НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ЯДЕРНЫЙ УНИВЕРСИТЕТ «МИФИ»
Кафедра информатики и процессов управления (№17)

Дисциплина «Информатика» (основной уровень), 2-й курс, 3-й семестр.

Методические указания

Тематическое занятие 21

Древовидные структуры данных.

Содержание

Деревья как структуры данных 2
Основные определения *2*
Двоичное дерево *2*
Организация связей в двоичном дереве *3*

Двоичное дерево поиска 3
Правила построения *3*
Указатели дерева *4*
Создание дерева *4*
Добавление узла в дерево *5*
Рекурсивный алгоритм добавления узла в дерево *7*
Создание дерева *8*

Структура двоичного дерева 8
Последовательность добавления узлов *8*
Структура связей в дереве *9*

Обход узлов двоичного дерева 10
Способы обхода *10*
Прямой обход (NLR) *10*
Симметричный обход (LNR, RNL) *11*
Обратный обход (LRN) *11*

Удаление узла из дерева 12
Возможные случаи *12*
Случай 1. Удаляемый узел не имеет потомков (удаление листа) *12*
Случай 2. Удаляемый узел имеет только одного потомка *13*
Случай 3. Удаляемый узел имеет обоих потомков *15*

Балансировка двоичного дерева 19
Сбалансированное дерево *19*
Методы балансировки *19*

Деревья как структуры данных

Основные определения

Древовидные структуры – способ представления нелинейных **иерархических** отношений между элементами. ***Дерево (tree)*** – структура данных, которая эмулирует древовидную структуру в виде набора связанных элементов.

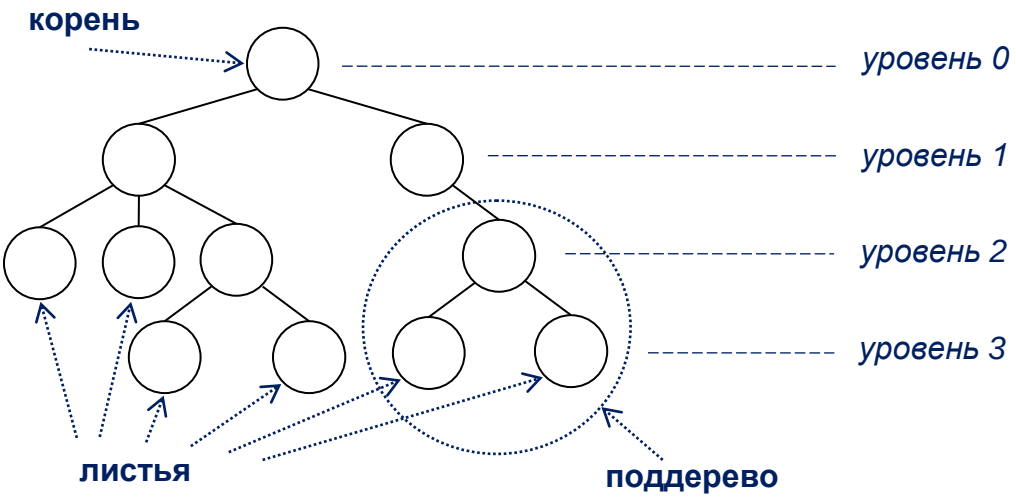
Элементы дерева называются ***узлами (nodes)***, а связывающие их ребра называются ***ветвями (edges, links)***.

Узлы могут быть связаны с вышестоящими элементами (***предками*** или ***родительскими*** узлами, ***parent nodes***) и нижестоящими элементами (***потомками***, ***наследниками*** или ***дочерними*** узлами, ***child nodes***).

Дерево с корнем (rooted tree) содержит единственный узел, не имеющий предка, который называется ***корневым*** узлом ***(root node)*** или просто ***корнем***. Все остальные узлы дерева имеют только **одного** предка.

Из корневого узла можно по ветвям достигнуть любого другого узла дерева. Количество ветвей на этом пути определяет ***уровень (level)***, на котором находится узел дерева.

Обычно на диаграммах корневой узел изображается сверху, а само дерево «растет» вниз:

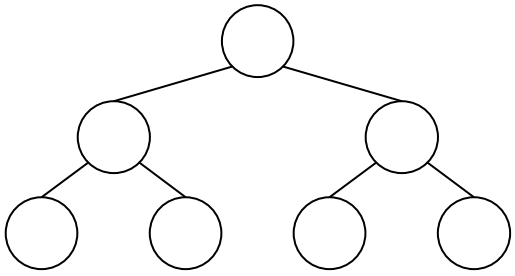


Если любой дочерний узел дерева отделить от его предка, то но станет корневым узлом ***поддерева (subtree)*** – дерева, образованного из части исходного дерева.

Узлы, не имеющие потомков, называются ***конечными (терминальными, листовыми, terminal node)*** узлами или просто ***листьями***.

Двоичное дерево

Двоичным (бинарным) деревом называется дерево с корнем, в котором каждый узел имеет **не более двух** потомков (которые обычно называют правым и левым).

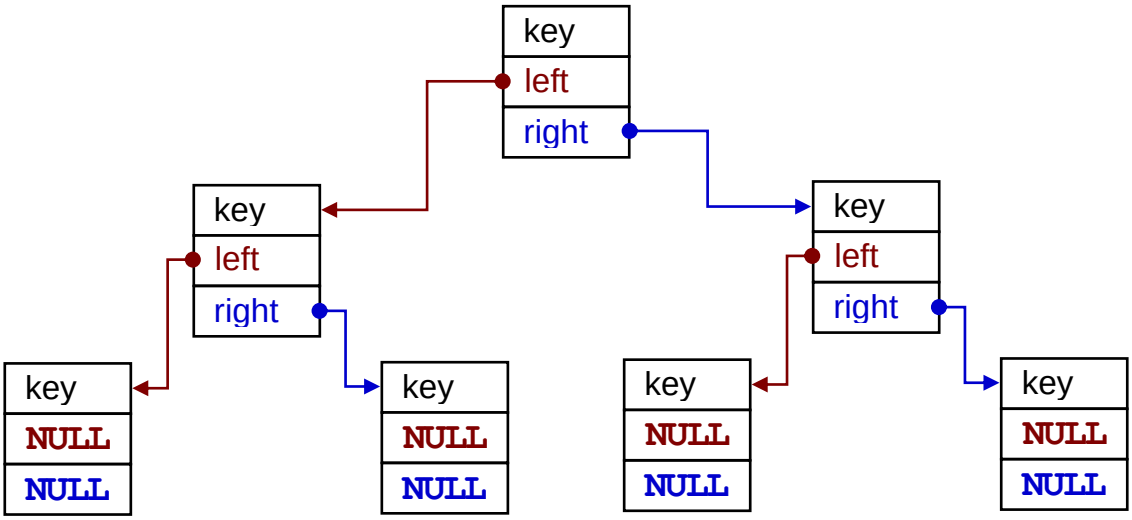


Организация связей в двоичном дереве

В простейшем случае элемент двоичного дерева должен состоять *из трёх полей:* ***ключевого*** (key) и двух ***указательных***.

Назовем указательные поля **left** (для узлов с меньшим значением поля key) и **right** (для узлов с большим значением поля key).

Схематичное изображение двоичного дерева:



Соответствующее объявление:

```
typedef struct node {
    int key;              /* key  - ключевое поле */
    struct node *left;    /* left - указатель на левого потомка */
    struct node *right;   /* right - указатель на правого потомка */
} Node;
```

Двоичное дерево поиска

Правила построения

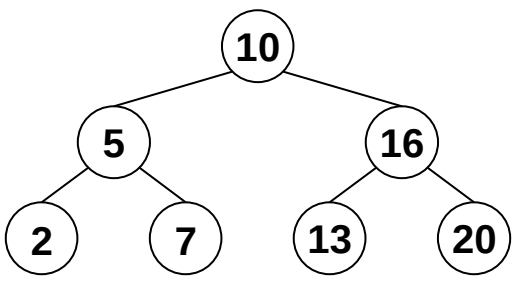
Двоичное дерево обеспечивает высокую эффективность реализации основанных на нём алгоритмов поиска и сортировки. Поэтому для этих целей часто используется ***двоичное дерево поиска (binary search tree, BST)***, у которого данные в каждом узле должны обладать ***ключами данных (keys)***, на которых определена операция сравнения **«меньше»**.

У двоичного дерева поиска для каждого узла X, имеющего потомков, выполняются следующие правила:

- у **левого** потомка значения ключа **меньше**, чем у самого узла X;
- у **правого** потомка значения ключа **не меньше** (т.е. **больше или равно**), чем у самого узла X.

Двоичное дерево поиска упрощает **исключение дубликатов**. Поэтому обычно используются деревья, не имеющие одинаковых значений ключей.

Пример:



Преимущество двоичного дерева по сравнению с линейными списками – поиск в дереве из n узлов имеет **порядок сложности в среднем** $O(\log_2 n)$.

Такое среднее значение времени поиска получается, если при построении дерева в него последовательно добавляются случайные значения. Реальное время поиска зависит от структуры дерева. Если дерево вырождается в список, то порядок сложности поиска увеличивается и становится равным $O(n)$, как в списке.

В двоичном дереве поиска узел с **минимальным** ключом – всегда крайний левый, а с **максимальным** – крайний правый.

Недостаток двоичного дерева поиска – не может быть эффективно решена задача, когда для заданного элемента требуется найти ближайшие больший и меньший элементы.

Двоичное дерево поиска не следует путать с пирамидой (двоичной кучей), которая используется для пирамидальной сортировки массива, и строится по другим правилам.

Указатели дерева

Для работы с двоичным деревом необходимо иметь один указатель на корневой узел дерева (назовем его **`root`**). Также могут потребоваться вспомогательные указатели для:

`p` – выделения и освобождения памяти узлов дерева;

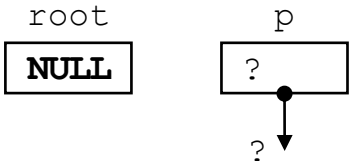
`t` – поиска узла в дереве;

`tp` – определения места добавления узла в дерево.

```
Node *root;
Node *p, *t, *tp;
```

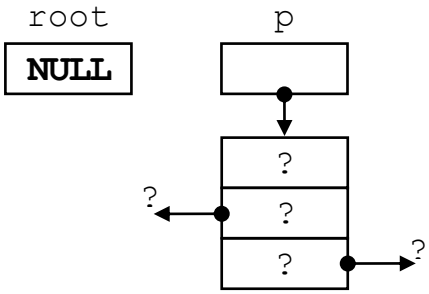
Создание дерева

1. Исходное состояние.



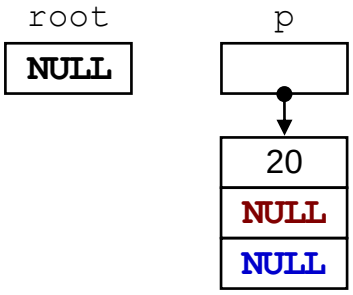

```
root = NULL;
```

2. Выделение памяти под первый узел дерева.



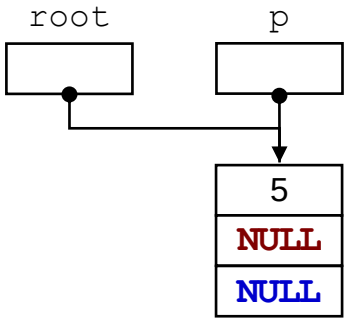

```
p = (Node *) malloc(sizeof(Node));
```

3. Занесение данных в первый узел дерева.




```
p->key = 20;
p->left  = NULL;
p->right = NULL;
```

4. Установка указателя `root` на созданный первый узел.



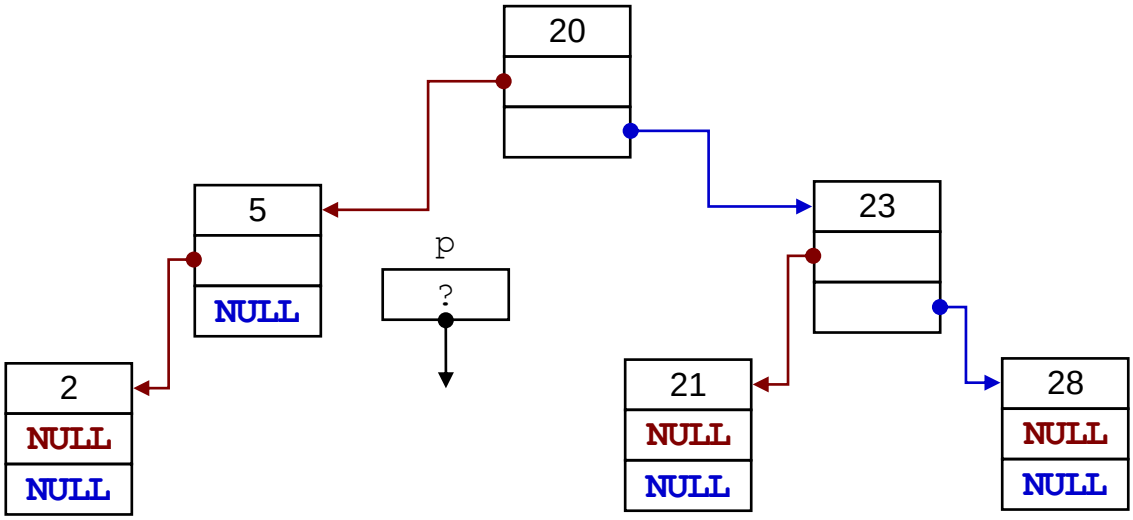

```
root = p;
```

Добавление узла в дерево

Цель – добавление нового узла в дерево в соответствии с правилами построения двоичного дерева поиска.

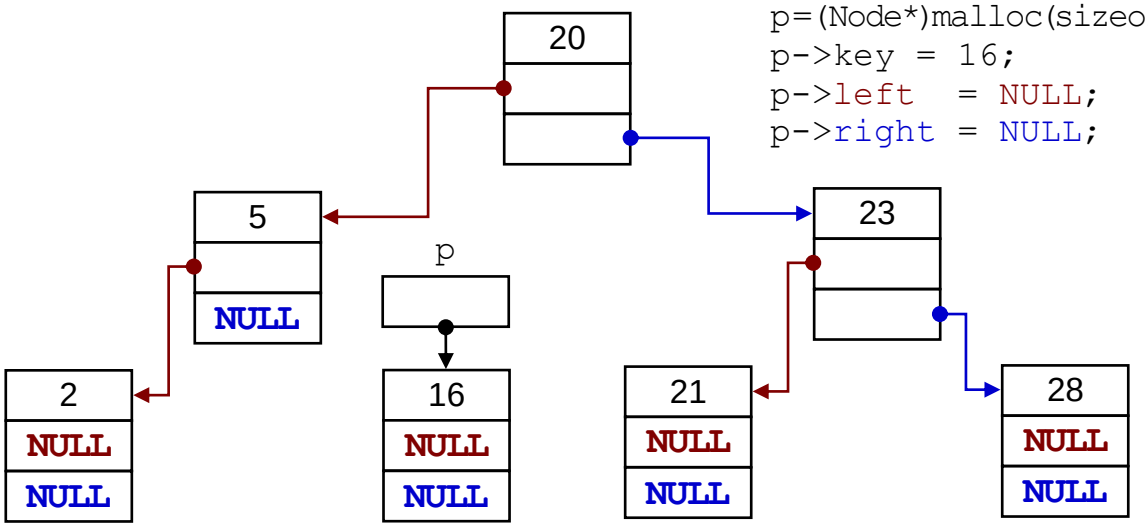
Следует заметить, что любой добавляемый узел не будет иметь потомков в дереве, т.е. он становится новым терминальным узлом (листом) дерева.

1. Исходное состояние.



2. Выделение памяти под новый узел и заполнение его полей.

```
p=(Node*)malloc(sizeof(Node));
p->key = 16;
p->left  = NULL;
p->right = NULL;
```



3. Определение места для добавления нового узла (в соответствии с правилами построения двоичного дерева поиска) и добавление нового узла (p) в найденное место.

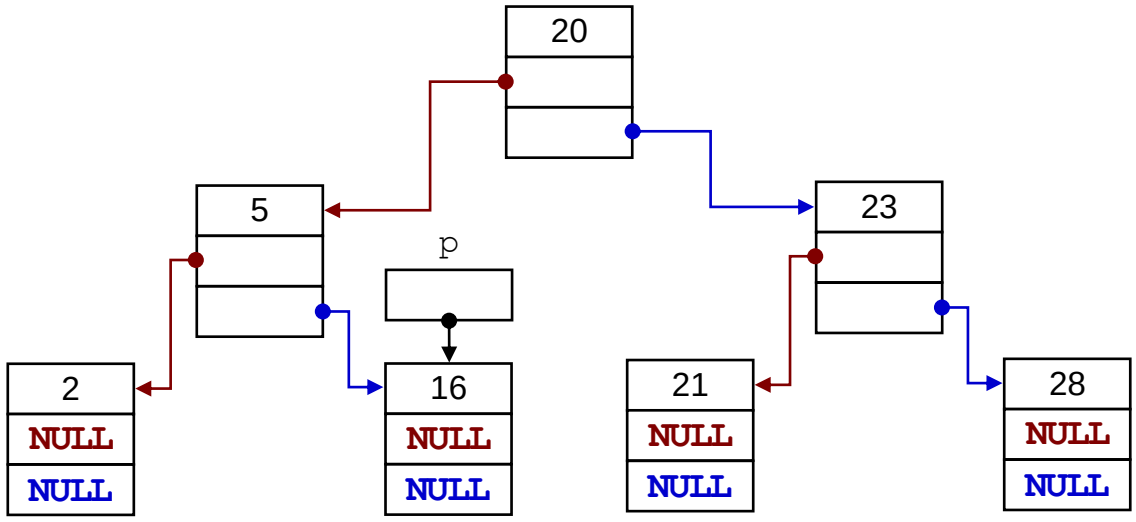
Для поиска места и добавления нового узла в дерево можно использовать следующий циклический алгоритм:

```
t = root; /*Поиск места добавления начинается с корня.*/
while (t != NULL) { /*Пока у текущего узла имеется нужный потомок.*/
     tp = t; /*Фиксирование текущего узла в указателе tp.*/
     if (p->key < t->key) { /*Сравнение ключей нового и текущего узлов.*/
          t = t->left; /*Переход по левой ветви дерева.*/
          if (t==NULL) /*Левый потомок отсутствует, найдено место добавления.*/
               tp->left = p; /*Добавление нового узла p.*/
     } else {
          t = t->right; /*Переход по правой ветви дерева.*/
          if (t==NULL) /*Правый потомок отсутствует, найдено место добавления.*/
               tp->right = p; /*Добавление нового узла p.*/
     }
}
```

Для рассматриваемого примера работу данного циклического алгоритма иллюстрирует таблица трассировки цикла:

№ итерации цикла `while`	Текущий узел `t` в начале итерации	Направление перехода по дереву	Текущий узел `t` в конце итерации	Место добавления нового узла `p`
1	`t->key` **=20**	`left`	`t->key` **=5**	*не найдено*
2	`t->key` **=5**	`right`	`NULL`	`tp->key` **=5** `tp->right`

4. Конечное состояние дерева.



Рекурсивный алгоритм добавления узла в дерево

Простую и наглядную реализацию поиска места и добавления нового узла в дерево дает рекурсивный алгоритм, который реализуется в форме рекурсивной функции.

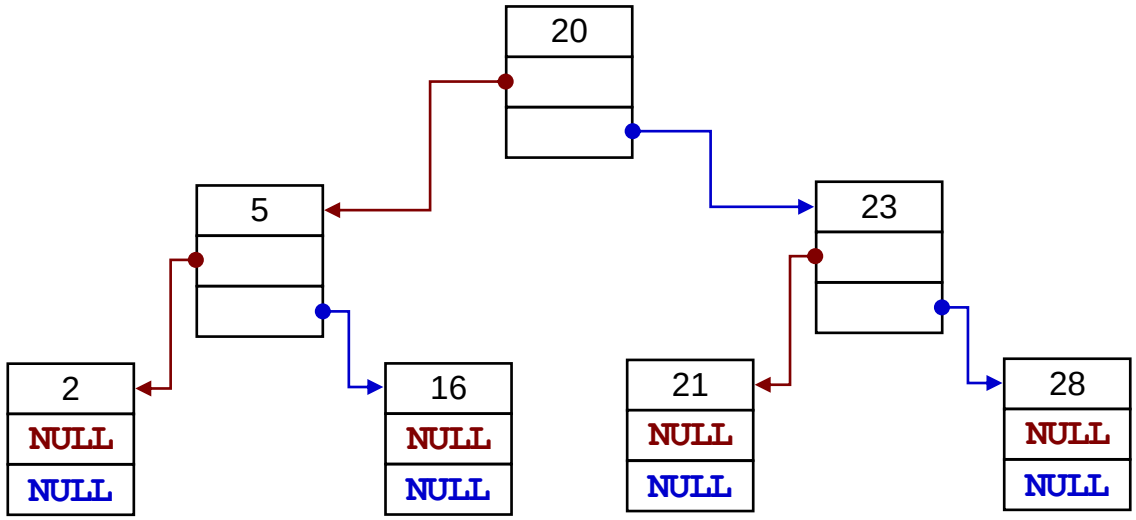
Такой подход позволяет выделять память под новый узел и заполнять его поля **после** нахождения места добавления узла.

```
typedef Node *pNode; /*Новый тип pNode - указатель на узел*/
...                  /*для удобства передачи в функцию.   */
void insert(pNode *p, int val) {
   if (*p == NULL) { /*Если дерево пусто.*/
      *p = (Node *) malloc(sizeof(Node)); /*Выделение памяти.*/
      (*p)->key = val;      /*                           */
      (*p)->left  = NULL;   /*Заполнение полей нового узла.*/
      (*p)->right = NULL;   /*                           */
   } else { /*Если дерево не пусто.*/
      if (val < (*p)->key) /*Сравнение ключей нового и текущего узлов.*/
         insert(&( (*p)->left ), val); /*Переход по левой ветви.*/
      else
         insert(&( (*p)->right ), val); /*Переход по правой ветви.*/
   }
}
```

Вызов этой рекурсивной функции для исходного дерева из рассматриваемого примера:

```
pNode root; /*Описание корня дерева с помощью нового тип pNode.*/
...
root = NULL; /*Создание пустого дерева.*/
insert(&root, 16);
```

Конечное состояние дерева станет таким же, как и в предыдущем случае.



Создание дерева

С помощью рекурсивного алгоритма добавления узлов можно создать дерево целиком.

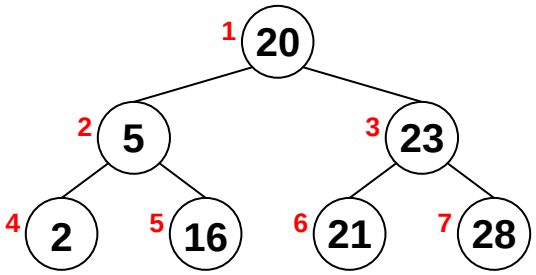
Например, для создания дерева из 7-и узлов, можно вызвать рекурсивную функцию в цикле 7 раз:

```
int n=7, a;
...
root = NULL;
for (i=0; i<n; ++i) {
   scanf("%d", &a);     /* Ввод значения узла. */
   insert(&root, a);    /* Добавление узла в дерево. */
}
```

Для создания дерева из рассматриваемого примера достаточно ввести с клавиатуры последовательность целых чисел, например:

20 5 23 2 16 21 28

В результате будет создано дерево из 7-и узлов:

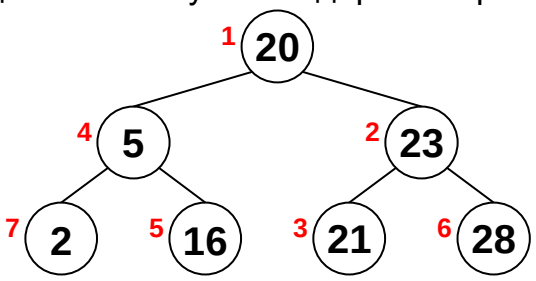


На схеме красным цветом выделены порядковые номера, определяющие последовательность добавления узлов в дерево.

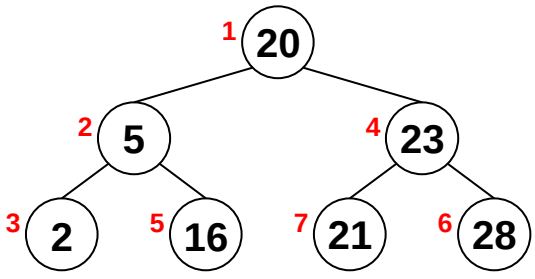
Структура двоичного дерева

Последовательность добавления узлов

То же самое дерево можно получить, если изменить последовательность добавления узлов в дерево. Примеры:



1	2	3	4	5	6	7
20	23	21	5	16	28	2



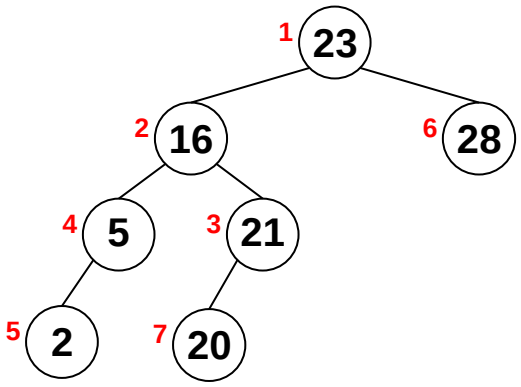
1	2	3	4	5	6	7
20	5	2	23	16	28	21

Все эти деревья имеют одинаковую структуру связей между узлами – у них все три уровня полностью заполнены узлами.

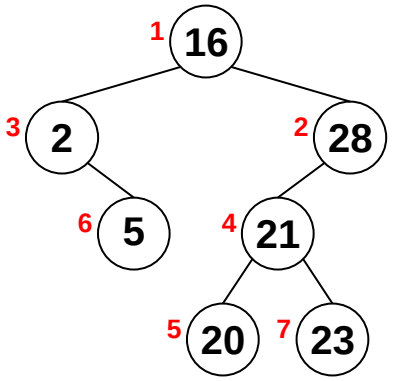
Но в общем случае структура связей создаваемого дерева **существенно зависит** от последовательности добавления узлов.

Структура связей в дереве

Различные последовательности добавления узлов могут приводить к построению деревьев, сильно отличающихся по своей структуре связей между узлами. Примеры деревьев из 7-и узлов с 4-мя уровнями:



1	2	3	4	5	6	7
23	16	21	5	2	28	20



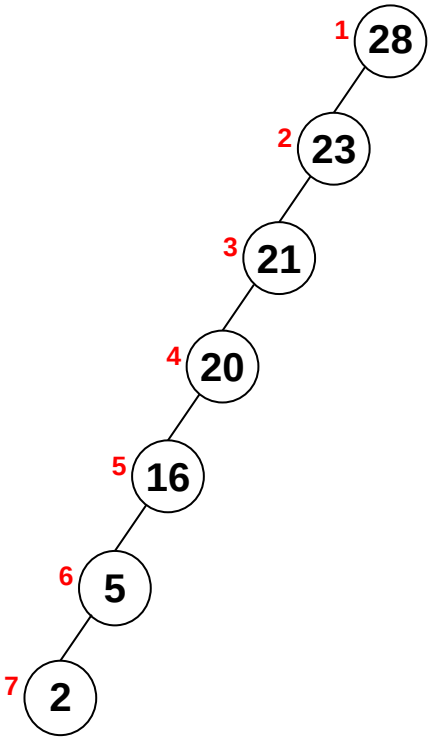
1	2	3	4	5	6	7
16	28	2	21	20	5	23

Для двоичного дерева, состоящего из 7-и узлов, количество уровней может варьироваться от 3-х до 6-и.

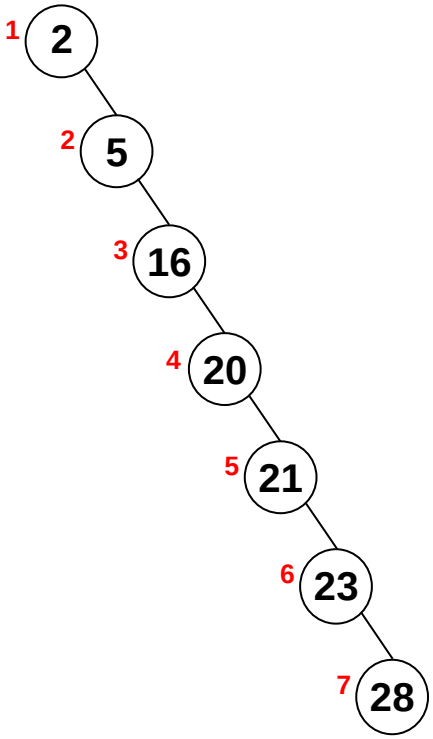
Деревья, содержащие одинаковое количество узлов, имеются два крайних случая построения структуры связей:

- все уровни, кроме возможно последнего, максимально заполнены узлами;
- на каждом уровне имеется только по одному узлу.

В последнем случае дерево вырождается в линейный однонаправленный список. Такое дерево может быть получено, если добавляемые узлы отсортированы по убыванию или по возрастанию:



1	2	3	4	5	6	7
28	23	21	20	16	5	2



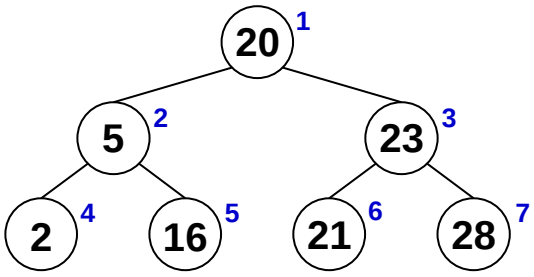
1	2	3	4	5	6	7
2	5	16	20	21	23	28

Обход узлов двоичного дерева

Способы обхода

При ***обходе дерева*** (или ***проходе по дереву***, ***traversing***) каждый узел посещается только один раз и возникает линейная последовательность всех узлов. Это позволяет использовать понятие *следующий узел* как узел, стоящий после данного при выбранном порядке обхода.

При обходе дерева ***в ширину (breadth-first)*** движение по узлам происходит в порядке уровней, посещая каждый узел на уровне, прежде чем перейти на следующий уровень. Каждый уровень обходится слева направо.
Например:



1	2	3	4	5	6	7
20	5	23	2	16	21	28

На схеме синим цветом выделены порядковые номера, определяющие последовательность обработки узлов дерева.

При обходе двоичного дерева ***в глубину*** сначала движутся вниз насколько это возможно для каждого потомка, а затем переходят к следующей ветви. При этом операции обработки вершины выполняются **рекурсивно** в каждом узле, начиная с корня. Существует несколько способов обхода дерева в глубину:

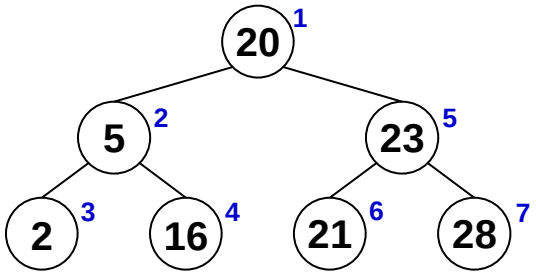
- ***прямой*** *(предварительная обработка, pre-order),*
- ***симметричный*** *(центрированный, порядковая обработка, in-order),*
- ***обратный*** *(отложенная обработка, post-order);*

Прямой обход (NLR)

Принцип: узел обрабатывается до того, как произошел переход к его левому и правому потомкам (движение сверху-вниз). Для корня дерева рекурсивно вызывается следующая процедура:

1) обработать узел (N),
2) обойти левое поддерево (L),
3) обойти правое поддерево (R).

Пример:



1	2	3	4	5	6	7
20	5	2	16	23	21	28

Рекурсивный обход реализуется в виде рекурсивной функции:

```
void preOrder(pNode p) {
   if (p != NULL) {
      printf("%d ", p->key); /* Обработать узел (N). */
      preOrder(p->left);     /* Обойти левое поддерево (L). */
      preOrder(p->right);    /* Обойти правое поддерево (R). */
   }
}
```

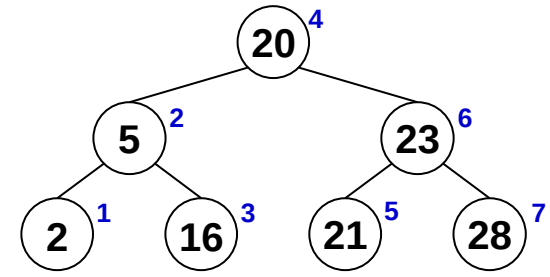
Вызов функции: `preOrder(root);`

Симметричный обход (LNR, RNL)

Принцип: сначала перейти к левому потомку, затем обработать сам узел, затем перейти к правому потомку. Рекурсивная процедура для корня:

1) обойти левое поддерево (L),
2) обработать узел (N),
3) обойти правое поддерево (R).

Пример:



1	2	3	4	5	6	7
2	5	16	20	21	23	28

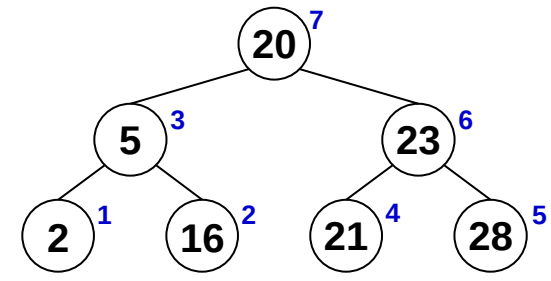
В двоичном дереве поиска симметричный обход извлекает данные в **отсортированном** порядке. При порядке обхода LNR последовательность узлов будет упорядочена по возрастанию, при RNL – по убыванию.

Обратный обход (LRN)

Принцип: сначала происходит переход к левому и правому потомкам узла, а затем обрабатывается сам узел (движение снизу-вверх). Рекурсивная процедура для корня:

1) обойти левое поддерево (L),
2) обойти правое поддерево (R),
3) обработать узел (N).

Пример:



1	2	3	4	5	6	7
2	16	5	21	28	23	20

Удаление узла из дерева

Возможные случаи

Цель – удаление заданного узла из дерева, так, чтобы после этого не нарушились правила построения двоичного дерева поиска.

Алгоритм удаления зависит от места удаляемого узла в структуре дерева. При этом возможны три случая, когда удаляемый узел:

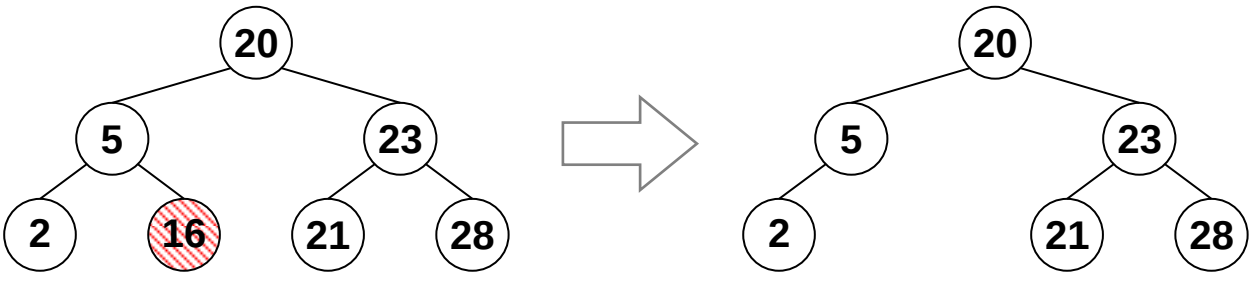
1) не имеет потомков (лист);
2) имеет только одного потомка (левого или правого);
3) имеет обоих потомков (и левого, и правого).

Для каждого из этих случаев вначале выполняется поиск удаляемого узла и его предка. После окончания поиска вспомогательные указатели содержат адреса:

`p` – удаляемого узла,
`par` – предка удаляемого узла.

Случай 1. Удаляемый узел не имеет потомков (удаление листа)

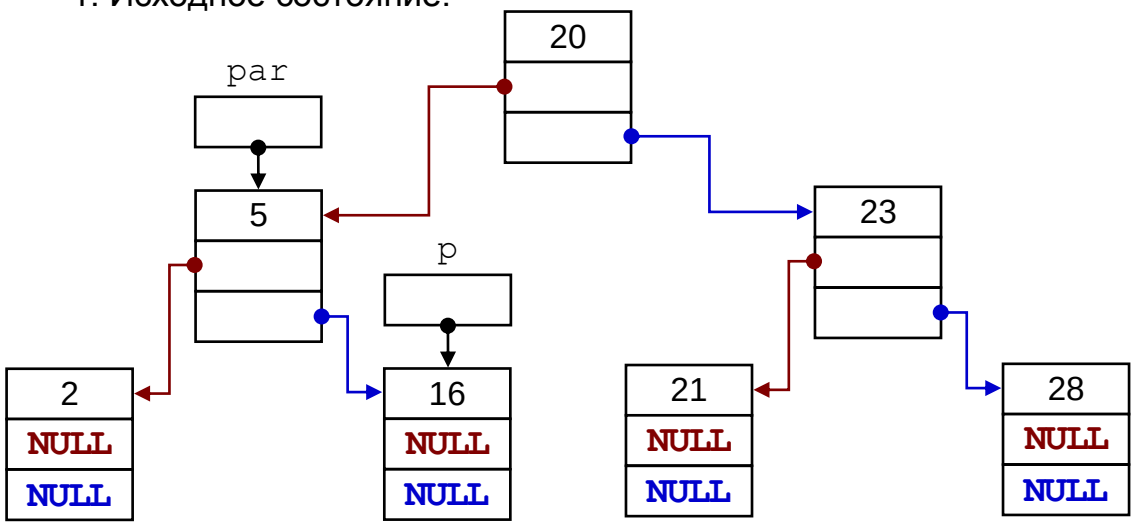
В этом случае при удалении узла, чтобы не нарушить правила построения дерева, достаточно обнулить соответствующее указательное поле предка удаляемого узла.



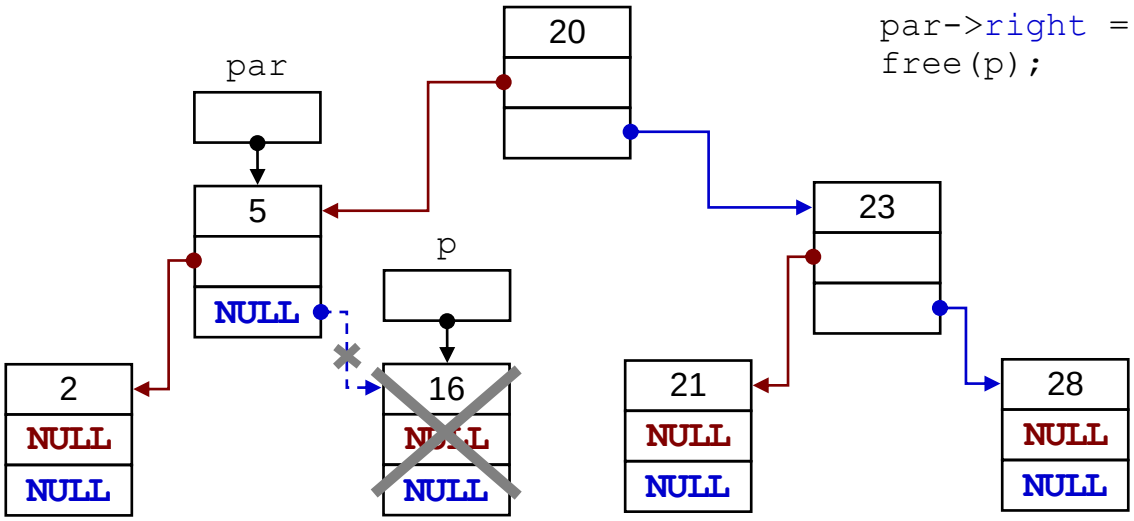
Исходное дерево
(удаляемый узел выделен штриховкой.)

Конечное дерево

1. Исходное состояние.

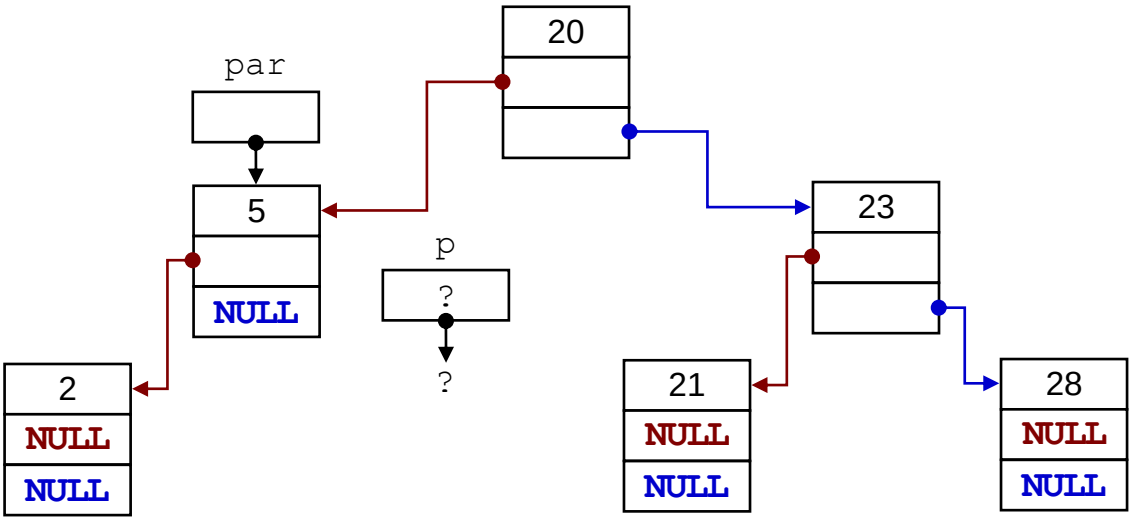


2. Удаление узла.



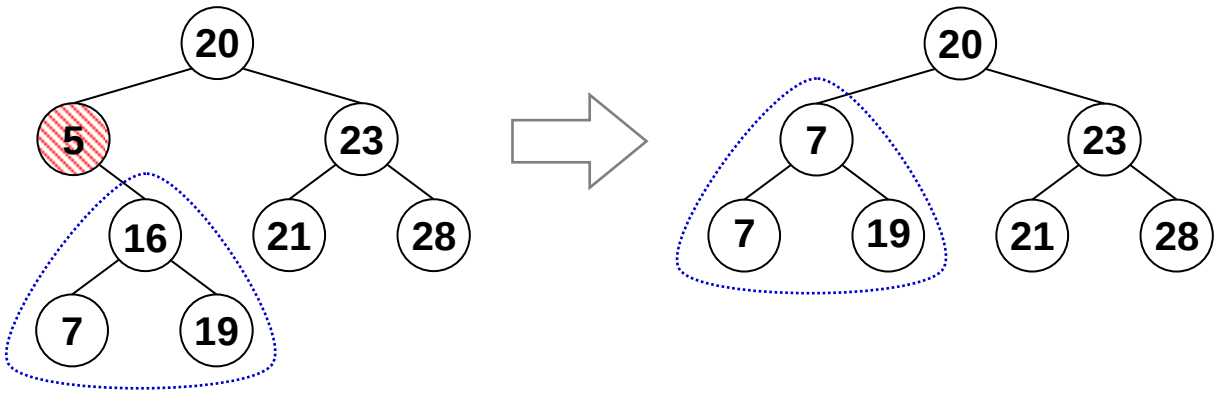

```
par->right = NULL;
free(p);
```

3. Конечное состояние.



Случай 2. Удаляемый узел имеет только одного потомка

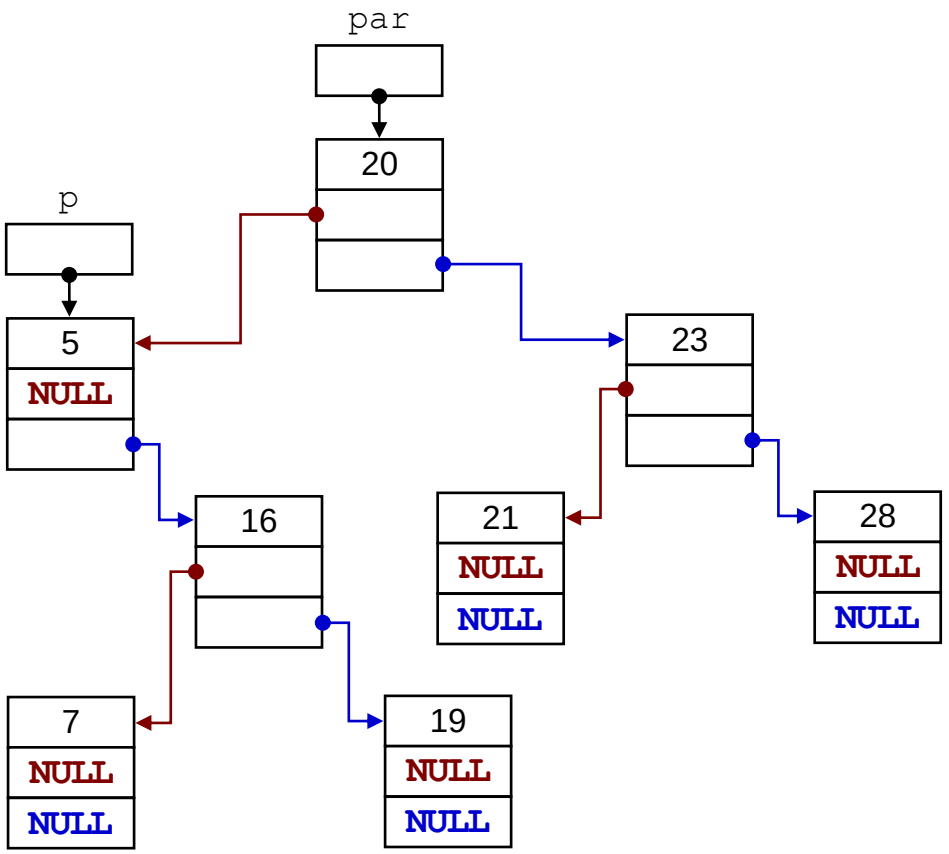
В этом случае в указательное поле предка удаляемого узла помещается адрес его единственного потомка. Эта операция выполняется независимо от того каким, левым или правым, является единственный потомок удаляемого узла.



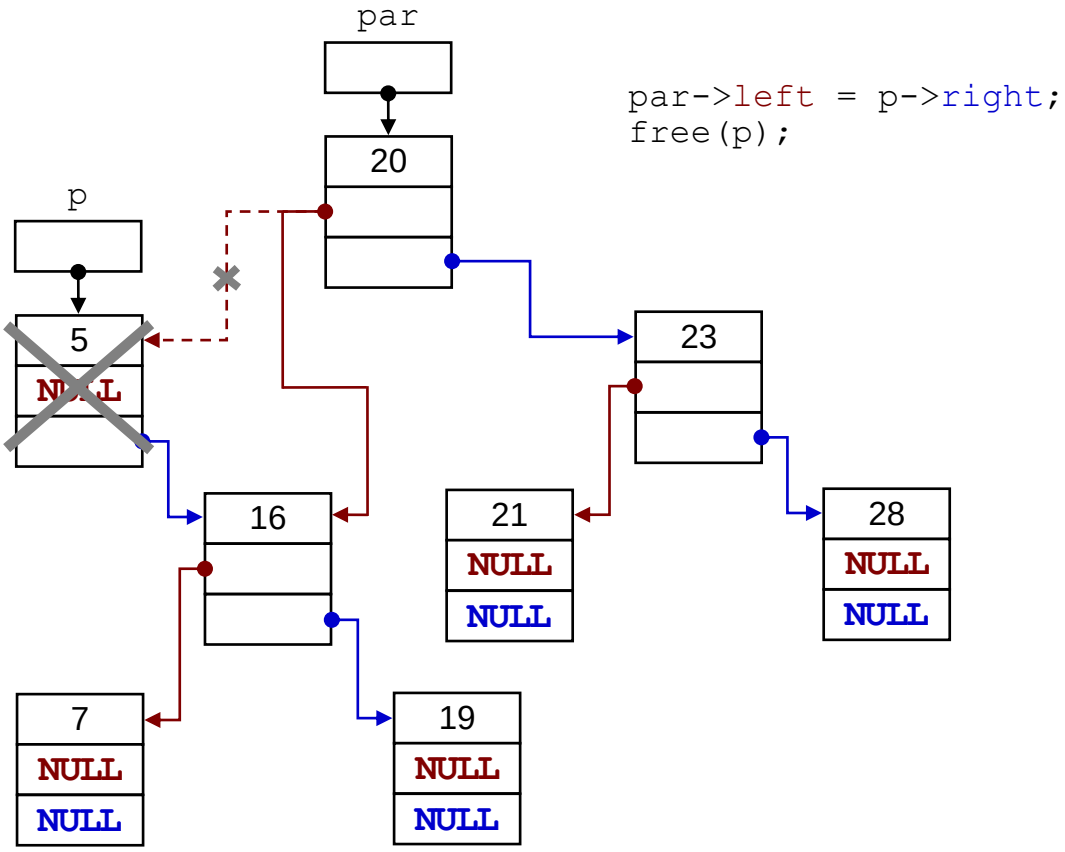
Исходное дерево
(удаляемый узел выделен штриховкой)

Конечное дерево

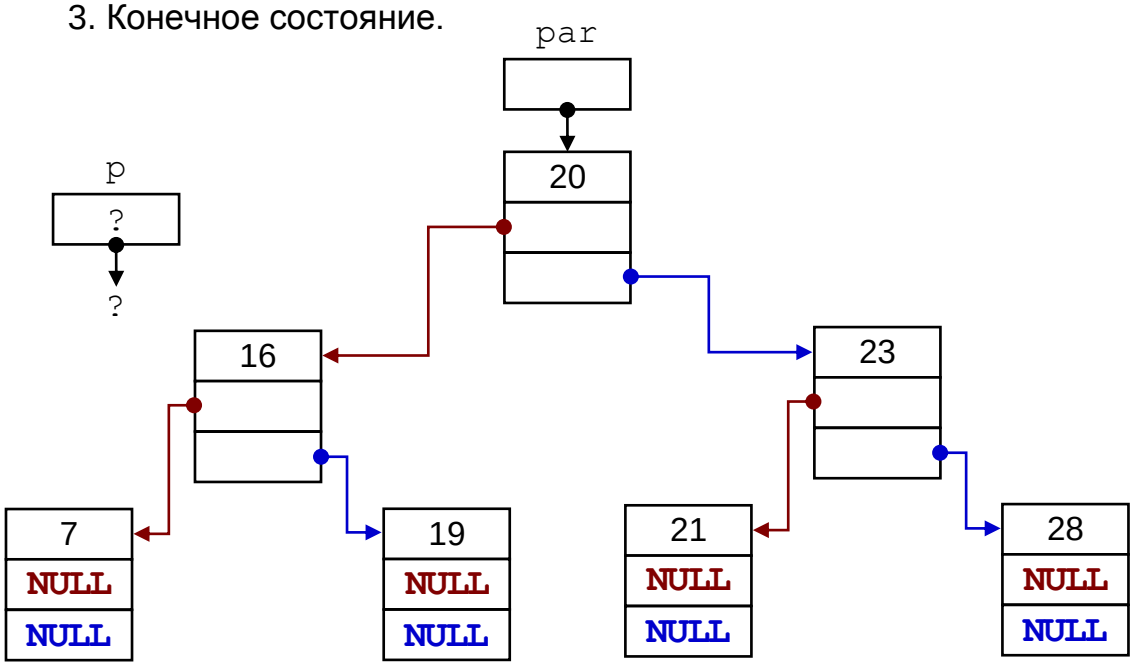
1. Исходное состояние.



2. Удаление узла.



3. Конечное состояние.



Случай 3. Удаляемый узел имеет обоих потомков

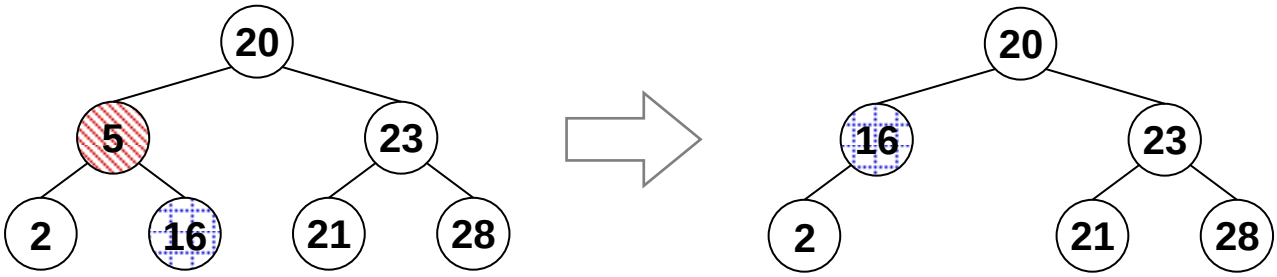
В этом случае простое «вырезание» удаляемого узла нарушит связи в дереве, поскольку одного предка невозможно связать с двумя потомками. Поэтому прибегают к следующему приему, который заключается не в удалении узла из дерева, а в замене его содержания, на содержание другого узла, следующего за удаляемым узлом по значению ключевого поля.

Этот узел называется ***последователем***, значение его ключа больше, чем значение ключа удаляемого узла, но меньше, чем значение ключей всех остальных узлов дерева. Поэтому последователь всегда располагается **в правом поддереве** удаляемого узла и является **крайним левым** узлом этого поддерева.

После замены содержания удаляемый узел перестает существовать логически (память не освобождается, изменяется только ключевое поле), а последователь удаляется из дерева «физически» (освобождая память).

Поскольку у последователя всегда отсутствует левый потомок, то удаление последователя происходит в соответствии с предыдущими ***Случаем 1*** или ***Случаем 2***.

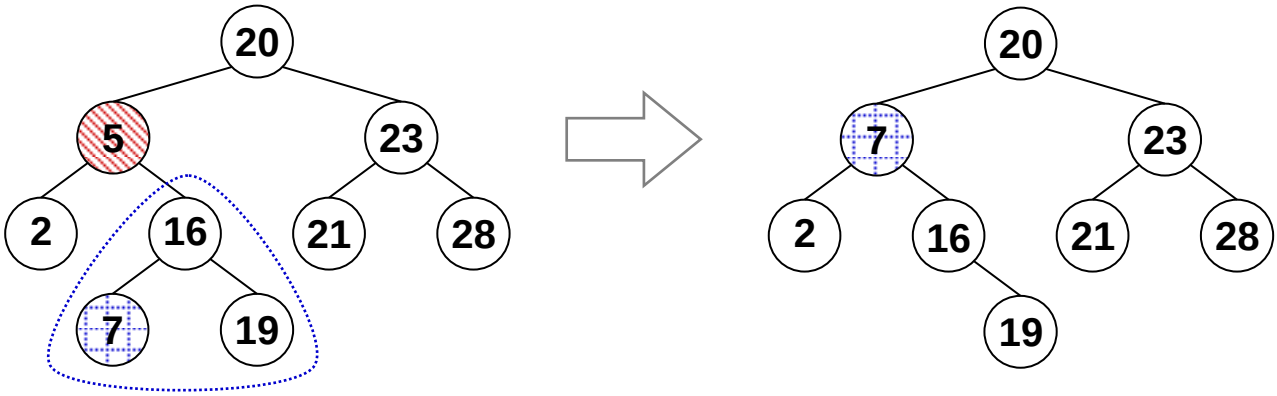
Пример А. Последователь – правый лист удаляемого узла. Удаление последователя происходит по схеме, описанной в ***Случае 1***.



Исходное дерево
(удаляемый узел и последователь
выделены штриховкой разного типа)

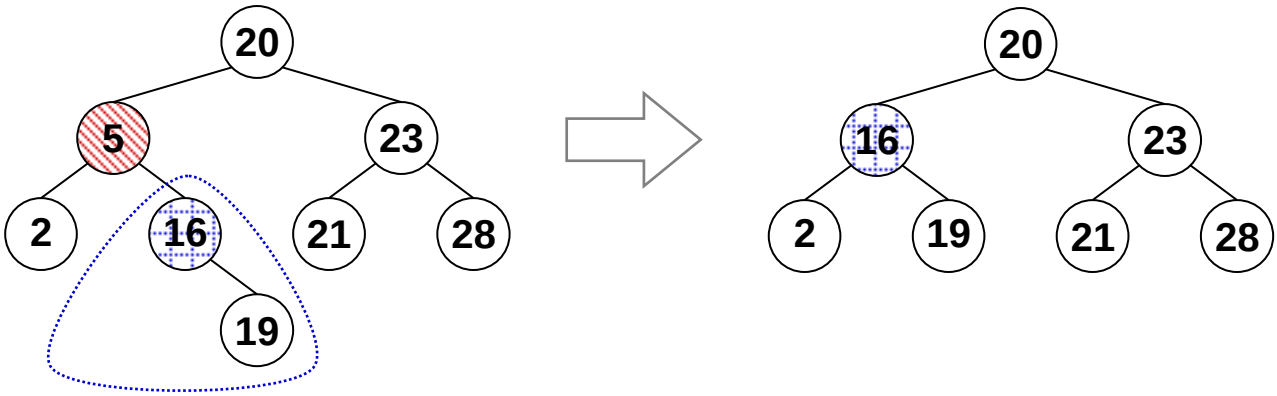
Конечное дерево

Пример Б. Последователь – лист правого поддерева удаляемого узла. Удаление последователя происходит по схеме, описанной в ***Случае 1***.



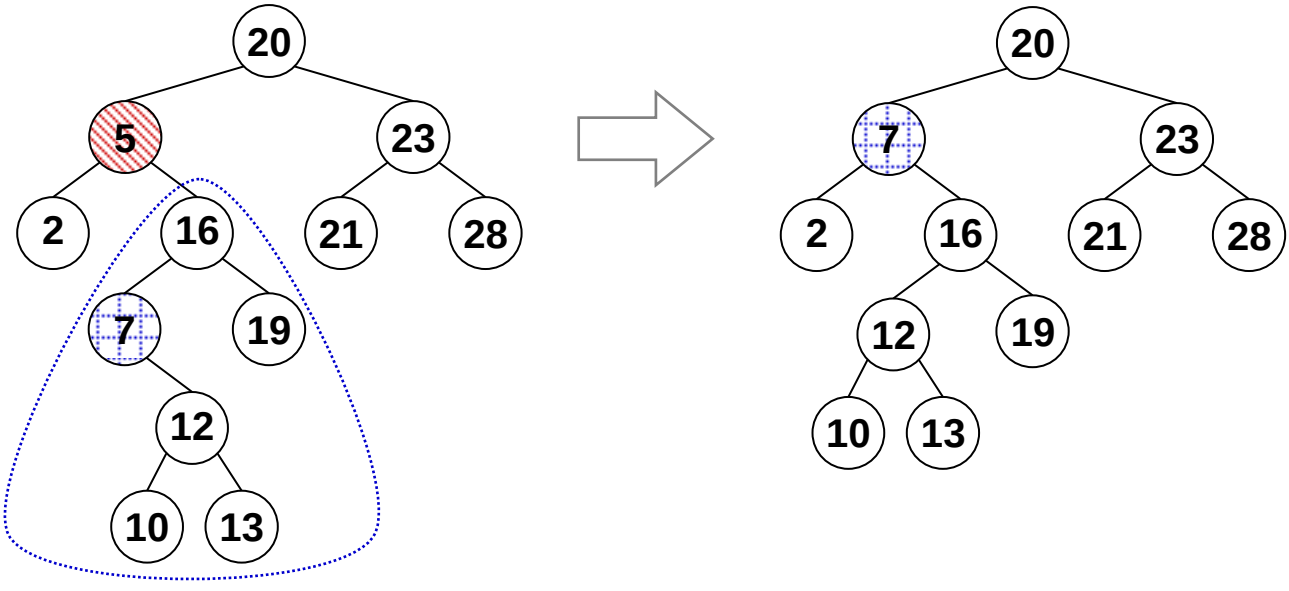
Исходное дерево Конечное дерево

Пример В. Последователь – корень правого поддерева удаляемого узла. Удаление последователя происходит по схеме, описанной в ***Случае 2***.



Исходное дерево Конечное дерево

Пример Г. Последователь – имеющий одного потомка узел правого поддерева удаляемого узла. Удаление последователя происходит по схеме, описанной в ***Случае 2***.

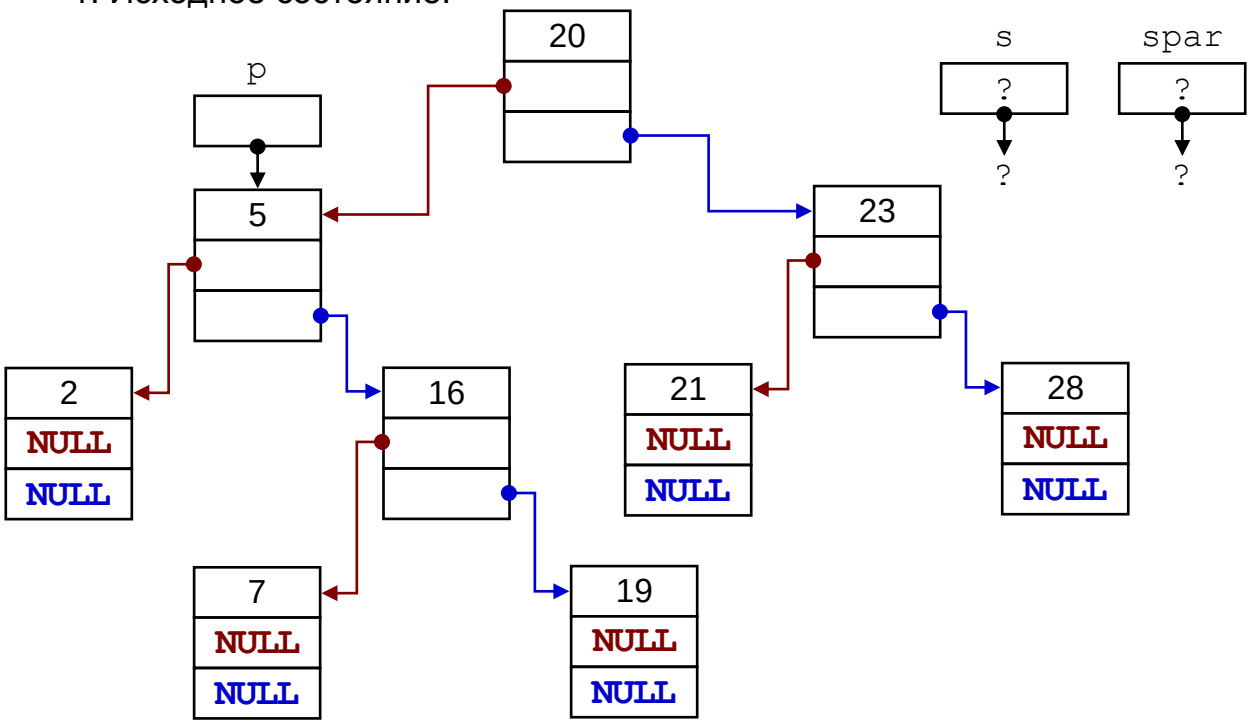


Исходное дерево Конечное дерево

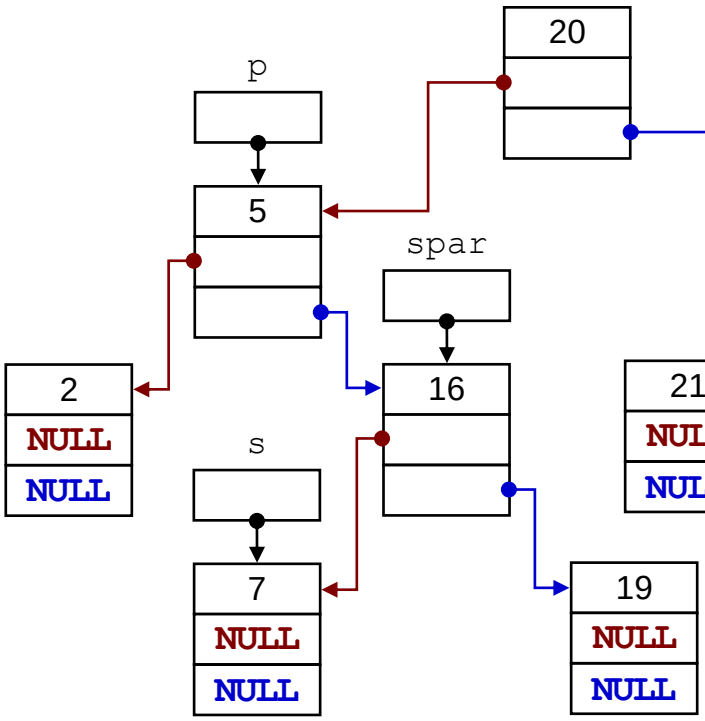
Рассмотрим удаление узла дерева из Примера Б. Обозначим вспомогательные указатели, которые содержат адреса:

p – удаляемого узла,
s – последователя,
spar – предка последователя.

1. Исходное состояние.

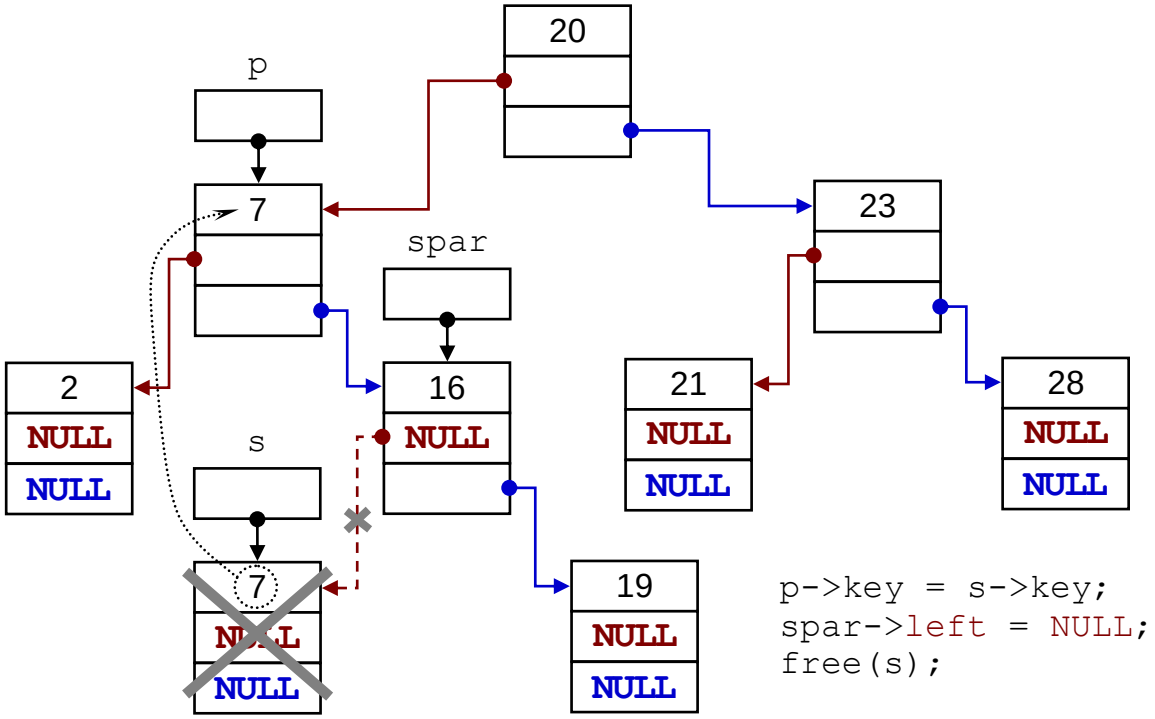


2. Поиск последователя (и его предка).

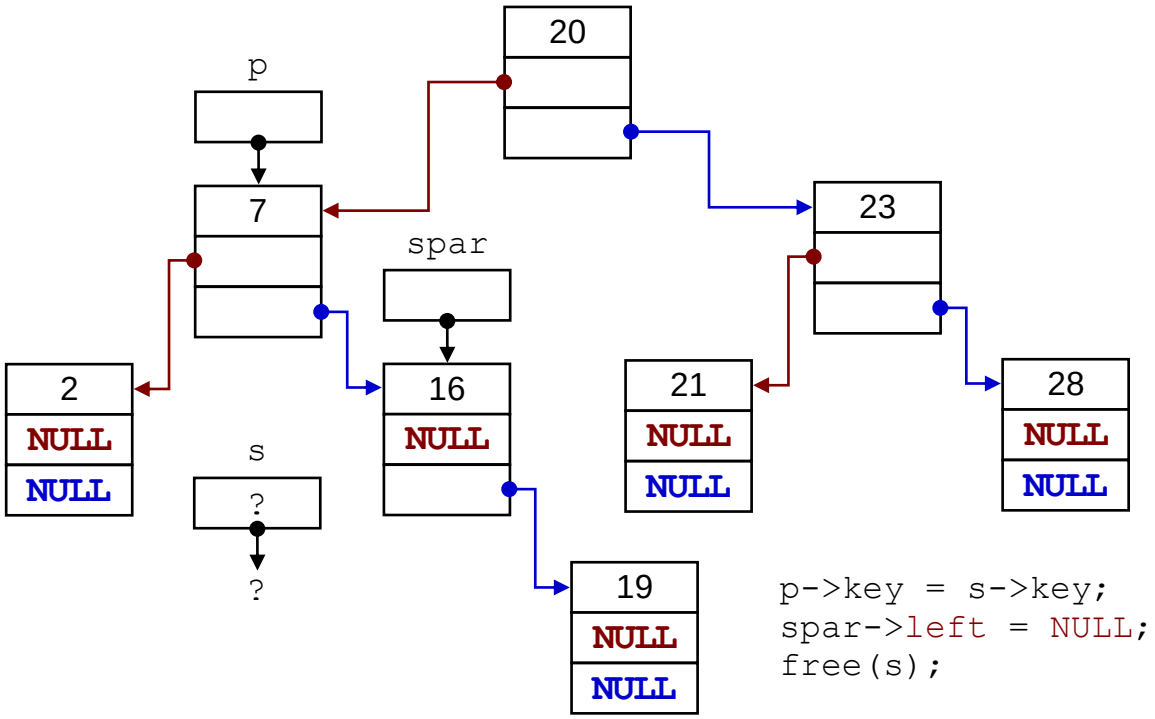



```
spar = p;
s = p->right;
while (s->left != NULL) {
    spar = s;
    s = s->left;
}
```

3. Замена ключа удаляемого узла `p` на ключ последователя `s`, и удаление последователя.



4. Конечное состояние.



Балансировка двоичного дерева

Сбалансированное дерево

Как уже отмечалось поиск в двоичном дереве имеет порядок сложности в среднем $O(\log_2 n)$. Но реальное время поиска зависит от структуры дерева.

Узел будет найден тем быстрее, чем ближе к корню он расположен. Поиск проходит быстрее в том дереве, в котором меньше сумма путей от корня до каждой из вершин.

Дерево называется ***идеально сбалансированным***, если все его уровни (кроме, возможно, последнего) полностью заполнены. В сбалансированном двоичном дереве полностью заполненный уровень n содержит $2 \cdot n$ узлов, а путь в каждую вершину не превышает $\lceil \log_2 n \rceil$. В таком дереве поиск выполняется за минимальное время.

Добавление и удаление узлов может приводить к изменению структуры дерева, которое влияет на его сбалансированность и скорость поиска. При работе с деревьями больших размеров крайне желательно, чтобы они были близки к сбалансированным.

Если дерево ***близко к сбалансированному***, то даже в худшем случае за время порядка $O(\log_2 n)$ в нем можно провести поиск заданного узла, добавить и удалить вершину.

Методы балансировки

Приведение уже существующего дерева к сбалансированному – сложный процесс. Проще балансировать дерево в процессе добавления или удаления каждого из его узлов.

Но даже после добавления/удаления одного узла приведение дерева к идеально сбалансированному почти всегда затрагивает все остальные его узлы, а значит требует значительного времени, превышающего $O(\log_2 n)$.

Поэтому на практике чаще используются другие виды сбалансированных деревьев, методы ***балансировки*** которых требуют после добавления/удаления узла лишь локальные изменения вдоль пути от корня к данному узлу. На такие операции затрачивается время, не превышающее $O(\log_2 n)$.

Широко известные виды сбалансированных двоичных деревьев:

- ***АВЛ-дерево*** (***AVL tree***, названное по первым буквам фамилий Адельсона-Вельского Г.М. и Ландиса Е.А., 1968): для каждой вершины высота её двух поддеревьев различается не более чем на единицу, при этом для узлов дерева вводится дополнительный атрибут – разность высот правого и левого поддеревьев (принимает три возможных значения: –1, 0, +1);
- ***красно-чёрное дерево*** (***red-black tree***, Байер Р., 1972): вводится дополнительный атрибута узлов дерева – цвет (принимает два возможных значения – чёрный или красный);
- ***расширяющееся*** или ***косое*** дерево (***splay tree***, Тарьян Р. и Слейтор Д., 1983): при каждом обращении к дереву выполняются расширяющие splay-операции, без использования дополнительных атрибутов узлов.