# Сбалансированные деревья

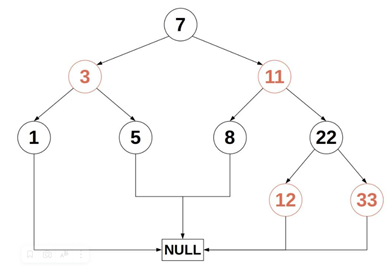
Сбалансированное дерево - это дерево поиска, которое не просто поддерживает порядок между узлами. Он также контролирует свою высоту, следя за тем, чтобы она оставалась после вставки или удаления.

Для этого сбалансированное дерево должно заново сбалансировать себя после добавления или удаления узла. Это приводит к вычислительным затратам и усложняет алгоритмы вставки и удаления. Однако это цена, которую мы готовы заплатить за дерево поиска логарифмической высоты с быстрыми операциями поиска, вставки и удаления.

Существует несколько типов таких деревьев. Они требуют, чтобы все их узлы были сбалансированы, но понятие баланса отличается от типа к типу.

# Красно-черные деревья

Красно-черные деревья (RBT) также уравновешивают высоту дочерних поддеревьев. Но RBT различает два типа узлов: красные и черные. RBT гарантирует, что все пути от узла к его потомку будут проходить через одинаковое количество черных узлов. Кроме того, количество черных узлов от узла до его листьев, исключая узел, называется высотой черного узла. Черная высота всего RBT равна черной высоте его корня. Например (с нулевыми листьями, объединенными в один узел для экономии места)

****

**По определению, RBT удовлетворяет этим условиям:**

* Каждый узел либо черный, либо красный.
* Корень черный.
* Каждый пустой узел (нулевой или ненулевой) черный.
* Если узел красный, то оба его дочерних элемента черные.
* Для каждого узла х пути от х, не включая его, к его дочерним листьям содержат одинаковое количество черных узлов.

Некоторые авторы не требуют, чтобы корень был черным, потому что мы в любом случае можем перекрасить дерево.

**Свойства RBT обеспечивают:**

что ни один путь от корня к листу не более чем в два раза длиннее пути к другому листу,и что высота дерева равна.

**Преимущества:**

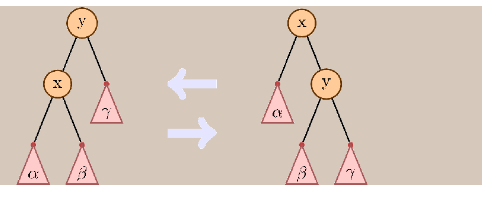
* Красно-черные деревья уравновешивают уровень дерева.
* Красно-черное дерево получает некоторый запас для структурирования дерева, восстанавливая уровень дерева.
* Временная сложность поисковой активности составляет O(log n).
* Он имеет такие же низкие константы в широком диапазоне ситуаций.

**Недостатки:**

* Сложно использовать из-за всех крайних случаев активности; как правило, вам нужно использовать стандартное выполнение библиотеки (например, набор STL в C++ и т. д.) вместо того, чтобы выполнять его самостоятельно без какой-либо подготовки.
* Если вы планируете сформировать дерево только один раз, а затем просто выполнять операции чтения, деревья AVL предлагают лучшее исполнение.
* Поскольку B-деревья могут иметь различное количество дочерних элементов, их часто предпочитают красно-черным деревьям за упорядочивание и размещение большого количества данных на пластинах, поскольку они могут быть несколько поверхностными, чтобы ограничить круговые задачи.

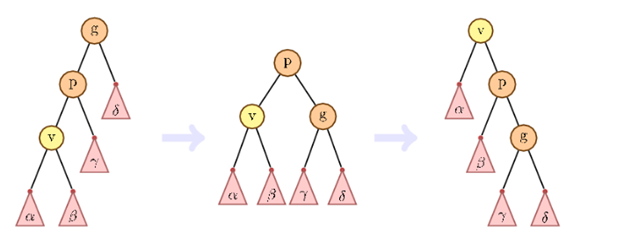
# Splay-деревья

# Splay-дерево — это самобалансирующееся бинарное дерево поиска. Дереву не нужно хранить никакой дополнительной информации, что делает его эффективным по памяти. После каждого обращения, даже поиска, splay-дерево меняет свою структуру. Основная эвристика splay-дерева — move-to-root. После обращения к любой вершине, она поднимается в корень. Подъем реализуется через повороты вершин. За один поворот, можно поменять местами родителя с ребенком, как показано на рисунке ниже.

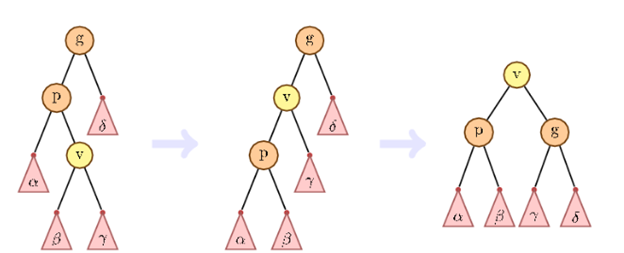


Но просто поворачивать вершину, пока она не станет корнем, недостаточно. Хитрость splay-дерева в том, что при продвижении вершины вверх, расстояние до корня сокращается не только для поднимаемой вершины, но и для всех ее потомков в текущих поддеревьях. Для этого используется техника zig-zig и zig-zag поворотов.

Основная идея zig-zig и zig-zag поворотов, рассмотреть путь от дедушки к ребенку. Если путь идет только по левым детям или только по правым, то такая ситуация называется zig-zig. Как ее обрабатывать показано на рисунке ниже. Сначала повернуть родителя, потом ребенка.



В противном случае, мы сначала меняем ребенка с текущим родителем, потом с новым.



Если у вершины дедушки нет, делаем обычный поворот:

Splay-деревья ведут себя хорошо при последовательной обработке большого набора запросов с произвольным распределением. Они экономичны в плане количества информации, которую требуется хранить в узлах.

Преимущества:

* Сопоставимая производительность: производительность в среднем случае такая же эффективная, как и у других деревьев.
* Небольшой объем памяти: деревьям Splay не нужно хранить какие-либо бухгалтерские данные.
* Сбалансировано время операций

**Недостатки:**

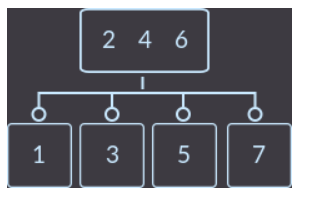
* Могут вырождаться в линеный список
* Например, это будет иметь место после доступа ко всем *n* элементам в порядке неубывания. Поскольку высота дерева соответствует наихудшему времени доступа, это означает, что фактическая стоимость одной операции может быть высокой. Однако [амортизированная](https://translated.turbopages.org/proxy_u/en-ru.ru.1c5916b4-63802361-5ab59c32-74722d776562/https/en.wikipedia.org/wiki/Amortized) стоимость доступа в этом наихудшем случае является логарифмической, O (log *n*). Кроме того, ожидаемая стоимость доступа может быть уменьшена до O (log *n*) с помощью рандомизированного варианта.[[4]](https://translated.turbopages.org/proxy_u/en-ru.ru.1c5916b4-63802361-5ab59c32-74722d776562/https/en.wikipedia.org/wiki/Splay_trees#cite_note-4)
* Представление splay-деревьев может измениться, даже если к ним осуществляется доступ "только для чтения" (т. Е. с помощью операций *поиска*). Это усложняет использование таких splay деревьев в многопоточной среде. В частности, требуется дополнительное управление, если нескольким потокам разрешено выполнять операции *поиска* одновременно. Это также делает их непригодными для общего использования в чисто функциональном программировании, хотя даже там они могут использоваться ограниченными способами для реализации очередей приоритетов.

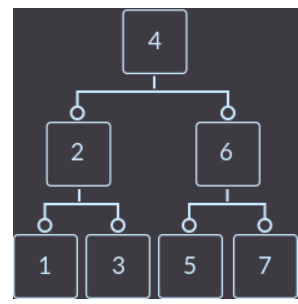
# 2-3-дерево

2-3-дерево — структура данных, страницы которой могут содержать только 2-вершины (вершины с одним полем и 2 детьми) и 3-вершины (вершины с 2 полями и 3 детьми). Листовые вершины являются исключением — у них нет детей (но может быть одно или два поля). 2-3-деревья сбалансированы, то есть, каждое левое, правое, и центральное поддерево имеет одну и ту же высоту, и, таким образом, содержат равные (или почти равные) объемы данных.

**Свойства:**

* Все нелистовые вершины содержат одно поле и 2 поддерева или 2 поля и 3 поддерева.
* Все листовые вершины находятся на одном уровне (на нижнем уровне) и содержат 1 или 2 поля.
* Все данные отсортированы (по принципу двоичного дерева поиска).
* Нелистовые вершины содержат одно или два поля, указывающие на диапазон значений в их поддеревьях. Значение первого поля строго больше наибольшего значения в левом поддереве и меньше или равно наименьшему значению в правом поддереве (или в центральном поддереве, если это 3-вершина); аналогично, значение второго поля (если оно есть) строго больше наибольшего значения в центральном поддереве и меньше или равно, чем наименьшее значение в правом поддереве. Эти нелистовые вершины используются для направления функции поиска к нужному поддереву и, в конечном итоге, к нужному листу. (*прим. Это свойство не будет выполняться, если у нас есть одинаковые ключи. Поэтому возможна ситуация, когда равные ключи находятся в левом и правом поддеревьях одновременно, тогда ключ в нелистовой вершине будет совпадать с этими ключами. Это никак не сказывается на правильности работы и производительности алгоритма.*).

****

****

**Преимущества:**

* Для наборов данных, которые помещаются в основную память, 2-3 дерева и BST обычно являются лучшим выбором (хотя были некоторые исследования, показывающие, что B-деревья низкого порядка могут превосходить BST в основной памяти из-за эффектов кэша.)
* Если вы планируете хранить огромный объем данных, которые не могут поместиться в основную память, B-деревья - отличный выбор для структуры данных.

**Недостатки:**

* В основной памяти B-деревья с очень большим коэффициентом ветвления будут медленнее, чем BSTS или 2-3 дерева, потому что для каждой вставки или удаления B-дерева может потребоваться большое количество переназначений указателей.

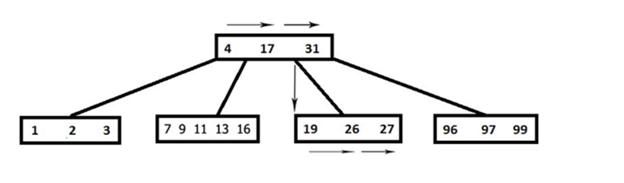
# B-tree

Большинство знакомо с такими сбалансированными деревьями, как «красно-черное дерево», «AVL-дерево», «Декартово дерево», поэтому не будем углубляться.

B-деревья также представляют собой сбалансированные деревья, поэтому время выполнения стандартных операций в них пропорционально высоте. Но, в отличие от остальных деревьев, они созданы специально для эффективной работы с дисковой памятью (в предыдущем примере – сторонним носителем), а точнее — они минимизируют обращения типа ввода-вывода.

Поиск в B-дереве очень схож с поиском в бинарном дереве, только здесь мы должны сделать выбор пути к потомку не из 2 вариантов, а из нескольких. В остальном — никаких отличий. На рисунке ниже показан поиск ключа 27. Поясним иллюстрацию (и соответственно стандартный алгоритм поиска):

* Идем по ключам корня, пока меньше необходимого. В данном случае дошли до 31.
* Спускаемся к ребенку, который находится левее этого ключа.
* Идем по ключам нового узла, пока меньше 27. В данном случае – нашли 27 и остановились.



B- деревья *сортированы*: ключи внутри самого узла B- дерева находятся в порядке сортировки. Благодаря этому, для определения места некого искомого ключа мы можем применять некий алгоритм, подобный двоичному поиску. Это также подразумевает, что поиск по B- деревьям обладает логарифмической сложностью. Например, поиск искомого ключа среди 4 миллиардов (**4 × 109**) элементов занимает около 32 сравнений (Когда нам приходится выполнять дисковое позиционирование для каждого из таких сравнений, это значительно замедлило бы нас, но так как узлы B- дерева хранят десятки и даже сотни элементов, нам всего лишь придётся выполнят по одному позиционированию диска для прыжка на следующий уровень.

Очевидно, что мы не можем хранить всё это дерево в оперативной памяти => в ней храним лишь часть информации, остальное же хранится на стороннем носителе (допустим, на жестком диске, скорость доступа к которому гораздо медленнее). Такие деревья как красно-черное или Декартово будут требовать от нас log n обращений к стороннему носителю. При больших n это очень много. Как раз эту проблему и призваны решить B-деревья!

**Иерархия В- деревьев**

B- состоят из множества узлов. Каждый узел содержит до **N** ключей и **N + 1** указателей на его дочерние узлы. Эти узлы логически разбиваются на три группы:

***Узел корня***

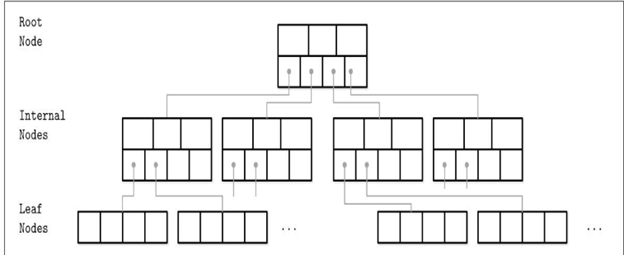
*Не имеет предков и является основной вершиной этого дерева.*

***Узлы листьев***

*Это самые узлы самого нижнего уровня, которые не имеют дочерних узлов {терминальные узлы}.*

***Внутренние узлы***

*Это все прочие узлы, соединяющие корень с листьями. Обычно имеется более одного уровня внутренних узлов {нетерминальные узлы}.*

**

Преимущества:

* Во всех случаях полезное использование памяти для хранения индексов составляет свыше 50%. Обеспечивается динамическое распределение и использование памяти. С ростом степени полезного использования памяти не происходит снижения качества обслуживания пользовательских запросов.
* Произвольный доступ к записи реализуется посредством небольшого количества подопераций (обращений к физическим блокам) и по эффективности сопоставим с методами перемешивания.
* В среднем достаточно эффективно реализуются операции включения и добавления записей. При этом сохраняется естественный порядок ключей с целью последовательной обработки, а также соответствующий баланс дерева для обеспечения быстрой произвольной выборки.
* Упорядоченность по ключу обеспечивает возможность эффективной пакетной обработки.

**Недостатки:**

* Основной недостаток В-деревьев состоит в отсутствии для них средств выборки данных по вторичному ключу.

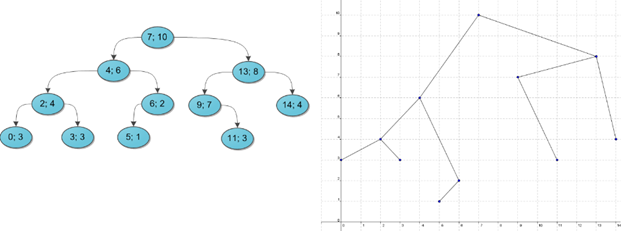
# Декартово дерево

**Декартово дерево** (cartesian tree, treap) — красивая и легко реализующаяся структура данных, которая с минимальными усилиями позволит вам производить многие скоростные операции над массивами ваших данных.

**Декартово дерево или дерамида** (англ. *Treap*) — это структура данных, объединяющая в себе [бинарное дерево поиска](https://neerc.ifmo.ru/wiki/index.php?title=%D0%94%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA%D0%B0,_%D0%BD%D0%B0%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D1%80%D0%B5%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F) и [бинарную кучу](https://neerc.ifmo.ru/wiki/index.php?title=%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%B0%D1%8F_%D0%BA%D1%83%D1%87%D0%B0) (отсюда и второе её название: treap (tree + heap) и дерамида (дерево + пирамида), также существует название курево (куча + дерево).

Более строго, это бинарное дерево, в узлах которого хранятся пары (x,y)(x,y), где xx — это ключ, а yy — это приоритет. Также оно является двоичным деревом поиска по xx и пирамидой по yy. Предполагая, что все xx и все yy являются различными, получаем, что если некоторый элемент дерева содержит (x0,y0)(x0,y0), то у всех элементов в левом поддереве x<x0x<x0, у всех элементов в правом поддереве x>x0x>x0, а также и в левом, и в правом поддереве имеем: y<y0y<y0.

Почему дерево называется декартовым? Это сразу станет ясно, как только мы попробуем его нарисовать. Возьмем какой-нибудь набор пар «ключ-приоритет» и расставим на координатной сетке соответствующие точки (x, y). А потом соединим соответствующие вершины линиями, образуя дерево. Таким образом, декартово дерево отлично укладывается на плоскости благодаря своим ограничениям, а два его основных параметра — ключ и приоритет — в некотором смысле, координаты. Результат построения показан на рисунке: слева в стандартной нотации дерева, справа — на декартовой плоскости.



* обладает почти гарантированно логарифмической высотой относительно количества своих вершин;
* позволяет за логарифмическое время искать любой ключ в дереве, добавлять его и удалять;
* исходный код всех её методов не превышает 20 строк, они легко понимаются и в них крайне сложно ошибиться.

**Преимущества:**

* В том применении, которое мы рассматриваем (мы будем рассматривать дерамиды, поскольку декартово дерево - это фактически более общая структура данных), X'ы являются ключами (и одновременно значениями, хранящимися в структуре данных), а Y'и - называются приоритетами. Если бы приоритетов не было, то было бы обычное бинарное дерево поиска по X, и заданному набору X'ов могло бы соответствовать много деревьев, некоторые из которых являются вырожденными (например, в виде цепочки), а потому чрезвычайно медленными (основные операции выполнялись бы за O (N)).
* В то же время, приоритеты позволяют однозначно указать дерево, которое будет построено (разумеется, не зависящее от порядка добавления элементов) (это доказывается соответствующей теоремой).
* Проще реализуется, по сравнению, например, с настоящими самобалансирующимися деревьями вроде красно-черного.
* Хорошо ведет себя «в среднем», если ключи *y* раздать случайно.

**Недостатки:**

* Большие накладные расходы на хранение: вместе с каждым элементом хранятся два-три указателя и случайный ключ *y*.
* Скорость доступа *O*(*n*) в худшем, хотя и маловероятном, случае. Поэтому декартово дерево недопустимо, например, в ядрах ОС.

**Источники:**

1) <https://translated.turbopages.org/proxy_u/en-ru.ru.318eeb2f-638017ce-bcdf8f87-74722d776562/https/www.baeldung.com/cs/balanced-trees>

2) <https://habr.com/ru/company/JetBrains-education/blog/210296/>

3) <https://habr.com/ru/post/303374/>

4) <https://habr.com/ru/post/114154/>

5) <http://onreader.mdl.ru/DatabaseInternals/content/Ch02.html>

6) <https://habr.com/ru/post/101818/>

7) [https://neerc.ifmo.ru/wiki/index.php?title=Декартово\_дерево](https://neerc.ifmo.ru/wiki/index.php?title=%D0%94%D0%B5%D0%BA%D0%B0%D1%80%D1%82%D0%BE%D0%B2%D0%BE_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE)

8) <https://progler.ru/blog/primenenie-preimuschestva-i-nedostatki-krasno-chernogo-dereva>

9) <https://translated.turbopages.org/proxy_u/en-ru.ru.1c5916b4-63802361-5ab59c32-74722d776562/https/en.wikipedia.org/wiki/Splay_trees#Disadvantages>

10)<https://studfile.net/preview/9433788/page:11/>

11)<https://translated.turbopages.org/proxy_u/en-ru.ru.2a6ee90a-63802441-648ad742-74722d776562/https/stackoverflow.com/questions/22125233/usage-pros-and-cons-for-binary-search-tree-2-3-tree-and-b-tree>

12)<https://e-maxx.ru/algo/treap>

13) <https://intellect.icu/derevya-poiska-avl-derevo-splej-derevo-dekartovo-derevo-65>

14)<https://helpiks.org/9-39073.html>

15)<https://helpiks.org/9-39073.html>