Основы построения файловых систем

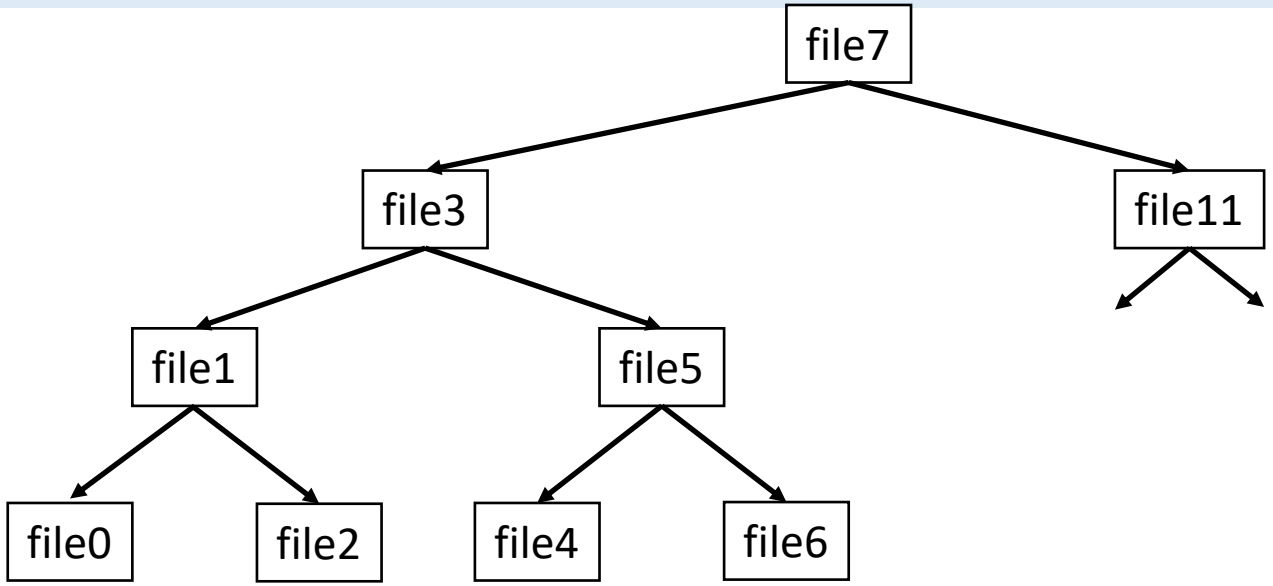
Сегодня мы рассмотрим задачу эффективного хранения списков файлов и экстентов

Напоминание: как организовать список файлов*?

Линейный список, где файлы идут в порядке создания

file15, file1, file2, file3, file4, file9, file6, file8, file7, file5, file12, file11, file10, file13, file14, file0

Дерево поиска



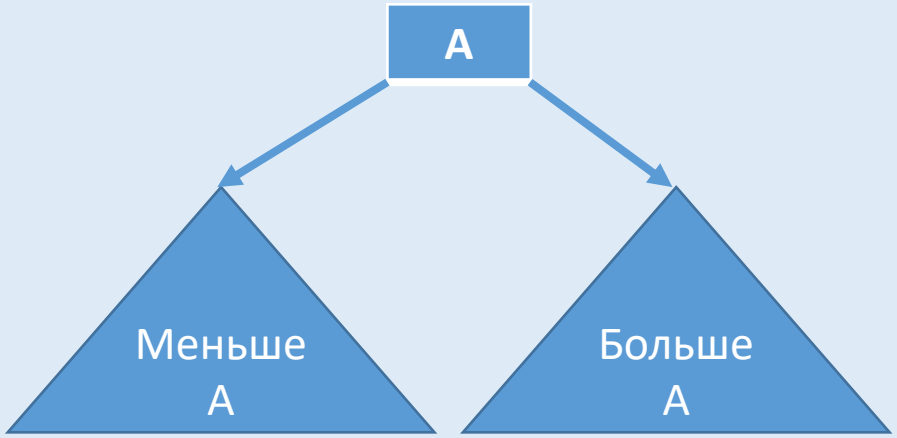
Переход на начало списка:	≈ 10msec	Тут нужны 4 позиционирования читающей головки, т.е. меньше, чем в 40msec мы не уложимся.
Чтение списка:	≈ 1msec (≈1MB)	
Поиск (список уместился в RAM):	<< 1msec	

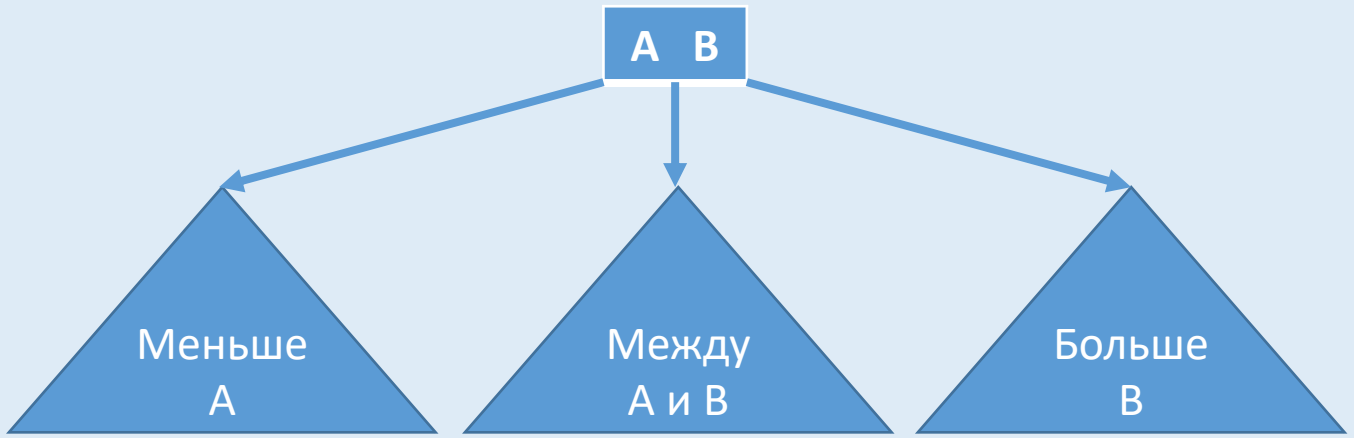
** для простоты пусть каждый прямоугольник на рисунке будет непрерывным блоком на диске*

2-3-деревья и красно-чёрные деревья (напоминание)

2-3-дерево – один из способов хранить множество элементов. Оно определяется следующими свойствами:

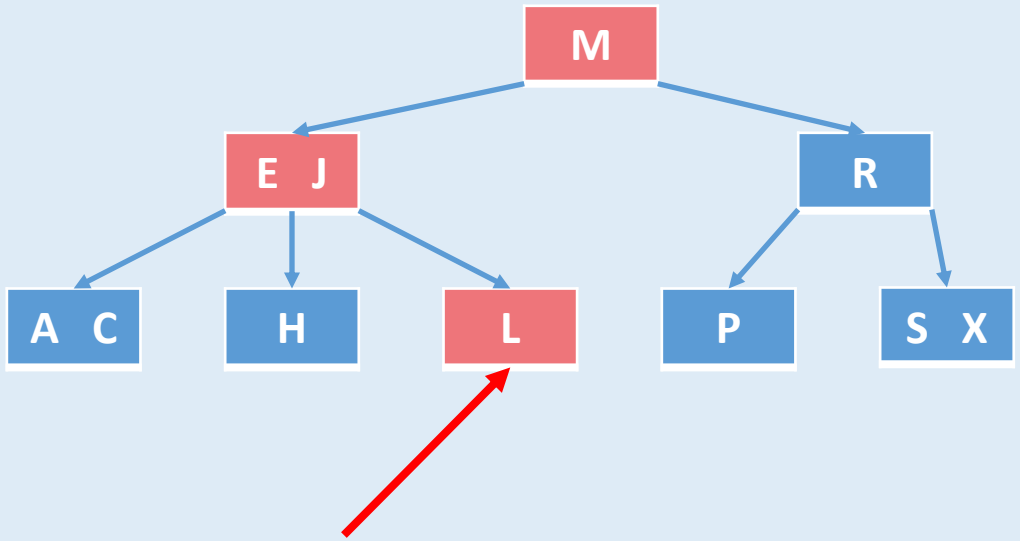
- каждый узел содержит один или два элемента из множества,
- узлы имеют 0, 2 или 3 потомка,
- дерево идеально сбалансировано,
- значения элементов в узлах упорядочены:





2-3-деревья и красно-чёрные деревья (напоминание)

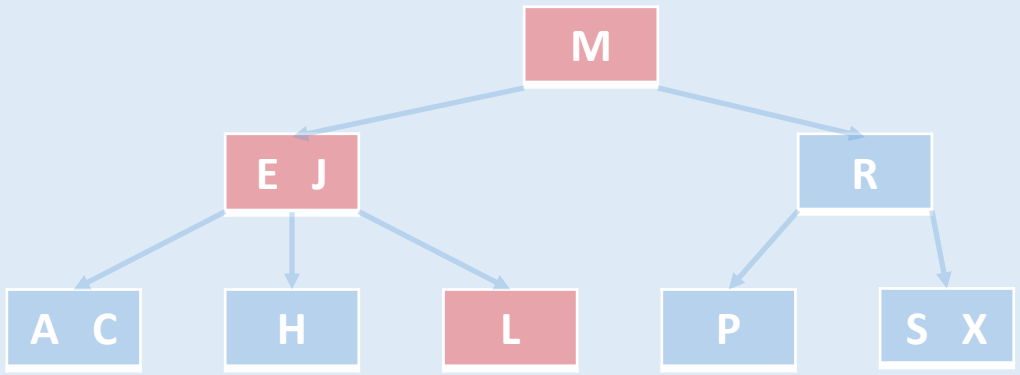
Вставка в узел с одним элементом:

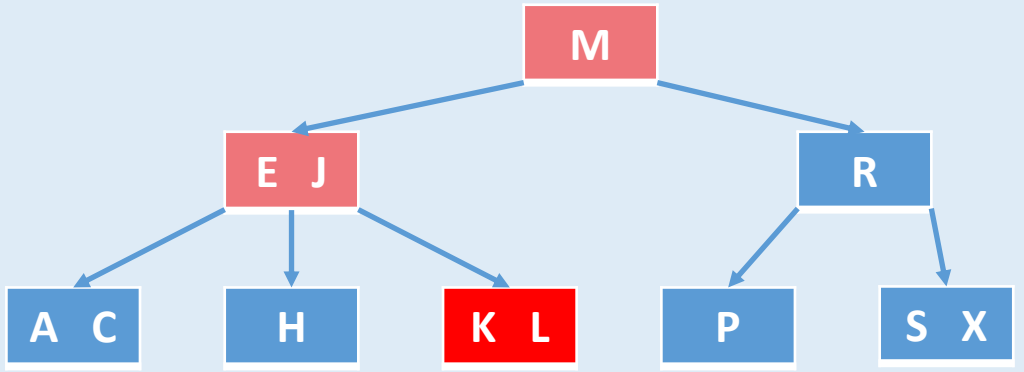


Поиск элемента **K**
закончится тут

2-3-деревья и красно-чёрные деревья (напоминание)

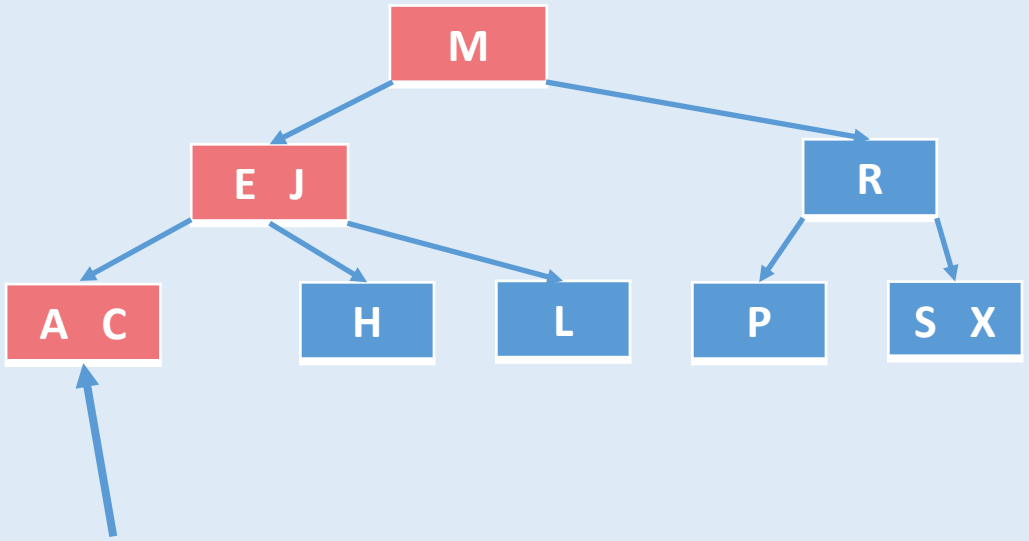
Вставка в узел с одним элементом:





2-3-деревья и красно-чёрные деревья (напоминание)

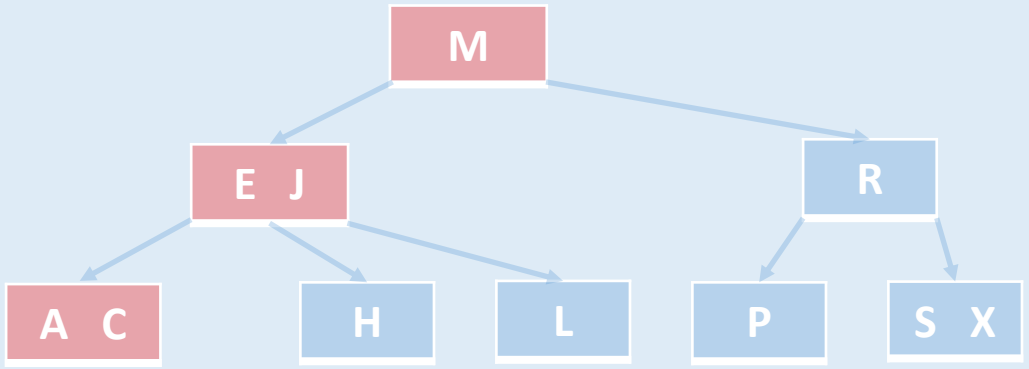
Вставка в узел с двумя элементами:

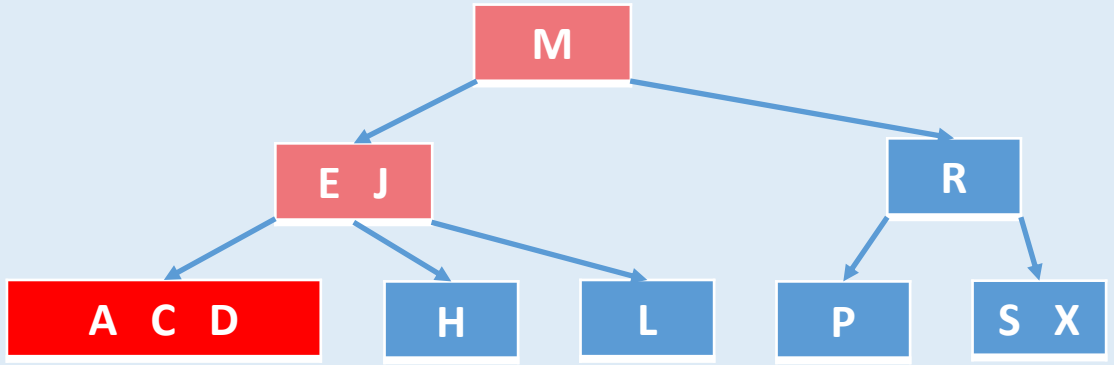


Поиск элемента **D**
закончится тут

2-3-деревья и красно-чёрные деревья (напоминание)

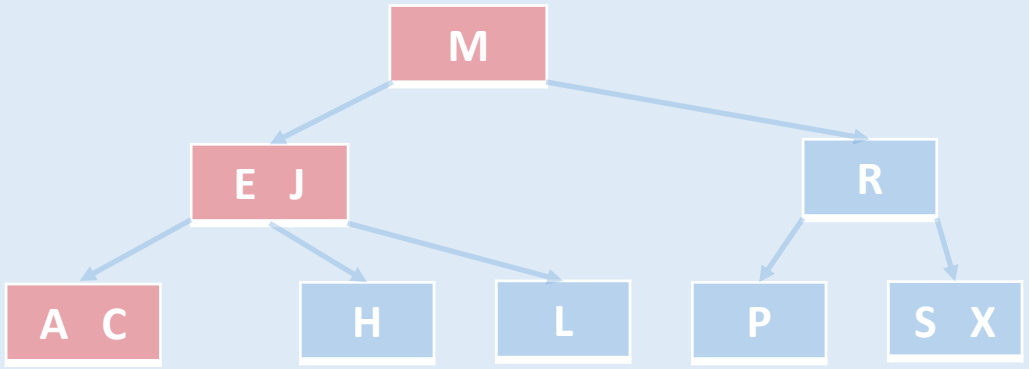
Вставка в узел с двумя элементами:

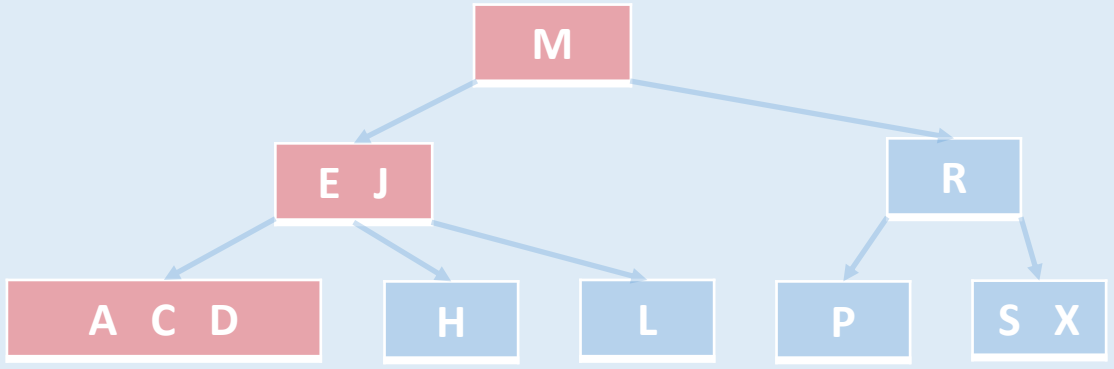


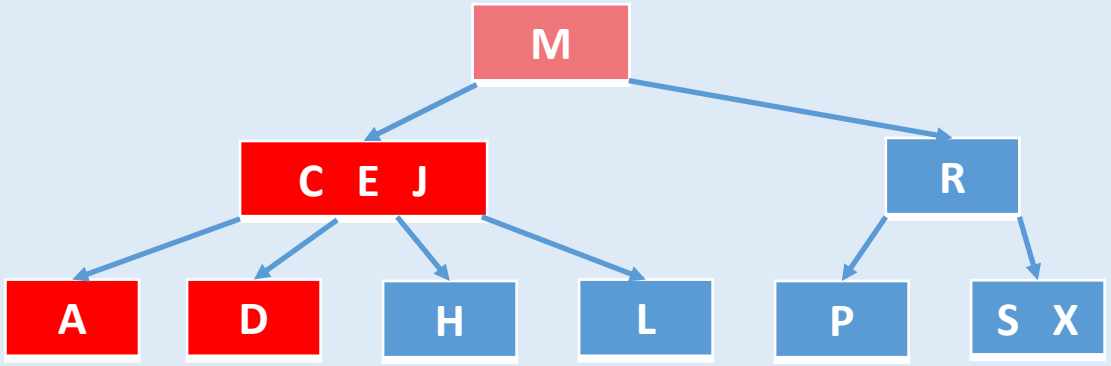


2-3-деревья и красно-чёрные деревья (напоминание)

Вставка в узел с двумя элементами:

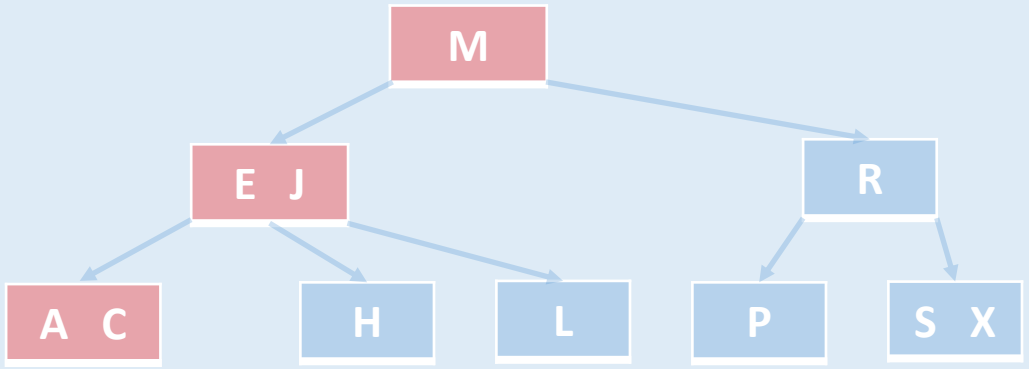


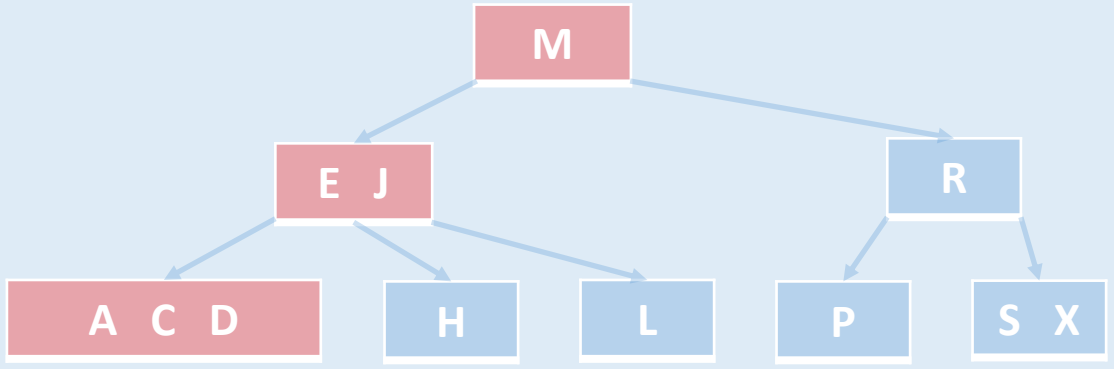


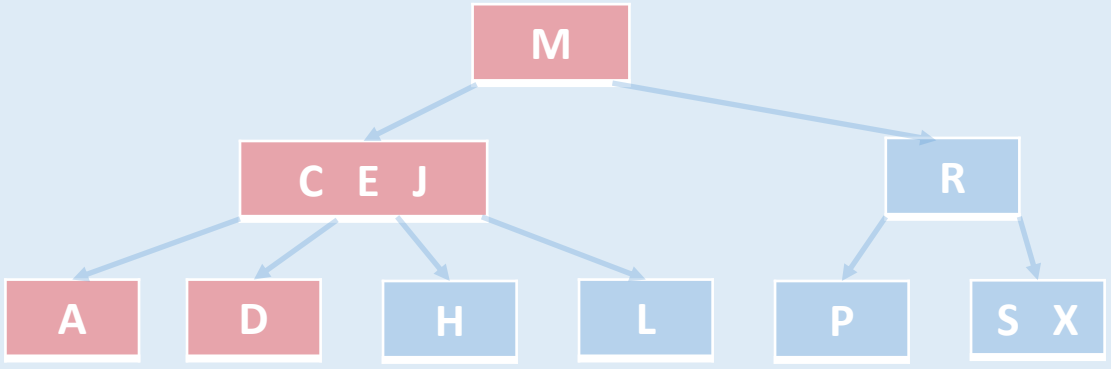


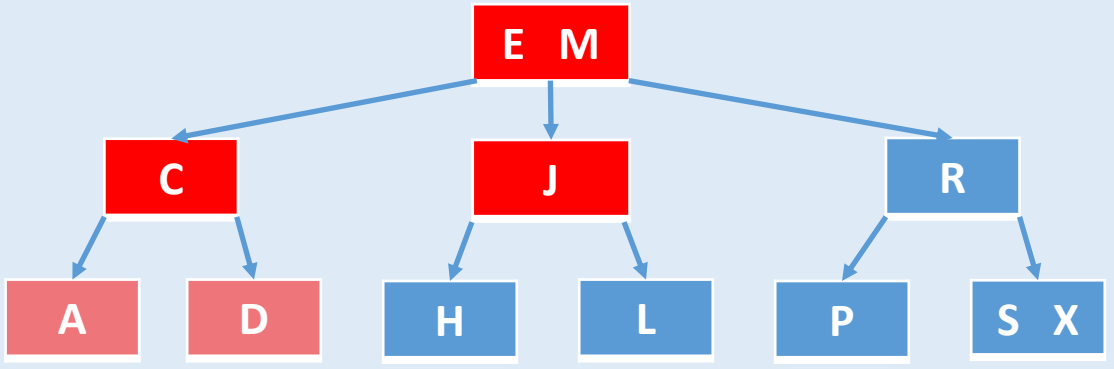
2-3-деревья и красно-чёрные деревья (напоминание)

Вставка в узел с двумя элементами:



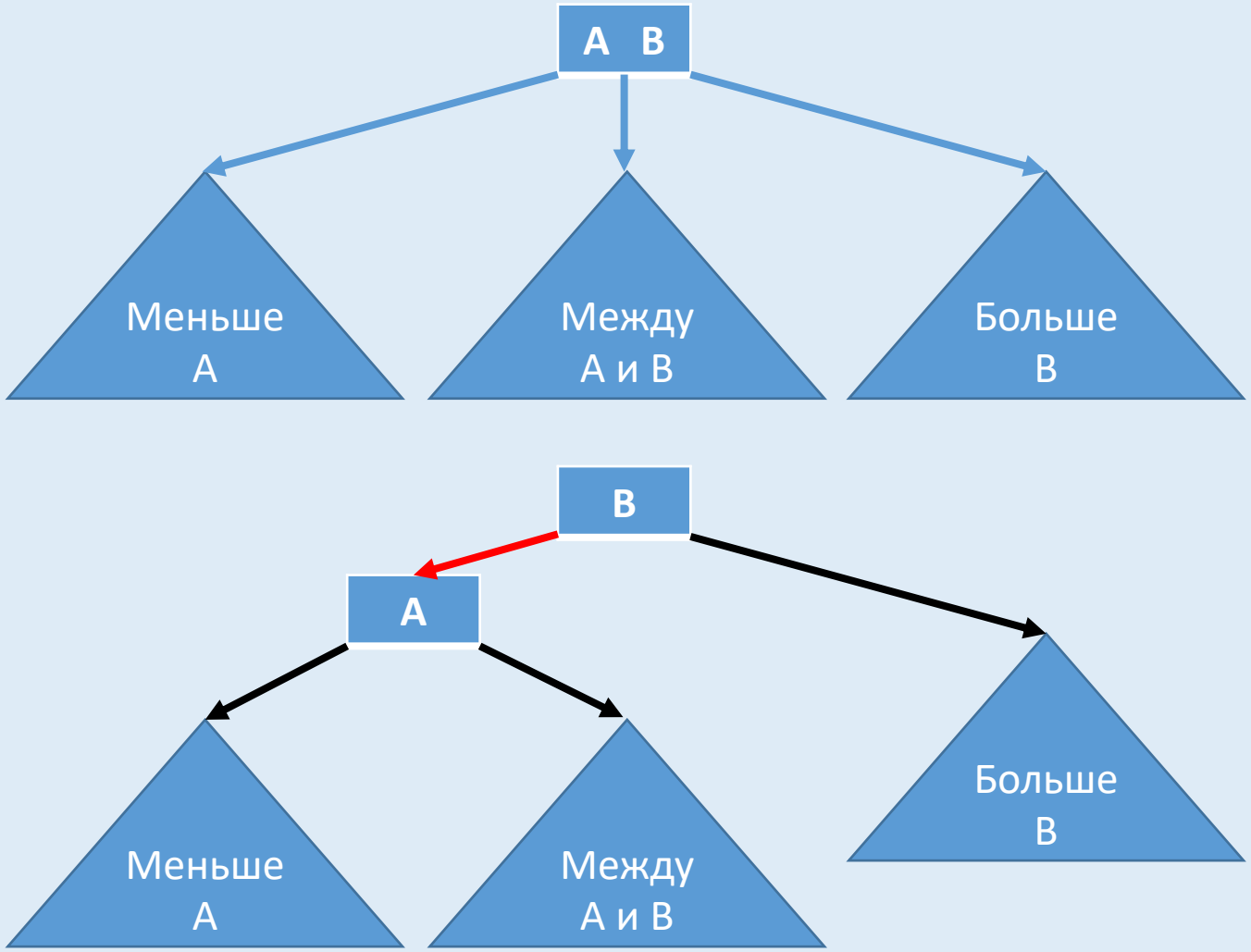






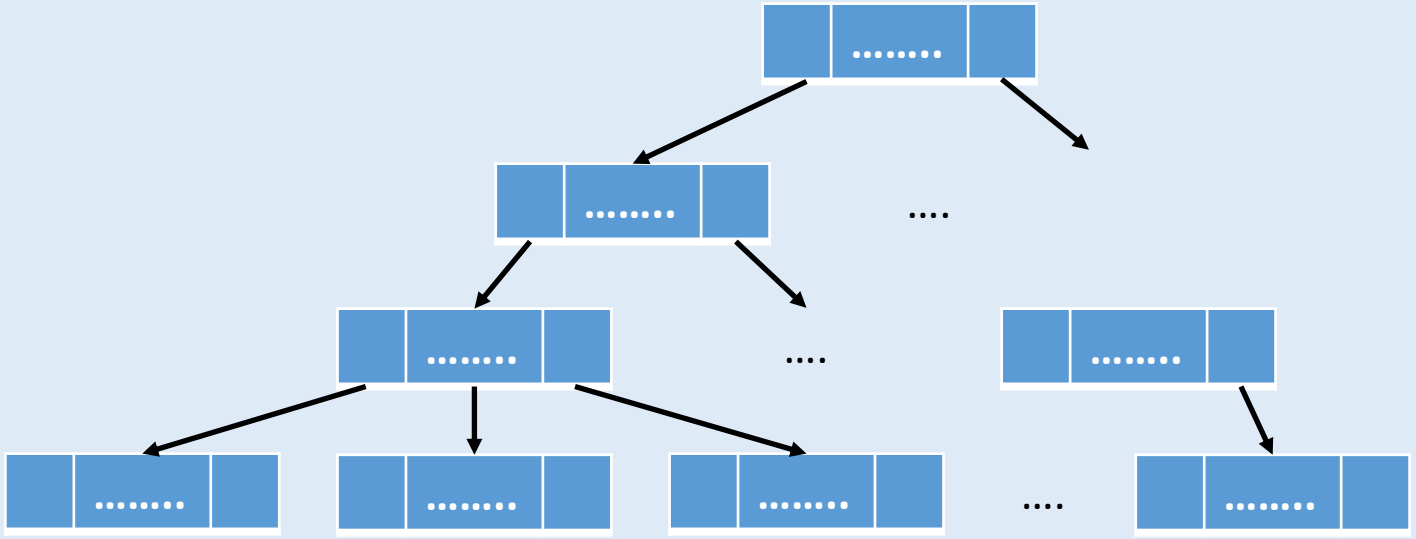
2-3-деревья и красно-чёрные деревья (напоминание)

2-3-деревья взаимно-однозначно соответствуют красно-чёрным:



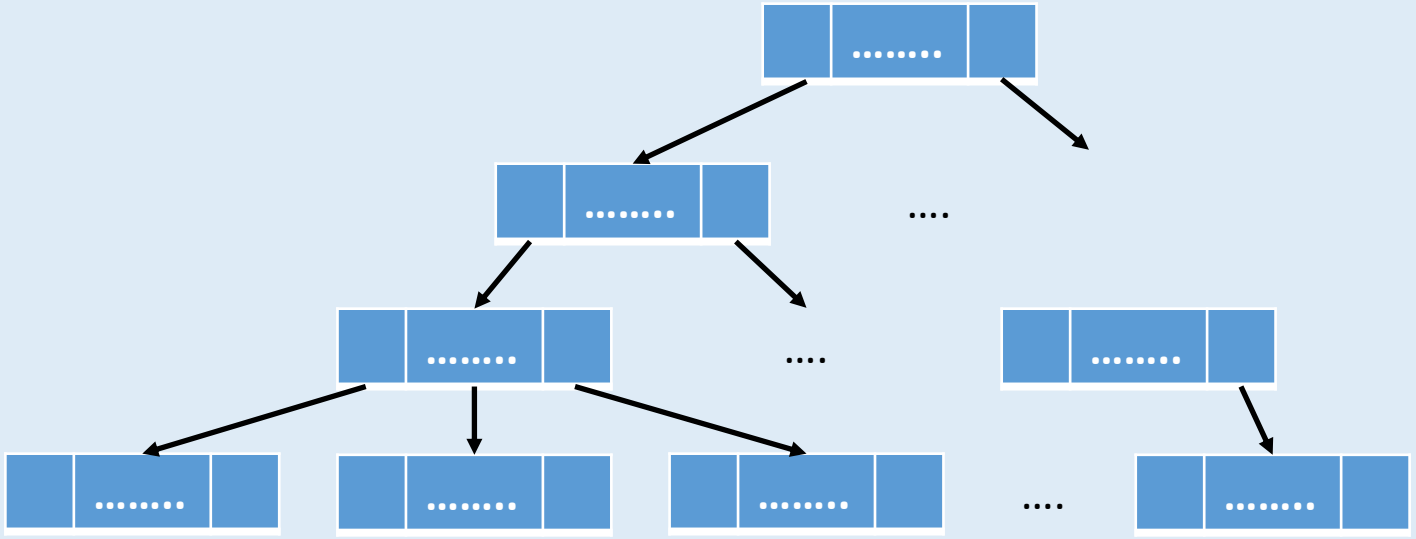
В деталях рассказано здесь:
https://algs4.cs.princeton.edu/33balanced/

B-деревья



- В узлах хранятся массивы пар (k_i, p_i), упорядоченные по возрастанию ключей,
- Массивы имеют ограниченную длину: от L до 2L элементов,
- Указатели на страницы данных хранятся в листьях, во внутренних узлах – ссылки на страницы с узлами-потомками,
- Все листья расположены на одной глубине,
- Указатель p_i ссылается на поддерево с ключами в диапазоне [k_i, k_{i+1}) (внимание: k_{m+1}!),
- Если при вставке происходит переполнение узла, то он разделяется на два узла длины L, а средний элемент перемещается в родительский узел.

B-деревья

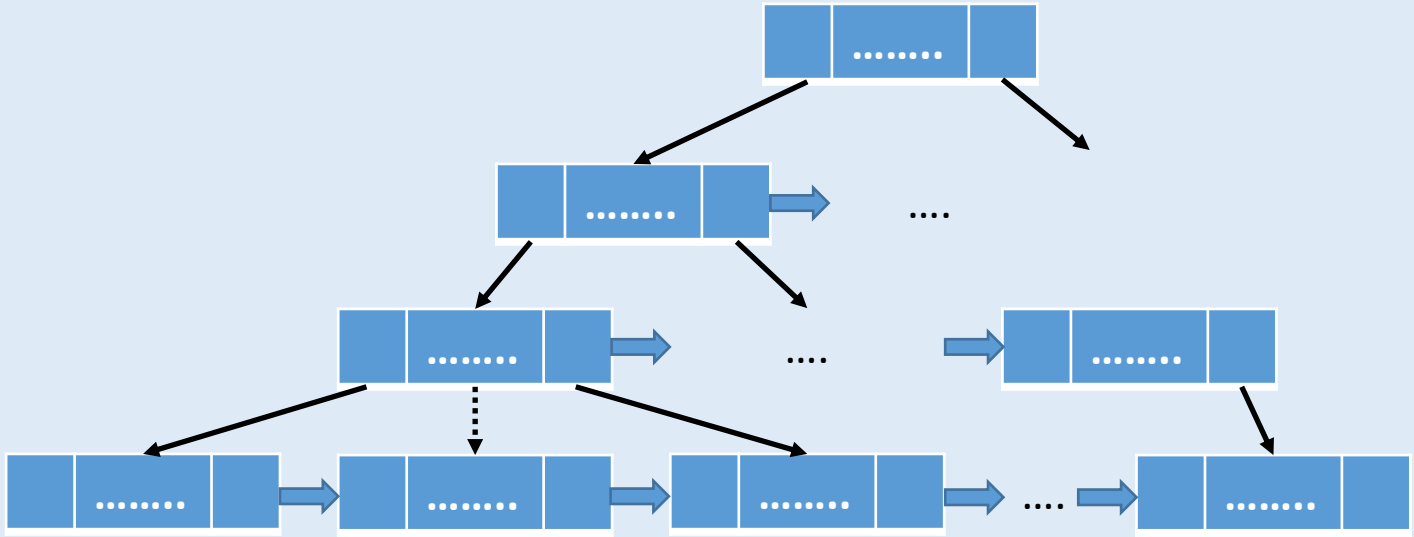


- В узлах хранятся массивы пар (k_i, p_i), упорядоченные по возрастанию ключей,
- Массивы имеют ограниченную длину: от L до 2L элементов,
- Указатели на страницы данных хранятся в листьях, во внутренних узлах – ссылки на страницы с узлами-потомками,
- Все листья расположены на одной глубине,
- Указатель p_i ссылается на поддерево с ключами в диапазоне [k_i, k_{i+1}) (внимание: k_{m+1}!),
- Если при вставке происходит переполнение узла, то он разделяется на два узла длины L, а средний элемент перемещается в родительский узел.

Есть проблемы:

- Вставки и удаления создают случайное IO,
- Удаление зачастую реализуется нетривиально, или оставляет много мусора,
- В многопоточной среде надо брать блокировки сразу на весь путь до листа.

B^{link}-деревья (Lehman, Yao)*



При расщеплении узла не обязательно модифицировать родителя – хватит проставить ссылку на правого соседа, а родительский узел можно модифицировать потом.

В итоге, в каждый момент времени достаточно держать блокировку только на одном узле.

https://www.csd.uoc.gr/~hy460/pdf/p650-lehman.pdf

** Используются, например, в PostgreSQL.*

Слияние двух B-деревьев

Заметим, что два множества, представленные B-деревьями, легко объединить за линейное время:

1. итерирование по листьям даёт элементы в порядке возрастания ключей; отсортированные списки ключей из двух деревьев можно объединить в один, как в сортировке слиянием,
2. отсортированный объединённый список выписываем в страницы, расположенные последовательно,
3. Для каждых 2L-1 подряд идущих страниц-узлов делаем директорную страницу; разные директорные страницы тоже выписываем последовательно,
4. Аналогично создаём директорные страницы более высокого уровня,
5. Повторяем до тех пор, пока не напишем одну директорную страницу, которую объявляем корнем объединённого дерева.

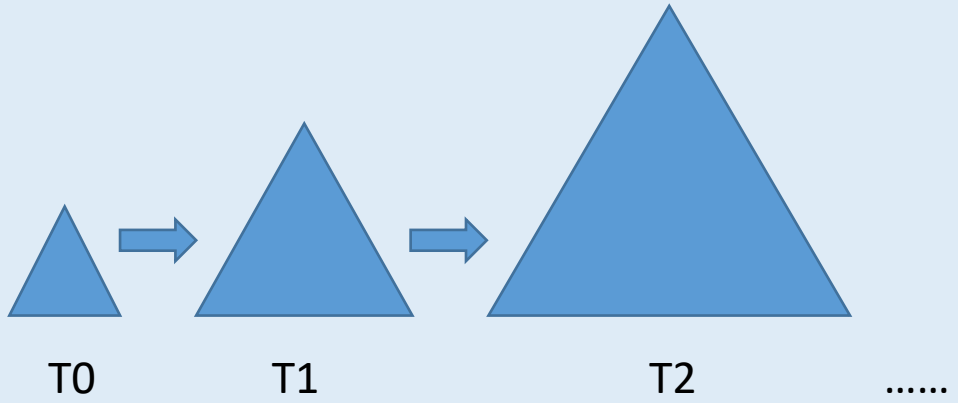
Важные свойства:

- Обход листьев объединяемых деревьев генерирует преимущественно линейное чтение,
- Страницы-листы объединённого дерева выписываются линейно.

Log-Structured Merge Tree

LSM-дерево – это иерархия B-деревьев.

- Поиск элемента делается по очереди в деревьях T_0, T_1, …,
- Вставки делаются только в дерево T_0,
- Дерево T_0 (возможно, несколько первых) располагается в RAM, гарантируя быструю вставку,
- При переполнении дерева T_i оно сливается с деревом T_{i+1}; полученное дерево объявляется новым T_{i+1},
- Удаление элемента реализуется как вставка элемента, помеченного флагом «удалённый»*. Фактическое удаление произойдёт при слиянии деревьев.



Факт: оптимальная производительность вставок в LSM-дерево (наименьшие накладные расходы на слияния) достигается, если число элементов в деревьях T_0, T_1, … образует геометрическую прогрессию.

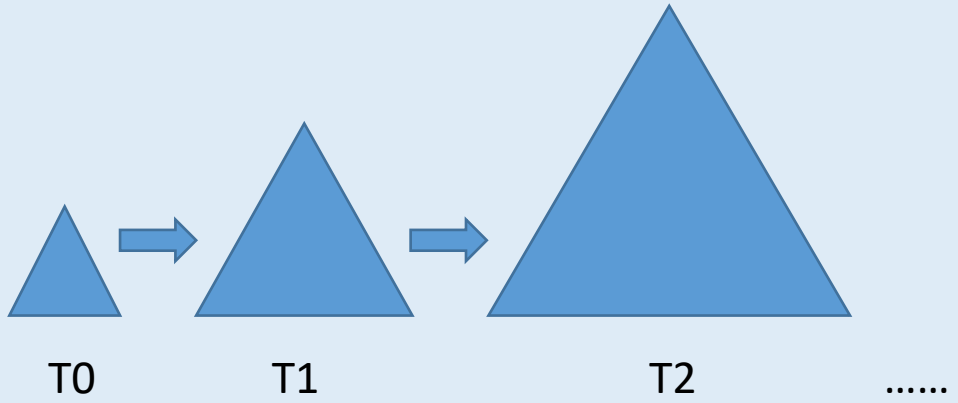
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.44.2782&rep=rep1&type=pdf
http://www.benstopford.com/2015/02/14/log-structured-merge-trees/

** Такой элемент называется “tombstone”.*

Log-Structured Merge Tree

При таком подходе вставки и удаления почти всегда получаются очень быстрыми, поскольку работают только с деревом в RAM.

Но теперь необходимо иногда выполнять слияние деревьев. Как мы обсудили, это можно сделать быстро и генерировать только эффективные последовательности чтений и записей.



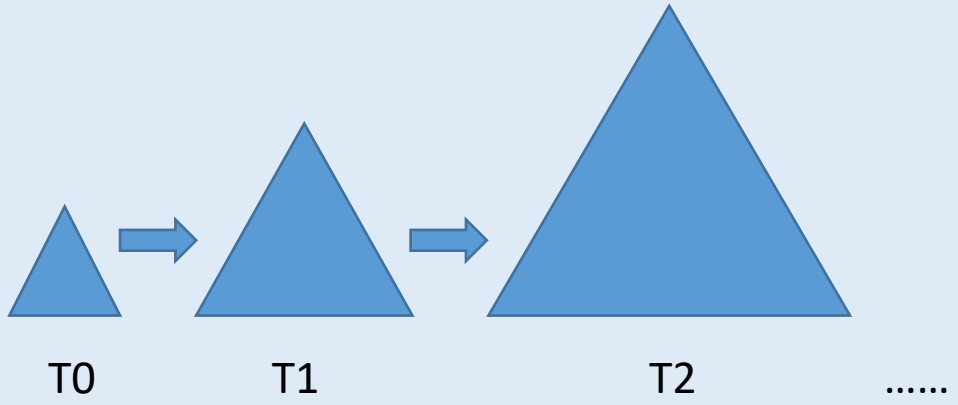
Log-Structured Merge Tree

При таком подходе вставки и удаления почти всегда получаются очень быстрыми, поскольку работают только с деревом в RAM.

Но теперь необходимо иногда выполнять слияние деревьев. Как мы обсудили, это можно сделать быстро и генерировать только эффективные последовательности чтений и записей.

Остаются две большие проблемы:

1. Поиск необходимо выполнять не в одном дереве, а во многих.
2. Write amplification и непредсказуемые задержки: у LSM-дерева мало амортизированное время вставки в дерево, но вставки, которые приводят к слиянию деревьев, будут исполняться на несколько порядков дольше, чем типичные вставки, и создадут много IO.



Bloom filters

Поиск в LSM-дереве приходится реализовывать как несколько поисков по его составляющим разных уровней.

Можно избежать поиска во многих деревьях T_i, если научиться быстро определять, что искомого ключа в T_i не содержится. Это делает фильтр Блума, вероятностная структура данных, которая по множеству и ключу может выдавать ответы

- элемента в множестве нет,
- элемент в множестве может присутствовать.

Bloom filters

Поиск в LSM-дереве приходится реализовывать как несколько поисков по его составляющим разных уровней.

Можно избежать поиска во многих деревьях T_i, если научиться быстро определять, что искомого ключа в T_i не содержится. Это делает фильтр Блума, вероятностная структура данных, которая по множеству и ключу может выдавать ответы

- элемента в множестве нет,
- элемент в множестве может присутствовать.

Конструкция фильтра Блума: пусть имеется битовый массив длиной **m** и **k** независимых хеш-функций $\mathbf{f_i}$, принимающих значения в диапазоне [0, m-1).

- При вставке элемента **x** установим в 1 биты, стоящие на местах $f_1(x)$, $f_2(x)$, … $f_k(x)$,
- Для проверки отсутствия элемента **y** проверим, установлены ли биты на позициях $f_1(y)$, $f_2(y)$, …, $f_k(y)$.

Если элементы x берутся из множества мощностью **N** и вероятность неправильного ответа «может присутствовать» не должна превышать **p**, то для построения фильтра надо взять

$$k >= -\log_2(p)$$

хеш-функций и битовый массив длины

$$m >= k*N / \ln(2)$$

B^ϵ-деревья

Как и в B-деревьях, узлы являются достаточно длинными блоками, но теперь они разделяются на две части:

- pivots (пары ключ-значение с указателями на пользовательские данные или на другие узлы дерева),
- commands (журнал вставок и удалений).

Узлы длиной B байт делятся в пропорции B^ϵ байт на pivots и $B - B^\epsilon$ байт на commands.

- Вставки и удаления добавляются только в журнал корневого узла,
- При переполнении журнала в корне он выталкивается в журналы дочерних узлов, причём выталкиваются только модификации наиболее изменённых поддеревьев.

Преимущества такой реализации:

- Нет необходимости поиска во многих деревьях или построения фильтров Блума,
- Журнал изменений расположен в корневом узле, который всегда в кеше,
- При расщеплении журнала изменений генерируется меньше IO, чем при слиянии компонент LSM-дерева,
- Если записи журнала поддерживать упорядоченными, то можно реализовать range queries, -- запросы диапазонов ключей
- Размер узлов B^ϵ-деревьев можно делать много больше, чем у B-деревьев, что уменьшает их глубину.

https://www.usenix.org/system/files/login/articles/login_oct15_05_bender.pdf

Домашнее задание

1. Написать B-дерево, которое хранит пары из 64-битного ключа и 64-битного значения. Удаление сделать как вставку маркера удаления.
2. Написать функцию для слияния двух B-деревьев.
3. Померять скорость вставки элементов со случайными ключами.