Разработка и анализ алгоритмов

Лекция 11
Сбалансированные деревья

Сергей Леонидович Бабичев

Полезные операции над деревьями

Какие операции сейчас имеются

Сейчас имеются следующие операции:

- $insert(key)$ — вставить ключ как терминальную вершину. $T = O(H)$.
- $rotateLeft(node)$ — повернуть в узле $node$ дерево налево, вернуть новый корень. $T = O(1)$.
- $rotateRight(node)$ — повернуть в узле $node$ дерево направо, вернуть новый корень. $T = O(1)$.
- $insertToRoot(key)$ — вставить ключ в корень, при подъёме вращая дерево по направлению к поднимающемуся узлу. $T = O(H)$.
- $find(key)$ — найти узел, содержащий ключ key. $T = O(H)$.
- $erase(key)$ — удалить узел, содержащий key. $T = O(H)$.

Что можно добавить?

Добавим операцию

- $moveToRoot(node)$ — переместить существующий узел в корень вращая дерево при подъёме. $T = O(H)$.

Какие операции появились:

- $split(key)$ — разбить дерево на два, T_1, содержащее все узлы, не большие key и T_2 — все узлы большие key.
  Для этого поднимем key в корень и отправим левую часть с корнем в T_1, правого ребёнка (если есть) — в T_2.
- $merge(T_1, T_2)$ — слить два дерева T_1, все элементы которого меньше любого из T_2 в одно. $T = O(H)$.
  Для этого поднимем наибольший элемент дерева T_1 в корень. Все элементы T_1, кроме наибольшего, окажутся в левом поддереве. Правым поддеревом сделаем T_2. $T = O(H)$.

Что можно добавить?

Добавим ещё операцию слияния произвольных деревьев T_1 и T_2 за линейное время.

1. Если одно из деревьев пусто — возвращаем второе.
2. Иначе произвольно выбираем корень из двух корней. Пусть это T_1.
3. Вставляем корень T_1 вставкой в корень в T_2. Это даст нам два поддерева, ключи которых меньше этого корня и два поддерева, ключи которого больше.
4. Рекурсивно вставляем левые поддеревья T_1 и T_2 в левую часть нового, правые — в правую часть нового.
5. Убедитесь сами, что сложность алгоритма линейна.

Слияние деревьев

```
node *join(node* t1, node *t2) {
   if (t1 == NULL) return t2;
   if (t2 == NULL) return t1;
   t2 = insertToToot(t2, t1->item);
   t2->left = join(t1->left, t2->left);
   t2->right = join(t2->right, t2->right);
   // destroy a
   return t2;
}
```

Какие полезные операции появились

- $split(key)$ — разбить дерево на два, T_1, содержащее все узлы, не большие key и T_2 — все узлы большие key.
  Для этого поднимем key в корень и отправим левую часть с корнем в T_1, правого ребёнка (если есть) — в T_2.
- $merge(T_1, T_2)$ — слить два дерева T_1, все элементы которого меньше любого из T_2 в одно. $T = O(H)$.
  Для этого поднимем наибольший элемент дерева T_1 в корень. Все элементы T_1, кроме наибольшего, окажутся в левом поддереве. Правым поддеревом сделаем T_2. $T = O(H)$.

Что нам это даёт

Теперь можно проводить операции $insert$ и $erase$, используя только $split$ и $merge$.

- $insert(key)$: разобьём дерево T на два по ключу key на T_1 и T_2, сделаем key вершиной, левым ребёнком сделаем T_1, правым — T_2. $T = O(H)$.
- $erase(key)$: поднимем key до корня, сольём T_{left} и T_{right}. $T = O(H)$.

Добавим ещё поле

К полями $left$, $right$ и $parent$ добавим поле $count$, которое будет содержать количество узлов во всех поддеревьях.
Тогда появляется операция

- $select(k)$ — дать $k-$ю порядковую статистику дерева. Спускаемся от корня дерева, переходя налево или направо в зависимости от суммарного количества детей в соответствующем поддереве. $T = O(H)$.

Дерево по нявному ключу

Вместо ключа можно использовать поле $count$, сохранив поле $data$. Тогда:

- $insertAfter(k, data)$ — вставить данные $data$ после номера k. Находим по $select(k)$ нужный узел и вставляем после него. $T = O(H)$.
- $insertBefore(k, data)$ — вставить данные $data$ после номера k. Находим по $select(k)$ нужный узел и вставляем перед ним. $T = O(H)$.

Теперь у нас есть массив данных, в который мы можем добавлять элементы в любое место, «раздвигая» элементы массива, удалять из любого места, «сдвигая» элементы массива.

Имеющиеся «внешние» операции

- $insert(key, data)$ — вставить ключ по значению. $T = O(H)$.
- $find(key)$ — найти узел, содержащий ключ key. $T = O(H)$.
- $erase(key)$ — удалить узел, содержащий key. $T = O(H)$.
- $select(k)$ — дать $k-$ю порядковую статистику дерева. $T = O(H)$.
- $insertAfter(k, data)$ — вставить данные $data$ после номера k. $T = O(H)$.
- $insertBefore(k, data)$ — вставить данные $data$ после номера k. $T = O(H)$.
- $enumerate()$ — перечислить все элементы в заданном порядке ключей. $T = O(N_{nodes})$.

У нас есть почти идеальная структура данных, где $H = O(\log N_{nodes})$ — декартово дерево. Тогда все операции проводятся за $O(\log N)$.
Хорошо подходит и более простая структура данных — splay-дерево.

Splay-деревья

- Алгоритм «Вставка в корень» перемещал только что вставленную вершину в корень дерева поиска.
- Мы уже убеждались, что вставка только в корень не приводит к избавлению от «бамбука» при вставке упорядоченных значений.
- Попробуем эту операцию совместить с переформировкой структуры дерева.
- Структура данных изобретена Слеатором и Тарджаном.

Свойства splay-дерева

- Каждый раз при поиске элемента x или вставке элемента x он перемещается в корень.
- Следующая операция с x производится быстро.
- ST не поддерживает инвариантов. Структура ST может быть произвольной.
- Каждая операция, даже поиска, изменяет форму.
- Узел отправляется наверх серией одиночных или двойных поворотов.
- Одиночный поворот поднимает узел на уровень вверх, двойной — на два уровня.
- Эти повороты происходят до тех пор, пока узел x не окажется в корне. Этот процесс называется $splaying$. Кроме подъёма вверх $splaying$ сокращает высоту дерева, делая его более сбалансированным.
- Имеется два варианта одиночных и четыре варианта двойных поворотов.
- В некоторой литературе их называют комбинацией слов zig (правый) и zag (левый).

- Будем обозначать за $p(x)$ родителя, а за $g(x)$ — родителя родителя.
- Если x левый ребёнок и $g(x)$ не существует, то возможен только простой одиночный правый поворот.
- Если $\mathtt{x}$ левый ребёнок и x не имеет дедушку, то возможен только простой одиночный левый поворот.
- $zag - zag$ — узел x находится справа-справа от $g(x)$.
- $zag - zag$ — узел x находится слева-слева от $g(x)$.
- $zig - zag$ — узел x находится справа-слева от $g(x)$.
- $zag - zig$ — узел x находится слева-справа от $g(x)$.

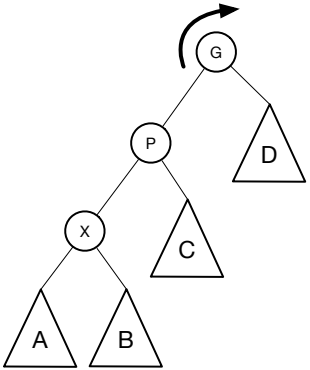
Операция $splay$

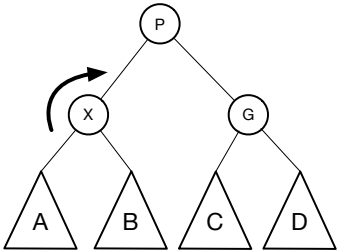
- Введём операцию $splay$, которая поднимает узел x в корень.
- Она состоит из нескольких подъёмов. Всё зависит от того, откуда x поднимается. Каждый пункт возвращает алгоритм к пункту 1. Итак:
  1. Если $p(x)$ не существует, то алгоритм завершён.
  2. Если $g(x)$ не существует:
     2.1 Если x — левый сын ($x = left(p(x))$), то вращаем родителя вправо.
     2.2 Если x — правый сын ($x = right(p(x))$), то вращаем родителя влево.
  3. $g(x)$ существует.
     3.1 Если ориентация x и $xp(x)$ а так же $p(x)$ и $g(x)$ одинаковая (или оба слева, или оба справа), то вначале поворачиваем $g(x)$, затем $p(x)$ ($zag - zag$ или $zig - zig$).
     3.2 Иначе сначала поворачиваем $p(x)$, потом $g(x)$ ($zig - zag$ или $zag - zig$).

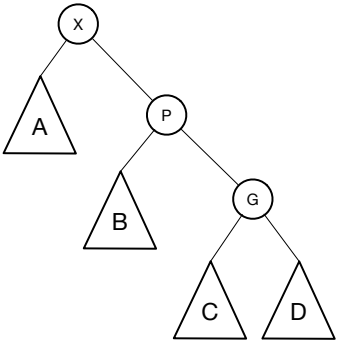
Псевдокод операции splay

```
while (x->p != NULL) {
    if (x->p->p == NULL) {
        if (x == x->p->left) x->p = rotateRight(x->p);
        else                 x->p = rotateLeft(x->p);
    } else {
        if (x == x->p->left && x->p == x->p->p->left) {
            x->p->p = rotateRight(x->p->p);
            x->p = rotateRight(x->p);
        } else if (x == x->p->right && x->p == x->p->p->right) {
            x->p->p = rotateLeft(x->p->p);
            x->p = rotateLeft(x->p);
        } else if (x == x->p->right && x->p == x->p->p->left) {
            x->p = rotateLeft(x->p);
            x->p = rotateRight(x->p);
        } else { // x == x->p->right && x->p == x->p->p->left
            x->p = rotateRight(x->p);
            x->p = rotateLeft(x->p);
        }
    }
}
```

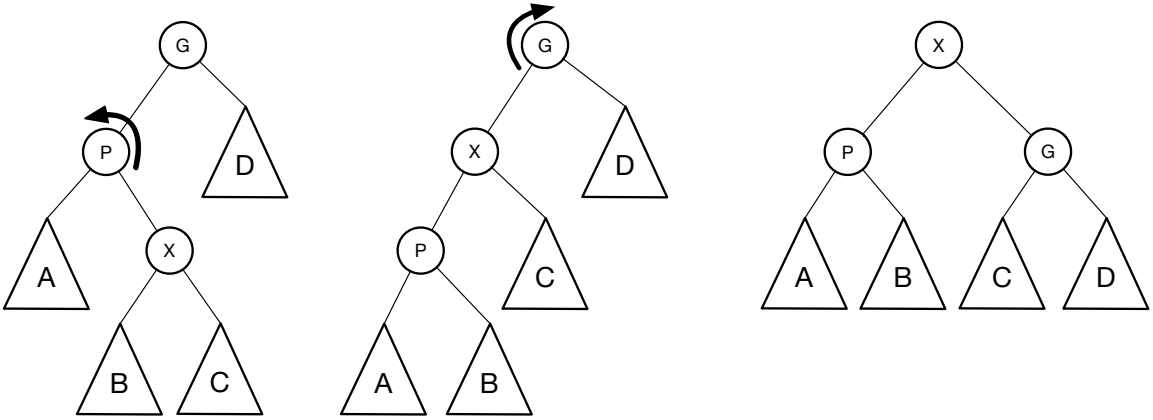
Операция $zag - zag$







Операция $zag - zig$



Почему это работает

Для определения амортизированной сложности используем метод потенциалов. Пусть первоначальное состояние дерева было T, после операции t стало T'. Тогда амортизированная стоимость операции $cost(a)$ есть реальная стоимость $cost(t)$ с добавлением изменения состояния от T до T'.

$$a = t + \Phi(T') - \Phi(T). \tag{1}$$

Если мы проделаем n операций, то:

$$\sum_n a = \sum_n t + \Phi(T_2) - \Phi(T_1) + \Phi(T_3) - \Phi(T_2) + \cdots + \Phi(T_n) - \Phi(T_{n-1},) \tag{2}$$

что даёт

$$\sum_n a = \sum_n t + \Phi(T_n) - \Phi(T_1) \tag{3}$$

Подсчёт сложности

Потенциальную функцию выбирают, чтобы разность потенциалов была всегда положительной, тогда амортизированная стоимость будет верхней границей реальной стоимости.
Пусть для узла x функция $size(x)$ есть количество подузлов x во всём поддереве, а $rank(x) = \log_2 size(x)$. Тогда определим потенциальную функцию как

$$\Phi(T) = \sum_{x \in T} rank(x). \tag{4}$$

Максимальный потенциал дерева $n \log n$.

Одиночные операции

$$\Delta\Phi = rank(p') - rank(p) + rank(x') - rank(x).$$

Так как $rank(x') = rank(p)$, $rank(p') \leqslant rank(p)$, $rank(x') \geqslant rank(x)$, то $\Delta\Phi = rank(p') - rank(x) \leqslant rank(x') - rank(x)$.

$$a_{zig} \leqslant 1 + rank(x') - rank(x) \leqslant 1 + 3(rank(x') - rank(x)) \tag{5}$$

Односторонние операции

$$a_{zigzig} = 2 + rank(g') - rank(g) + rank(p') - rank(p) - rank(x') - rank(x).$$

Заметим, что $size(x') \geqslant size(x) + size(g')$

$$a_{zigzig} = 2 + rank(g') + rank(p') - rank(p) - rank(x).$$

Так как $rank(x) < rank(p)$ и $rank(x') > rank(p')$

$$a_{zigzig} = 2 + rank(g') + rank(x') - 2rank(x).$$

Докажем, что $2rank(x') - rank(x) - rank(g') \geqslant 2$.

$$2rank(x') - rank(x) - rank(g') = (rank(x') - rank(x)) + (rank(g') - rank(x') = \log_2\left(\frac{size(x')}{size(x)}\right) + \log_2\left(\frac{size(x')}{size(g')}\right) \geqslant \log_2\left(\frac{2size(x)}{size(x)}\right) + \log_2\left(\frac{2size(g')}{size(g')}\right)$$

Итого: $a_{zigzig} \leqslant 2rank(x') - rank(x) - rank(g') + rank(g') + rank(x') - 2rank(x)$
$a_{zigzig} \leqslant 2rank(x') - rank(x) - rank(g') + rank(g') + rank(x') - 2rank(x)$

$$a_{zigzig} \leqslant 3(rank(x') - rank(x))$$

Окончательный баланс

$a_{total} = a_1 + a_2 + \cdots + a_n = 3(rank(T_n) - rank(T_1))$.
Потенциал максимален, когда узел поднимается от терминального, тогда оценка $3 \log n$. В остальных случаях — меньше.

Сбалансированные деревья поиска

Сбалансированные деревья поиска

- Задача: реализовать операции с деревьями, имеющие время в худшем $\Theta(\log N)$.

$$H < A \cdot \log N + B,$$

  где A и B — некоторые фиксированные константы.
- Решение: использовать сбалансированные деревья и не нарушающие сбалансированность.

Сбалансированные деревья поиска: критерии сбалансированности

Высота дерева H_t не превосходит $A \log N + B$, если в бинарном дереве с N узлами выполнено хотя бы одно из условий:

1. для любого узла количество узлов в левом и правом поддереве N_l, N_r отличаются не более, чем на 1

$$N_r \leqslant N_l + 1, \quad N_l \leqslant N_r + 1$$

2. для любого узла количество подузлов в левом и правом поддеревьях удовлетворяют условиям

$$N_r \leqslant 2N_l + 1, \quad N_l \leqslant 2N_r + 1$$

3. для любого узла высоты левого и правого поддеревьев H_l, H_r удовлетворяют условиям

$$H_r \leqslant H_l + 1, H_l \leqslant H_r + 1$$

Сбалансированные деревья поиска

Случай 1. Идеально сбалансированное дерево.
Пусть $H_{ideal}(N)$ — максимальная высота идеально сбалансированного дерева.

- N — нечётно и равно $2M + 1$. Тогда левое и правое поддеревья должны содержать ровно по M вершин.

$$H_{ideal}(2M + 1) = 1 + H_{ideal}(M)$$

- N — чётно и равно $2M$. Тогда

$$H_{ideal}(2M) = 1 + \max(H_{ideal}(M - 1), H_{ideal}(M))$$

Так как $H_{ideal}(M)$ — неубывающая функция, то

$$H_{ideal}(2M) = 1 + H_{ideal}(M)$$

$$H_{ideal}(N) \leqslant \log_2 N$$

Сбалансированные деревья поиска

Случай 2. Примерная сбалансированность количества узлов.
Пусть $H(M)$ — максимальная высота сбалансированного дерева со свойством 2.

- Тогда $H(1) = 0, H(2) = H(3) = 1$.
- При добавлении узла один из узлов будет корнем, остальные $N - 1$ распределятся в отношении $N_l : N_r$, где $N_l + N_r = N - 1$.
- Не умаляя общности, предположим, что $N_r \geqslant N_l$, тогда $N_r \leqslant 2N_l + 1$.

$$H(N) = \max_{N_l, N_r}(1 + \max(H(N_l), H(N_r)))$$

Сбалансированные деревья поиска

Функция $H(N)$ — неубывающая, поэтому

$$H(N) = 1 + H(\max(N_r, N_l))$$

При ограничениях $N_r \leqslant 2N_l + 1$ и $N_l + N_r = N + 1$ получаем

$$H(N) = 1 + H\left(\left\lfloor \frac{2N-1}{3} \right\rfloor\right)$$

$$H(N) > 1 + H\left(\left\lfloor \frac{2N}{3} \right\rfloor\right)$$

$$H(N) > \log_{3/2} N + 1 \approx 1.71 \log_2 N + 1$$

Сбалансированные деревья поиска

Случай 3. Примерная сбалансированность высот. АВЛ-деревья.
Пусть $N(H)$ — минимальное число узлов в АВЛ-дереве с высотой H (минимальное АВЛ-дерево).

- Пусть левое дерево имеет высоту $H-1$.
- Правое дерево будет иметь высоту $H-1$ или $H-2$.
- $N(H)$ — неубывающая, для минимального АВЛ-дерева высота правого равна $H-2$.
- Число узлов в минимальном АВЛ-дереве:

$$N(H) = 1 + N(H-1) + N(H-2)$$

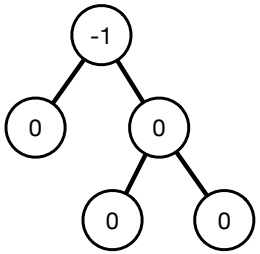
$$\lim_{h\to\infty} \frac{N(h+1)}{N(h)} = \varphi = \frac{\sqrt{5}+1}{2}$$

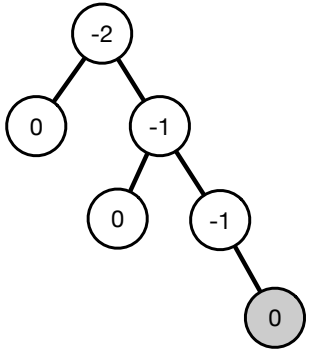
$$H(N) \approx \log_\varphi(N-1) + 1 \approx 1.44 \log_2 N + 1$$

AVL-деревья

AVL-деревья

- Добавим к каждому узлу поле высоты поддерева, начинающемся в этом узле.
- Высота терминального узла будет равна 1.
- Тогда можно вычислить $balance$ — разницу между высотами поддеревьев.
- Инвариант: $-1 \leqslant balance \leqslant 1$.
- Баланс может измениться при операциях $insert$ и $erase$.



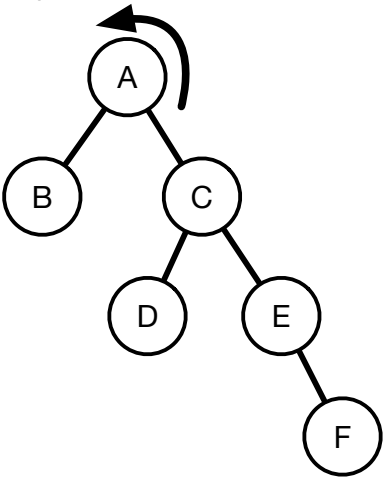


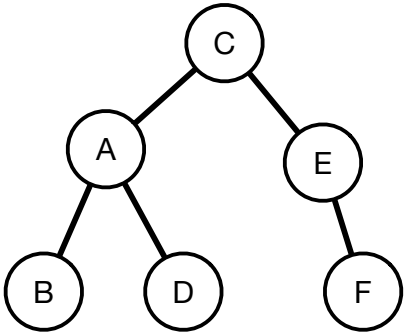
AVL: insert

- Вставка производится стандартно — в терминальный лист.
- После операции $insert$ может нарушиться баланс.
- Утверждение: баланс в предках вставленного узла не может выйти за границы $[-2, 2]$.
- Значение баланса -2 и 2 — дисбаланс.
- Дисбаланс лечится поворотами.
- Верно ли утверждение, что достаточно повернуть в дисбалансном узле корень в направлении от перекоса?

AVL: перебалансировка

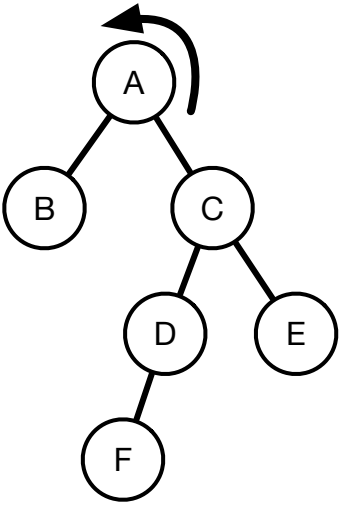
Верно?

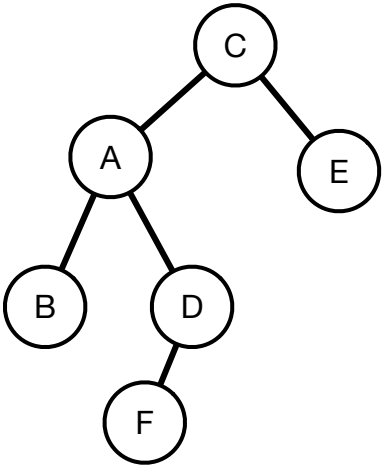




AVL: перебалансировка

Неверно!





AVL: правила перебалансировки

- Корректировка баланса проводится снизу вверх.
- При дисбалансе возможны как простые повороты, так и «большие».
- Правило: если при дисбаланс знак произведения балансов корня и проблемного ребёнка отрицателен, сначала надо развернуть ребёнка, и только после этого разворачивать корень.

AVL: правила перебалансировки

```
node *balance(node *n) {
  calcBalance(n);
  switch (delta(n)) {
  case -2:
    if (delta(n->left) > 0)
      n->left = rotateLeft(n->left);
    return rotateRight(n);
  case 2:
    if (delta(n->right) < 0)
      n->right = rotateRight(n->right);
    return rotateLeft(n);
  default:
    return n;
  }
}
```

B-деревья.

- В AVL деревьях высоты поддеревьев всё же различаются.
- Имеется ли структура данных, в которой все высоты поддеревьев во всех узлах гарантировано равны?
- Это, например, B-деревья.
- Их можно использовать и при обычном построении, но чаще — для работы с внешней памятью.

Внешний поиск с использованием B-деревьев

- Основной носитель информации — жёсткий или SSD «диск».
- Информация на жёстком диске располагается в *секторах*, которые логически расположены на *дорожках*.
- Размер сектора типично 512/2048/4096 байт.
- Информация считывается и записывается *головками чтения/записи*.
- Для чтения/записи информации требуется *подвести* головку чтения записи к нужной дорожке и дождаться подхода нужного сектора.
- Типичные скорости вращения жёстких дисков — 5400/7200/10033/15000 оборотов в минуту.
- Один оборот совершается за время от 1/90 до 1/250 секунды.
- Операция перехода на соседнюю дорожку примерно 1/1000 секунды.

Работа с внешними носителями

- Внешние сортировки используют многократный последовательный проход по данным, расположенным на носителях информации.
- Последовательное считывание информации с жёсткого диска 100-150 мибибайт в секунду.
- Смена позиции в файле часто требует:
  - ожидания подвода головки на нужную дорожку;
  - ожидания подхода нужного сектора к головкам чтения/записи;
- Операция последовательного чтения 4096 байт занимает $\frac{4096}{100 \times 10^6} \approx 40 \times 10^{-6}$ секунд
- Операция случайного чтения 4096 байт занимает не менее $5 - 10 \times 10^{-3}$ секунд.

Работа с внешними носителями

- Второй популярный носитель — SSD диск.
- Информация хранится в энергонезависимой памяти на микросхемах.
- Операции производятся блоками размером 64-1024 кибибайт.
- Время доступа к блоку $\approx 10^{-6}$ секунд.
- HDD и SSD используют *буферизацию* для ускорения работы.
- Алгоритмы поиска во внешней памяти должны минимизировать число обращений к внешней памяти и выровненные операции.

Работа с SSD носителями

- На логическом уровне обращения происходят блоками любого размера, кратного 512 байт.
- На физическом уровне всё сложнее.
- Размер физического блока от 512 байт до 1024 кибибайт.
- Операция частичной записи 512 байт:
  1. Считывается полный блок (всегда).
  2. Заменяется 512 байт в требуемом месте.
  3. Записывается полный блок (всегда).
- Выровненная запись целого блока — минимум двукратное ускорение.

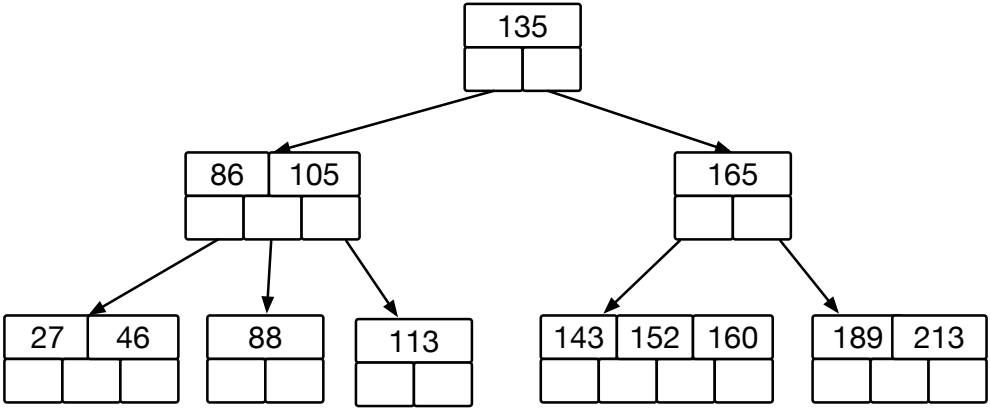
Оценка применимости внешнего поиска

- Пусть имеется бинарное дерево поиска, состоящее из:
  1. Данных размером 64 байта.
  2. Ключа размером 8 байт.
  3. Указателей left и right размером 8 байт.
- Общий размер узла — 88 байт.
- В оперативную память размером 16 гибибайт поместится $\frac{16 \times 2^{30}}{88} \approx 195 \times 10^6$ узлов (не учитываем фрагментацию аллокаторов).
- Как хранить словарь из 10^9 элементов?

B-деревья

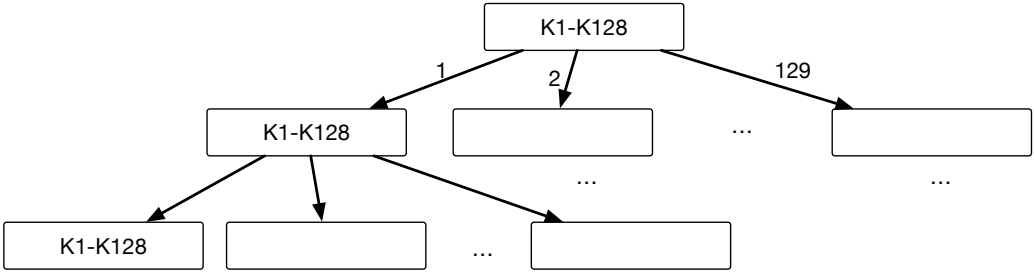
- *B-дерево* — сбалансированное дерево поиска, узлы которого хранятся во внешней памяти.
- В оперативной памяти хранится часть узлов.

B-деревья: свойства



- Высота дерева не более $O(\log N)$, где N— количество узлов.
- Каждый узел может содержать 1 ключ и больше.
- Количество детей узла равно $K + 1$, где K — количество ключей в узле.

B-деревья: свойства



- Пусть в узле помещается 128 ключей.
- Высота дерева — 3
- Тогда общее количество узлов

$$1 + 129 + 129^2 = 16771$$

- Общее количество ключей

$$16771 \times 128 = 2146688$$

B-деревья: определение

- **B-дерево** — корневое дерево, обладающее свойствами:
  1. Каждый узел содержит:
     - количество ключей n, хранящихся в узле.
     - индикатор листа `final`.
     - `n` ключей в порядке возрастания.
     - `n+1` указатель на детей, если узел не корневой.
  2. Ключи есть границы диапазонов ключей в поддеревьях.
  3. Все листья расположены на одинаковой глубине `h`.
  4. Имеется показатель `t` — минимальная степень дерева.
  5. В корневом узле от 1 до `2t-1` ключей.
  6. Во внутренних узлах минимум `t-1` ключей.
  7. Во внешних узлах максимум `2t-1` ключей.
  8. Заполненный узел имеет `2t-1` ключ.

B-деревья: высота

- **Теорема:** Высота B-дерева с $n \geqslant 1$ ключами и минимальной степенью $t \geqslant 2$ в худшем случае не превышает $\log_t \frac{n+1}{2}$
- **Доказательство.** В максимально высоком дереве высоты h в каждом узле, кроме корневого, содержится $t-1$ ключ.
  Тогда общее количество ключей в дереве есть

$$1 + 2 + 2t + 2t^2 + \cdots + 2t^{h-1} =$$

$$= 1 + 2(t-1)(1 + t + t^2 + \cdots + t^{h-1}) =$$

$$= 1 + 2(t-1)\frac{t^h - 1}{t-1}$$

  Отсюда

$$h = \log_t \frac{n+1}{2}$$

B-деревья: операции

- Используем операции `Load` и `Store`.
- Корень сохраняем в оперативной памяти.
- Минимизируем количество операций.

B-деревья: операция `find` поиска ключа `k`

1. Операцией бинарного поиска ищем самый левый ключ $key_i \geqslant k$
2. Если $key_i = k$, то узел найден.
3. Исполняем `Load` для дочернего узла и рекурсивно повторяем операцию.
4. Если $final = true$, то ключ не найден.

Количество операций $T_{load} = O(h) = O(\log_t n)$

Добавление ключа

1. Операцией `find` находим узел для вставки.
2. Если лист не заполнен, сохраняя упорядоченность вставляем ключ.
3. Если лист заполнен (`2t-1` ключей), разбиваем его на два листа по `t-1` ключу поиском медианы.
4. Медиана рекурсивно вставляется в родительский узел.

Сложность в худшем случае: каждый раз разбивается узел на каждом уровне, $O(t \log_t n)$
Количество операций: $T_{ext} = O(h) = O(\log_t n)$

Разновидности B-деревьев

- $\mathbf{B}^+$-дерево содержит информацию только в листьях, ключи — только во внутренних узлах.
- Используется в файловых системах `XFS`, `JFS`, `NTFS`, `Btrfs`, `HFS`, `APFS`, ...
- Используется для хранения индексов в базах данных `Oracle`, `Microsoft SQL`, `IBM DB2`, `Informix`, ...