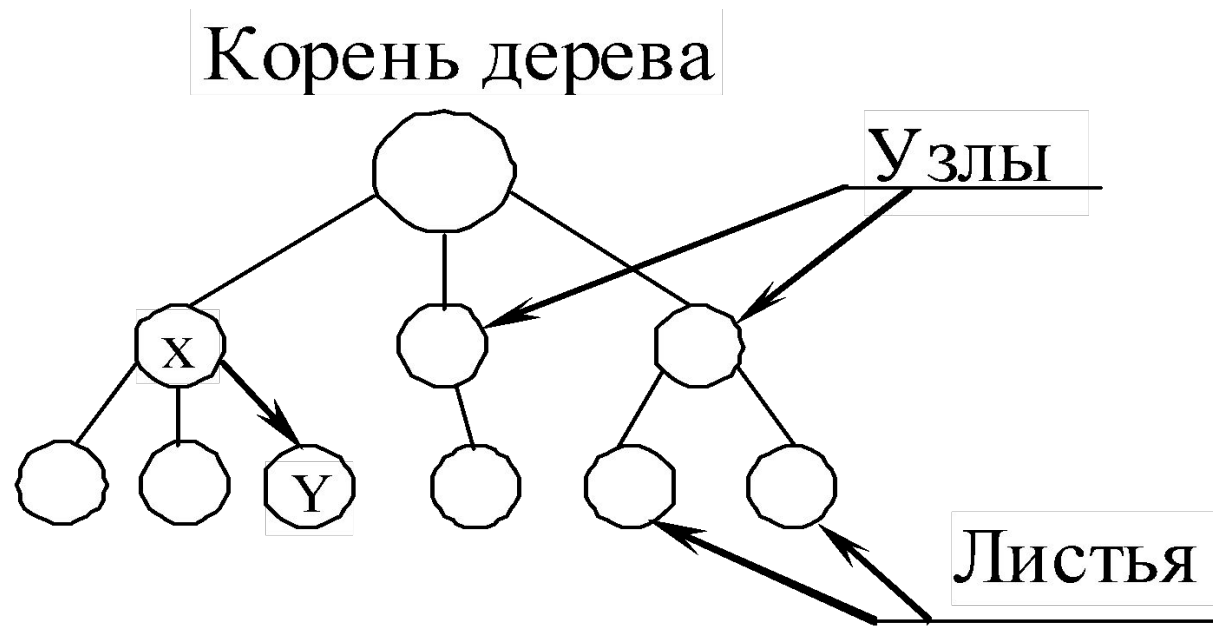


Нелинейные структуры данных

Введение в динамические данные двух и более указателей позволяет получить нелинейные структуры. Наиболее распространенными являются структуры, имеющие следующий вид:



Такая структура получила название «дерево».

Дерево состоит из элементов, называемых *узлами* (вершинами). Узлы соединены между собой дугами. В случае $X \rightarrow Y$, вершина X называется *родителем*, а Y – *сыном* (дочерью).

Дерево имеет единственный узел, не имеющий родителей (ссылок на этот узел), который называется *корнем*. Любой другой узел имеет ровно одного родителя, т.е. на каждый узел дерева имеется ровно одна ссылка.

Узел, не имеющий сыновей, называется *листом*.

Внутренний – это узел, не являющийся ни листом, ни корнем.

Порядок узла равен количеству его узлов-сыновей.

Степень дерева – максимальный порядок его узлов.

Высота (глубина) узла равна числу его родителей плюс один.

Высота дерева – это наибольшая высота его узлов.

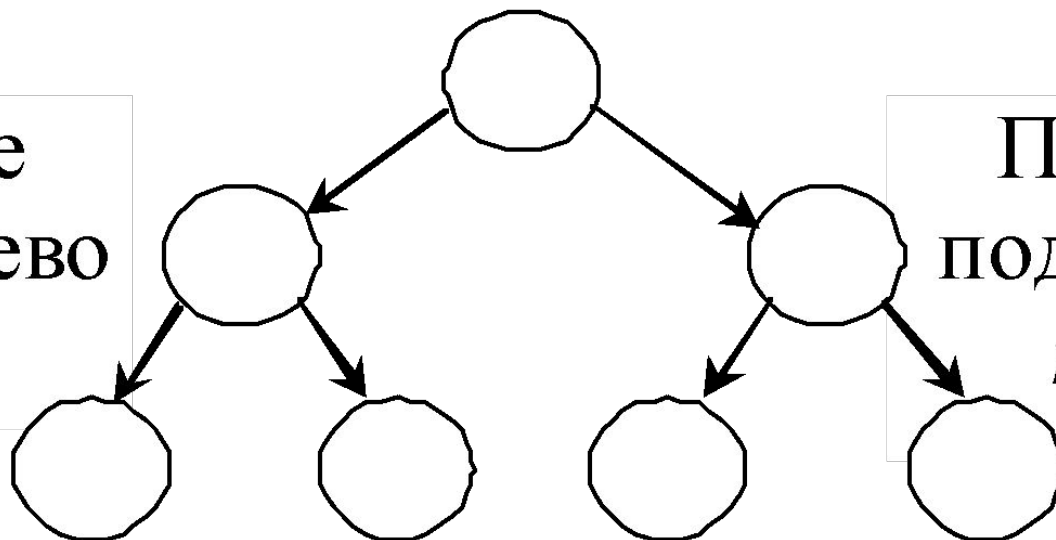
Бинарные деревья

Бинарное – это дерево, в котором каждый узел-родитель содержит, кроме данных, не более двух сыновей (левый и правый).

Пример бинарного дерева (корень обычно изображается сверху, хотя изображение можно и перевернуть):

Корень дерева *root*

Левое
поддерево
left



Правое
поддерево
right

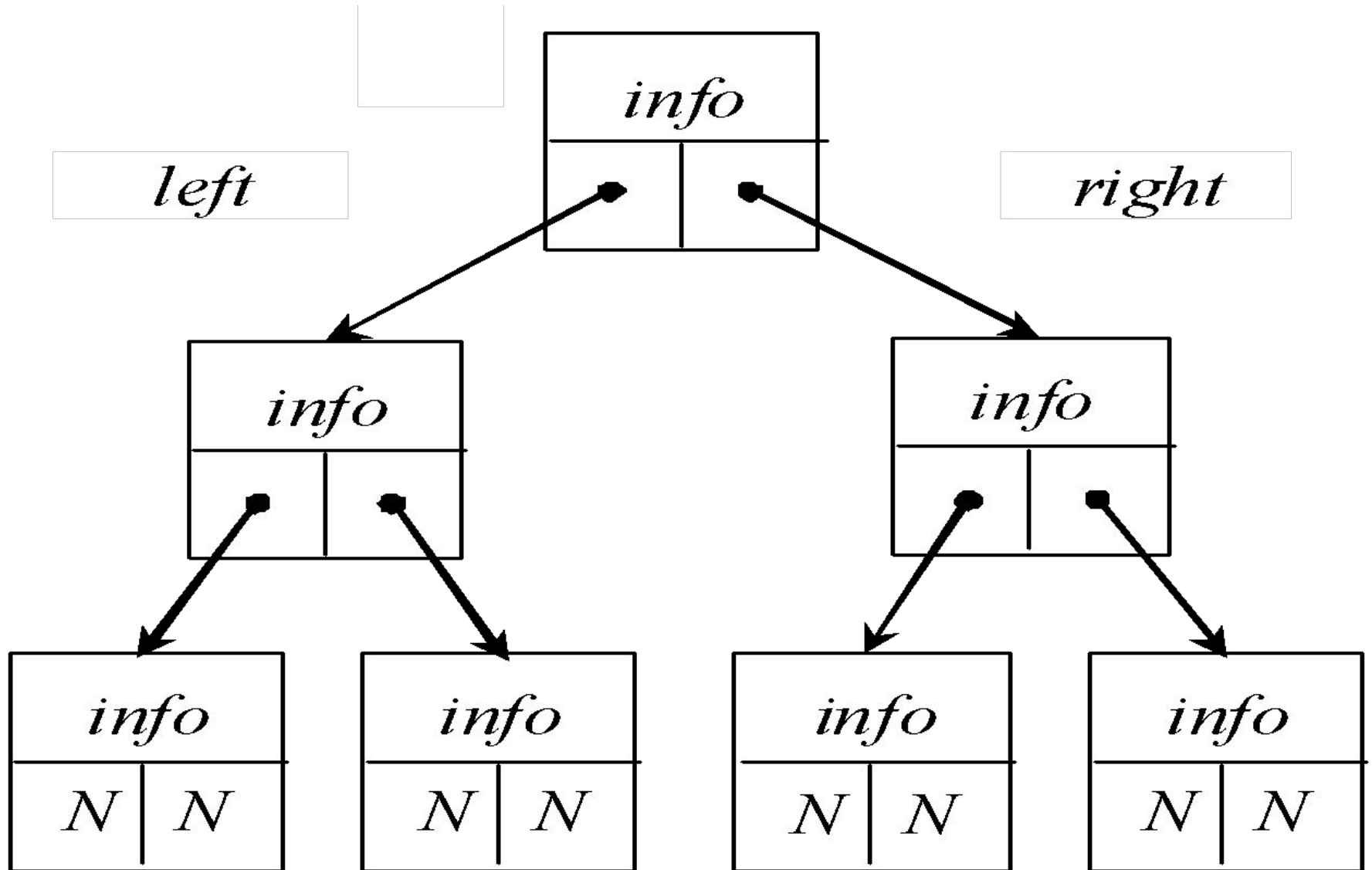
Для работы с бинарными деревьями объявляется структурный тип (шаблон) следующего вида:

```
struct Tree {  
    Информационная Часть (info)  
    Tree *left, *right;    – Адресная Часть  
} * root;
```

Адресная часть – указатели на левую *left* и правую *right* ветви;

root – указатель корня.

Такая структура данных организуется следующим образом ($N - NULL$):



Высота дерева определяется количеством уровней, на которых располагаются его узлы.

Если дерево организовано таким образом, что для каждого узла все ключи его левого поддеревья меньше ключа этого узла, а все ключи его правого поддеревья – больше, оно называется *деревом поиска*. Одинаковые ключи здесь не допускаются.

Древовидные структуры удобны и эффективны для быстрого поиска информации.

Сбалансированными называются деревья, для каждого узла которых количество узлов в его левом и правом поддеревьях различаются не более чем на единицу.

Дерево является *рекурсивной* структурой данных, поскольку каждое его поддерево также является деревом. В связи с этим действия с такими структурами чаще всего описываются с помощью рекурсивных алгоритмов.

Основные алгоритмы

При работе с деревьями необходимо уметь:

- создать дерево;
- добавить новый элемент;
- просмотреть все элементы дерева;
- найти элемент с указанным значением;
- удалить заданный элемент;
- освободить память.

Обычно бинарное дерево строится сразу в виде *дерева поиска*.

Все алгоритмы работы с деревьями будем рассматривать для данных, информационной частью которых являются целые числа (*ключи*).

Структурный тип будет иметь вид:

```
struct Tree {  
    int info;           // ИЧ  
    Tree *left, *right; // АЧ  
} *root;                // Лучше локально!!!
```

Создание дерева

Сначала создаем КОРЕНЬ (лист). Далее для того чтобы добавить новый элемент, необходимо найти для него место.

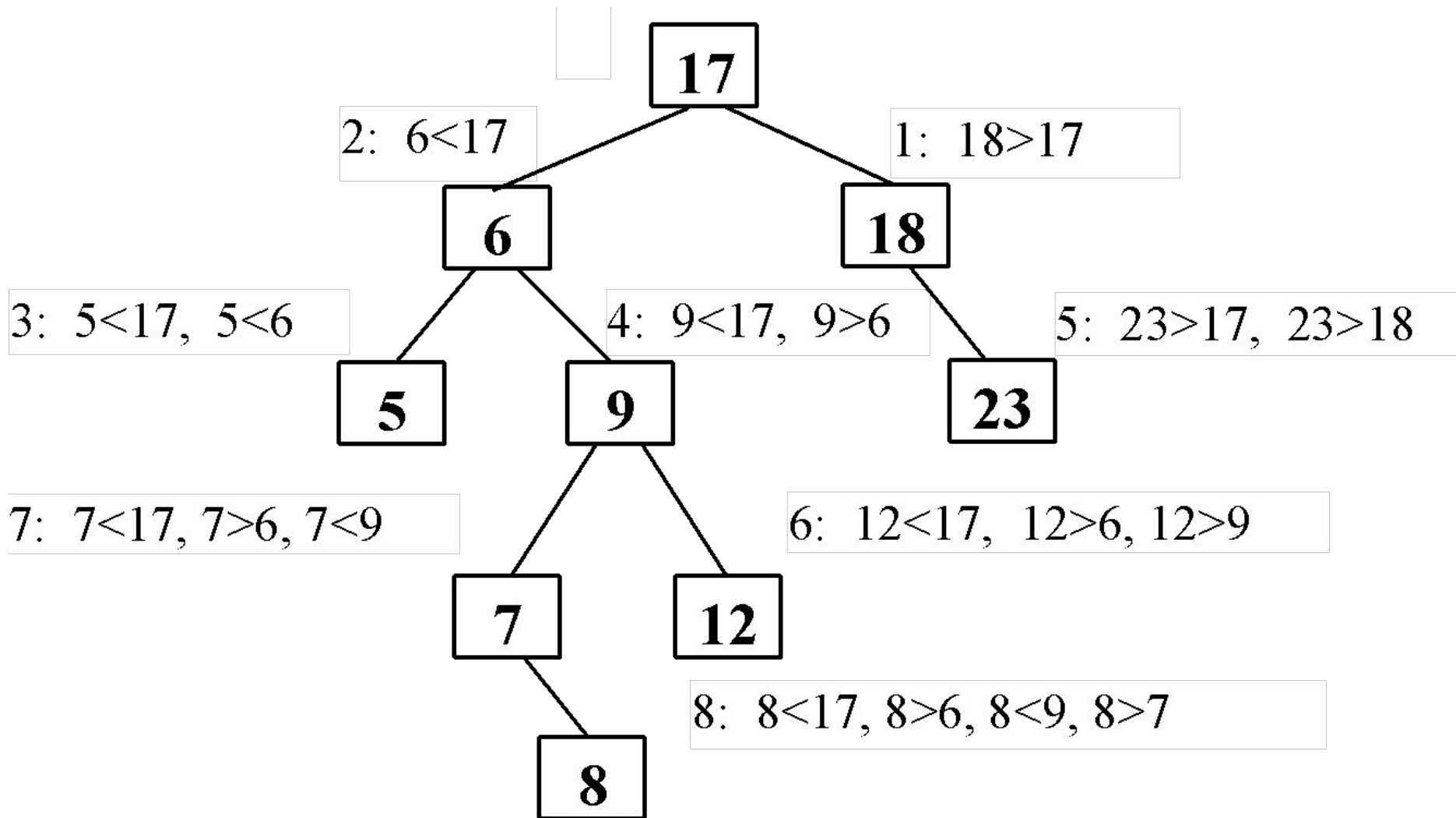
Для этого, начиная с корня, сравниваем значения узлов ($t \rightarrow info$, где t – текущий указатель) со значением нового элемента (b).

Если $b < t \rightarrow info$, то идем по левой ветви, в противном случае – по правой ветви.

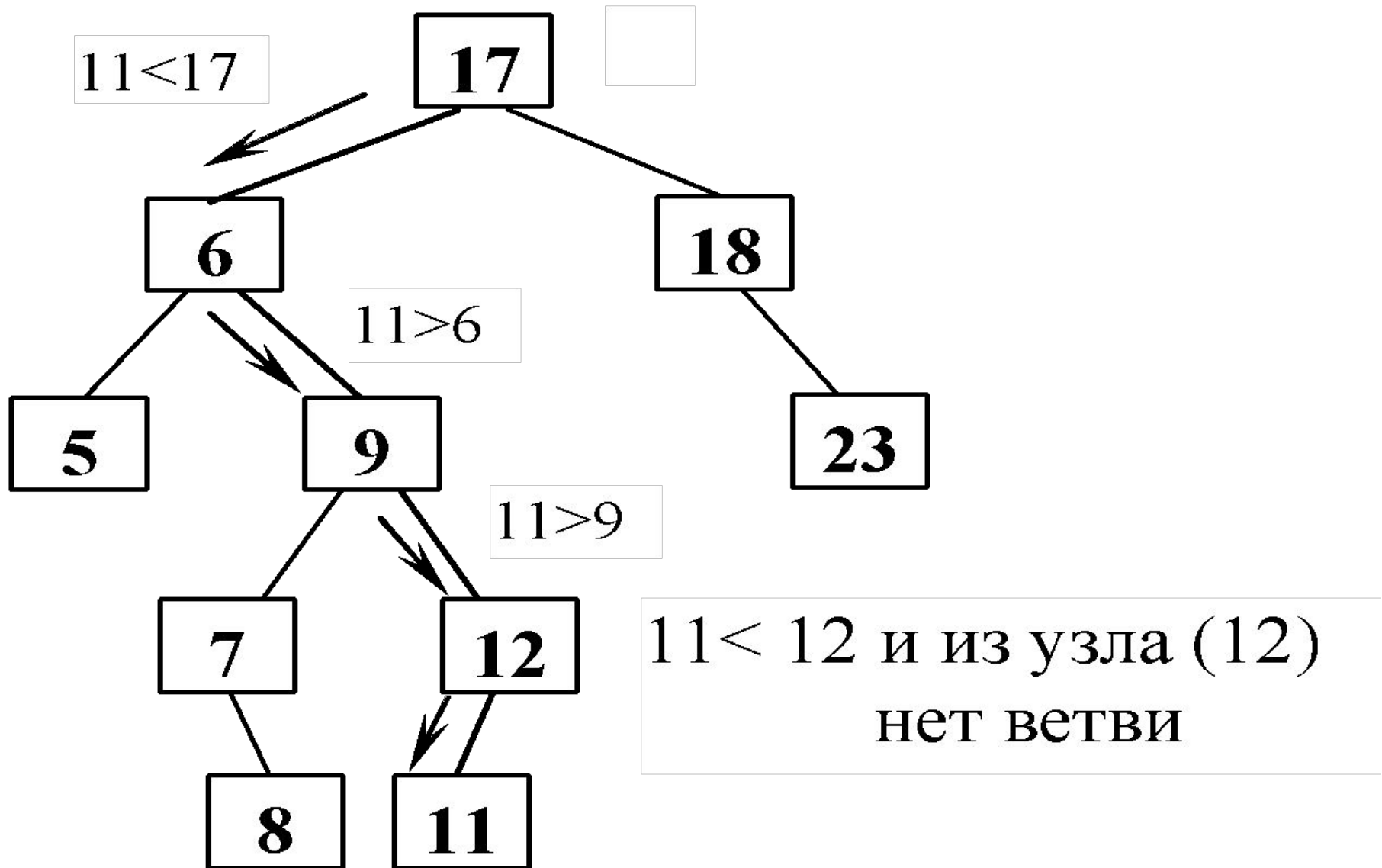
Когда дойдем до узла, из которого не выходит нужная ветвь для дальнейшего поиска, это означает, что место под новый элемент найдено.

Пример создания дерева

Построим дерево поиска для следующих ключей
17, 18, 6, 5, 9, 23, 12, 7, 8:



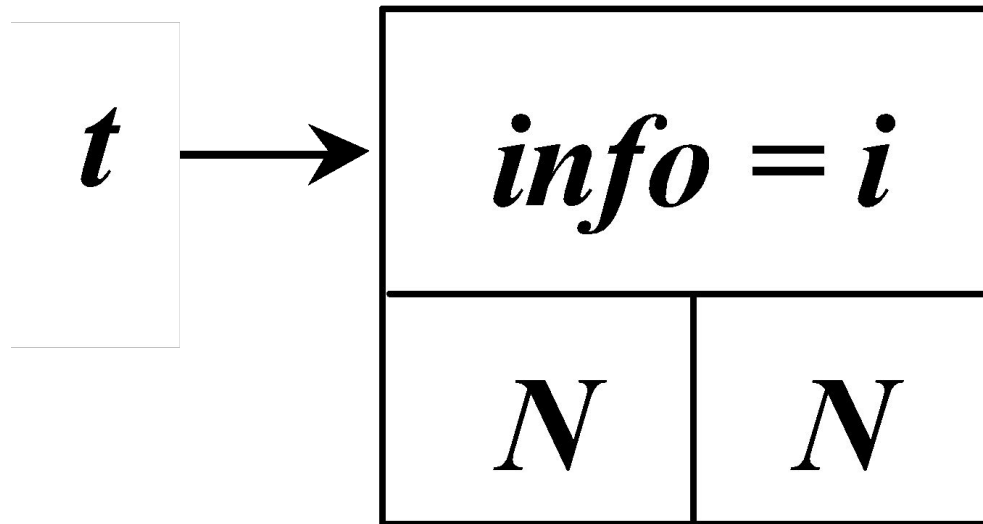
Поиск места для узла **11** в построенном дереве:



Рассмотрим функцию *List* для создания нового элемента ЛИСТА (*i* – информационная часть (ИЧ), в нашем случае *ключ*):

```
Tree* List ( int i )  
{  
    Tree *t = new Tree; // Захват памяти  
    t -> info = i;      // Формирование ИЧ  
    // Формирование адресных частей  
    t -> left = t -> right = NULL;  
    return t;          // Адрес созданного листа  
}
```

В результате выполнения функции *List* создается новый элемент-лист ($N - NULL$):



Функция создания дерева из N ключей с добавлением:

```
Tree* Create (Tree *root) {
    Tree *Prev, *t;
// Prev – родитель текущего элемента
    int N, b, find;
    cout << " N = ";    cin >> N;
    if ( ! root ) {    N--;    // Если дерево не создано
        cout << " Input Root : “;    cin >> b;
        root = List (b);    /* Создаем адрес корня root,
который первоначально – лист*/
    }
```



```
//----- Добавление элементов -----
```

```
for (int i=1; i <= N; ++i) {  
    cout << "Input Info : ";  
    cin >> b;
```

```
// Текущий указатель установили на корень
```

```
t = root;
```

```
find = 0;      // Признак поиска
```

```
while ( t && ! find) {  
    Prev = t;  
    if( b == t->info) {  
        find = 1;  
        cout << " Такой уже есть! " << endl;  
    }  
    // Ключи должны быть уникальны  
    else  
        if ( b < t -> info ) t = t -> left; // На лево  
        else t = t -> right; // На право  
}
```

```
// Если нашли место с адресом Prev
  if ( ! find ) {      // if (find == 0)
// Создаем новый узел, являющийся листом
    t = List ( b );
// и присоединяем его, либо
    if ( b < Prev -> info )
// на левую ветвь,
        Prev -> left = t;
// либо на правую ветвь
    else Prev -> right = t;
  } // Конец if
} // Конец цикла for()
return root;
}
```

Участок кода с обращением к функции *Create* будет иметь вид:

```
...  
Tree *root = NULL;      // Указатель корня  
...  
root = Create (root);  
...
```

Рассмотрим функцию добавления *одного* листа в уже созданное дерево:

```
void Add_List (Tree *root, int key)
{
    Tree *prev, *t;
    int find = 1;
    t = root;
    while ( t && find) {
        prev = t;
        if( key == t -> info) {
            find = 0;
            cout << " Такой уже есть !" << endl;
        }
    }
```

```
else
  if ( key < t -> info ) t = t -> left;
  else
    t = t -> right;
}
if (find) {
  t = List(key);
  if ( key < prev -> info )
    prev -> left = t;
  else
    prev -> right = t;
}
}
```

Участок кода с обращением к функции *Add_List* может иметь вид:

```
...  
cout << " Input info : ";  
cin >> in;  
if (root == NULL)  
    root = List (in);  
else  
    Add_List ( root, in );  
...
```

Удаление узла

Удаление узла из дерева зависит от того, сколько сыновей (потомков) имеет удаляемый узел. Возможны три варианта.

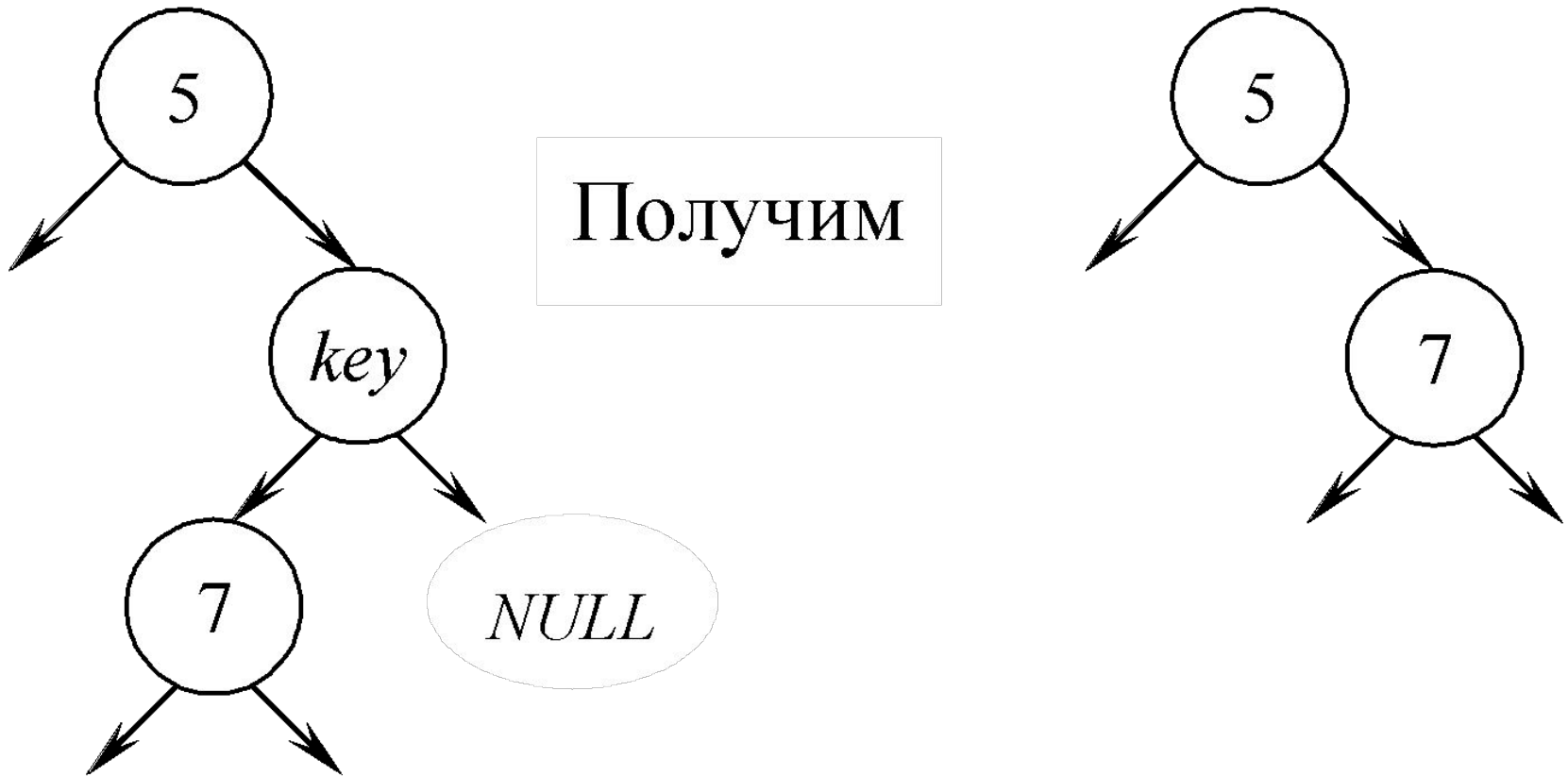
1. Удаляемый узел является *листом* – просто удаляем ссылку на него.

Пример схемы удаления *листа* с ключом *key*:



2. Удаляемый узел имеет *одного потомка*, т.е. из удаляемого узла выходит ровно одна ветвь.

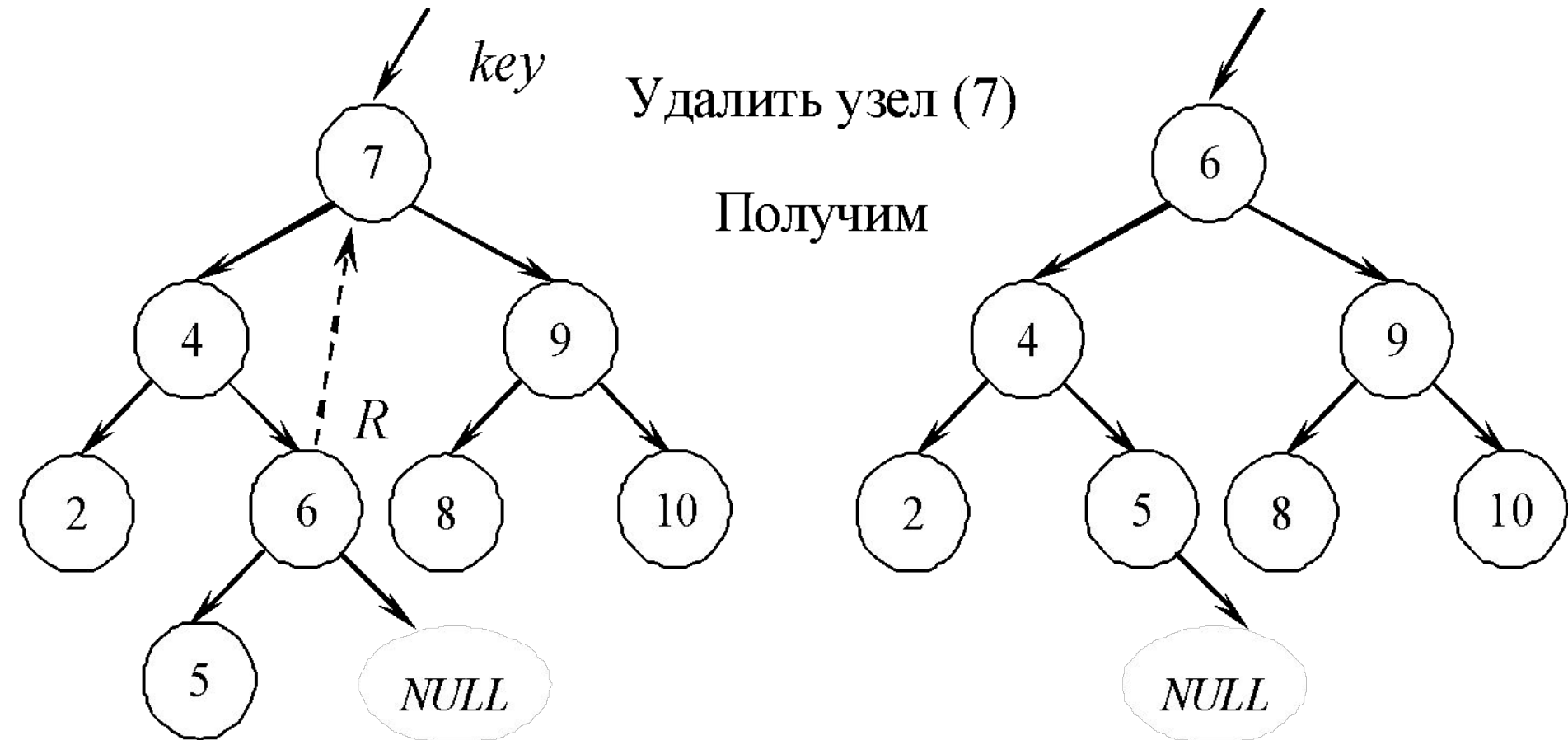
Пример схемы удаления узла *key*, имеющего одного сына:



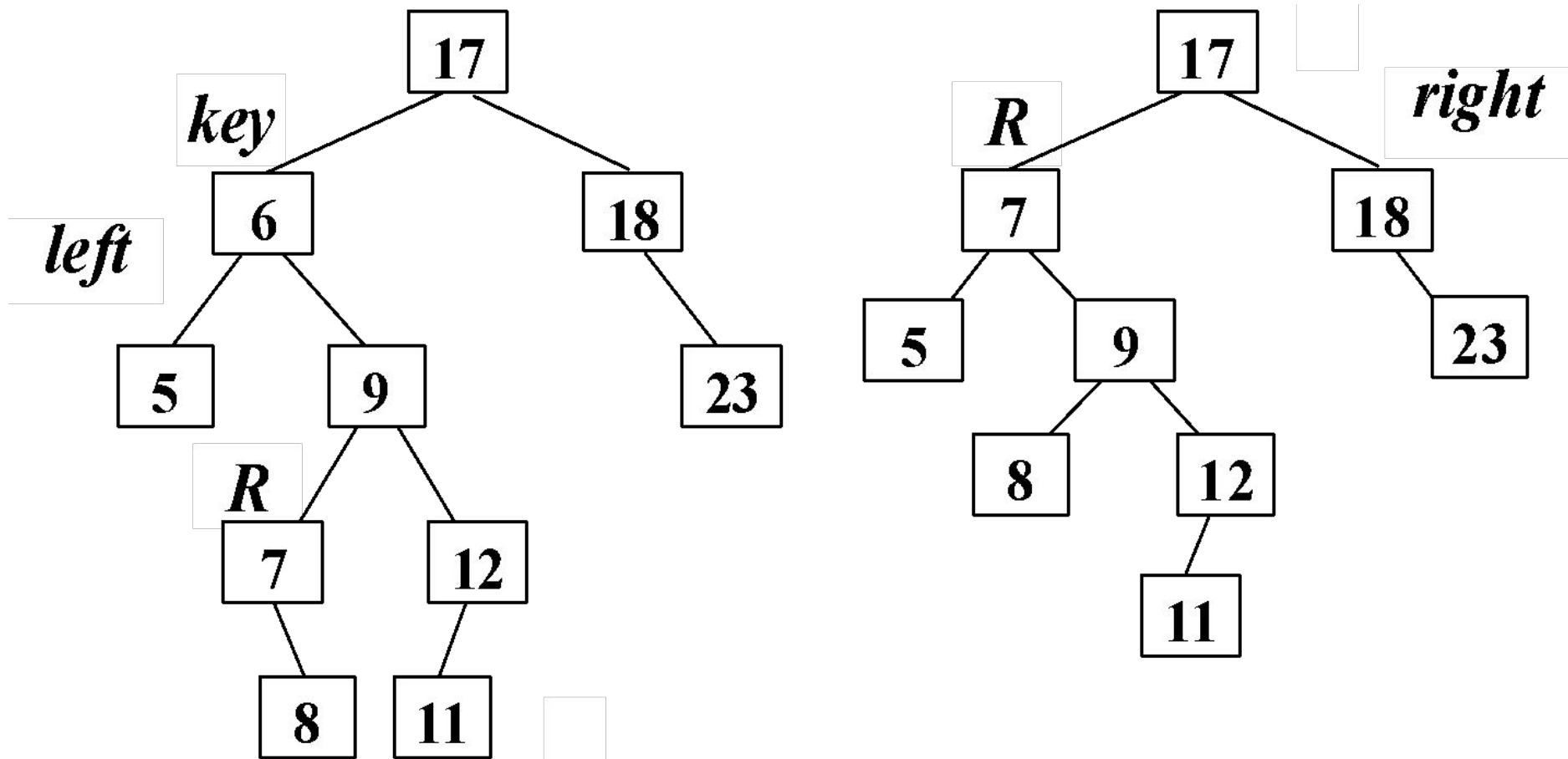
3. Удаление узла, имеющего *двух сыновей*, сложнее рассмотренных выше.

Если *key* – удаляемый узел, то его следует заменить узлом *R*, который содержит либо наибольший ключ в левом поддереве (самый правый, у которого указатель *right* равен *NULL*), либо наименьший ключ в правом поддереве (самый левый, у которого указатель *left* равен *NULL*).

Используя первое условие, находим узел R , который является самым правым узлом поддерева узла key , у него имеется только левый сын:



В построенном ранее дереве удалим узел *key* (6), используя второе условие, т.е. ищем самый левый узел в правом поддереве – это узел *R* (указатель *left* равен *NULL*):



Функция удаления узла по заданному ключу *key* может иметь вид:

```
Tree* Del (Tree *root, int key) {  
    Tree *Del, *Prev_Del, *R, *Prev_R;  
    /* Del, Prev_Del – удаляемый элемент и его  
пре-дыдущий (родитель);  
    R, Prev_R – элемент, на который заменяется  
удаленный, и его родитель; */  
    Del = root;  
    Prev_Del = NULL;
```

```
// Поиск удаляемого элемента и его родителя
while (Del != NULL && Del -> info != key) {
    Prev_Del = Del;
    if (Del -> info > key) Del = Del -> left;
        else Del = Del -> right;
}
if (Del == NULL) { // Элемент не найден
    cout << "\n NO Key!" << endl;
    return root;
}
```

```
// ----- Поиск элемента R для замены -----  
if (Del -> right == NULL) R = Del -> left;  
else  
    if (Del -> left == NULL) R = Del -> right;  
    else {  
// Ищем самый правый узел в левом поддереве  
    Prev_R = Del;  
    R = Del -> left;  
    while (R -> right != NULL) {  
        Prev_R = R;  
        R = R -> right;  
    }  
}
```

```
/* Нашли элемент для замены R и его родителя  
Prev_R */
```

```
if( Prev_R == Del)  
    R -> right = Del -> right;  
else {  
    R -> right = Del -> right;  
    Prev_R -> right = R -> left;  
    R -> left = Prev_R;  
}  
}
```



```
if (Del == root) root = R;
// Удаляя корень, заменяем его на R
else
/* Поддерево R присоединяем к родителю уда-
ляемого узла */
if ( Del ->info < Prev_Del -> info)
    Prev_Del -> left = R; // На левую ветвь
else
    Prev_Del -> right = R; // На правую
cout << " Delete " << Del -> info << endl;
delete Del;
return root;
}
```

Участок программы с обращением к данной функции будет иметь вид

...

```
cout << " Input Del Info : ";
```

```
cin >> key;
```

```
root = Del ( root, key );
```

Функция просмотра

Рекурсивная функция вывода *узлов* дерева:

```
void View ( Tree *t, int level ) {  
    if ( t ) {  
        View ( t -> right , level + 1 );  
        // Вывод узлов правого поддеревья  
        for ( int i=0; i < level; ++i )  
            cout << "  ";  
        cout << t -> info << endl;  
        View ( t -> left , level + 1 );  
        // Вывод узлов левого поддеревья  
    }  
}
```

Обращение к функции *View* будет иметь вид
View (root, 0);

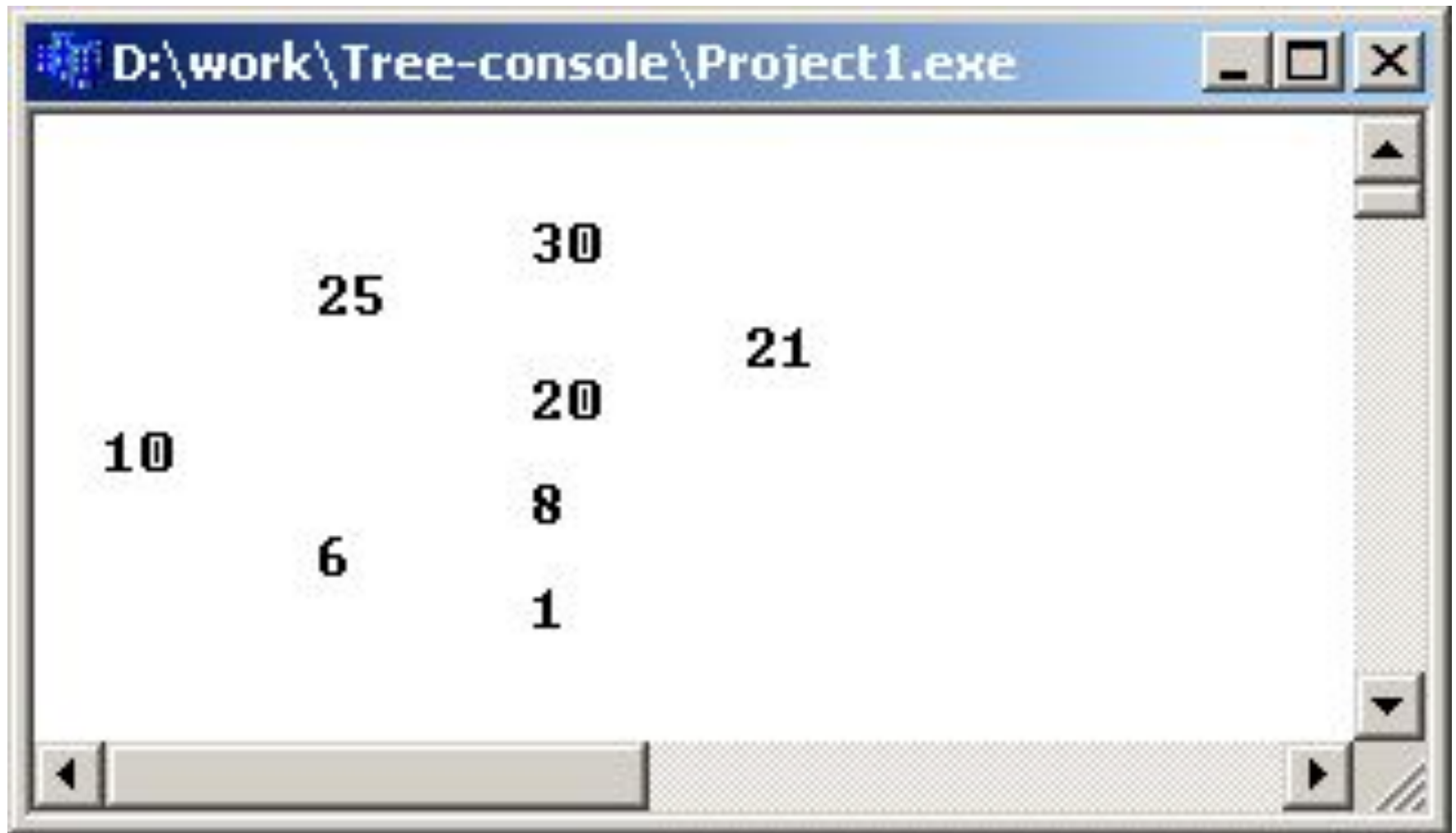
Второй параметр определяет уровень (*level*), на котором находится узел.

Корень находится на уровне 0. Значения узлов выводятся по горизонтали так, что корень находится слева.

Перед значением узла для имитации структуры дерева выводится количество пробелов, пропорциональное уровню узла.

Если закомментировать цикл печати пробелов, ключи будут выведены просто в столбик.

Для ключей 10 (корень), 25, 20, 6, 21, 8, 1, 30, построенное дерево будет выведено на экран функцией *View* в следующем виде:



Освобождение памяти

Функция освобождения памяти может быть реализована аналогично функции *View* :

```
void Del_All (Tree *t) {  
    if ( t != NULL) {  
        Del_All ( t -> left);  
        Del_All ( t -> right);  
        delete t;  
    }  
}
```

Что надо сделать с корнем после вызова этой функции???

Поиск узла с минимальным (макс.) ключом

```
Tree* Min_Key (Tree *p) { // Max_Key  
    while (p -> left != NULL)  
        p = p -> left;      // p = p -> right;  
    return p;  
}
```

Вызов функции для нахождения минимального ключа *p_min -> info* :

```
Tree *p_min = Min_Key (root);
```

Для построения сбалансированного дерева поиска из ключей необходимо сформировать массив (динамический), отсортировать его в порядке возрастания и после этого обратиться к функции

Make_Blns (root, 0, k, a); // ??????

k – размер массива.


```
void Make_Blns (Tree **p, int n, int k, int *a)
{
    if (n == k) {
        *p = NULL;
        return;
    }
    else {
        int m = (n + k) / 2;
        *p = new Tree;
        (*p) -> info = a[m];
        Make_Blns ( &(*p) -> left, n, m, a);
        Make_Blns ( &(*p) -> right, m+1, k, a);
    }
}
```

Алгоритмы обхода дерева

Существуют три алгоритма обхода деревьев, которые естественно следуют из самой структуры дерева.

1. Обход слева направо: *Left-Root-Right* (сначала посещаем левое поддерево, затем – корень и, наконец, правое поддерево).

2. Обход сверху вниз: *Root-Left-Right* (посещаем корень до поддеревьев).

3. Обход снизу вверх: *Left-Right-Root* (посещаем корень после поддеревьев).

Рассмотрим результаты этих обходов на примере записи формулы в виде дерева, т.к. они и позволяют получить различные формы записи арифметических выражений.

Пусть для операндов A и B выполняется операция сложения.

Привычная форма записи $A + B$ называется *инфиксной*.

Запись, в которой знак операции перед операндами $+AB$, называется *префиксной*.

Если операция после операндов $AB+$ это *постфиксная* форма.

Рассмотрим обходы дерева на примере формулы:

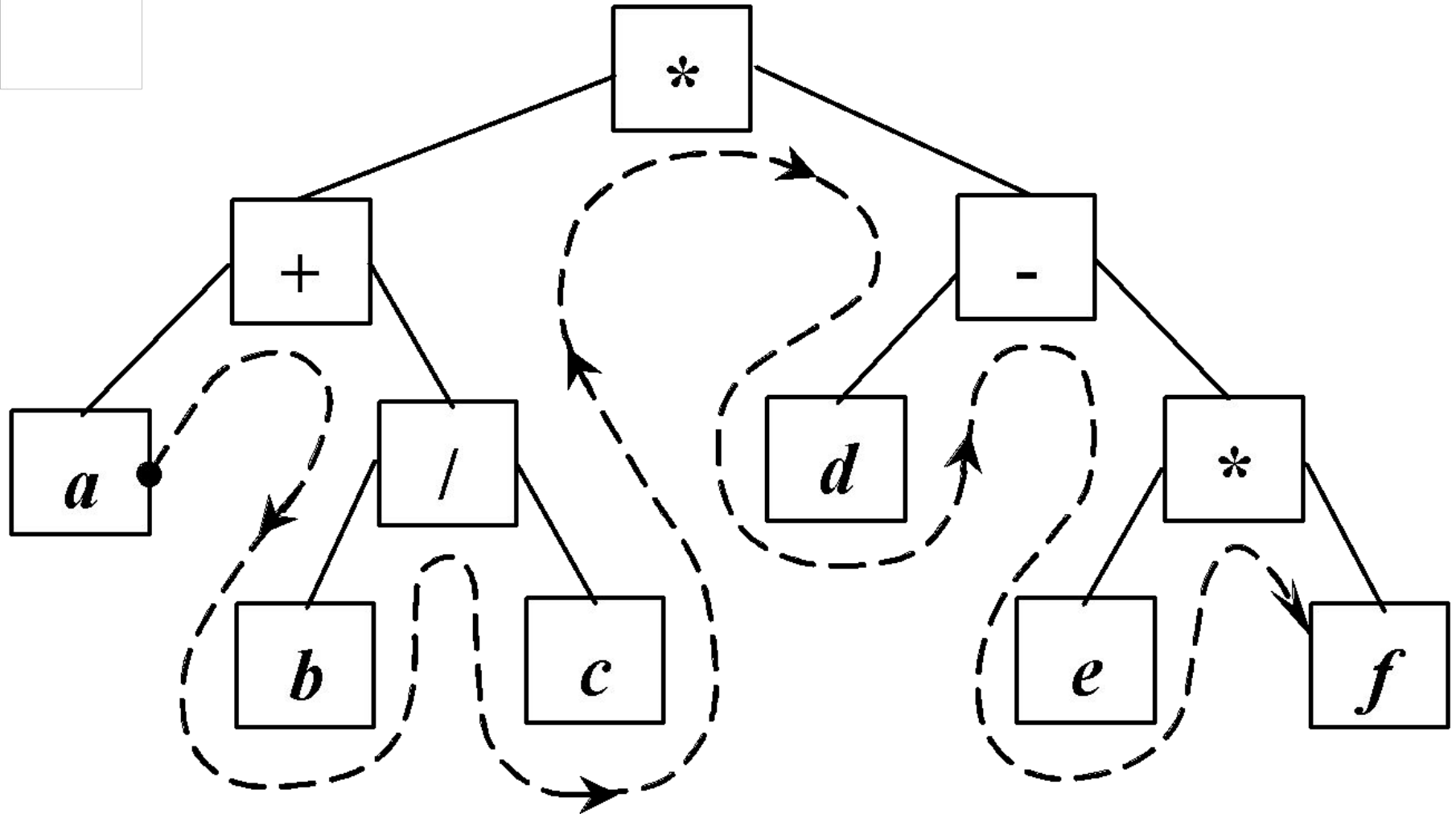
$$(a + b / c) * (d - e * f).$$

Дерево формируется по принципу:

- в корне размещаем операцию, которая выполнится последней;
- далее узлы-операции, операнды – листья дерева.

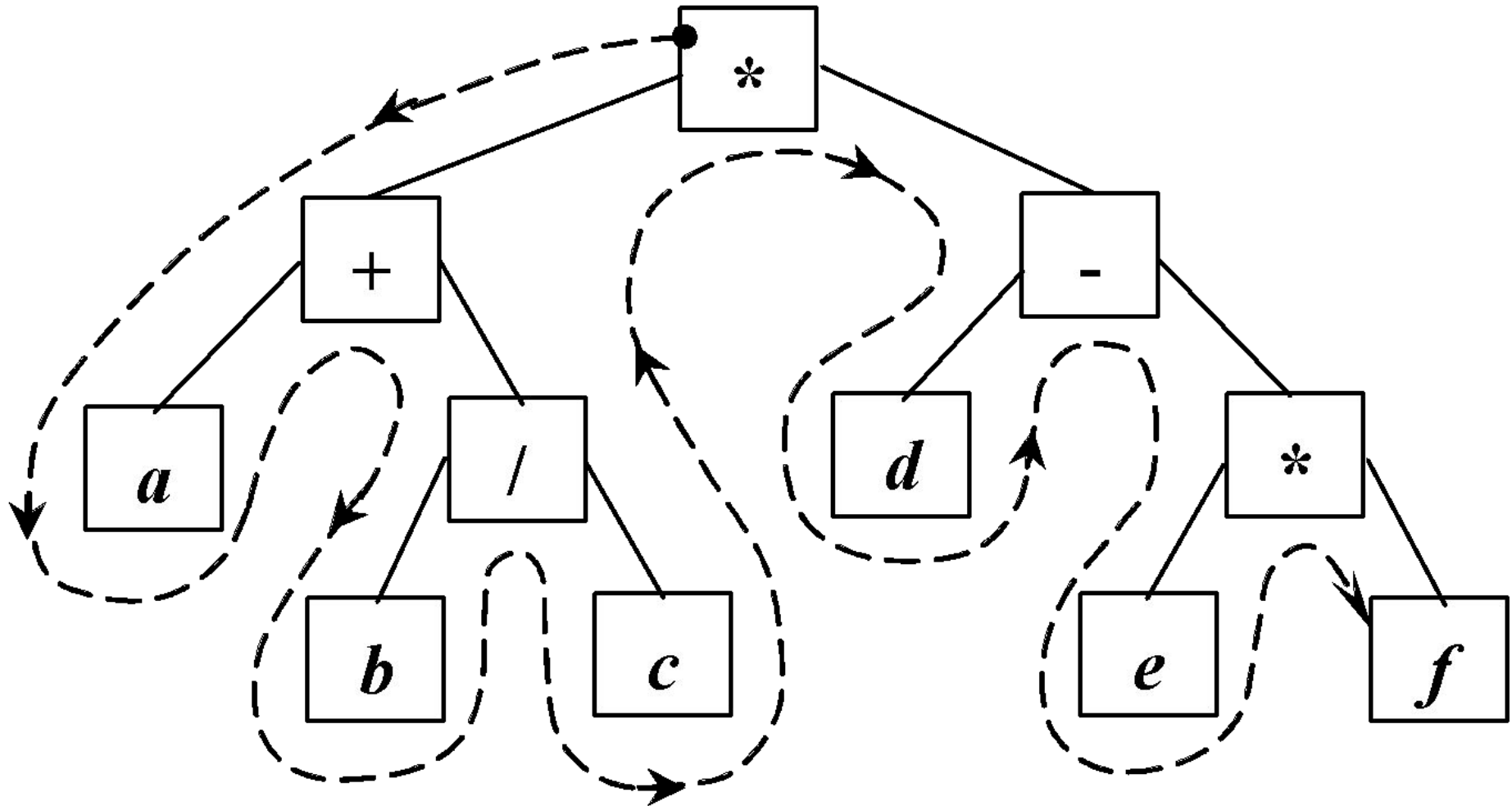
Обход 1 (Left-Root-Right) дает инфиксную запись выражения (без скобок):

$$a + b / c * d - e * f$$



Обход 2 (Root-Left-Right) дает префиксную запись выражения (без скобок):

$$* + a / b c - d * e f$$



Обход 3 (Left-Right-Root) дает постфиксную запись выражения:

$a b c / + d e f * - *$

