**Красно-чёрное дерево (Red-Black Tree)**

**Определение и основные характеристики:**

Красно-чёрное дерево – это сбалансированное двоичное дерево поиска, которое гарантирует логарифмическую высоту по числу узлов. Каждому узлу дерева добавляется бит «цвета» (красный или чёрный), что позволяет поддерживать баланс без хранения высот или доп. полей. Благодаря этому поиск, вставка и удаление выполняются за O(\og n) в худшем случае. Красно-чёрное дерево изобрёл Байер (Rudolf Bayer) и использует три основных операции поворота и перекраску узлов для самобалансирования.

**Свойства и инварианты:**

Дерево удовлетворяет следующим условиям:

* Каждый узел окрашен **красным** или **чёрным**.
* Корневой узел и все (фиктивные) листья считаются **чёрными**.
* Любой красный узел имеет **чёрного** родителя (нет двух подряд красных).
* Все простые пути от корня до листьев содержат одинаковое число чёрных узлов. Этот инвариант (чёрная высота) обеспечивает сбалансированность дерева.

**Поиск:**

Осуществляется как в обычном BST – путём сравнения ключа с текущим узлом и спуска в левое или правое поддерево. Цвет узлов при поиске игнорируется – баланс только гарантирует O(log n) высоту. Сложность поиска – O(log n) в среднем и в худшем случае.

**Вставка:**

Новый ключ добавляется в листовом узле обычным образом (как в BST), узел окрашивается **красным**. Затем выполняется процедура восстановления свойств:

1. Если отец нового узла был чёрным, то свойства не нарушены – работа закончена.
2. Если отец красный (нарушено «два красных подряд»), смотрим на «дядю» (другого ребёнка деда):
   * **Дядя красный:** перекрасить отца и дядю в **чёрный**, а деда – в **красный**. После этого продолжаем проверку балансировки для деда (возможно потребуется подняться выше).
   * **Дядя чёрный:** необходимо сделать поворот. Если вставленный узел – правый сын, сначала выполнить левый поворот, чтобы он стал левым, затем правый поворот относительно деда. После поворотов скорректировать цвета: бывший отец стать чёрным, а бывший дед – красным.
3. В конце обеспечить, чтобы корень был **чёрным** (если он стал красным).

Ниже схематично описана последовательность действий при вставке:

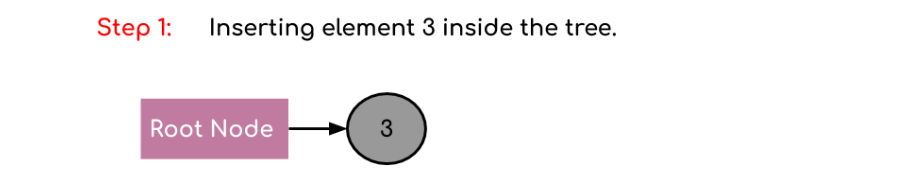
* Найти место вставки и добавить узел красным.
* Если возникает красный-родитель, применить описанные кейсы (перекраска или поворот).
* Окончательно перекрасить корень в чёрный.

**Удаление:**

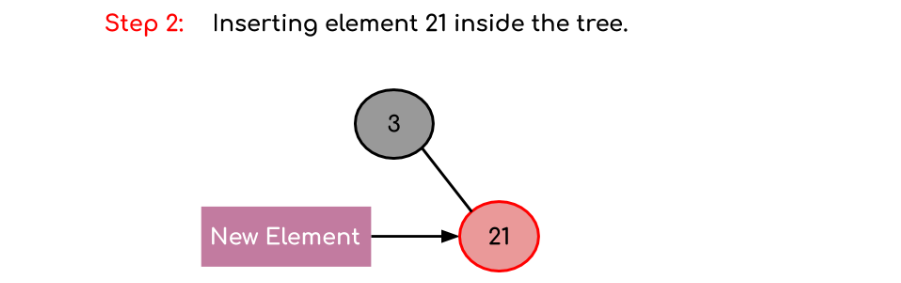
Удаление более сложное. Сначала удаляют ключ как в BST (с заменой через преемника/предшественника, если удаляемый узел – внутренний). Если удаляется красный узел, структура остаётся корректной. При удалении чёрного узла может нарушиться чёрная высота, и тогда применяются следующие приёмы восстановления:

* Если «брат» узла (сосед в дереве) **красный**, делают поворот и перекраску: брат становится чёрным, отец – красным. Это приводит к ситуации, когда брат становится чёрным и даёт возможность перейти к обычным случаям.
* Если брат **чёрный**:
  + **У брата оба ребёнка чёрные:** перекрасить брата в красный и продолжить корректировку на уровне родителя. Это компенсирует удаление чёрного узла за счёт увеличения чёрной глубины по пути к родителю.
  + **У брата есть хотя бы один красный ребёнок:** выполняют вращение (и перекраску) так, чтобы перенести «лишний» красный узел к удаляемому месту. Например, если левый ребёнок брата красный: перекрашивают брата и его левого сына и делают правый (или левый) поворот. Ситуация разбирается отдельными подслучаями (левый/правый сын красный). После такого поворота дерево выравнивается и можно окончательно перекрасить узлы, восстановив свойства.

Таким образом, при удалении используется серия поворотов и перекраски, аналогичных описанным в алгоритме вставки. Итоговый алгоритм гарантирует, что свойства красно-чёрного дерева восстановятся.



*Рисунок 1.* **Шаг 1:** Вставка элемента 3. Поскольку дерево пусто, узел 3 становится корневым и перекрашивается в чёрный.  
**Шаг 2:** Вставляем 21 (красный, так как 21>3). Узел 21 уходит в правое поддерево корня. Двойного красного не возникло (отец 3 чёрный), баланс сохранён.  
**Шаг 3:** Вставляем 32 (красный, 32>3). Получилась пара «отец-красный» (узел 21 – красный с красным потомком). Это нарушает правило, поэтому выполняем правый поворот вокруг корня и перекрасим: в итоге 21 становится корнем (чёрным), 3 и 32 – его красными детьми.



*Рисунок 2.* **Шаг 4:** Вставляем 15 (красный). Он вставляется как левый ребёнок узла 21. При этом родители 3 и 21 – красные, что снова нарушает правило. Производится перекраска: узлы 3 и 21 становятся чёрными, а их «дед» (то есть новый корень 21) – красным. Затем обязательно делаем корень чёрным.

**Вырожденные ситуации:**

* Теоретически RB-дерево не вырождается в список: высота всегда благодаря чёрной высоте, так что линейного вырождения не происходит.
* На практике при крайне неблагоприятных последовательностях вставок/удалений (например, чередование крайних значений) может происходить максимальное число перекрасок и поворотов, но асимптотика остаётся логарифмической.

**Итог:**

В результате всех операций окончательное дерево имеет корень 21 (чёрный) с двумя потомками 3 (красный, слева) и 32 (красный, справа), при этом под 3 находится ещё 15 (красный).

**Сравнительный анализ:**

* **Достоинства:** Гарантируется O(log n) в худшем случае для всех операций (поиск, вставка, удаление). В каждом узле хранится только 1 бит цвета, поэтому по памяти экономичнее AVL (у AVL храним разницу высот, 2 бита). Красно-чёрные деревья широко применяются в библиотеках С++ (например, std::map реализован как RB-дерево), Java (TreeMap) и других системах, где важна гарантированная сложность операций.
* **Недостатки:** Более высокая «жёсткая» высота, чем у AVL: красно-чёрное дерево может быть в ~1.38 раза выше при том же числе узлов. Вставка/удаление требуют до 3 вращений (но не более константы), в то время как у AVL может быть до O(log n) поворотов. Однако на практике в быстрых реализациях RB-деревья иногда уступают AVL по производительности из-за накладных расходов балансировки.

**Splay-дерево**

**Определение и основные характеристики:**

Splay-дерево (расширяющееся дерево) – саморегулирующееся двоичное дерево поиска. Особенность состоит в том, что после каждой операции доступа (поиск, вставка или удаление) целевой узел «растаскивается» (splay) к корню с помощью последовательности вращений. Дополнительных полей (цветов, высот) не хранится – баланс достигается чисто за счёт реорганизации дерева при доступе. Такая стратегия обеспечивает **амортизированную** сложность O(log n) на операцию (за последовательность операций общее время O(log n) при n). Придуман Робертом Тарьяном и Дэниелом Слейтором (1983).

**Свойства и инварианты:**

Единственный неизменный атрибут – свойство BST (левое поддерево содержит меньшие ключи, правое – большие). Фактический баланс не гарантируется при каждом доступе: после одной операции дерево может ухудшиться (вырост итерационный худшего случая до O(n)). Главная «гарантия» – амортизированное время. Нет жёстких свойств наподобие красно-чёрных (запрета двух подряд красных и т.д.).

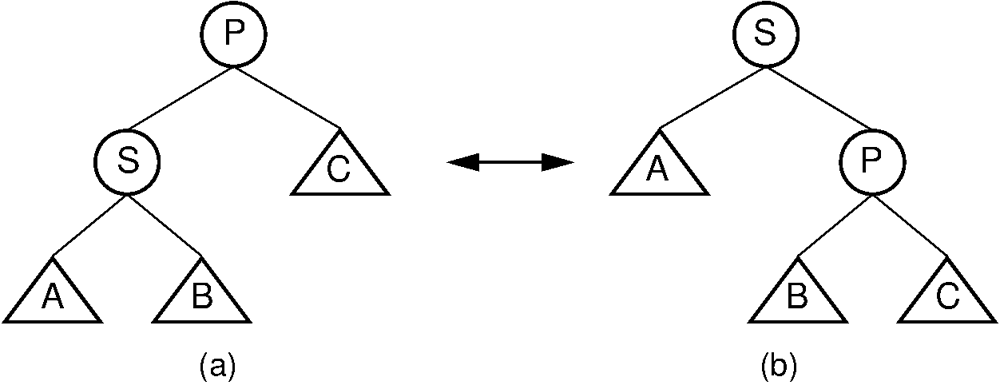
**Алгоритмы:**

* **Поиск:** Выполняется обычным BST-поиском. Если узел найден, выполняется splay: выбранный узел поднимается к корню через серию вращений. Если узел не найден, то поднимается к корню последний посещённый узел (предок отсутствующего ключа). Это облегчает будущие поиски близких значений.
* **Вставка:** Вставляем ключ как в обычное BST (в лист) и отмечаем его как найденный. Затем выполняем splay для нововставленного узла. Процедура вращений та же самая, что и при поиске. В итоге новый узел окажется в корне, адаптируя структуру под частые обращения.
* **Удаление:** Ищем узел, удаляем его как в BST (если два потомка – меняем ключ с преемником/предшественником, затем удаляем в листе). После удаления обычно сплайним родителя удалённого узла или другой близкий узел к корню. Альтернативно, удалённый узел сначала splay’ят к корню, затем удаляют корень и сливают левое и правое поддеревья. В любом случае после операции глубина дерева вновь сбалансируется амортизированно.

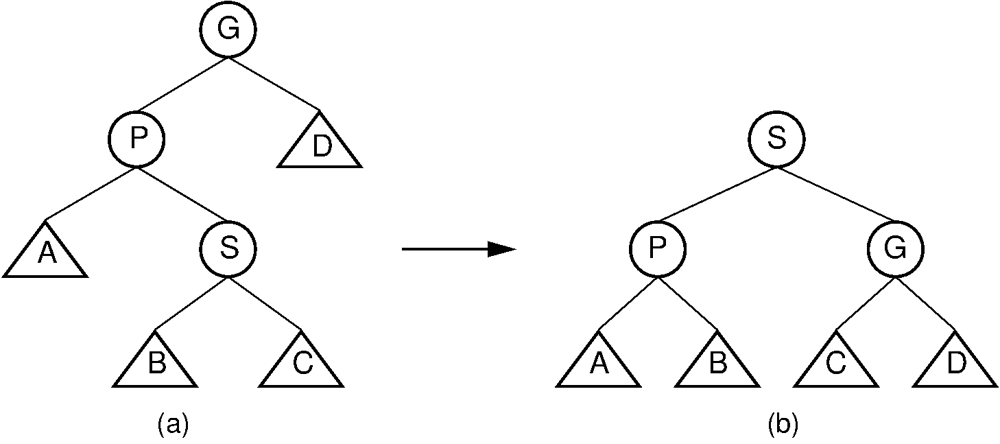
**Операция *splay* (расширение):**

Состоит из повторных **вращений** (Zig, Zig-Zig, Zig-Zag). Пусть xxx – текущий узел, ppp – его родитель, ggg – дед.

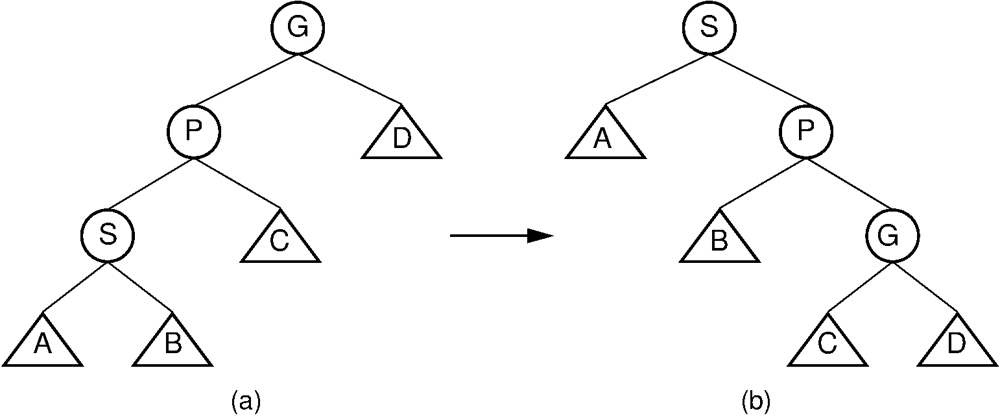
* *Zig:* когда ppp – корень. Проводим один поворот по ребру ppp–xxx, поднимая xxx на место ppp.
* *Zig-Zig:* когда xxx и ppp оба левые (или оба правые) дети. Сначала поворот по ребру ggg–ppp, затем по ребру ppp–xxx.
* *Zig-Zag:* когда xxx и ppp находятся по разные стороны (один левый, другой правый). Выполняем два поворота: сначала по ребру ppp–xxx, затем по ребру ggg–xxx.



*Рисунок 3.* **Zig (одинарное вращение):** узел xxx (S) поднимается к корню, меняясь местами с родителем ppp. На рисунке узел S переместился наверх, а P стал его потомком.



*Рисунок 4.* **Zig-Zag:** узел S сначала поворачивается с родителем P, затем с дедом G. В результате он поднимается на два уровня (S становится корнем поддерева).



*Рисунок 5.* **Zig-Zig:** узел S и его отец P оба – левые (или правые) дети. Сначала делается вращение G↔P, затем P↔S. После двух вращений S оказывается наверху, сохраняя порядок BST.

**Вырожденные ситуации:**

* Одна операция поиска/вставки/удаления может занять O(n), если дерево превратилось в цепочку (например, вставка или последовательный доступ по возрастанию без перетасовки).
* При последовательных отсортированных запросах без случайных прерываний дерево будет вырождаться в список, однако амортизированная сложность сохраняется.

**Пошаговый пример:**

Например, при поиске узла S в дереве дважды применяются операции Zig-Zig или Zig-Zag до поднятия S в корень. На рисунках 3–5 показаны возможные сценарии этих вращений. После splay наиболее часто запрашиваемые узлы оказываются близко к корню, ускоряя последующие операции.

**Сравнительный анализ:**

* **Достоинства:** Простая реализация (нет необходимости хранить балансирующие поля). Часто эффективно адаптирует дерево под реальный паттерн запросов: недавно или часто запрашиваемые элементы двигаются к корню. Это выгодно для кэширования и динамических задач (упорядоченные множества, сжатие данных, сетевые алгоритмы). Амортизированно обеспечивает O(log n) время на операцию при общем объёме запросов.
* **Недостатки:** Нет жёстких гарантий баланса. В худшем случае одна операция может занять O(n), дерево может вырождаться (например, при вставке отсортированных данных). Поэтому splay-дерево **не подходит** для систем с жёсткими требованиями к времени (реальное время, safety-critical). Бывает хуже обычных сбалансированных деревьев по стабильности производительности.

**B-дерево**

**Определение и основные характеристики:**

B-дерево – сбалансированное **m-арное** (или *B*-дерево порядка *m*) дерево поиска с высокой степенью ветвистости. В узле может храниться сразу несколько ключей (до m-1) и m указателей на потомков. Все листья находятся на одном уровне, и глубина дерева минимальна. B-дерево изобретено Байером и МакКрейтом (1970) для эффективной работы с внешней памятью (базами данных, дисками). Баланс обеспечивается тем, что длины любых двух корень–лист путей отличаются не более чем на 1. Типичный порядок B-дерева – большой (например, 50–2000), чтобы соответствовать размеру страниц диска.

**Свойства и инварианты:**

Пусть минимальная степень дерева = 2 (иногда обозначают m=2 – максимальное число потомков). Тогда соблюдаются условия:

* Каждый узел (кроме корня) содержит от t-1 до 2t-2 ключей; корень может содержать от 1 до 2t-1 ключей.
* Ключи внутри узла упорядочены по возрастанию. Если узел содержит ключи K\_1…K\_n, то он имеет n+1 потомков. Причём первый поддерево содержит ключи K\_1, второе – между K\_1 и K\_2, …, (n+1)-e – K\_n.
* Все листья имеют одинаковую глубину (одно количество обращений от корня к листу). Это даёт гарантированную логарифмическую высоту O(log n) при любом раскладе данных.

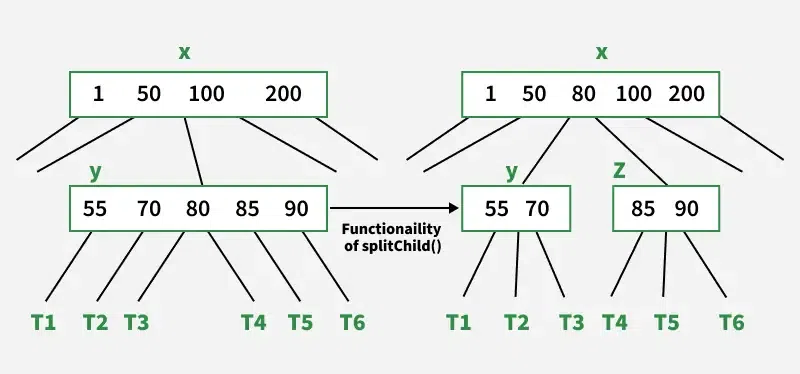
**Поиск:**

В корне находим интервал для ключа (обычно бинарным поиском по ключам узла), затем спускаемся в соответствующего потомка. Процесс повторяется, пока не достигнем листа или найдём ключ. Поиск за O(log n) при разумном выборе t (высота мала благодаря большой ветвистости).

**Вставка:**

Новый ключ всегда добавляется в лист. Алгоритм (метод «спуск, разделение по необходимости»):

1. Найти лист, в который должен попасть ключ (обычным BST образом по ключам узла).
2. Вставить ключ в этот узел (вставка в отсортированный массив ключей).
3. Если после вставки узел превысил максимальное число ключей (2t-1), то разделить его: старый узел y делится на два (y\_1 и y\_2), в каждом остаётся по t-1 ключей, а медианный ключ из y поднимается в родительский узел. Указатели распределяются аналогичным образом (каждый потомок y переходит в y\_1 или y\_2).
4. Если разделился корень (т.е. родитель был пустым), то медианный ключ становится **новым корнем**, а два полученных узла – его детьми (высота дерева увеличивается на 1).
5. Разделение может рекурсивно подниматься вверх, если и родитель переполнится.



*Рисунок 6.* **Пример вставки в B-дерево:**

Пусть t=3 (каждый узел хранит 2..5 ключей). При вставке ключа 80 узел Y (содержит ключи 55, 70, 80, 85, 90) переполняется (5 ключей). Он делится на два узла Y\_1=[55,70] и Y\_2=[85,90], а медиана 80 поднимается в родителя. Указатели на поддеревья также делятся пополам. В итоге в родительском узле вместо ссылки на Y появляются ссылки на Y\_1 и Y\_2. Таким образом B-дерево остаётся сбалансированным и сохраняет отсортированность ключей.

**Удаление:**

Удаление ключа из B-дерева сложнее и сводится к поддержанию инвариантов t-1{ключей}-2t-1 в каждом узле. Алгоритм:

1. Найти узел x, содержащий удаляемый ключ K.
2. Если x – лист, удалить ключ непосредственно. Если в x осталось ≥(t-1) ключей, завершить. Иначе (узел стал «недоукомплектованным»), применяются случаи заимствования или слияния:
   * Если у «брата» (соседа того же родителя) ≥ t ключей, можно **заимствовать**: взять соответствующий ключ-разделитель из родителя и один ключ из брата, выполнить поворот и обновить цвет (аналогично поворотам в RB-дереве, но в контексте B-дерева).
   * Если брать нельзя, выполняется **слияние**: узел x объединяется с одним из соседей и соответствующим ключом из родителя (в сумме получается 2t-2 ключей). Родитель при этом теряет ключ-разделитель; если в нём стало меньше t-1 ключей, процесс повторяется рекурсивно вверх.
3. Если удаляемый ключ K находится во внутреннем узле, его обычно заменяют на преемника (максимальный ключ в левом поддереве) или предшественника (минимальный в правом). После этого удаление продолжается как из листа (описано выше).
4. Если в процессе стало пустым само корневое дерево, корень заменяется единственным потомком (высота уменьшается на 1).

Таким образом все инварианты B-дерева сохраняются. Удаление гарантирует, что дерево остаётся сбалансированным, хотя может потребоваться несколько слияний или переносов между узлами.

**Вырожденные ситуации:**

* При минимальной степени t=2 B-дерево превращается в 2-3-4 дерево; оно не вырождается в список, но при малых t высота может стать значительной.
* Если данные некорректно подобраны (частые разделения без слияний), может наблюдаться временное увеличение числа узлов и операций разделения, но глубина остаётся O(log n).

**Сравнительный анализ:**

* **Достоинства:** Благодаря высокой ветвистости (много ключей в узле) высота дерева мала, что минимизирует число дисковых обращений при поиске (важно для внешней памяти). Операции поиска, вставки, удаления работают в среднем эффективно и сохраняют порядок ключей, что важно для последовательной обработки больших данных. Память используется экономно (более 50% объёма страницы) и операции ввода-вывода сведены к минимуму. B-деревья широко применяются для **индексов** в СУБД, файловых системах и хранилищах данных. Например, MySQL, PostgreSQL, SQLite и многие файловые системы (NTFS, Ext4 и др.) используют B-деревья или их вариации (B+-деревья) для быстрого поиска по диску.
* **Недостатки:** Сложность реализации намного выше по сравнению с BST: требуется управление множеством ключей и указателей. При малых данных и в памяти B-дерево неэффективно (большой накладной размер узла). Кроме того, традиционные B-деревья не обладают эффективными обходами по критериям, отличным от ключа (т.е. сложно делать быстрые выборки по «вторичным» атрибутам).

**Источники**

[Красно-чёрное дерево — Википедия](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B0%D1%81%D0%BD%D0%BE-%D1%87%D1%91%D1%80%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE#:~:text=%D0%9A%D1%80%D0%B0%D1%81%D0%BD%D0%BE,%D0%B2%D0%BE%D0%B7%D0%BC%D0%BE%D0%B6%D0%BD%D1%8B%D1%85%20%D0%B7%D0%BD%D0%B0%D1%87%D0%B5%D0%BD%D0%B8%D0%B9%C2%A0%E2%80%94%20%C2%AB%D1%87%D1%91%D1%80%D0%BD%D1%8B%D0%B9%C2%BB%20%D0%B8%D0%BB%D0%B8%20%C2%AB%D0%BA%D1%80%D0%B0%D1%81%D0%BD%D1%8B%D0%B9%C2%BB)

[Splay-дерево — Википедия](https://ru.wikipedia.org/wiki/Splay-%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE#:~:text=%D0%A0%D0%B0%D1%81%D1%88%D0%B8%D1%80%D1%8F%D1%8E%D1%89%D0%B5%D0%B5%D1%81%D1%8F%20%28%D0%B0%D0%BD%D0%B3%D0%BB,%D0%BF%D1%80%D0%B8%20%D0%BA%D0%B0%D0%B6%D0%B4%D0%BE%D0%BC%20%D0%BE%D0%B1%D1%80%D0%B0%D1%89%D0%B5%D0%BD%D0%B8%D0%B8%20%D0%BA%20%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D1%83)

[Красно-черное дерево — Викиконспекты](https://neerc.ifmo.ru/wiki/index.php?title=%D0%9A%D1%80%D0%B0%D1%81%D0%BD%D0%BE-%D1%87%D0%B5%D1%80%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE#:~:text=%D0%9A%D1%80%D0%B0%D1%81%D0%BD%D0%BE,%D0%B4%D0%BB%D1%8F%20%D0%BA%D0%BE%D1%82%D0%BE%D1%80%D0%BE%D0%B3%D0%BE%20%D0%B2%D1%8B%D0%BF%D0%BE%D0%BB%D0%BD%D1%8F%D1%8E%D1%82%D1%81%D1%8F%20%D1%81%D0%BB%D0%B5%D0%B4%D1%83%D1%8E%D1%89%D0%B8%D0%B5%20%D1%81%D0%B2%D0%BE%D0%B9%D1%81%D1%82%D0%B2%D0%B0)

[Introduction to Splay tree data structure | GeeksforGeeks](https://www.geeksforgeeks.org/introduction-to-splay-tree-data-structure/)

[B-дерево — Википедия](https://ru.wikipedia.org/wiki/B-%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE#:~:text=B,7)