Б-деревья

Лекция 19

Б-деревья

Б-деревья – сбалансированные деревья, обеспечивающие эффективное хранение информации на дисках и других устройствах с прямым доступом.

Каждая вершина x *в Б-дереве* хранит $n(x)$- элементов (ключей)

и имеет $n(x)$+1 детей.

Ключи служат границами, разделяющими всех ее потомков на

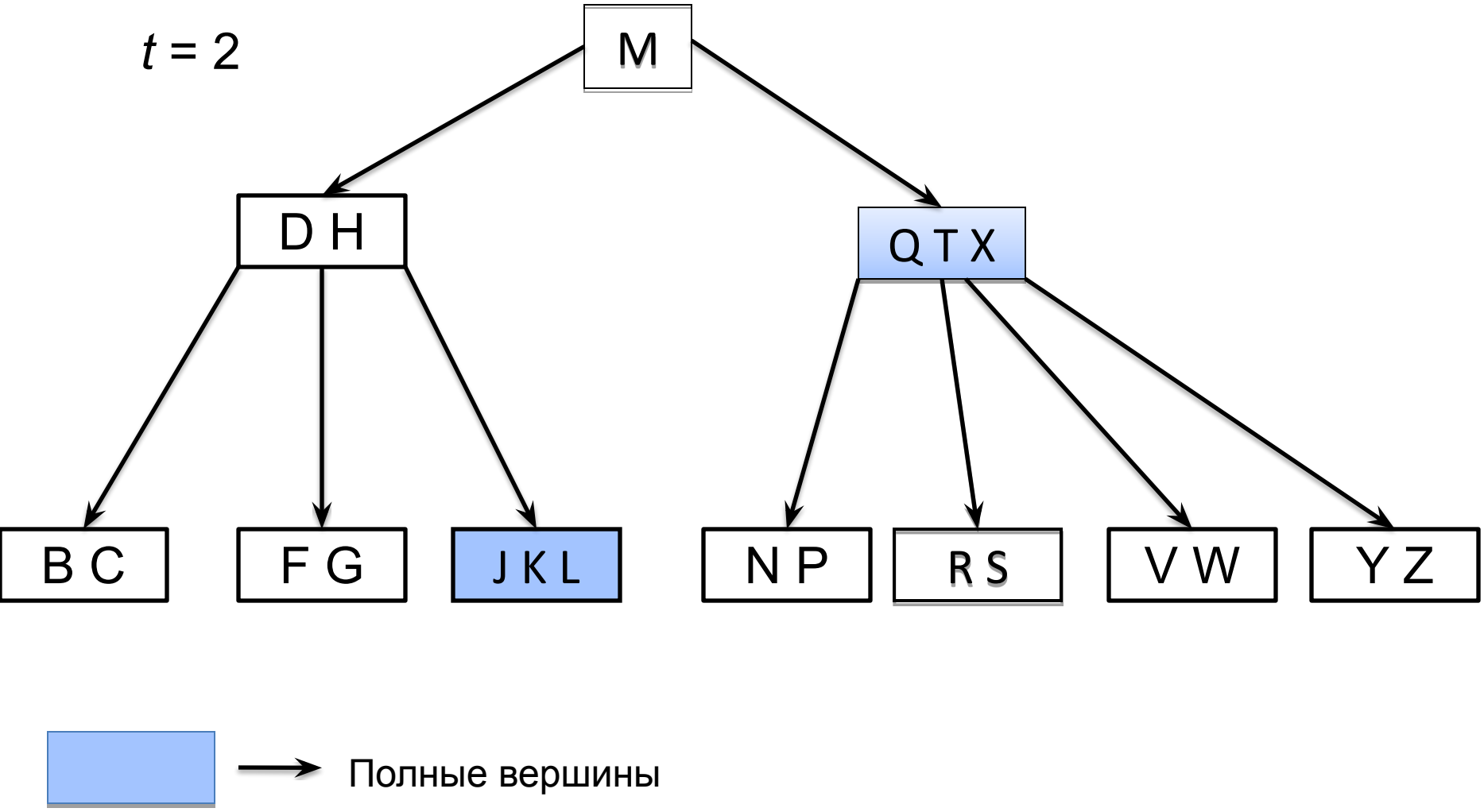
$n(x)$+1 групп.

При поиске в Б-дереве мы сравниваем искомый ключ с

$n(x)$- ключами из x и по результатам сравнения выбираем один из

$n(x)$+1-путей.

Пример Б-дерева



Определение Б-дерева

Для простоты будем считать, что вся дополнительная информация, связанная с ключами храниться в той же вершине дерева (на практике это не всегда так).

Б-дерево – корневое дерево, устроенное следующим образом:

1. Каждая вершина x содержит поля, в которых хранятся:

   а) n - количество ключей, хранящихся в данной вершине;

   б) сами ключи $k_0 \le k_1 \le \ldots \le k_{n-1}$ в неубывающем порядке;

   в) булевское значегие *leaf*[x], истинное, если вершина x - лист;

2. Если x – внутренняя вершина, то она также содержит $n(x)+1$-указателей: $C_0, C_1, \ldots, C_{n(x)}$ на ее детей;

Определение Б-дерева(продолжение)

3. Ключи $key_i[x]$ служат границами, разделяющими значения ключей в поддеревьях:

   $k_0 \le key_0[x] \le k_1 \le key_2[x] \le \ldots \le key_{n[x]-1}[x] \le K_{n[x]}$, где k_i - множество ключей, хранящихся в поддереве с корнем $C_i[x]$;
4. Все листья находятся на одной и той же глубине, равной высоте дерева;
5. Число ключей, хранящихся в одной вершине, ограничено сверху и снизу единым для Б-дерева числом $t \ge 2$, которое называется - *минимальной степенью Б-дерева*. А именно:

Определение Б-дерева (продолжение)

а) каждая вершина, кроме корня содержит по меньшей мере t-1 ключей. Т.о. внутренняя вершина(кроме корня) имеет t-детей.

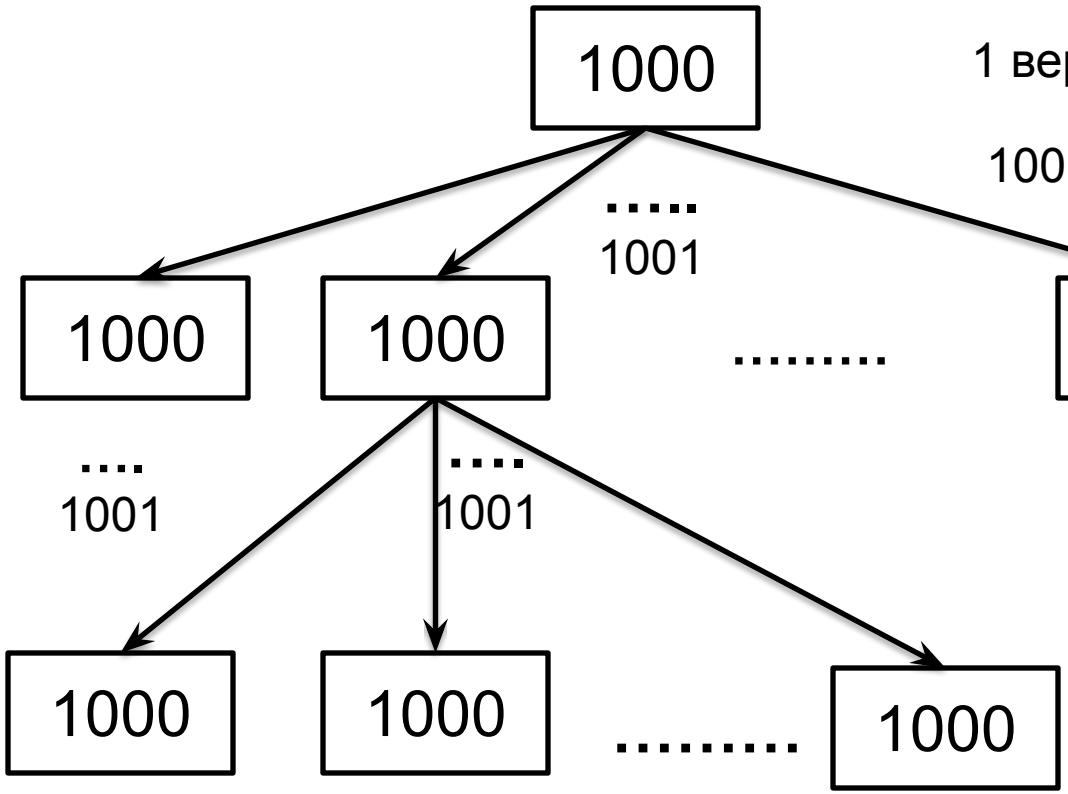
Если дерево не пусто, то в корне должен храниться хотя бы один ключ.

б) В каждой вершине хранится не более $2t$-1 ключей, внутренняя вершина имеет не более $2t$-детей . Вершину, хранящую $2t$-1 ключей, называют *полной*.

Например, t = 2, то у каждой вершины 2, 3 или 4 ребенка.

Такое дерево называется *2-3-4 деревом*.

Для эффективной работы t надо брать гораздо большим.



1 вершина – 1000 ключей

1001 вершина – 1001000 ключей

1 002 001 вершина -1 002 001 000 ключей

Б-дерево высоты 2 – содержит более миллиарда ключей.
Каждая вершина содержит 1000 ключей.
Более миллиона листьев на глубине 2.

У таких деревьев, как правило, только корень находиться в ОП, остальное дерево – на диске.

Диск разбит на сектора (дорожки на сектора).

Обычно записывают или считывают сектор целиком.

Время доступа, чтобы подвести головку к нужному месту на диске, может быть достаточно большим (до 20 миллисекунд).

Как только головка диска установлена, запись или чтение происходит довольно быстро.

Часто получатся, что обработка прочитанного занимает меньше времени, чем поиск нужного сектора.

Важно количество обращений к диску!

Реализация в ОП

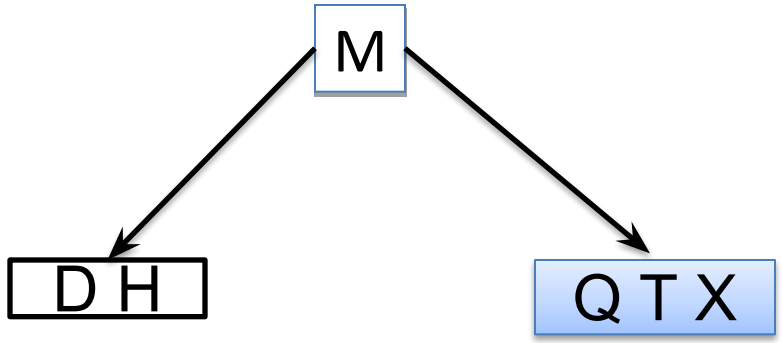
```
typedef struct tree{
    int n; //количество ключей
    int *key; //key[0]<key[1]< … <key[n-1]
    struct tree **child; //указатели на детей
} B_tree;
```

Обозначим ссылки на детей: $C_i(x)$.

Создание корня дерева

```
B_tree *B;
B=(B_tree*) malloc (sizeof(B_tree));
B->key=(int*) malloc (sizeof(int));
B->n=1;
(B->key)[0]='M';
B->child=NULL;
```

Создание дерева




```
B->child = (B_tree**)malloc(sizeof(B_tree*)*2);
B->child)[0]=(B_tree*)malloc(sizeof(B_tree));
B->child)[1]=(B_tree*)malloc(sizeof(B_tree));
x=(B->child)[0];
x->n=2;
(x->key)=(int*)malloc(2*sizeof(int));
(x->key)[0]='D';
(x->key)[1]='H';
X->child=NULL;
```

Аналогичные действия для вершины: QTX

Можно выполнить реализацию с использованием файлов, где каждый ребенок есть отдельный файл.

В общем случае можно считать, что:

- имеется операция Disk_READ(x) – чтение с диска;
- Diks_Write(x) – запись на диск.

Учитываем только количество обращений к диску!

Теорема. Для любого Б-дерева высоты h и минимальной степени $t \geq 2$, хранящего $n \geq 1$ ключей, выполнено неравенство:

$$h \leq \log_t \frac{n+1}{2}.$$

Высота Б-дерева с *n*-вершинами есть *O*(*log n*),
но основание логарифма для Б-дерева гораздо больше,
что примерно в *log t* раз сокращает количество обращений к диску.

Глубина вершины - путь из корня в вершину.
Высота (уровень) – max путь из вершины в лист.

Алгоритм поиска

Поиск в Б-дереве похож на поиск в двоичном дереве.

Разница в том, что в вершине *x* мы выбираем один вариант из (*n*(*x*)+1), а не из двух.

Процедура поиска получает на вход указатель *x* на корень поддерева и ключ *k*, который мы ищем в этом поддереве.

Если процедура обнаруживает в дереве ключ *k*, то она возвращает пару **(*y*, *i*)**, где *y* - вершина, *i* - порядковый номер указателя, для которого $key_i(y) = k$.

Иначе операция возвращает NULL.

Реализация поиска

```
B_tree_search(x,k){
int i = 0;
  while ((i < n(x)) && (k > key_i(x)))
     i++;
  if ((i<n(x))&&(k==key_i(x))
     return(x,i);
  if leaf(x) return NULL;
  else {     //считать C_i(x)-файл
     Disk_READ(C_i(x));
     return B_tree_search(C_i(x),k);
  }
}
```

Разбиение вершины Б-дерева

Добавление эелемента в Б-дерево – более сложная операция по сравнению с бинарными деревьями.

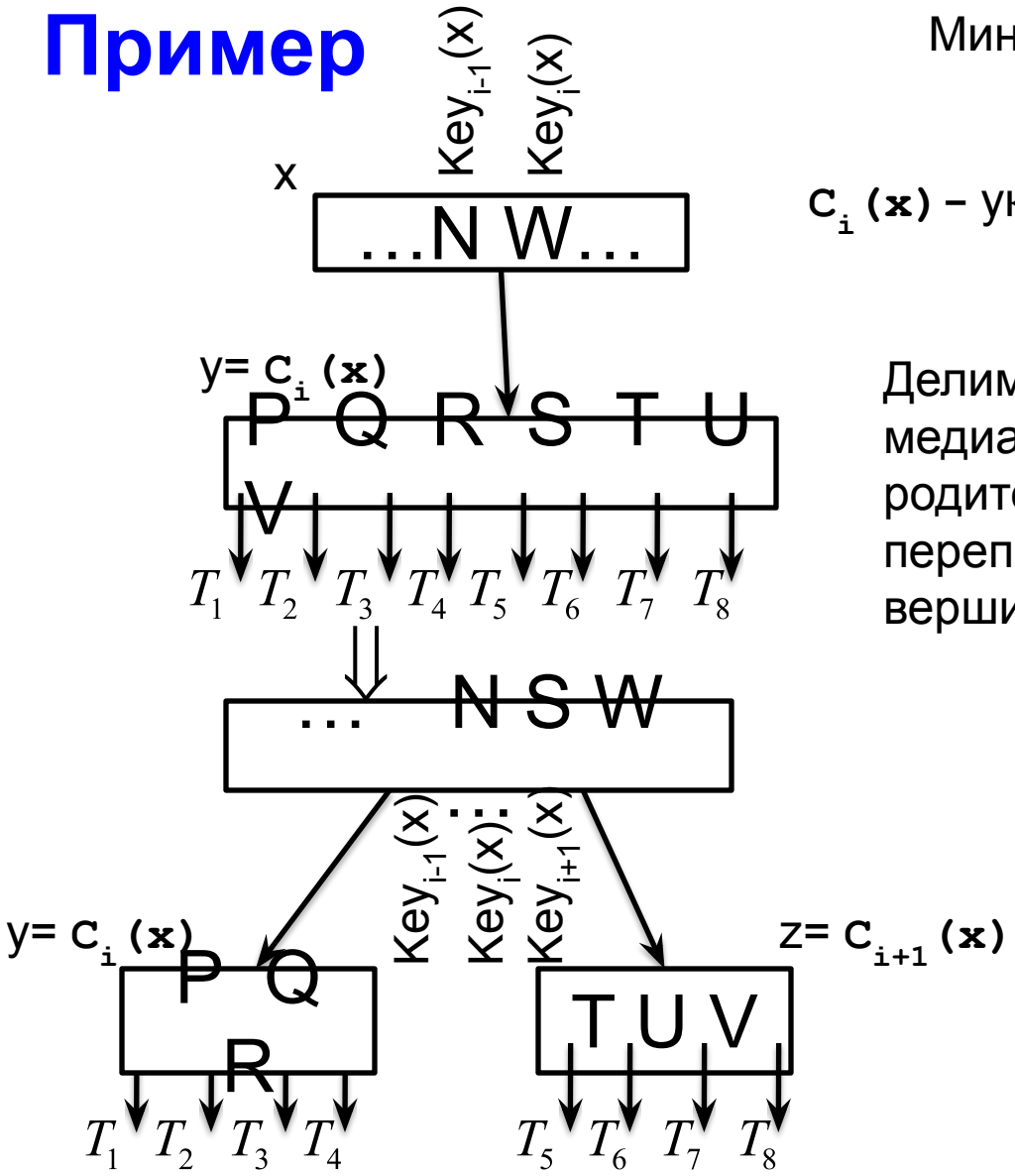
Ключевым местом является разбиение полной (с $2t$-1 ключами) вершины на две вершины, имеющие по t-1 ключей в каждой.

При этом ключ-медиана $key_{t1}(y)$ отправляется к родителю x вершины y и становится разделителем двух полученных вершин. Это возможно, если вершина x неполна.

Если y – корень, то высота дерева увеличивается на 1.

Пример

Минимальная степень t=4.



$c_i(x)$ – укакзатель на i –го ребенка в x

Делим вершину y на две: y и z. Ключ медиана *S* вершины *y* переходит к ее родителю *x* . Ключи, больше *S,* переписываются в нового ребенка z вершины x.

Входные данные: неполная внутренняя вершина *x*, число *i* и полная вершина *y*: $y = C_i(x)$ (считаем, что *x* и y уже в ОП).

```
B_tree_SPLIT_Child (x, i, y)
{
  z – создать узел;(файл,отвести место).
  leaf(z)=leaf(y);
  n(z)=t-1;
  for(j=0;j<t-1;j++)
      key_j(z)=key_{j+t}(y);
  if (!leaf(y))
   for(j=0;j<t;j++)
      C_j(z)= C_{j+t}(y);
  n(y)=t-1;
```

```
    for(j=n(x)+1; j ≤ i; j--)
         C_{j+1}(x)= C_j(x);
    C_{i+1}[x]=z;
    for(j=n(x); j ≤ i; j--)
       key_{j+1}(x)=key_j(x);
    key_i(x)=key_j(y);
    n(x)=n(x)+1;
    Переписать вершины: y, z, x.
    (Disk_Write [x])
}
```

Вершина *y*-имела 2*t*-детей, после разбиения в ней осталось *t*-детей. Остальные *t*-детей стали детьми новой вершины z.

Добавление элемента в Б-дерево

Процедура *B_tree_insert* (*T*, *k*) – добавляет элемент *k* в Б-дерево *T*, пройдя один раз от корня к листу.

На это требуется время $O(t \log_t n)$ и $O(h)$-обращений к диску, если высота дерева равна *h*.

По ходу дела с помощью процедуры *B_tree_Split_child* разделяются вершины, которые являются полными и которые имеют неполного родителя.

В результате, доходим до неполного листа, куда и добавляем новый элемент.

```
B_tree_insert (T, k) {
      // добавление в дерево с полным корнем
   r = root(T);
   if (n(r)== 2t-1){
    s= выделяем память/файл для нового узла;
    root(T)= s;     //он становится корнем     leaf(s)=
   0;
    n(s)= 0;
    C1(s)= r;
    B_tree_split_child (S, 1, r);
    B_tree_insert_nonfull (s, k);//добавляет
   }        // элемент в k в поддерево с корнем в неполной
    else    //вершине
    B_tree_insert_nonfull (r, k);
}
```

Добавление элемента в неполную вершину

`B_tree_insert_nonfull (r, k)`-
рекурсивно вызывает себя, при необходимости, выполнив разделение.

Если вершина *x* – лист, то ключ *k* в него добавляется.

Иначе *k* добавляется к поддереву, корень которого является ребенком *x*. Для этого определяется нужный ребенок вершины *x*.

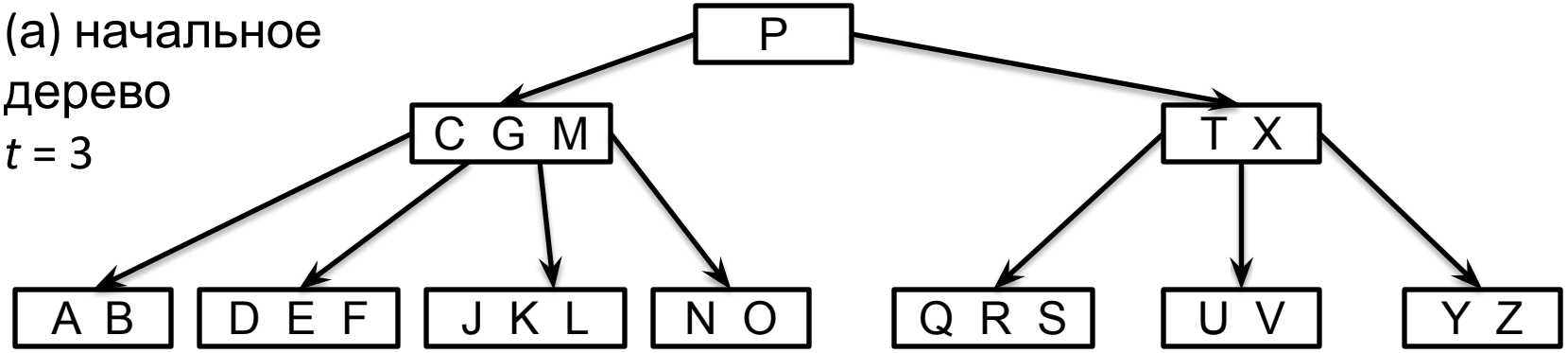
Если ребенок – полная вершина, то он разделяется.

```
B_tree_insert_nonfull(x, k){
  i = n(x);
  if leaf(x){ // ключ вставляется в лист
      while((i≥0)&&(k<key_i(x))){
        key_{i+1}(x)=key_i(x);
        i--;
      }
       key_{i+1}(x)=k;
       n(x) = n(x)+1;
      Disk_WRITE (x);
  }
  else {while((i ≥ 0) &&(k < key_i(x)))
       i--;  // поиск нужного ребенка
```

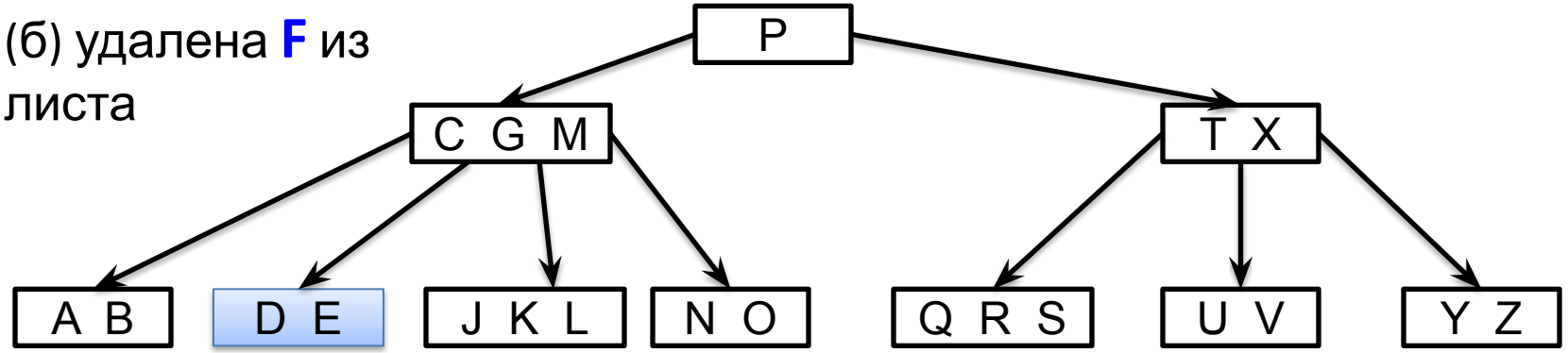
```
    i= i+1;
    Disk_READ(C_i(x));
    if (n(C_i(x))== 2t-1)
          //если ребенок-полная вершина
    B_tree_split_child (x, i, C_i(x));
          // разделение
    if (k > key_i(x)) i=i+1;
    B_tree_ insert_nonfull (C_i(x), k);
}
```

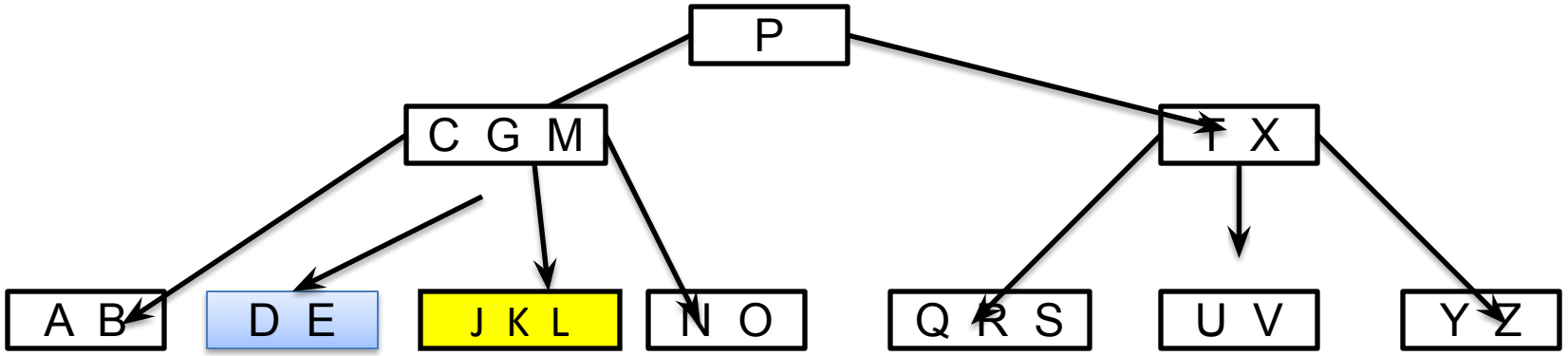
Удаление элемента из Б-дерева

(а) начальное дерево
$t = 3$

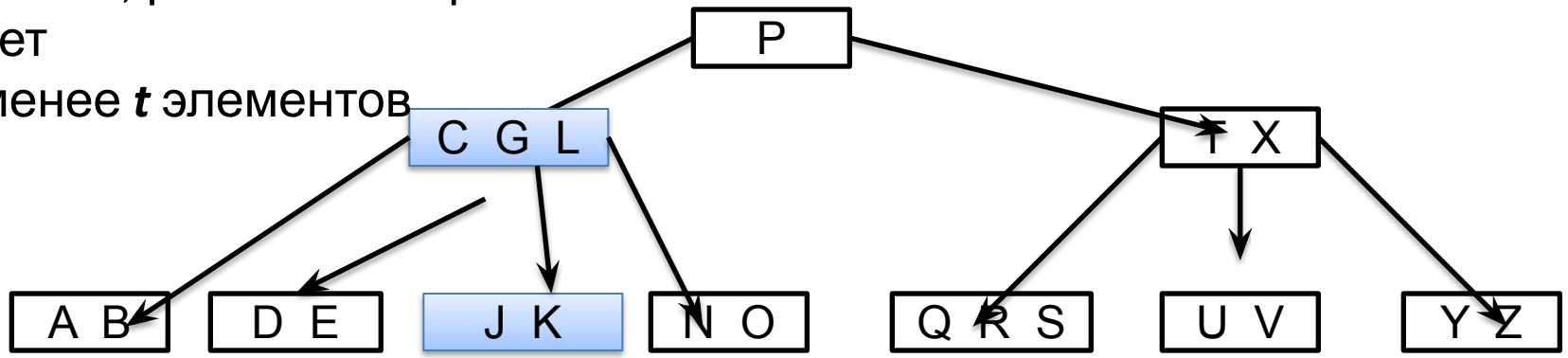


(б) удалена **F** из листа

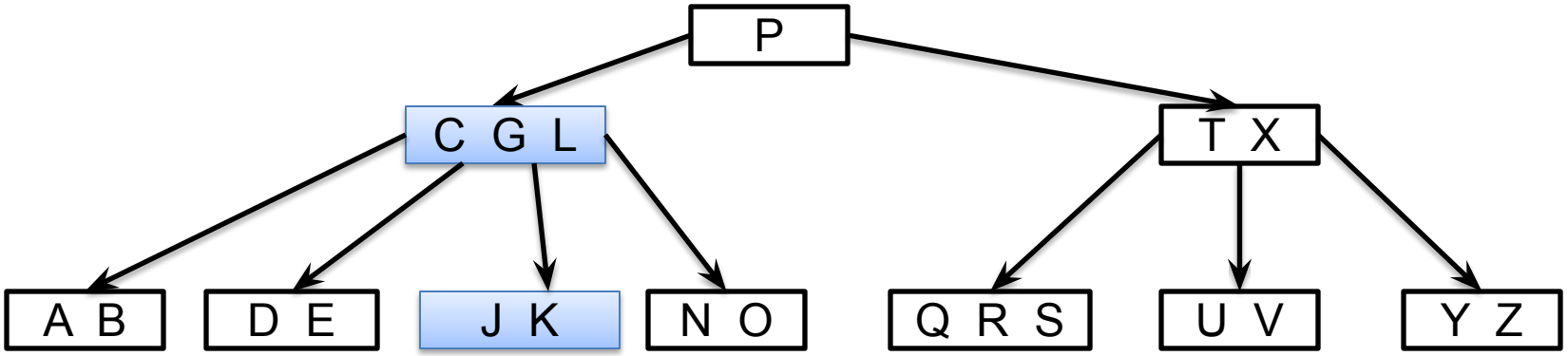




(в) **удалена M** из внутренней вершины, ребенок которой имеет
не менее ***t*** элементов

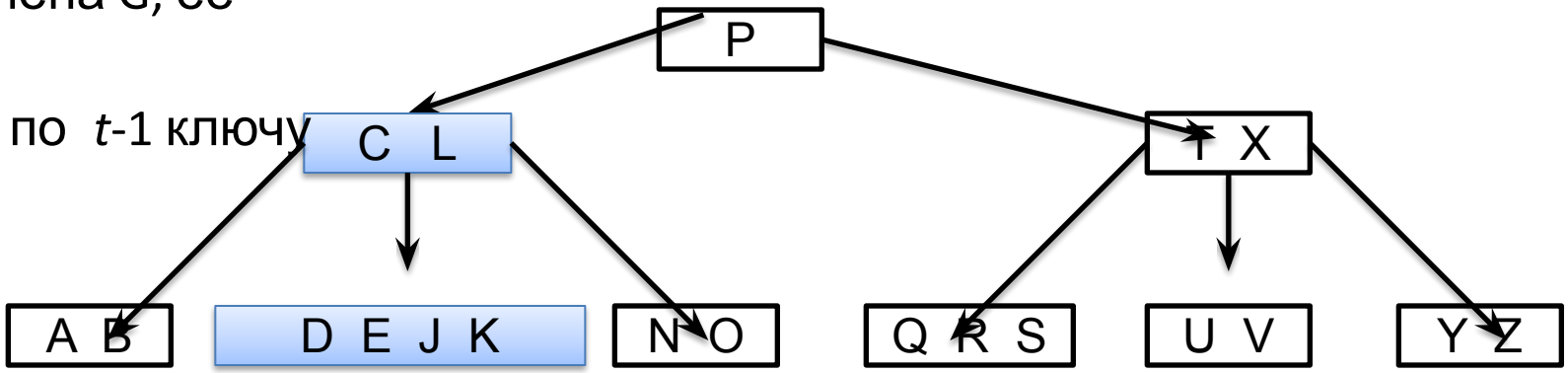


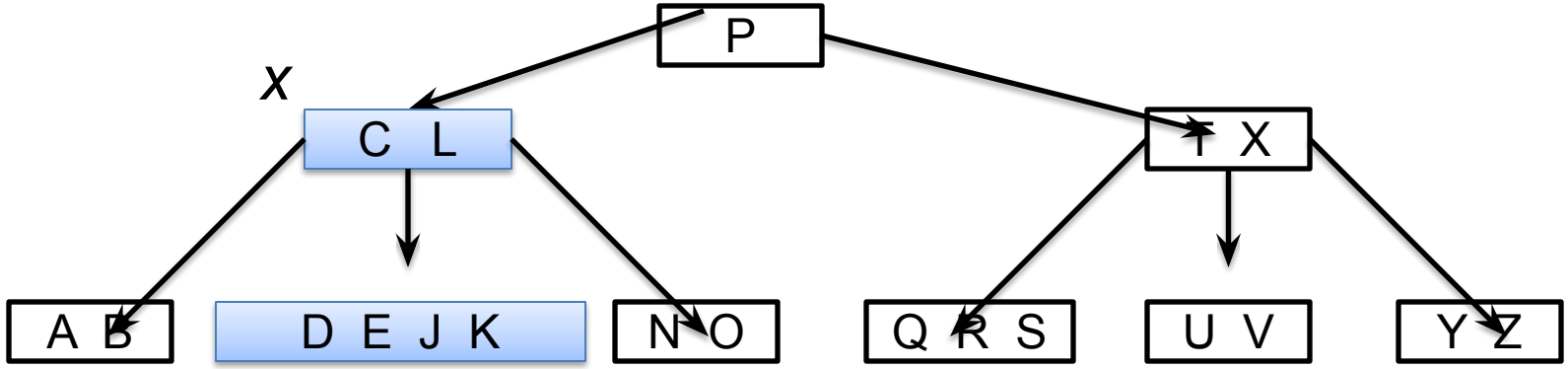
Если ребенок, следующий за удаляемым ключом, имеет не менее *t* элементов,
Поступаем аналогично (в)



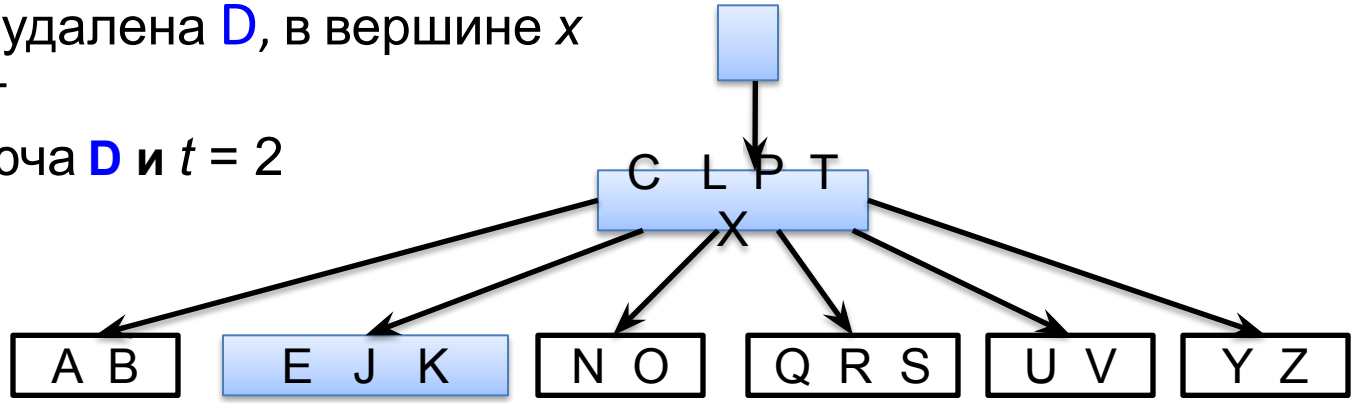
(г) удалена G, ее дети

имеют по *t*-1 ключу

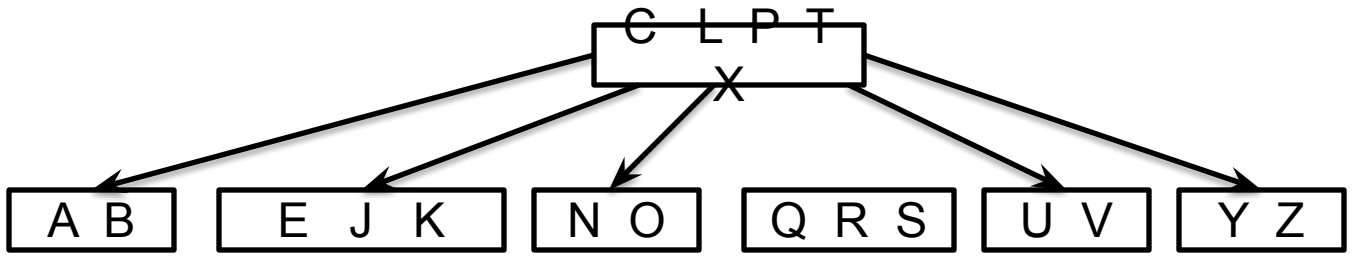




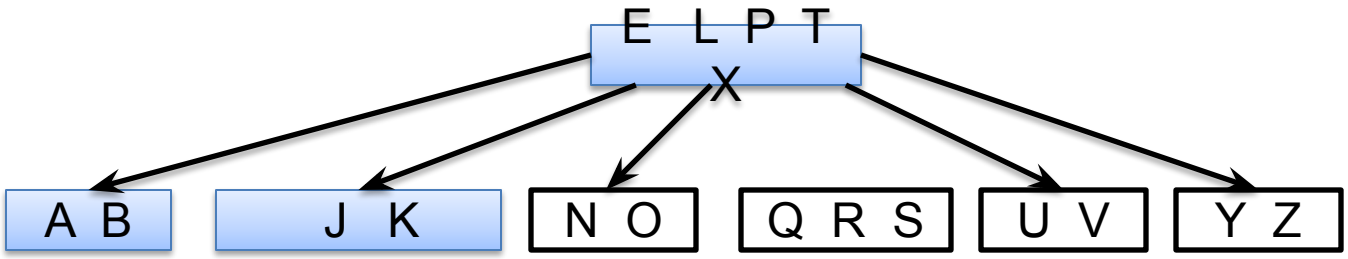
(д) удалена D, в вершине *x* нет ключа **D и** *t* = 2



(д’) уменьшение высоты дерева



(е) удалена
C



$O(h)$ – обращений к диску!