# Алгоритмы и структуры данных

## Лекция 8

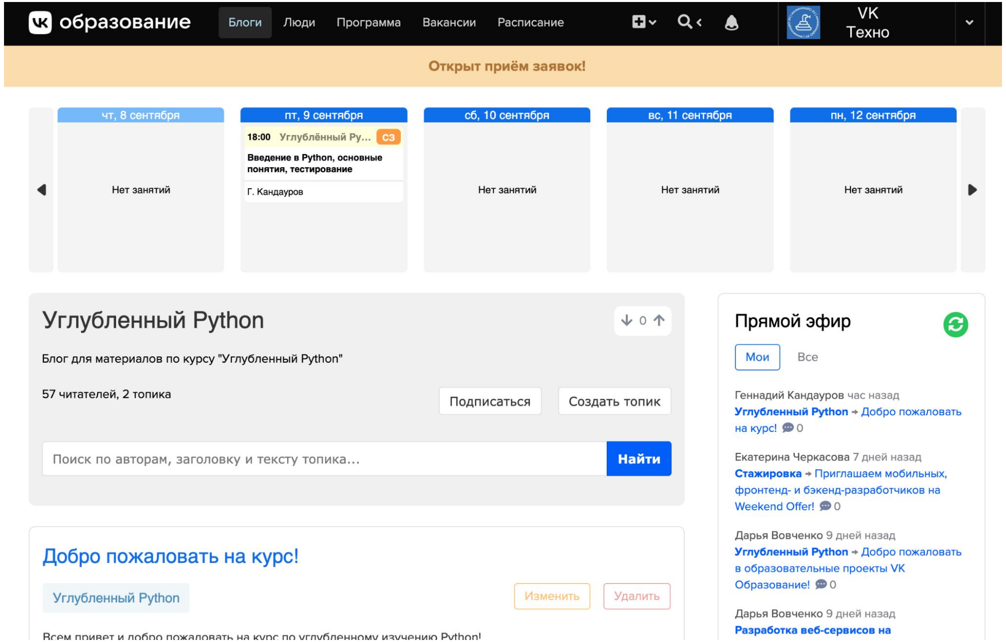
## Продвинутые деревья

Кандауров Геннадий

education

# Напоминание отметиться на портале

+ оставить отзыв



# Квиз про прошлой лекции

# Содержание занятия

1. Красно-черные деревья
2. B-дерево
3. Trie

# Красно-черное дерево

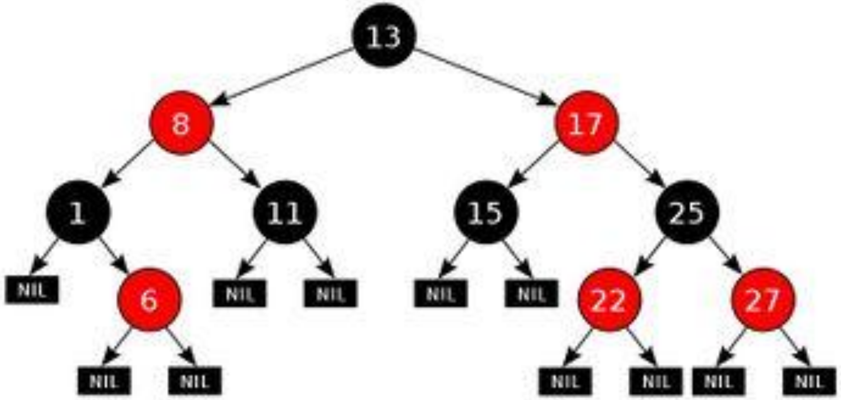
# Красно-черные деревья

**Красно-черное дерево** – двоичное дерево поиска, в котором баланс осуществляется на основе "цвета" узла дерева, который принимает только два значения: "красный" и "чёрный".

Все листья дерева являются фиктивными и не содержат данных, но относятся к дереву и являются чёрными.

Для экономии памяти фиктивные листья делают одним общим фиктивным листом.

Изобретатель – Рудольф Байер (1972г).



# Красно-черные деревья

**Красно-черное дерево** – двоичное дерево поиска, у которого каждому узлу сопоставлен дополнительный атрибут – цвет и для которого выполняются следующие свойства:

1. Каждый узел промаркирован красным или чёрным цветом
2. Корень и конечные узлы (листья) дерева — чёрные
3. У красного узла родительский узел — чёрный
4. Все простые пути из любого узла ***x*** до листьев содержат одинаковое количество чёрных узлов

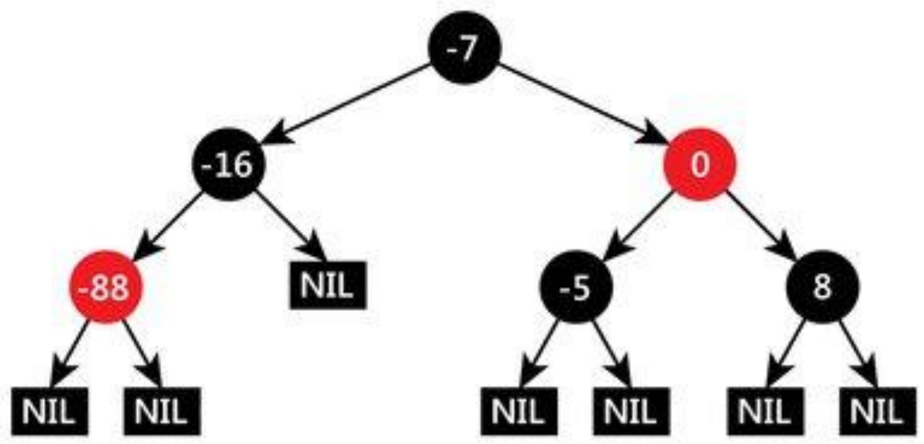
# Красно-черные деревья

**Черная высота вершины x** – число черных вершин на пути из x в лист, не учитывая саму вершину x.

<u>Пример.</u>

Для вершин «-16» и «-88» черная высота = 1.

Для вершин «-7» и «0» черная высота = 2.



# Красно-черные деревья

**В красно-черном дереве с корнем в узле x содержится по крайней мере $2^{bh(x)}$ - 1 внутренних вершин.**

Докажем по индукции по h(x):

- h(x) = 0
  Значит, x – лист (NIL). Тогда дерево с корнем x содержит $2^{bh(x)} - 1 = 2^0 - 1 = 0$ внутренних вершин.

- h(x) > 0.
  X – внутренняя вершина, у нее 2 потомка. У каждого потомка черная высота либо bh(x), либо bh(x) – 1, в зависимости от цвета потомка (красный и черный, соответственно).

  Так как h(потомок x) < h(x), то к нему применимо предположение индукции: у каждого потомка по крайней мере $2^{bh(x)-1} - 1$ внутренняя вершина.
  Тогда в дереве с корнем в x их по крайней мере
  $$(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$$
  Переход доказан. Следовательно, утверждение верно и для всего дерева.

# Красно-черные деревья

**Теорема**:

Красно-чёрное дерево с $N$ ключами имеет высоту $h = O(\log N)$.

**Доказательство:**

$\triangleright$

Рассмотрим красно-чёрное дерево с высотой $h$. Так как у красной вершины чёрные дети, чёрных вершин не меньше, чем $h/2$.

По доказанной лемме, для количества внутренних вершин в дереве $N$ выполняется неравенство:

$$N \geqslant 2^{h/2} - 1$$

Прологарифмировав неравенство, имеем:

$$\log(N+1) \geqslant h/2$$

$$2\log(N+1) \geqslant h$$

$$h \leqslant 2\log(N+1)$$

$\triangleleft$

# Красно-черные деревья: вставка

- Каждый элемент вставляется вместо листа.
- Для выбора места вставки идём от корня в нужную сторону, как в наивном методе построения дерева поиска. До тех пор, пока не остановимся в листе (в фиктивной вершине).
- Вставляем вместо листа новый элемент красного цвета с двумя листами-потомками.
- Теперь восстанавливаем свойства красно-черного дерева.

# Красно-черные деревья: вставка

**Что можем сломать?**

Поскольку добавленный узел автоматически окрашивается в красный цвет, то нарушить можем только эти свойства красно-черного дерева:

- Корень дерева — чёрный
- У красного узла родительский узел — чёрный

# Красно-черные деревья: вставка

Ситуации после вставки:

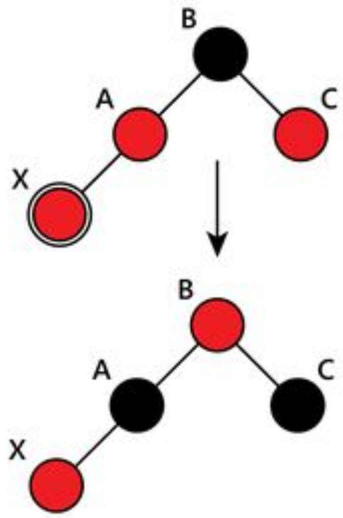
- Если отец нового элемента черный, то ничего делать не надо.
- Если отец нового элемента красный, то возможны 3 случая (без учета симметрии):

    1. Отец красный, дядя красный.
    2. Отец красный, дядя черный, новый элемент левый потомок.
    3. Отец красный, дядя черный, новый элемент правый потомок.

# Красно-черные деревья: вставка

<u>Случай 1.</u> Отец и дядя красные.

- Перекрашиваем «отца» и «дядю» в чёрный цвет, а «деда» - в красный.
- Поскольку «дед» может нарушать свойство дерева (вдруг его отец красный), придется рекурсивно восстанавливать свойства дерева, двигаясь к предкам.
- Если мы таки образом дойдём до корня, то в нём в любом случае ставим чёрный цвет.

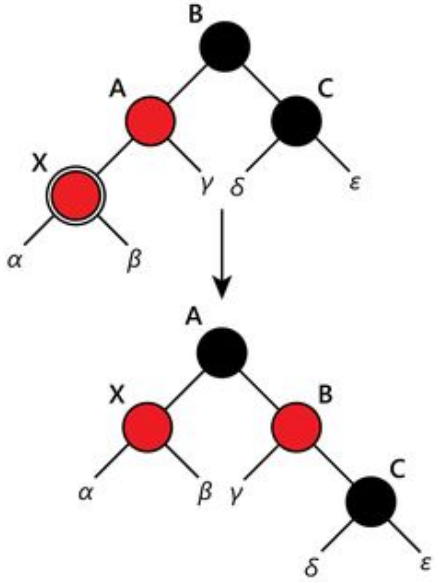


# Красно-черные деревья: вставка

Случай 2. «Дядя» черный и правый, новый элемент — левый потомок.

Просто выполнить перекрашивание отца в черный цвет нельзя, чтобы не нарушить постоянство чёрной высоты дерева по ветви с отцом.

- Выполняем правый поворот B.
- Перекрашиваем A и B.
- Останавливаемся — больше ничего делать не требуется.

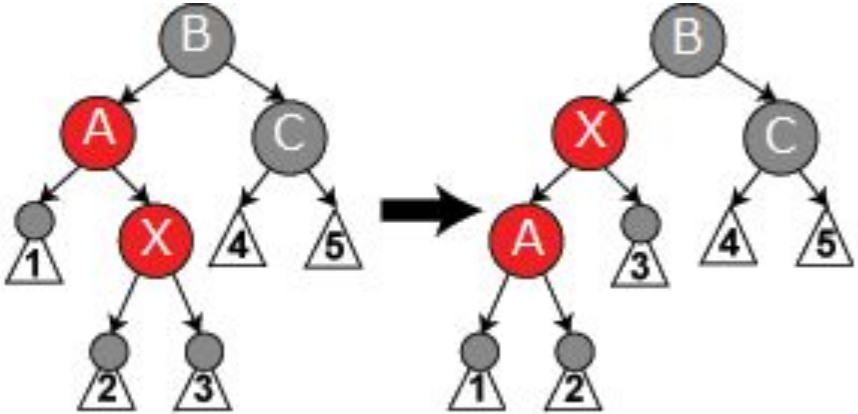


# Красно-черные деревья: вставка

<u>Случай 3.</u> «Дядя» черный и правый, новый элемент — правый потомок.

Добавленный узел *X* — правый потомок отца А. Выполним левое вращение в А, тем самым сделав А левым потомком *X*.

Случай сводится к предыдущему.



# Красно-черные деревья: вставка

Всего один шаг, если:

- Родитель черный, тогда вообще ничего делать не нужно
- Случай 1, когда отец деда черный или дед — корень.
- Случай 2
- Случай 3

Длинная цепочка действий возможна только при многократном повторении случая 1.

Сложность вставки ***O(logN)***.

# Красно-черные деревья: удаление

**Как производится удаление:**

1. Если у удаляемой вершины нет детей, у родителя перенаправляем указатель на фиктивный лист.

2. Если только один потомок, у родителя перенаправляем указатель на этого потомка.

3. Если потомка два, ищем в поддеревьях следующую или предыдущую вершину. Вместо исходной, удаляем именно эту вершину способом из п.1 или п.2, предварительно скопировав её ключ в изначальную вершину.

**Таким образом, удаление всегда выполняется для вершины, имеющей не более одной дочерней.**

# Красно-черные деревья: удаление

**Что можем сломать?**

- Корень дерева — чёрный
- У красного узла родительский узел — чёрный
- Все простые пути из любого узла x до листьев содержат одинаковое количество чёрных узлов

# Красно-черные деревья: удаление

**А. Удаление красной вершины**

При удалении красной вершины свойства дерева не нарушаются.

<u>Если потомок единственный</u>

Красная вершина, не может иметь единственного потомка. Если бы потомок существовал, то он был бы черным и нарушилось бы свойство постоянства черной глубины для потомка и его соседней фиктивной вершины.

<u>Если потомков нет</u>


Действия:

- Удалить красную вершину (заменить на лист)
- Конец

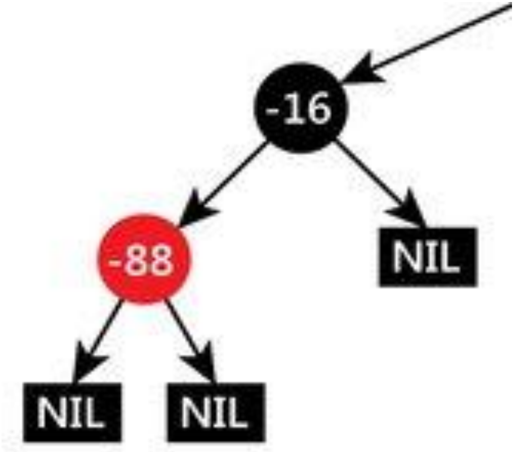
# Красно-черные деревья: удаление

**Б. Удаление черной вершины с единственным потомком**

Единственным потомком черной вершины может быть только красная вершина. Иначе нарушилось бы свойство постоянства черной глубины для потомка и его соседней фиктивной вершины.

<u>Действия:</u>

- В черную вершину заносим данные красной.
- Удаляем красную (заменяем на лист)
- Конец



# Красно-черные деревья: удаление

**B. Удаление черной вершины без потомков. Это самый сложный случай.**

<u>Действия:</u>

- Удалим черную вершину (заменим на лист)
- Лист на месте удаленной вершины обозначим «*x*»

Путь в «*x*» имеет меньшее количество черных вершин (черную глубину), чем в другие вершины. Будем помнить об этом и называть «*x*» дважды черным.

Теперь с помощью перекрашиваний и вращений будем пытаться восстановить свойства красно-черного дерева.

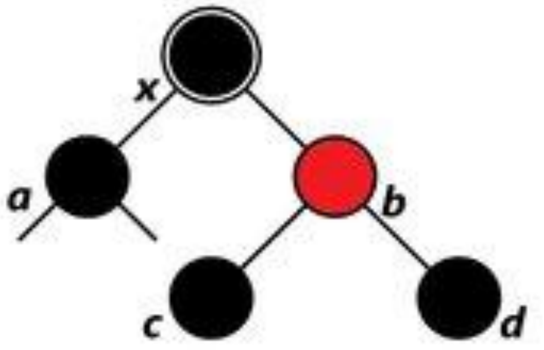
# Красно-черные деревья: удаление

Восстановление свойств. Случай 0.

Если дважды черная вершина ***x*** – корень.

- Оставим корень просто черным (один раз черным)
- Конец

Так черная глубина всего дерева уменьшится на 1.



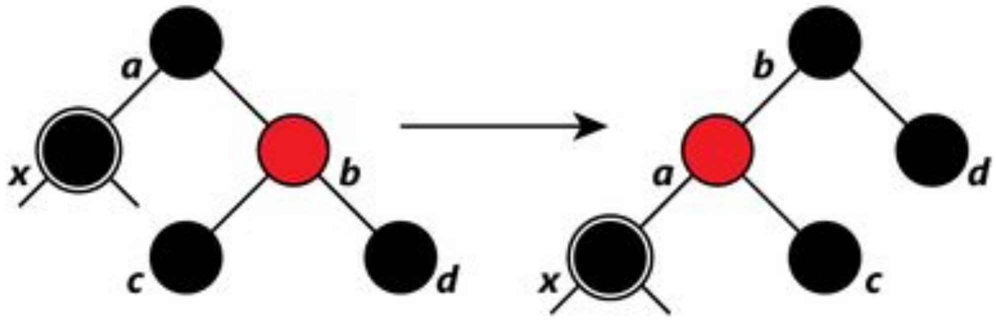
# Красно-черные деревья: удаление

Восстановление свойств. Случай 1.

Если брат ***b*** дважды черной вершины ***x*** – красный.

- Делаем малый левый поворот в ***a***. Бывший брат ***b*** становится дедом.
- Красим ***b*** в чёрный, ***a*** – в красный цвет.

Теперь брат ***x*** – черный, переходим в случай 2, 3 или 4 в зависимости от цветов детей узла *c*.

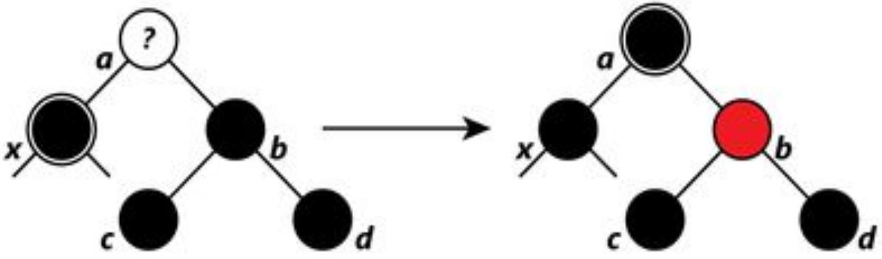


# Красно-черные деревья: удаление

Восстановление свойств. Случай 2.

Если брат ***b*** вершины ***x*** – черный, и оба дочерних узла брата ***c*** и ***d*** – черные. ***c*** и ***d*** могут быть листьями. Красим брата ***b*** в красный цвет.

- Цвет отца ***a*** - красный:
  - Красим отца *a* в черный цвет, так черная глубина ***a*** восстановится
- Цвет отца ***a*** - черный:
  - Считаем отца ***a*** дважды черным, рассматриваем его как *x*
  - Теперь может быть любой случай



# Красно-черные деревья: удаление

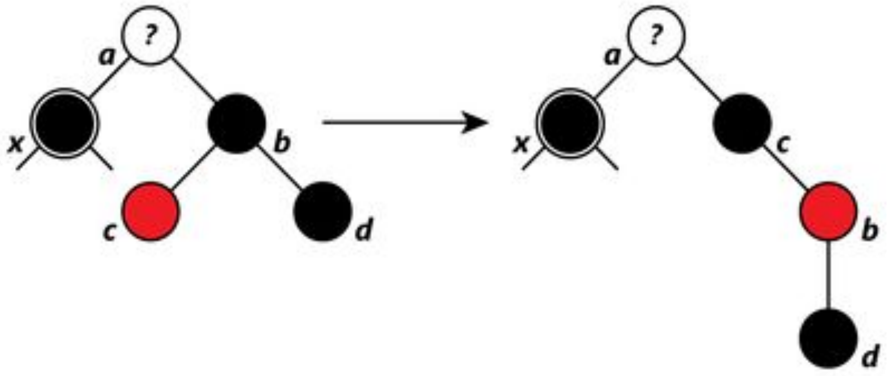
Восстановление свойств. Случай 3.

Если брат ***b*** дважды черной вершины ***x*** – черный, левый ребенок брата ***c*** – красный, а правый ***d*** – черный.

- Делаем малое правое вращение в ***b***
- Красим ***b*** в красный цвет
- Красим ***c*** в черный цвет

Так у брата правый ребенок станет красным.

После случая 3 всегда случай 4, затем конец.



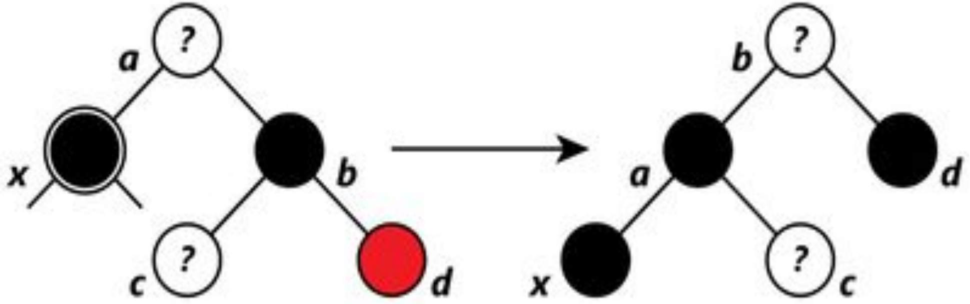
# Красно-черные деревья: удаление

Восстановление свойств. Случай 4.

Если брат ***b*** дважды черной вершины ***x*** – черный, правый ребенок брата ***d*** – красный.

- Делаем малое левое вращение в ***a***
- Красим ***b*** в цвет, который был у ***a***
- Красим ***a*** в черный цвет

Так черная глубина ***x*** увеличится на 1, то есть восстановится. Готово.



# Красно-черные деревья: удаление

**Восстановление свойств**

Последовательность обработки случаев:

- Случай 1 → Случай 2 → Конец (отец узла x гарантированно красный после Случая 1)
- Случай 1 → Случай 3
- Случай 1 → Случай 4
- Случай 3 → Случай 4
- Случай 4 → Конец
- Случай 2 → Любой случай (если отец узла x черный)

Длинная цепочка возможна только при переходах Случай 2 → Случай 2.

# Красно-черные деревья

**Восстановление свойств**

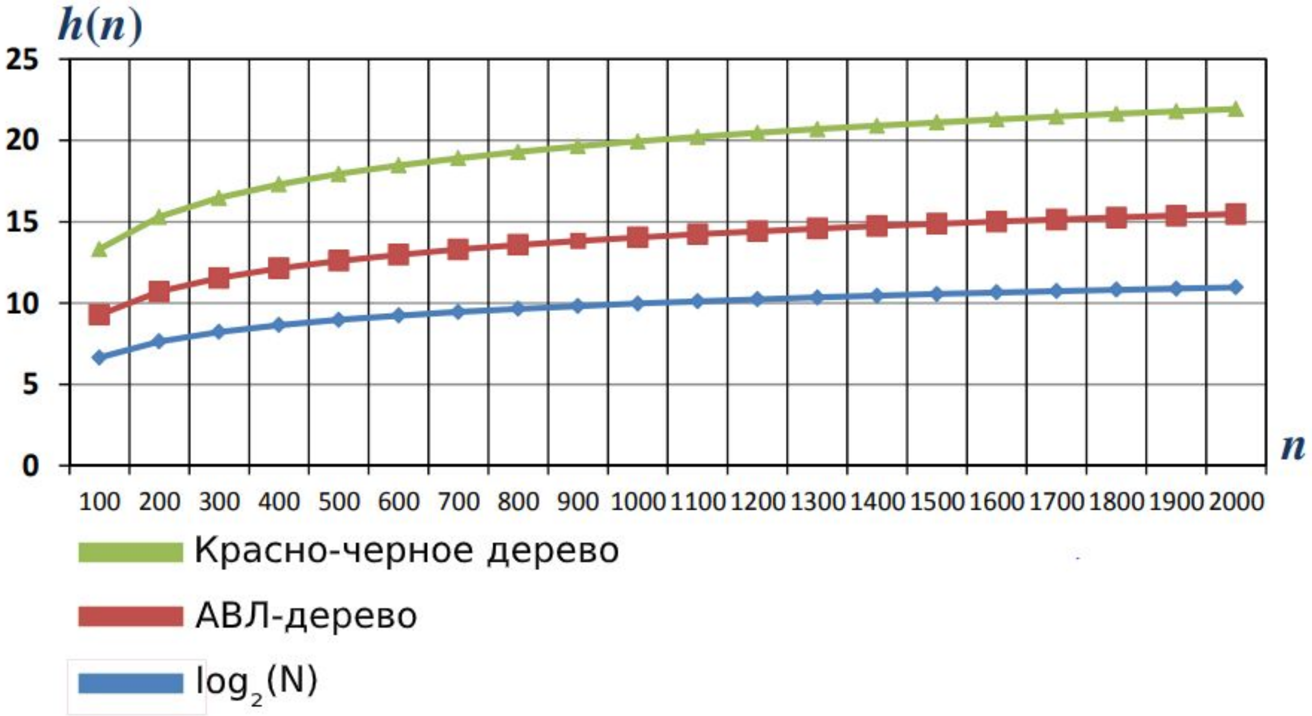
| Вставка | Удаление |
| --- | --- |
| ▪ Случай выбирается по цвету «дяди».<br>▪ Максимум 2 вращения | ▪ Случай выбирается по цвету «брата» и его потомков.<br>▪ Максимум 3 вращения |

# Красно-черные деревья

**Расход памяти и время работы**

| | В среднем случае | В худшем случае |
|---|---|---|
| Расход памяти | O(n) | O(n) |
| Поиск | O(log(n)) | O(log(n)) |
| Вставка | O(log(n)) | O(log(n)) |
| Удаление | O(log(n)) | O(log(n)) |

# Оценка высот АВЛ и красно-черных деревьев



# Сравнение АВЛ и красно-черных деревьев

- AVL деревья более строго сбалансированы, быстрее выполняют поиск элемента. Максимальная высота ~1.44 * $\log_2 n$.
- Красно-черные деревья быстрее выполняют вставку и удаление элемента.
- AVL деревья хранят в узлах значение баланса или высоту узла, нужно тратить память на целочисленную переменную.
- Красно-черные деревья хранят в узлах цвет (всего 2 возможных значения).
- Красно-черные деревья получили более широкое распространение: стандартные контейнеры в C++ STL, Java, ядро Linux (например, планировщик) и пр.

# B-дерево

# B-дерево

- Сбалансированное дерево поиска, обобщает понятие двоичного дерева поиска.
- Узлы B-деревьев могут иметь тысячи потомков
- Оптимизирует работу с диском, минимизируя число операций чтения/записи.
- Многие БД хранят данные в B-деревьях

Придумано Р. Бэйером (англ. R. Bayer) и Э. МакКрейтом (англ. E. McCreight) в 1970 году.

# B-дерево

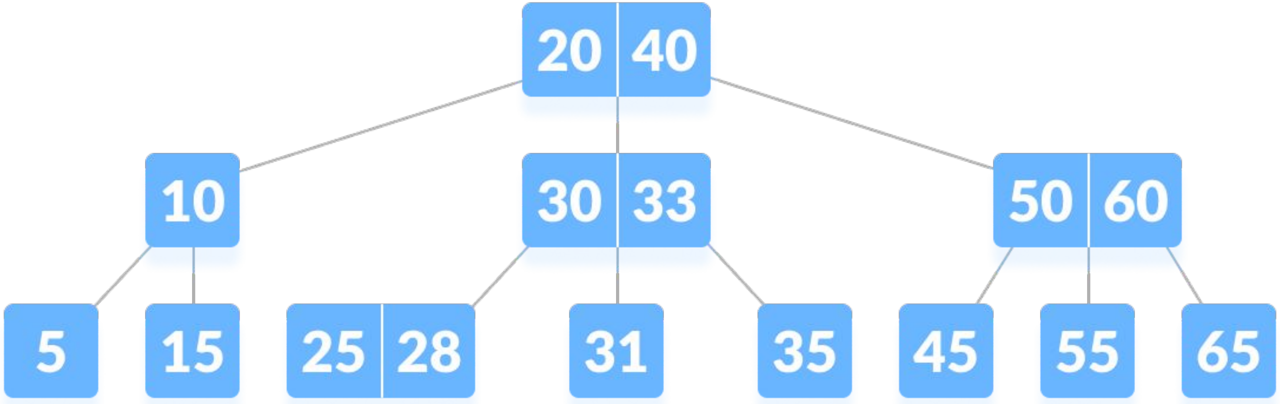
**B-дерево** – дерево с корнем, обладающее следующими свойствами:

1) Каждый узел X содержит:
   - X.n – число ключей в узле
   - Сами ключи в отсортированном порядке: $X.key_1 \le X.key_2 \le \ldots \le X.key_n$
   - X.leaf – булево значение “узел является листом”
2) Каждый внутренний узел также содержит:
   - X.n + 1 указателей $x.c_{1,}$ $x.c_{2, \ldots,}$ $x.c_{n+1}$ на потомков.

# В-дерево

3) Ключи $X.key_i$ внутри узла определяют, в какое поддерево переходим при поиске/вставке/удалении.

4) Пусть $\{k_i\}$ – все ключи из поддерева, на которое указывает $X.c_i$, тогда $\{k_1\} \le X.key_1 \le \{k_2\} \le X.key_2 \le \ldots \le X.key_{X.n} \le \{k_{X.n+1}\}$.



# B-дерево

4) Все листья B-дерева находятся на одной глубине

5) Минимальная степень B-дерева – целое число $t \geq 2$, определяет минимум и максимум числа ключей в узле.

- Все внутренние узлы, помимо корня, должны хранить не менее $t - 1$ ключей (следовательно, иметь не менее $t$ потомков). В непустом дереве в корне хранится по крайней мере 1 ключ.

- Все узлы хранят максимум $2t - 1$ ключ (следовательно, имеют не более $2t$ потомков). Если в узле $2t - 1$ ключ, то его называют заполненным.

- Чем больше $t$, тем меньше высота дерева.

# B-дерево: высота

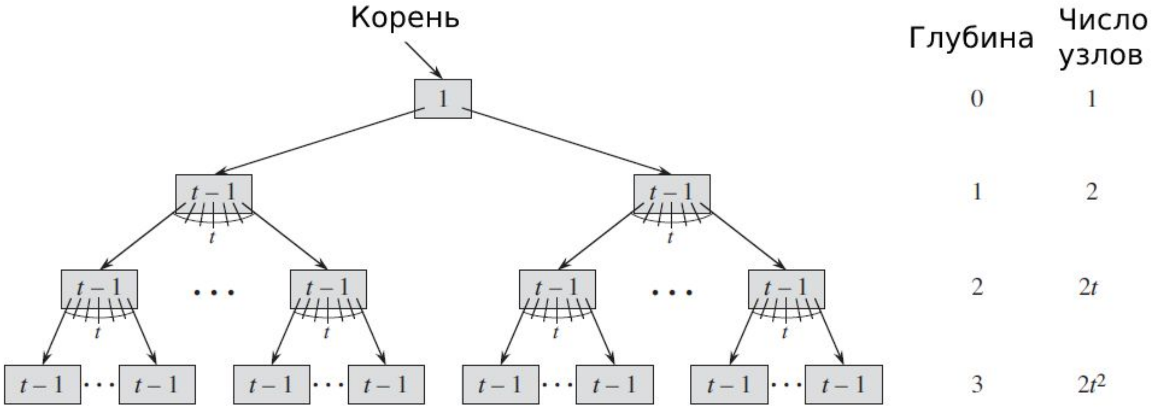
**Теорема:**
Если N ≥ 1, то для любого B-дерева, содержащего N ключей и минимальной степенью t ≥ 2, верно утверждение
$h \leq \log_t\left(\frac{n+1}{2}\right)$.

**Доказательство**:
Корень B-дерева содержит по крайней мере 1 ключ, прочие узлы – по крайней мере t - 1 ключ. Следовательно, дерево T с высотой h имеет по крайней мере 2 узла на глубине 1, 2t узлов на глубине 2, $2t^2$ узлов на глубине 3, ..., $2t^{h-1}$ узлов на глубине h.
(Продолжение на следующем слайде)

# B-дерево: высота

**Доказательство**: (продолжение)

Иллюстрация для $h$ = 3:




Таким образом, имеет место неравенство:

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} = 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) = 2t^h - 1$$

Следовательно, $t^h \leq \left(\frac{n+1}{2}\right)$. Возьмем $\log_t$ от обеих частей неравенства: $h \leq \log_t\left(\frac{n+1}{2}\right)$. Что и требовалось доказать.

# B-дерево: устройство жесткого диска

- Данные хранятся на одной или несколько пластинах
- Пластины вращаются с постоянной скоростью вокруг шпинделя
- Считывающая головка перемещается к шпинделю/от шпинделя

# B-дерево: RAM и HDD

- Стоимость 1Гб оперативной памяти в ~100 раз выше, чем стоимость 1Гб на жестком диске.

- Как правило, емкость установленных в компьютере жестких дисков как минимум на 2 порядка превышает объем доступной оперативной памяти.

- Доступ к данным в RAM занимает ~50 нс, в HDD ~8-11 мс (в 160,000 - 220,000 раз медленнее). Все из-за механики – нужно прокрутить дисковые пластины и переместить считывающую головку.

# B-дерево: особенности HDD

- Минимальная единица чтения/записи – страница.

- Все страницы равного размера (обычно от 2 до 16 килобайт).
  Все биты в странице хранятся на диске единым неделимым блоком.

- Выгодно сразу читать/писать несколько страниц: основные траты времени приходятся на механическое позиционирование пластин и считывающей головки на начало первой страницы.
  Само чтение/запись происходят быстро – там одна электроника, без механики (без учета постоянного вращения пластины).

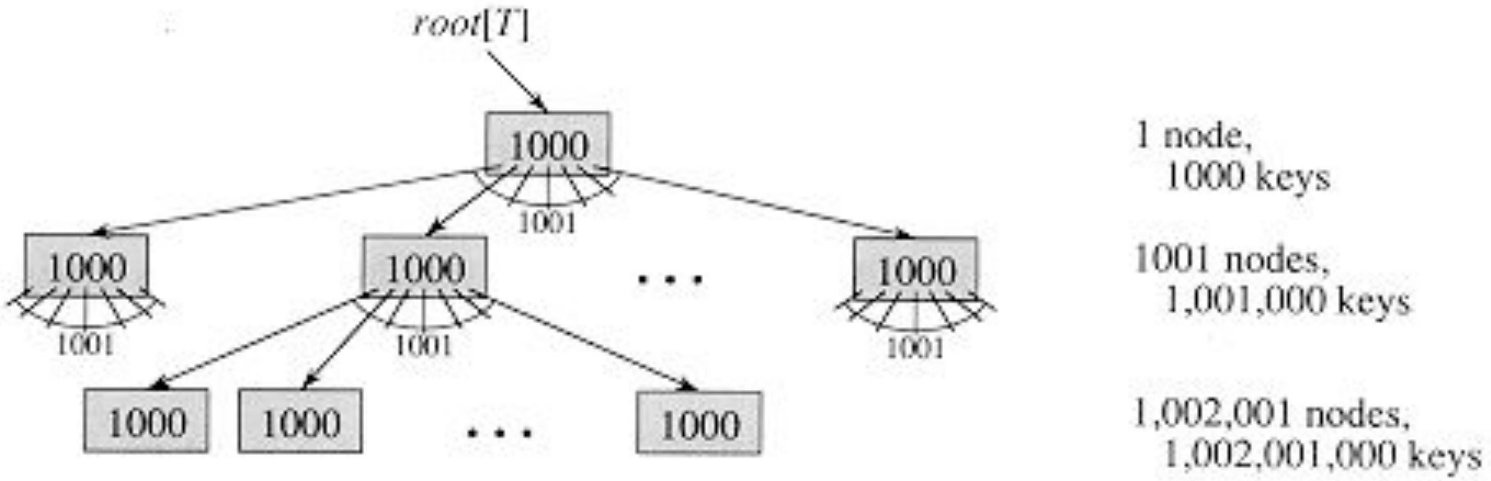
# B-дерево: учитываем HDD

- Предполагается, что хранимые данные не помещаются в оперативную память.
- B-дерево держит в оперативной памяти ограниченное константой число страниц. По мере надобности страницы подгружаются с диска в память, а при модификации записываются на диск.
- Для оптимизации чтения/записи размер узла B-дерева подбирается под размер страницы жесткого диска. Это накладывает ограничения на число хранимых в узле ключей и, соответственно, число потомков.
- В больших B-деревьях фактор ветвления обычно от 50 до 2000 (зависит от размера ключа). Чем он больше, тем меньше высота дерева, а значит меньше доступов к диску.

# B-дерево: учитываем HDD

Пример: фактор ветвления 1001, высота 2.

Помещается более миллиарда ключей. Если корень всегда в RAM, любой ключ достижим не более чем за 2 доступа к диску.



# B-дерево

- Нет гарантий, что нужный нам узел X находится в RAM или не будет вскоре вытеснен из нее.

- Введем операцию DISK-READ(X), зачитывающую узел X с диска в RAM. Если X уже в ней, то DISK-READ не будет обращаться к диску.

- Операция DISK-WRITE(X) записывает на диск узел X. Необходимо ее вызывать после любой модификации узла X.

- Никогда не будем вытеснять корень дерева из RAM, поэтому для него DISK-READ не понадобится. Но при модификации корня все равно нужен DISK-WRITE.

# B-дерево: создание

B-TREE-CREATE создает пустой корневой узел.

```
B-TREE-CREATE(T)
  X = ALLOCATE-NODE()
  X.leaf = TRUE
  X.n = 0
  DISK-WRITE(X)
  T.root = X
```

Вспомогательная операция ALLOCATE-NODE выделяет страницу для нового узла за O(1).

B-TREE-CREATE требует O(1) дисковых операций и O(1) процессорного времени.

# B-дерево: поиск ключа

B-TREE-SEARCH(X, k) – поиск ключа k в B-дереве с корнем X.
Если k в дереве, то вернет (Y, i) (где У – узел, i – индекс ключа в Y, такой что $Y.key_i$ == k), иначе NIL.

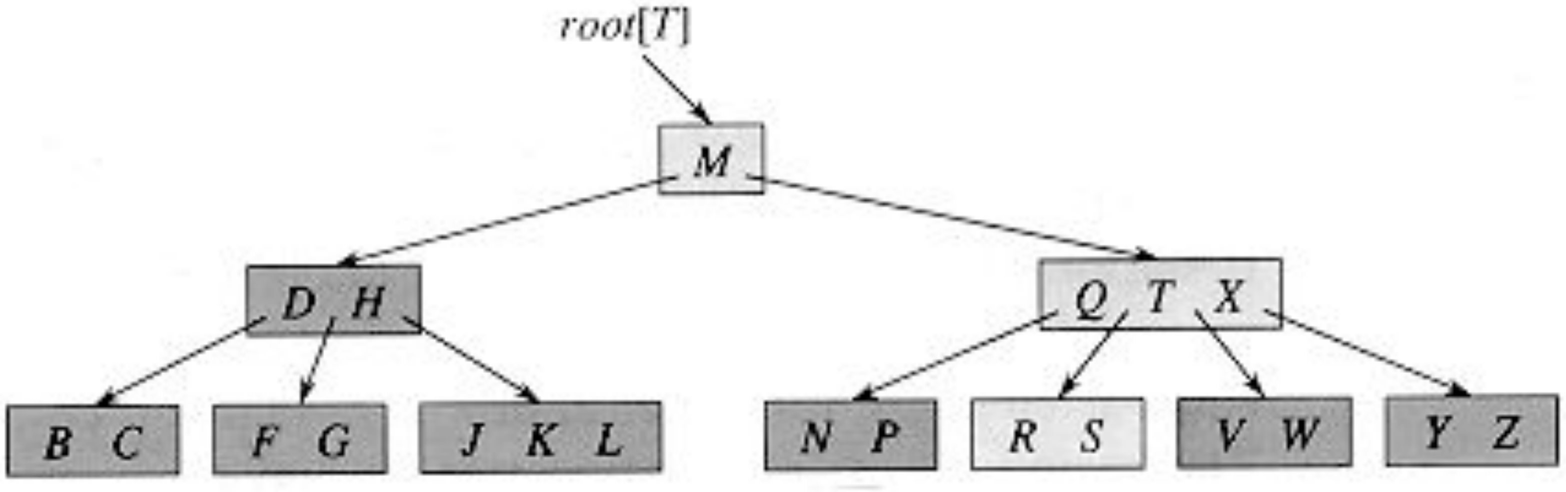
```
B-TREE-SEARCH(X, k)
  i = 1
  while i ⩽ X.n and k > X.key_i
      i = i + 1
  if i ⩽ X.n and k == x.key_i
      return (X, i)
  elseif X.leaf
      return NIL
  else
      DISK-READ(X.c_i)
      return B-TREE-SEARCH(X.c_i, k)
```

# B-дерево: поиск ключа

Сложность поиска: $O(t * \log_t N)$.

Пример:

B-дерево построено на согласных буквах английского алфавита, ищем в нем ключ R. Светлым подсвечены узлы, по которым пройдем при поиске ключа.

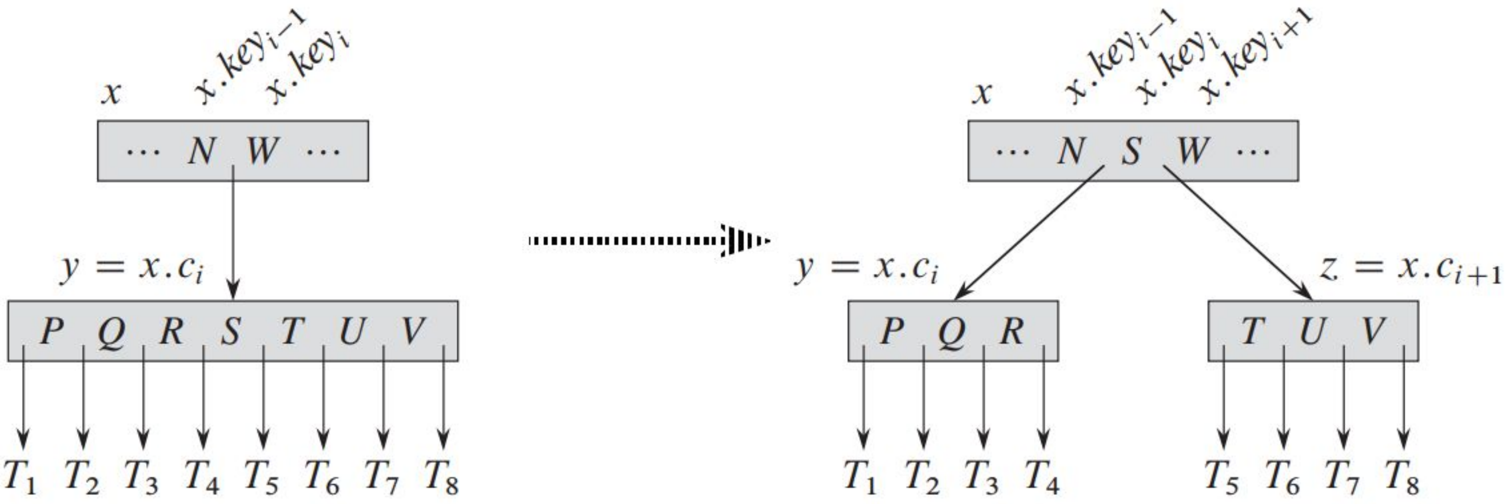


# B-дерево: вставка ключа

- Вставка сложнее, чем в бинарном дереве поиска: мы не можем создать новый лист в дереве – результат перестанет быть B-деревом.
- Будем добавлять ключ в какой-то существующий лист.
- Нельзя добавлять ключ в полный узел (хранящий $2t - 1$ ключей).
- Чтобы бороться с переполнением узлов, введем операцию B-TREE-SPLIT-CHILD, разбивающую полный узел Y по медианному ключу $Y.key_t$ на 2 новых узла, в каждом по $t - 1$ ключей. Ключ $Y.key_t$ перенесем в родителя Y. Новые деревья подцепим к нему же, $Y.key_t$ станет для них разделителем.

# B-дерево: вставка ключа

Пример применения B-TREE-SPLIT-CHILD: B-дерево, минимальная степень t = 4. Узел y содержит 2t - 1 = 7 ключей, он полный. Разбиваем по медиане (S) на 2 новых узла, S добавляем к ключам родителя (x), новые деревья цепляем по указателям $x.c_i$ и $xc_{i+1}$. У родителя +1 к ключам и +1 к потомкам.



# B-дерево: вставка ключа

- Внутри B-TREE-SPLIT-CHILD один из ключей разбиваемого узла поднимается к родителю. Но что если родитель тоже уже полный? Придется рекурсивно вызывать B-TREE-SPLIT-CHILD. Так можно до корня дойти.
- Дополнительная оптимизация: мы не будем ждать, пока потребуется добавлять ключ в полный узел. В процессе спуска по дереву для всех полных узлов заблаговременно вызываем B-TREE-SPLIT-CHILD (включая сам лист).
- Благодаря этому при вызове B-TREE-SPLIT-CHILD будем уверены, что у родителя есть место для нового ключа.

# B-дерево: вставка ключа

B-TREE-SPLIT-CHILD(X, i) – для неполного внутреннего узла X вызвать разбиение полного потомка по указателю $X.c_i$.

```
B-TREE-SPLIT-CHILD(X, i)
  Z = ALLOCATE-NODE() // новый узел-потомок X
  Z.leaf = Y.leaf // если Y - лист, то Z тоже
  Z.n = t – 1 // половина ключей Y уходит в Z
  for j = 1 to t - 1 // переносим их
      Z.key_j = Y.key_j+t
  if not Y.leaf // если у Y есть потомки
      for j = 1 to t // половину потомков Y
          Z.c_j = Y.c_j+t // переносим в Z
   Y.n = t – 1 // поправим счетчик ключей в Y
  for j = X.n + 1 downto i + 1
      X.c_j+1 = x.c_j // сдвиг указателей на 1 вправо
  x.c_i+1 = z // чтобы добавить указатель на Z
  // продолжение на следующем слайде
```

# B-дерево: вставка ключа

B-TREE-SPLIT-CHILD(X, i) – для неполного внутреннего узла X вызвать разбиение полного потомка по указателю $X.c_i$.

```
B-TREE-SPLIT-CHILD(X, i)
  ... //начало на предыдущем слайде
  for j = X.n downto i  // сдвиг ключей в X вправо
      X.key_{j+1} = X.key_j // на 1 для вставки медианы
  X.key_i = Y.key_t // бывшего узла Y (до разбиения)
  X.n = X.n + 1 // из-за этого +1 к ключам в X
  DISK-WRITE(Y)
  DISK-WRITE(Z)
  DISK-WRITE(X)
```

Сложность разбиения $O(t)$.

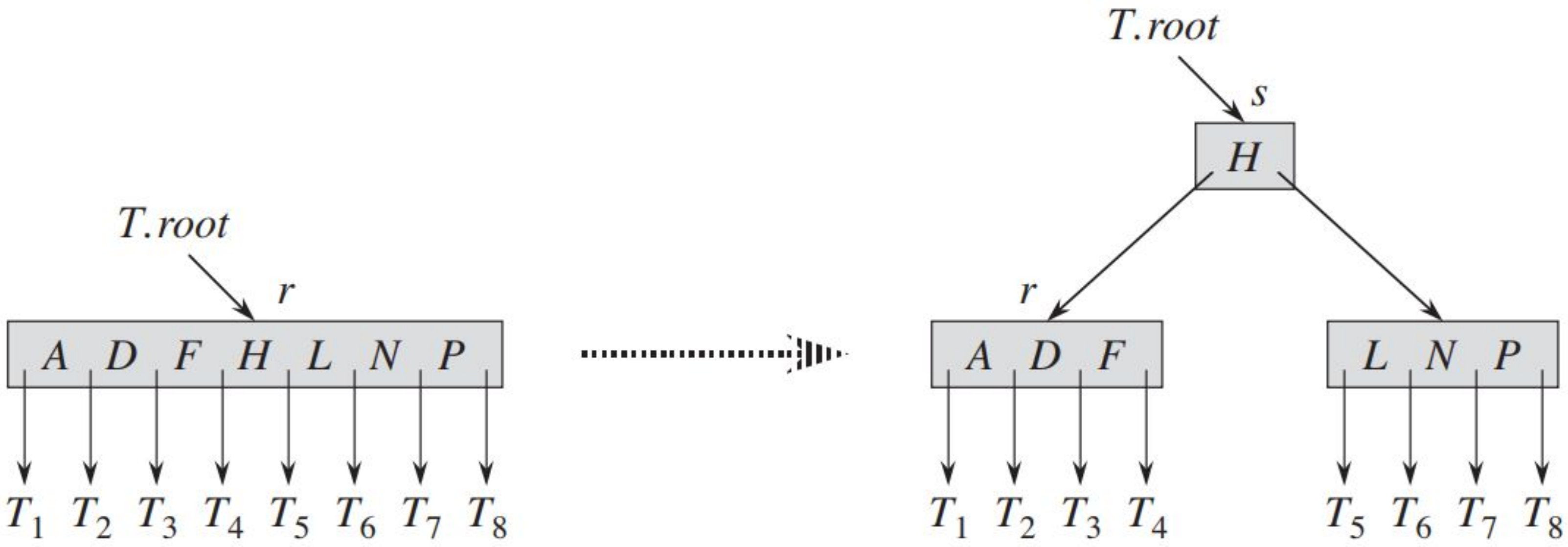
# В-дерево: вставка ключа

Как разбить полный корневой узел:

- Создаем новый пустой корневой узел и цепляем к нему потомком старый заполненный корневой узел.
- Вызываем B-TREE-SPLIT-CHILD на новом корневом узле. Старый корень распадется на 2 узла, они подцепятся к новому корню, бывшая медиана старого корня переместится в новый корень.
- В результате высота дерева увеличится на 1.
- Разбиение узлов – единственный способ увеличить высоту В-дерева.

# B-дерево: вставка ключа

Пример вызова B-TREE-SPLIT-CHILD для полностью заполненного корня (t = 4):
В отличие от бинарного дерева поиска, B-дерево растет вверх, а не вниз.



# B-дерево: вставка ключа

B-TREE-INSERT(T, k) добавляет ключ k в B-дерево T.

Фактически вставка производится в B-TREE-INSERT-NONFULL.

```
B-TREE-INSERT(T, k)
  r = T.root
  if r.n == 2t - 1 // если корень заполнен
      s = ALLOCATE-NODE() // создаем новый узел
      T.root = s // делаем его корнем
      s.leaf = FALSE // у него будут потомки
      s.n = 0
      s.c1 = r // цепляем к нему старый корень
      B-TREE-SPLIT-CHILD(s, 1) // старый корень
      B-TREE-INSERT-NONFULL(s, k)
  else
      B-TREE-INSERT-NONFULL(r, k)
```

# B-дерево: вставка ключа

B-TREE-INSERT-NONFULL(X, k) добавляет ключ k в неполный (обязательно!) узел X.

```
B-TREE-INSERT-NONFULL(X, k)
  i = X.n
  if X.leaf // если X - лист, в этот узел запишем ключ k
      // ищем позицию вставки справа налево, сдвигая обойденные ключи
на 1 вправо
      while i ⩾ 1 and k < X.key_i
          X.key_i+1 = X.key_i
           i = i – 1
      // добавляем ключ на освободившееся место
      X.key_i+1 = k
      X.n = X.n + 1 // +1 к хранимым ключам
      DISK-WRITE(X)
  // конец if, продолжение на следующем слайде
```

# B-дерево: вставка ключа

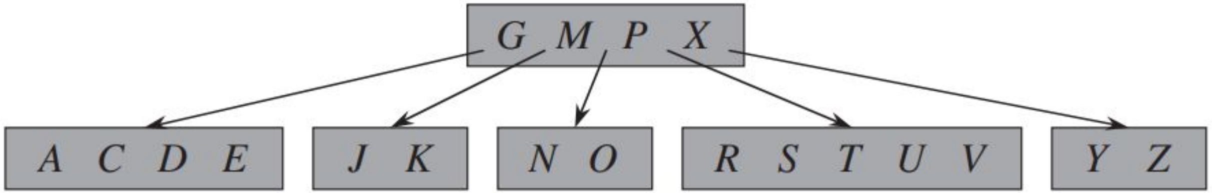
B-TREE-INSERT-NONFULL(X, k) добавляет ключ k в неполный (обязательно!) узел X.

```
B-TREE-INSERT-NONFULL(X, k)
  else // если же X - внутренний узел
      // надо найти потомка, куда будем писать
      while i ⩾ 1 and k < X.key_i
           i = i – 1
      i = i + 1
      DISK-READ(X.c_i)
      // если потомок полон, сначала разобъем его
      if X.c_i.n == 2t – 1
          B-TREE-SPLIT-CHILD(X, i)
          // в какого из новых потомков пойдем?
          if k > X.key_i
              i = i + 1
      B-TREE-INSERT-NONFULL(X.c_i, k)
```

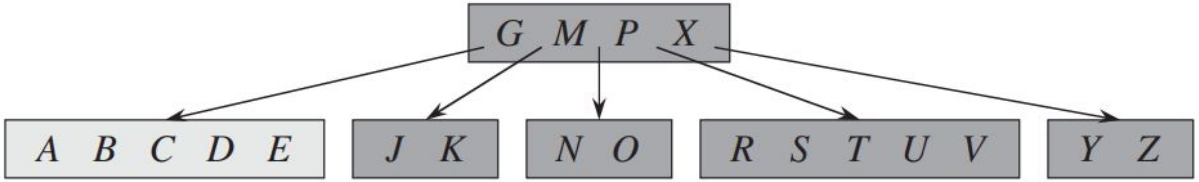
# B-дерево: вставка ключа

Примеры ситуаций при вставке.

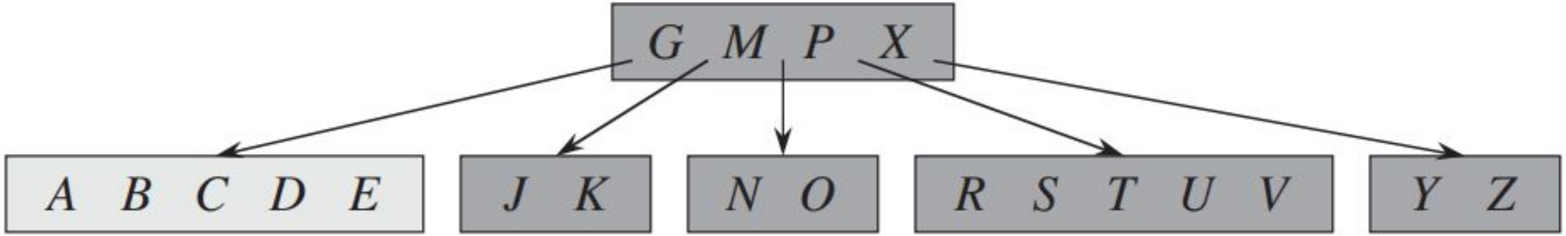
Исходное B-дерево, t = 3. Узел может содержать от 2-х до 5 ключей (корень должен содержать хотя бы 1).



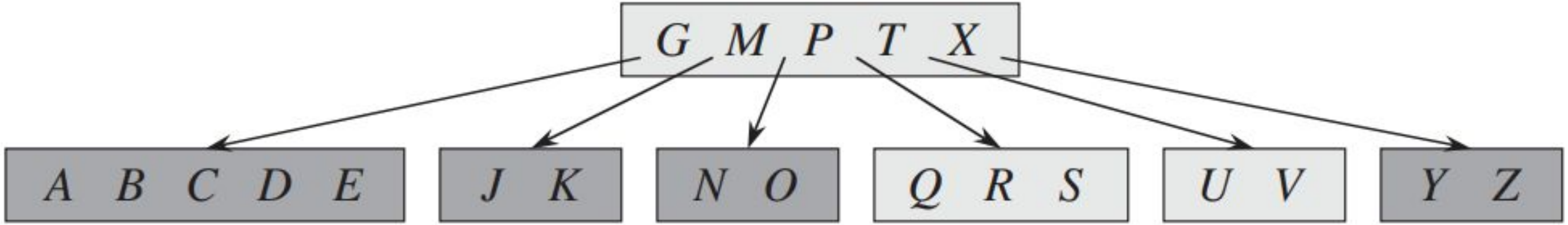

Добавляем ключ B, он просто записывается в лист.



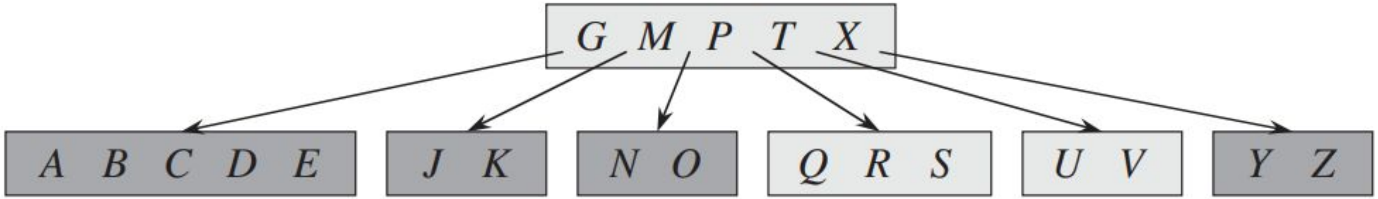
# B-дерево: вставка ключа



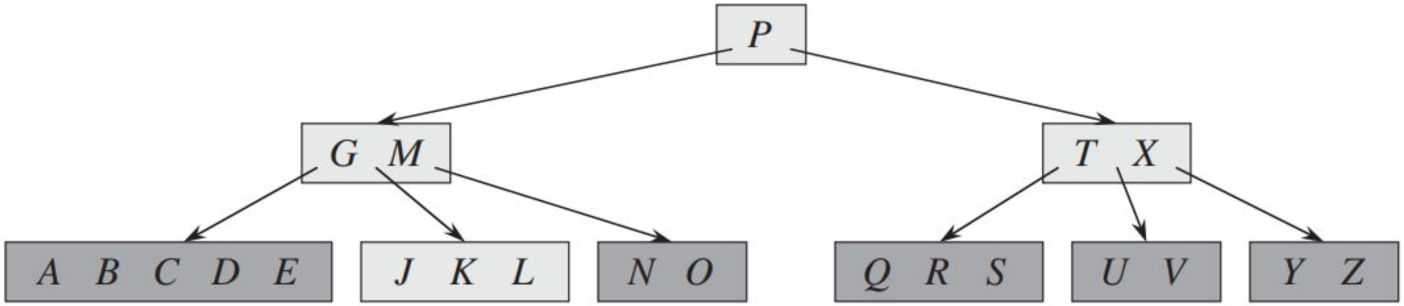

Добавляем ключ Q. Он должен попасть в лист RSTUV, но тот полон. Разбиваем его на RS и UV, T переносим в родительский узел, Q добавляем в левого потомка (RS).



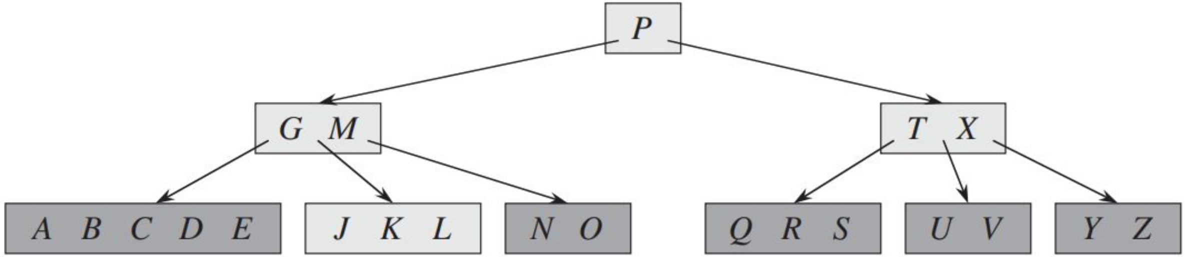
# В-дерево: вставка ключа



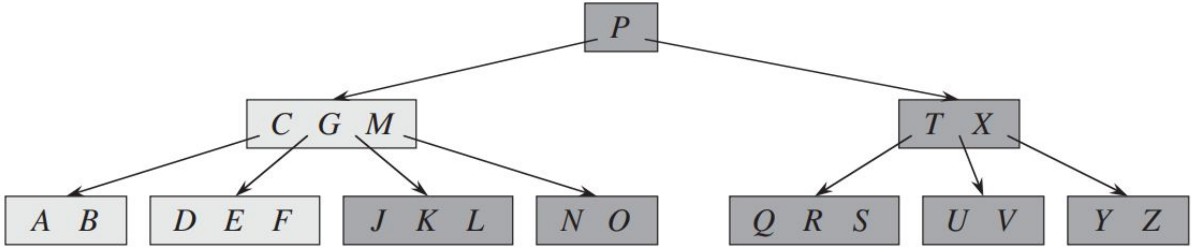

Добавляем ключ L. По пути обнаруживаем, что корень полон, надо разбивать. Получаем новый корень P с потомками GM и TX. Высота дерева увеличилась на 1. L будет записан в лист JK.



# B-дерево: вставка ключа




Добавляем ключ F. Лист ABCDE полон, разбиваем на AB и DE. Ключ C переносим к родителю. F попадет в узел DE.



# B-дерево: вставка ключа

В рамках вызова B-TREE-INSERT-NONFULL производится O(1) вызовов DISK-READ и DISK-WRITE, значит вызов B-TREE-INSERT выполняет $O(h) = O(\log_t N)$ дисковых операций.

Общая сложность алгоритма вставки ключа $O(t * h) = O(t * \log_t N)$.

# B-дерево: удаление ключа

Удаление сложнее вставки – удалять можно из произвольного узла, не только листа.

При вставке мы могли переполнить узел – превысить лимит на $2t - 1$ ключ.

При удалении может произойти обратное – в узле станет менее $t - 1$ ключей.

При проходе по дереву в поиске удаляемого ключа будем принимать меры, чтобы у потомка, в который переходим, было по крайней мере $t$ ключей (на 1 больше минимума).

# B-дерево: удаление ключа

B-TREE-DELETE(X, k) удаляет ключ k из поддерева с корнем в узле X.

Псевдокод слишком громоздкий, чтобы привести его на слайдах, поэтому просто рассмотрим возможные случаи. С псевдокодом можно ознакомиться по ссылке: https://sites.math.rutgers.edu/~ajl213/CLRS/Ch18.pdf (страницы 9 и 10)

# B-дерево: удаление ключа

1. Ключ k нашелся в узле X, X – лист.
   Удаляем k из X.

2. Ключ k нашелся в узле X, X – внутренний узел.
   a) Если потомок Y, предшествующий k в узле X, содержит не менее t ключей.
      Для ключа k находим в узле Y предшественника -- k′. Рекурсивно удаляем k′ из Y, а в X вместо k ставим k′.
   b) Если в потомке Y только t - 1 ключ, то пытаемся произвести симметричные действия с потомком Z, следующим за k в узле X. Если в Z хотя бы t ключей, тогда для ключа k находим в узле Z последующий ключ -- k′. Рекурсивно удаляем k′ из Z, а в X вместо k ставим k′.
   c) Если оба потомка Y и Z хранят всего по t – 1 ключей каждый.
      Сливаем ключ k и весь узел Z в Y. Из X уходят ключ k и указатель на Z. Узел Y теперь содержит 2t - 1 ключ. Освобождаем Z, рекурсивно удаляем k из Y.

# B-дерево: удаление ключа

3. Ключ k не нашелся во внутреннем узле X. Найдем корень $X.c_i$ поддерева, в котором продолжим поиски k. Если в $X.c_i$ только t - 1 ключ, выполним шаг 3a или 3b, чтобы в $X.c_i$ стало t ключей. Делаем B-TREE-DELETE($X.c_i$, k).

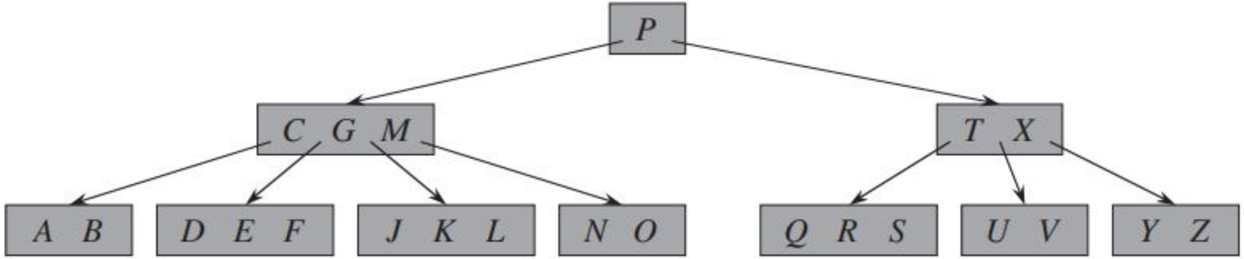
   a) Если в $X.c_i$ только t - 1 ключ, но у ближайшего соседа справа или слева по крайней мере t ключей.
   Возьмем из X ключ «между $X.c_i$ и соседом $X.c_i$» и спустим его в $X.c_i$, поднимем ключ из соседа $X.c_i$ в X, на место перенесенного. Также перенесем соответствующий указатель на потомка из соседа $X.c_i$ в $X.c_i$. Не забудем поправить указатель на потомка в $X.c_i$.

   b) Если и в $X.c_i$ только t - 1 ключ, и у обоих ближайших соседей справа и слева по t - 1 ключу.
   Возьмем одного из соседей, ключ из X, разделяющий соседа и $X.c_i$, и сольем их в новый узел. В X теперь на 1 ключ и на 1 ссылку меньше.

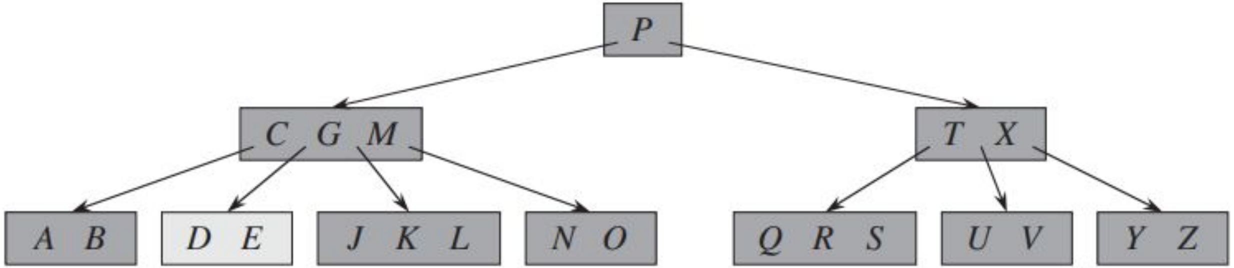
# B-дерево: удаление ключа

Примеры ситуаций при удалении.

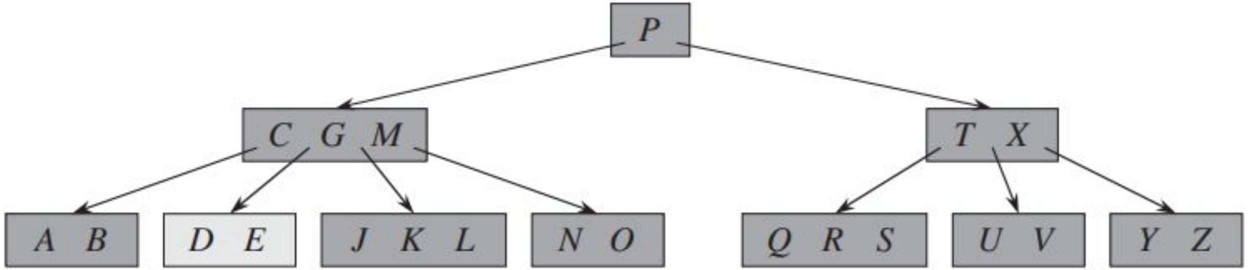
Исходное B-дерево, t = 3. Узел может содержать от 2-х до 5 ключей (корень должен содержать хотя бы 1).



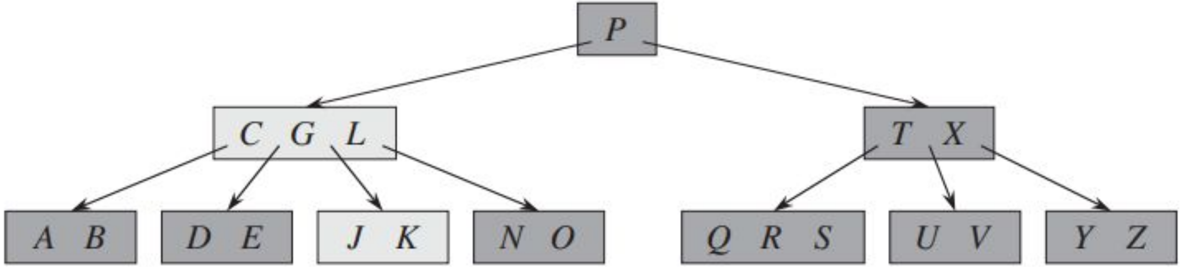

Удаляем ключ F. Тут простое удаление из листа. Случай 1.



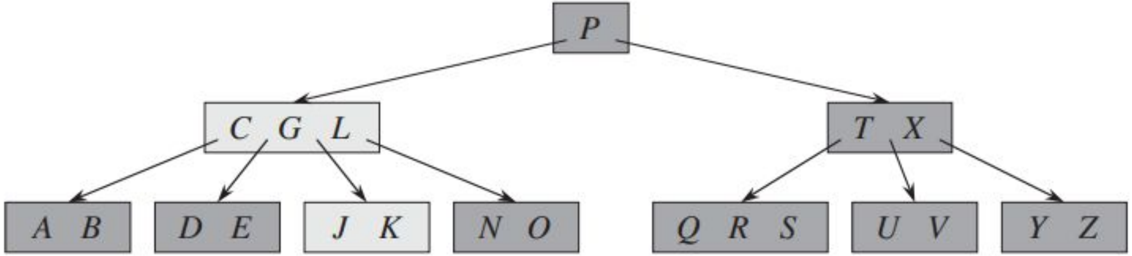
# B-дерево: удаление ключа



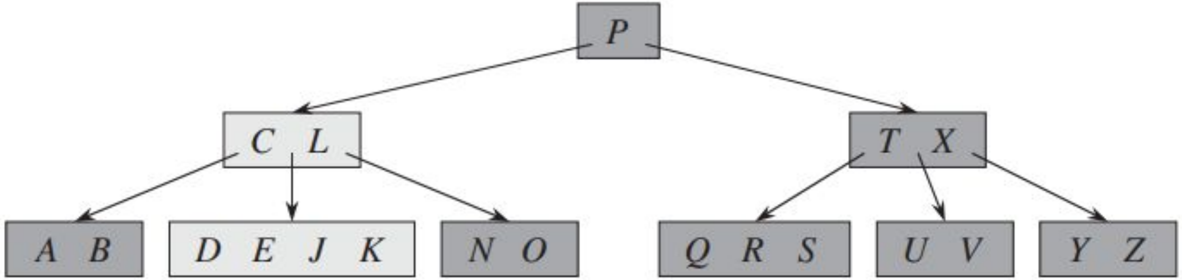

Удаляем ключ M. Случай 2a. Вместо M подставим предшественника из левого потомка – L.



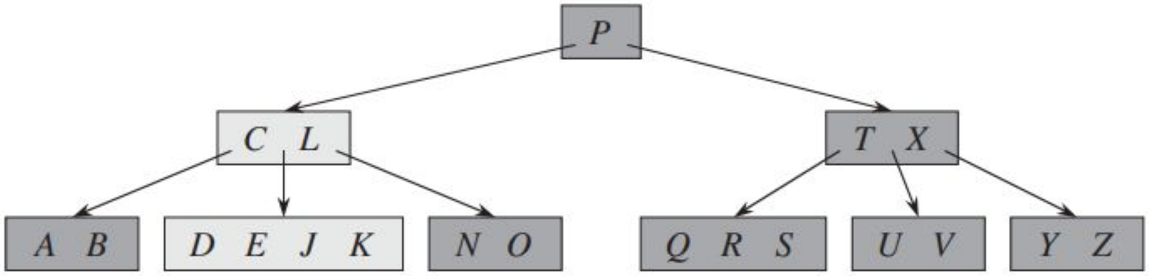
# B-дерево: удаление ключа



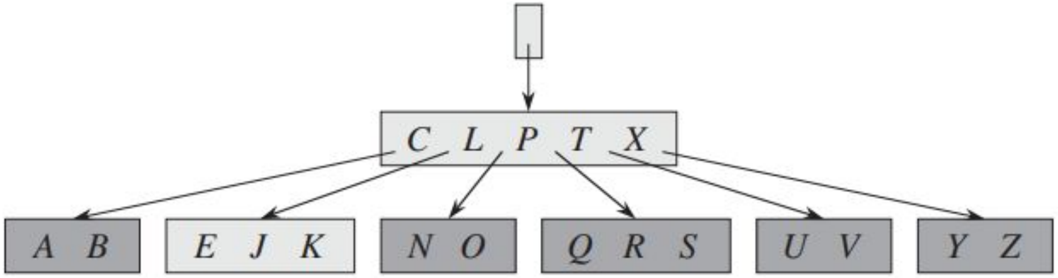

Удаляем ключ G. Случай 2c. Объединяем ключ G и узлы DE и JK в один узел DEGJK. Из CGL убрали 1 ключ и 1 указатель. Рекурсивно удаляем G из нового узла (1-й случай).



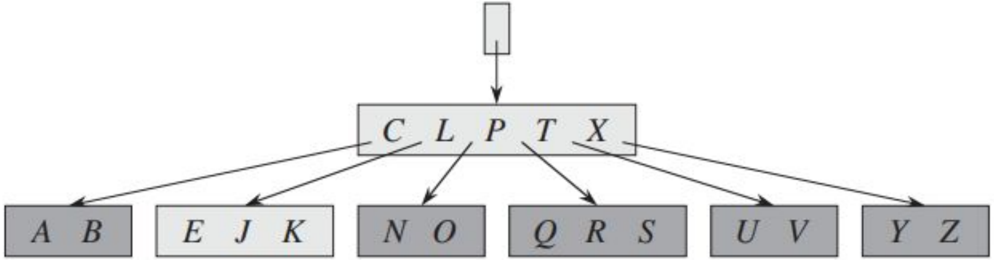
# B-дерево: удаление ключа



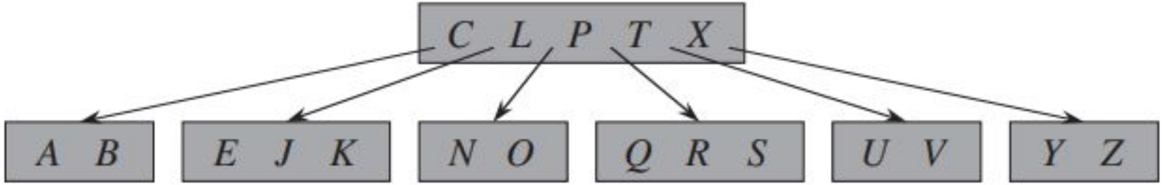

Удаляем ключ D. Случай 3b. Мы не можем просто спуститься в CL – он имеет минимальное допустимое количество ключей. Сливаем CL, P и TX в новый узел.
Затем удаляем D из листа (случай 1).



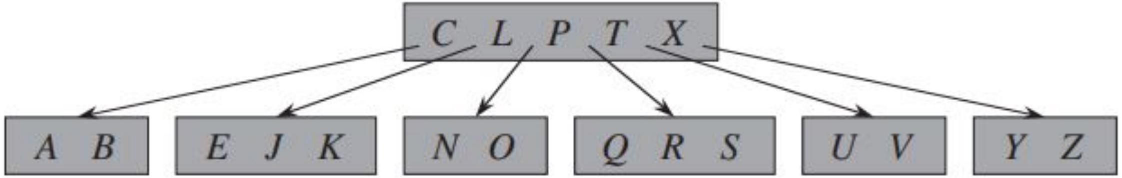
# B-дерево: удаление ключа



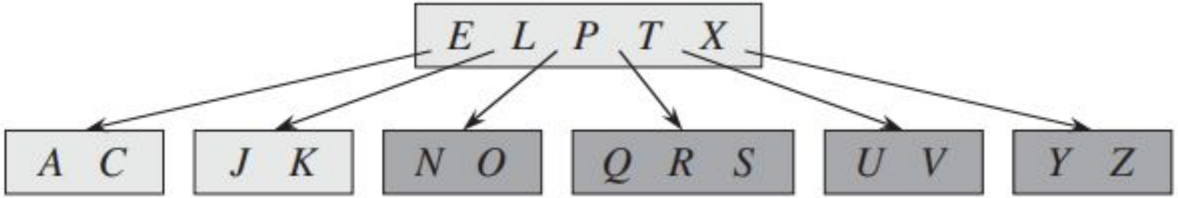

Завершение случая с прошлого слайда. Удаляем пустой корень, теперь корнем стал узел CLPTX. Высота дерева уменьшилась на 1.



# B-дерево: удаление ключа




Удаляем B. Случай 3а. C переносим на место B, E переносим на место C.



# B-дерево: удаление ключа

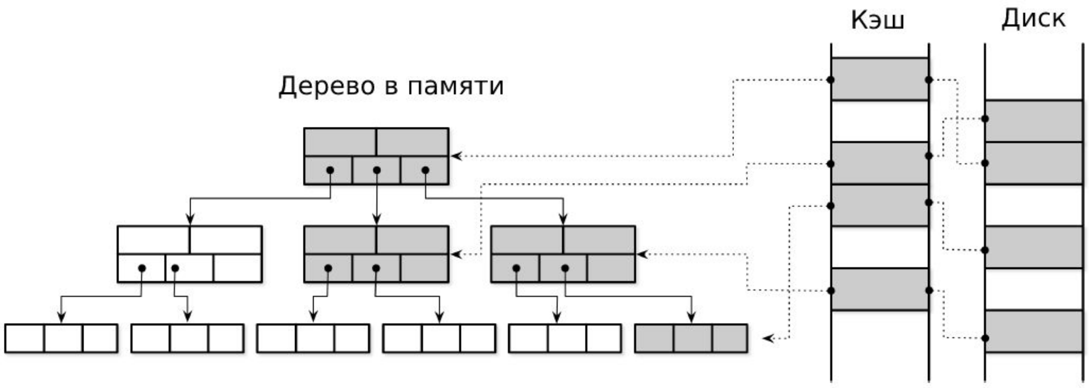
В рамках вызовов B-TREE-DELETE производится O(h) вызовов DISK-READ и DISK-WRITE, значит происходит $O(h) = O(\log_t N)$ дисковых операций.

Общая сложность алгоритма удаления ключа $O(t * h) = O(t * \log_t N)$.

# B-дерево: дисковый кэш

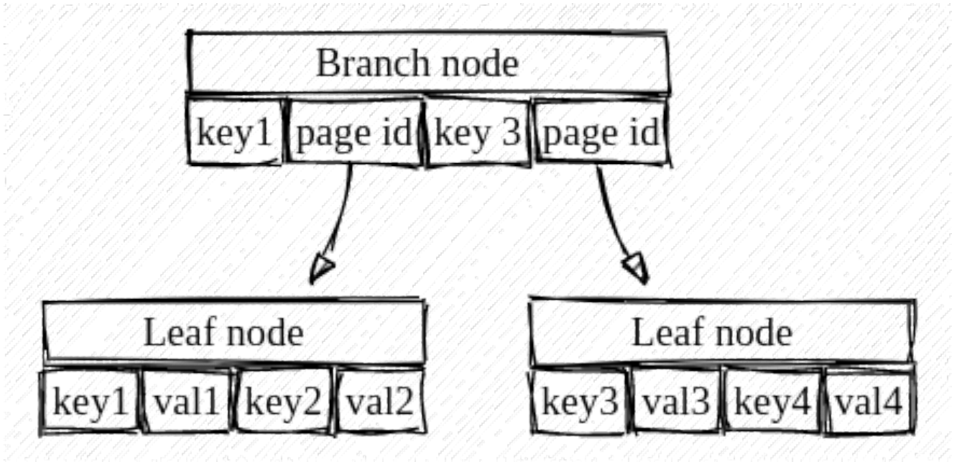
**Назначение дискового кэша:**

- Хранит содержимое страницы (узла дерева) в RAM.
- Если запрошенной страницы нет в RAM, и там для нее есть место, загружает ее содержимое с диска. Если места в RAM нет, вытесняет одну из загруженных страниц, при этом записывает ее актуальную версию на диск.
- Если запрошенная страница уже загружена в RAM, возвращает закэшированную версию.
- Буферизует запись в страницы.



# $B^+$-дерево

$B^+$-дерево – разновидность B-дерева, в котором ассоциированные с ключами значения хранятся только в листьях.
Внутренние узлы хранят лишь копии ключей-разделителей и ссылки на потомков.
Обычно $B^+$-деревья также хранят в листьях ссылки на соседние листья.



# $B^+$-дерево и B-дерево

Преимущества $B^+$-дерева:

- Во внутренних узлах не храним значения, поэтому в них помещается больше ключей. Это позволяет увеличить коэффициент ветвления дерева, уменьшить его высоту.

- Если листья связаны ссылками на соседей, удобно делать обход диапазонов ключей.

Недостатки:

- Внутренние узлы не могут содержать значений, поэтому всегда придется спускаться до листьев. В обычном B-дереве путь до значения может быть короче.

# $B^+$-дерево: использование

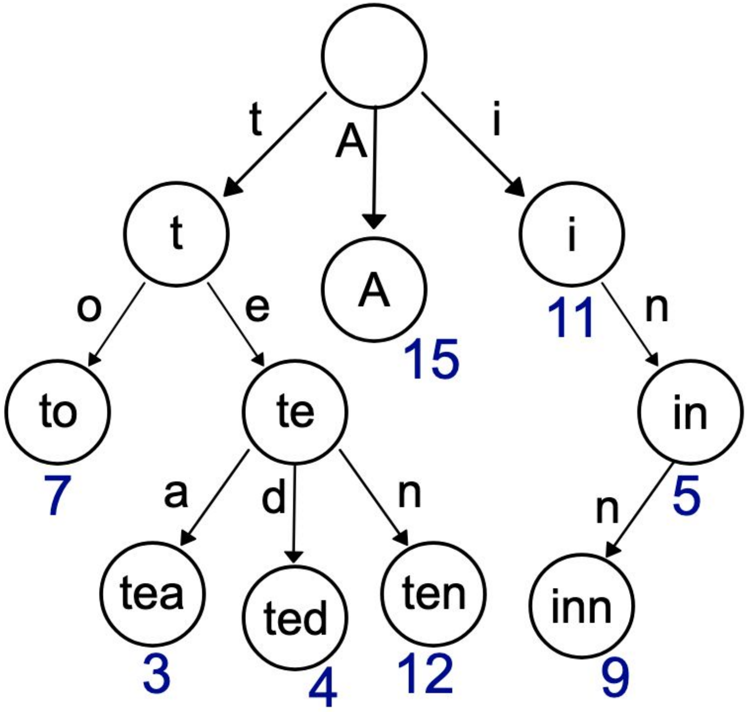
- В файловых системах. Например, NTFS, APFS, ext4 (использует extent trees -- модифицированные $B^+$-деревья).

- В реляционных БД. Например, Microsoft SQL Server, Oracle (с 8-й версии), SQlite.

- No-SQL БД. Например, CouchDB, MongoDB (при использовании подсистемы хранения WiredTiger), Tokyo Cabinet.

# Trie

# Trie



# Trie

Префиксное дерево содержит данную строку-ключ тогда и только тогда, когда эту строку можно прочитать на пути из корня до некоторого (единственного для этой строки) выделенного узла.
В отличие от бинарных деревьев поиска, ключ, идентифицирующий конкретный узел дерева, не явно хранится в данном узле, а задаётся положением данного узла в дереве. Получить ключ можно выписыванием подряд символов, помечающих рёбра на пути от корня до узла.

```
structure Node
    Children Node[Alphabet-Size]
    Is-Terminal Boolean
    Value Data-Type
end structure
```

```
Trie-Insert(x, key, value)
    for 0 ⩽ i < key.length do
        if x.Children[key[i]] = nil then
            x.Children[key[i]] := Node()
        end if
        x := x.Children[key[i]]
    repeat
    x.Value := value
    x.Is-Terminal := True
```

# Trie

**Расход памяти и время работы**

| | В среднем случае | В худшем случае |
|---|---|---|
| Расход памяти | O(n) | O(n) |
| Поиск | O(n) | O(n) |
| Вставка | O(n) | O(n) |
| Удаление | O(n) | O(n) |

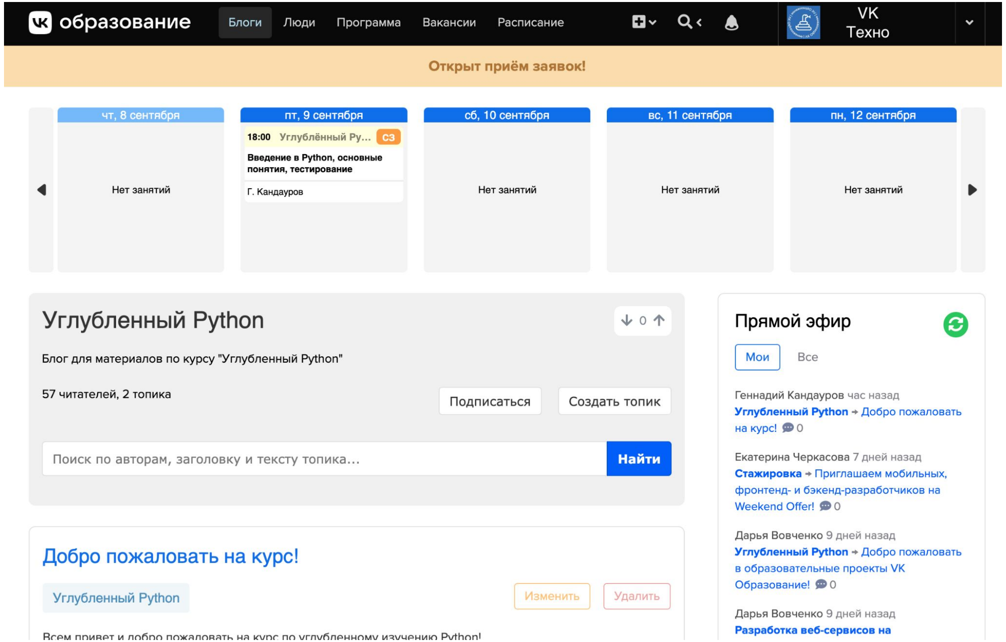
# Визуализация

- https://www.cs.usfca.edu/~galles/visualization/RedBlack.html
- https://www.cs.usfca.edu/~galles/visualization/AVLtree.html
- https://www.cs.usfca.edu/~galles/visualization/BTree.html

# Домашнее задание #06

# Напоминание отметиться на портале Vol 2

+ оставить отзыв после лекции



# Спасибо за внимание

education