аргумента.
- Поиск начинается от корня дерева. Если содержащийся в корневой вершине ключ не совпадает с аргументом поиска, то анализируется самый левый бит аргумента. Если он равен 0, происходит переход по левой ветви, если 1 - по правой. Если не обнаруживается совпадение ключа с аргументом поиска, то анализируется следующий бит аргумента и т.д., пока либо не будут проверены все биты аргумента, или мы не наткнемся на вершину с отсутствующей левой или правой ссылкой.

- На рисунке показан пример дерева цифрового поиска для некоторых заглавных букв латинского алфавита. Считается, что буквы кодируются в соответствии с кодовым набором ASCII, а для их представления и поиска используются 5 младших бит кода. Например, код буквы A равен 41(16), а представляться A будет как последовательность бит 00001.


T
10100
G
00111
10400
U
10110
E
00101
10400
I
01001
S
10011
Z
11010
F
01111
P
10001
10400

- При построении оптимального дерева мы не будем требовать, чтобы сумма всех p и q равнялась 1 . Ведь обычно эти вероятности определяются экспериментально, подсчетом обращений к вершинам. Поэтому вместо вероятностей pi и qj мы далее будем пользоваться просто "счетчиками" частоты обращений. Обозначим их следующим образом:
- ai = число поисков с аргументом x, равным ki
- bj = число поисков с аргументом x, лежащим между ki и kj+1.
-
- Условимся, что b0 — число поисков с аргументом x, меньшим k1, a bn — число поисков с x, большим чем kn (рис. 4.37). Далее, вместо средней длины пути мы будем обозначать через P *общую взвешенную длину пути*:
- P = (**S**i : 1 <= i <= n : ai * hi) + (**S**j : 0 <= j <= m: bj *h’j)

- Т. О., мало того, что избегается вычисление вероятностей по измеренным частотам, но еще имеется выигрыш от использования целых чисел для построения оптимального дерева.
-

- Учитывая, что число возможных конфигураций из n вершин растет экспоненциально с ростом n, задача построения оптимального дерева при больших n кажется совершенно безнадежной. Однако оптимальные деревья обладают одним важным свойством, которое помогает их обнаруживать: все их поддеревья тоже оптимальны. Если, например, дерево на рис. 4.37 оптимально, то поддерево с ключами k3 и k1, как показано, также оптимально. Такая особенность предполагает алгоритм, который систематически находит все большие и большие деревья, начиная с отдельных вершин как наименьших возможных поддеревьев. Таким образом, дерево растет "от листьев к корню", "снизу вверх", если учесть, что мы рисуем деревья сверху вниз [4.7].

- В основе нашего алгоритма лежит уравнение (4.60). Пусть P — взвешенная длина пути всего дерева, а PL и PR — соответствующие длины для левого и правого поддеревьев его корня.
- Ясно, что P — сумма PL и PR и числа случаев W, когда поиск проходит по единственному пути к корню, т. е. W — просто общее число поисков:
-
- P = PL + W + PR

(4.60)

-

- $W = (\mathbf{S}i: 1 <= i <= n:ai) + (\mathbf{S}j: 0 <= j <= m: bj)$ **(4.61)**
-
- Назовем W *весом* дерева. Тогда средняя длина пути будет P/ W.
- Из этих рассуждений видно, что необходимо ввести обозначения для веса и длины пути любого из поддеревьев, включающего "соседние" ключи. Пусть Tij — оптимальное поддерево, состоящее из вершин с ключами ki+1, ki+2, …, kj. Тогда обозначим его вес через wij, а длину пути — через pij-. Ясно, что P = p0,n и W= wo,n. Эти величины определяются следующими рекуррентными соотношениями:
- wii = bi (0 <= i <= n),
- wij = wi,j-1 +aj + bj (0 <= i < j <= n), **(4.62)**
-
- pii = wii, (0 <= i <= n), **(4.63)**
- pij = wij + **MIN** *k* : i < *k* <= j : (pi,k-1 + pkj) (0 <= i < j <= n)
-

- Последнее равенство следует непосредственно из (4.60) и определения оптимальности. Поскольку существует около n2/2 значений pij, а (4.63) требует выбора одного из 0 <j-i<=n значений, то весь процесс минимизации будет занимать приблизительно n3/6 операций. Кнут отмечает, что можно избавиться от одного множителя n и тем самым сохранить практическую ценность данного алгоритма. Делается это так.
-
- Пусть rij — значение k, при котором в (4.63) достигается минимум. Поиск rij можно ограничить значительно меньшим интервалом, т. е. сократить число вычислений до j-i. Это можно сделать, поскольку если мы нашли корень rij оптимального поддерева Tij, то ни расширение при добавлении к дереву справа новой вершины, ни сжатие при отбрасывании самого левого узла не могут сдвинуть вправо, этот оптимальный корень. Такое свойство выражается отношением
- ri,j-1 <= rij <= ri+1,j, **(4.64)**
- которое и ограничивает поиск возможных решений для rij диапазоном ri,j-1… ri+1,j. Это в конце концов и ограничивает число элементарных шагов — около n2.

- Теперь мы уже готовы составить детальный алгоритм оптимизации. Введем следующие определения исходя из того, что Tij -оптимальные деревья, содержащие узлы с ключами ki+1 ... kj: