The basics of file systems

Today we will see some ways to store lists of files and extents efficiently.

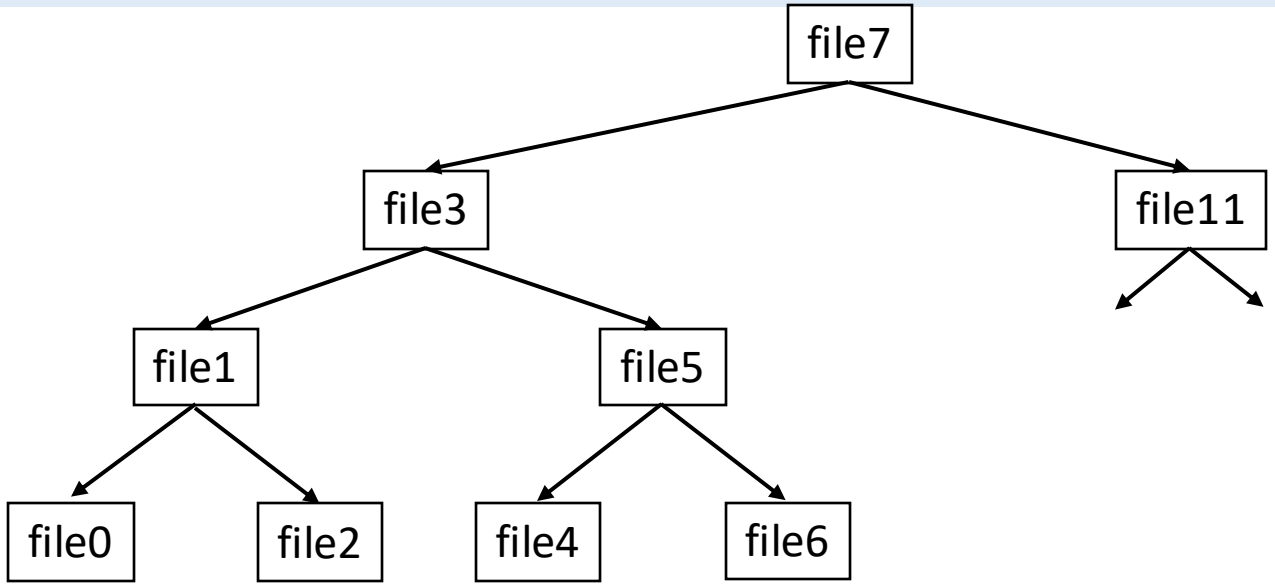
Warning: choosing parameters of B-trees and LSM trees is very dependent on your task at hand. We will only make an overview.

A reminder: how does one store a list of files*?

An array with file names, unsorted:

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

A balanced binary search tree:



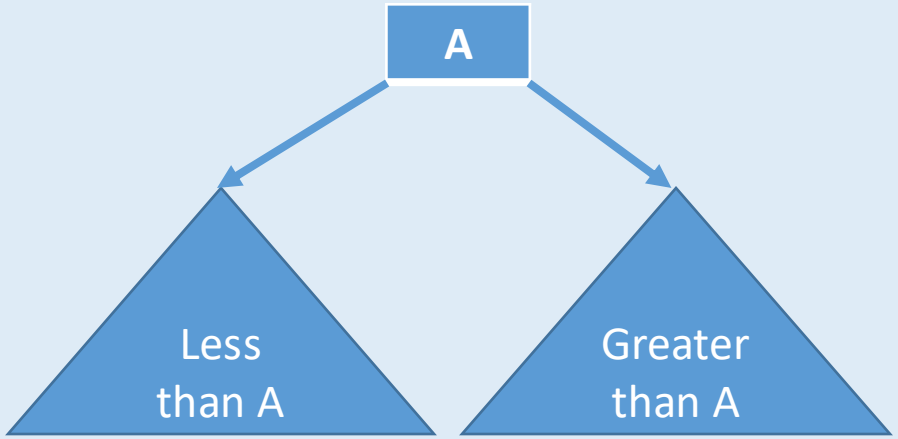
Jump to the start of the list:	≈10msec	Most files need 4 or 3 random accesses which translates to latencies above 30 msec.
Read the whole list:	<1msec (<100K)	
Scan the array in RAM:	<1msec	

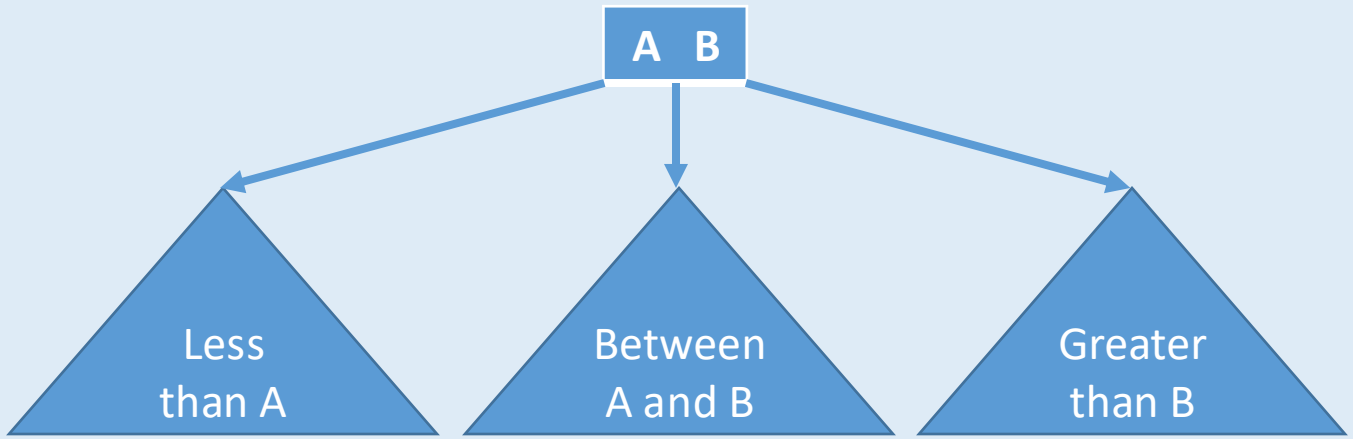
** boxes in diagrams depict contiguous areas of a disk; different boxes are assumed not to be adjacent*

2-3-trees and red-black trees (a reminder)

2-3-tree is a way to represent a set of comparable items. It is defined to have the following properties:

- every node contains one or two items from the set,
- every node has 0, 2 or 3 child nodes,
- the tree is fully balanced which means all leaf nodes are equidistant from the root,
- the tree is a search tree:

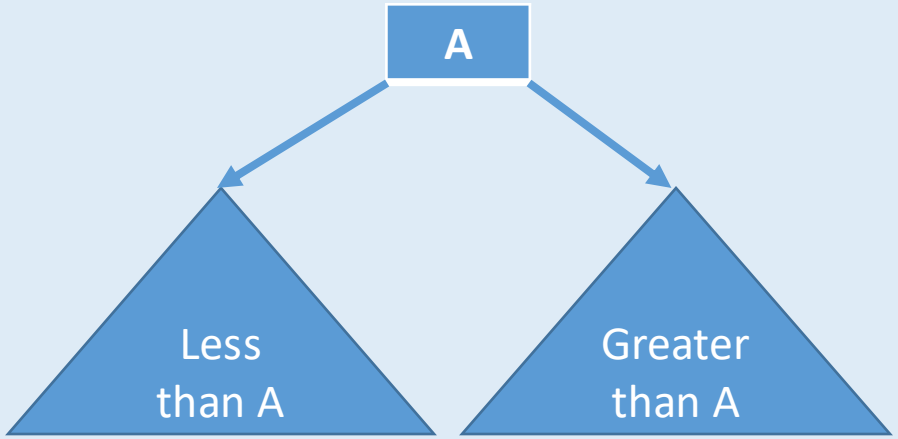


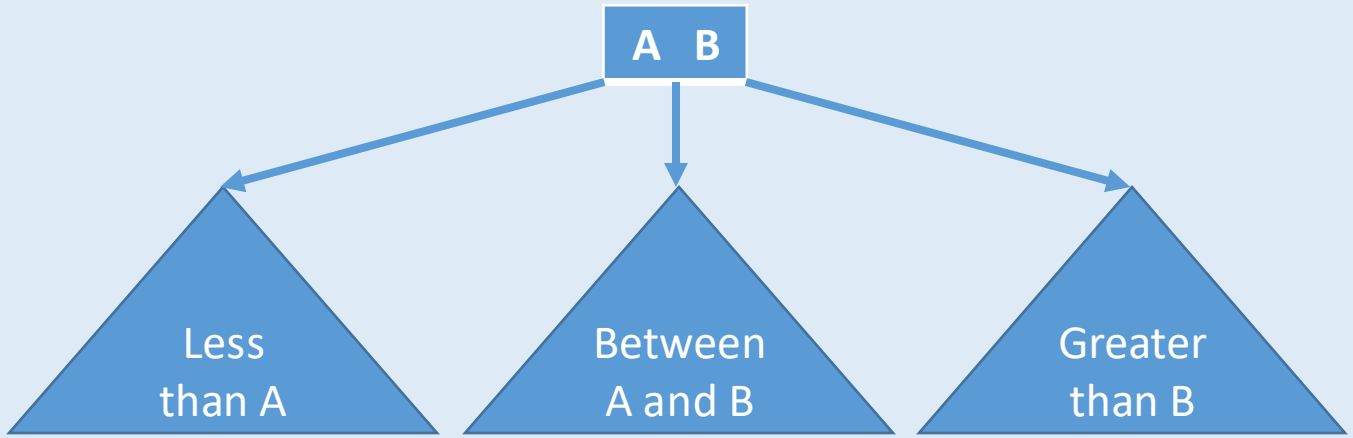


2-3-trees and red-black trees (a reminder)

2-3-tree is a way to represent a set of comparable items. It is defined to have the following properties:

- every node contains one or two items from the set,
- every node has 0, 2 or 3 child nodes,
- the tree is fully balanced which means all leaf nodes are equidistant from the root,
- the tree is a search tree:

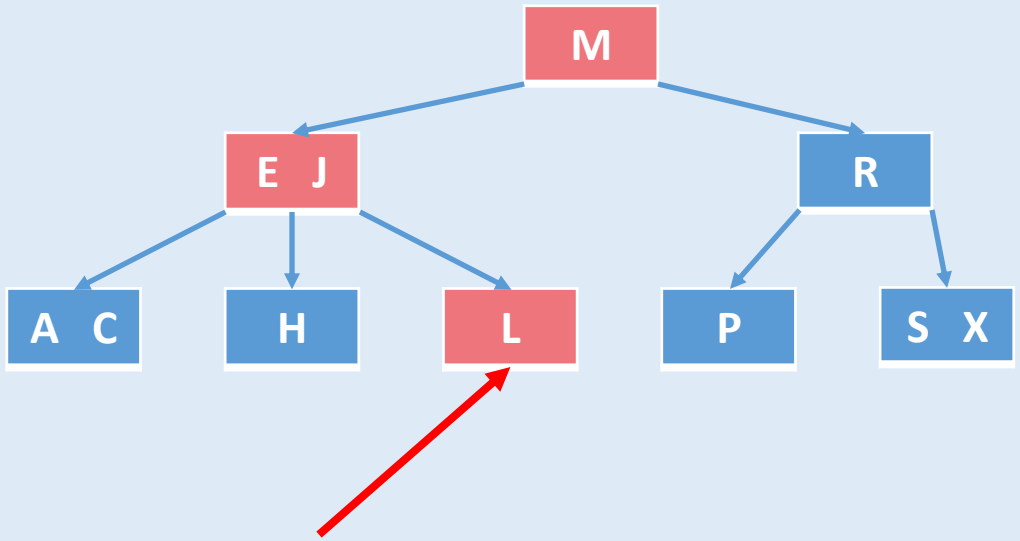




Quiz: if we know how to represent an ordered set A as a tree, how de we represent a map from A to B?

2-3-trees and red-black trees (a reminder)

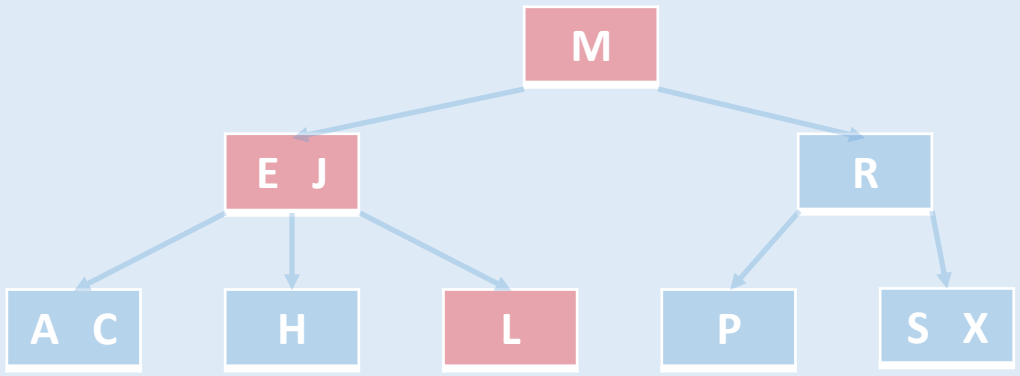
Inserting an item into a 1-node:

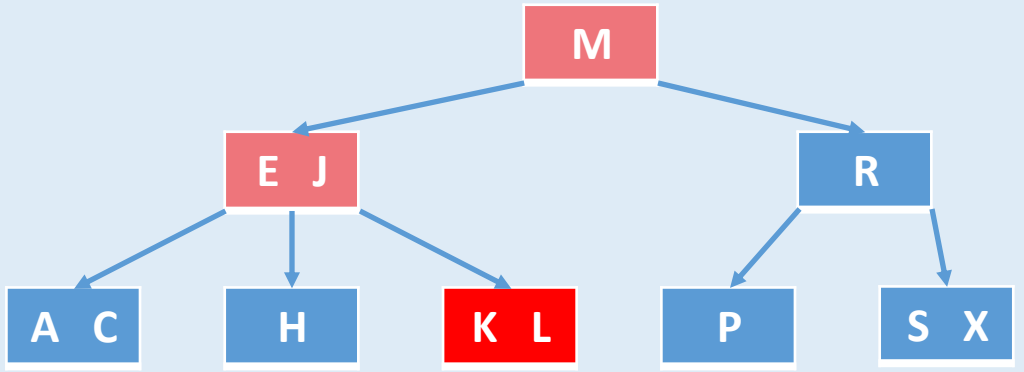


The search for **K**
stops here

2-3-trees and red-black trees (a reminder)

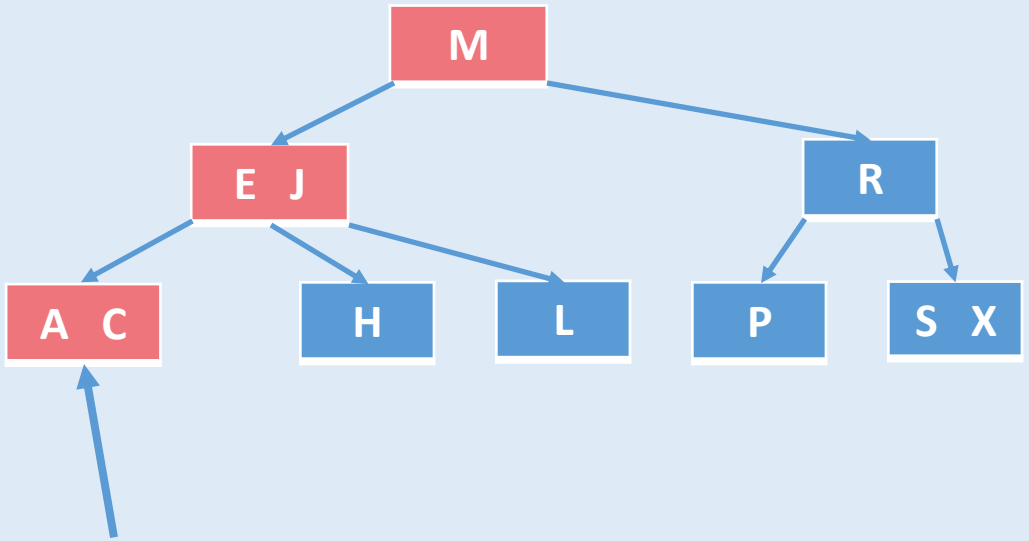
Inserting an item into a 1-node:





2-3-trees and red-black trees (a reminder)

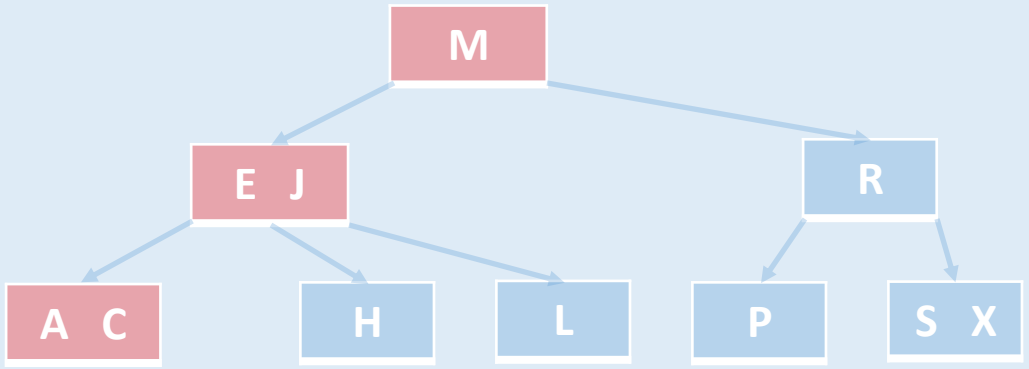
Inserting an item into a 2-node:

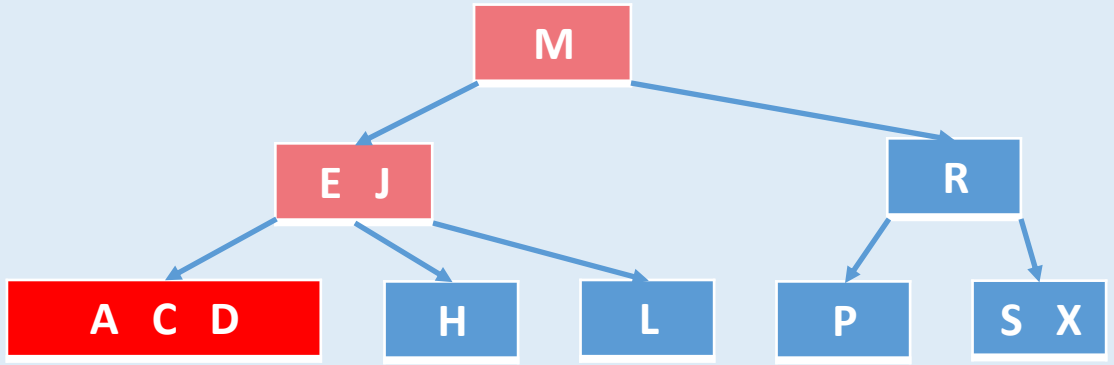


The search for **D** stops here

2-3-trees and red-black trees (a reminder)

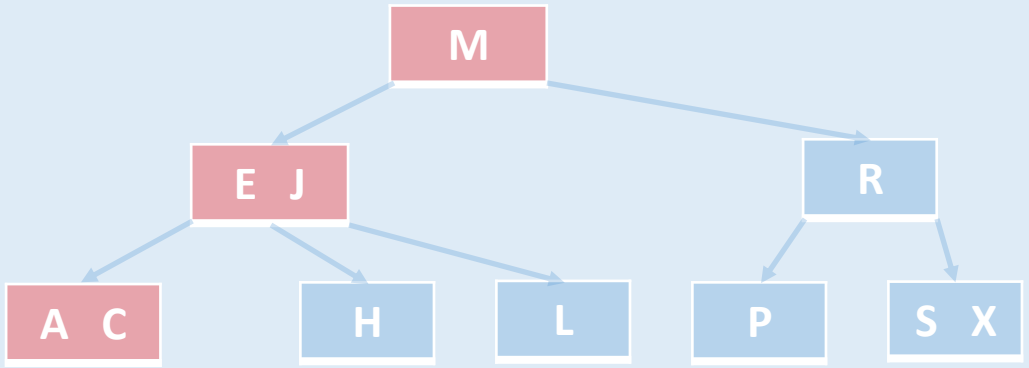
Inserting an item into a 2-node:

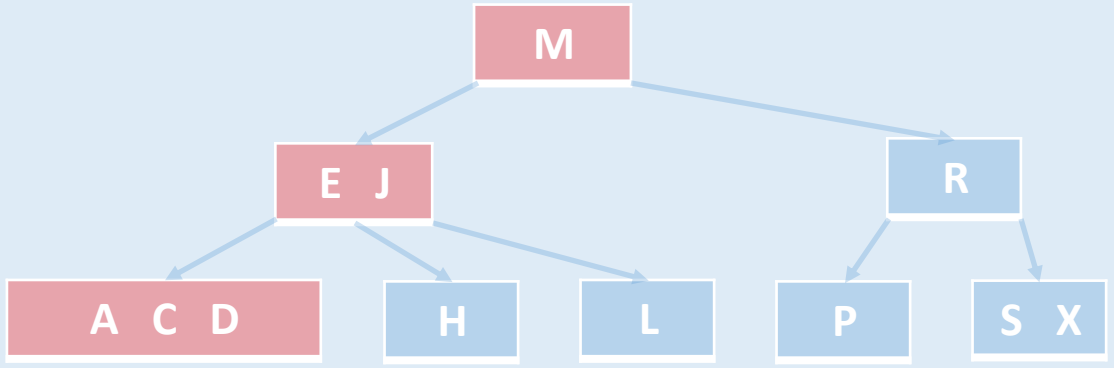


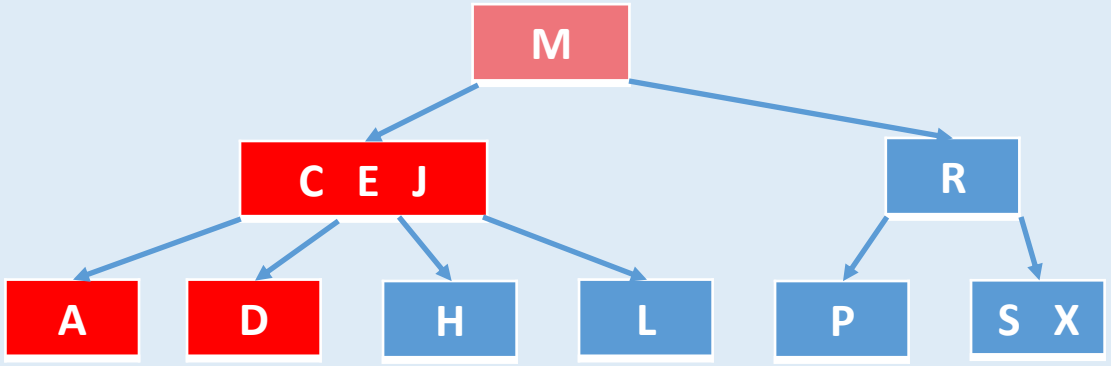


2-3-trees and red-black trees (a reminder)

Inserting an item into a 2-node:

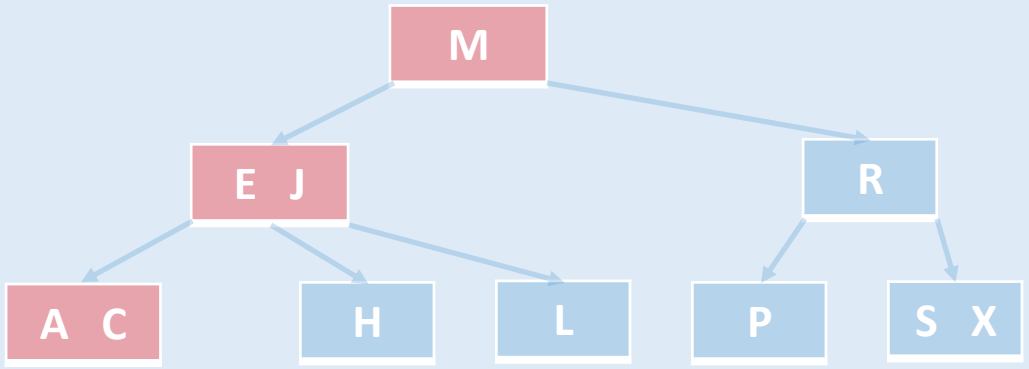


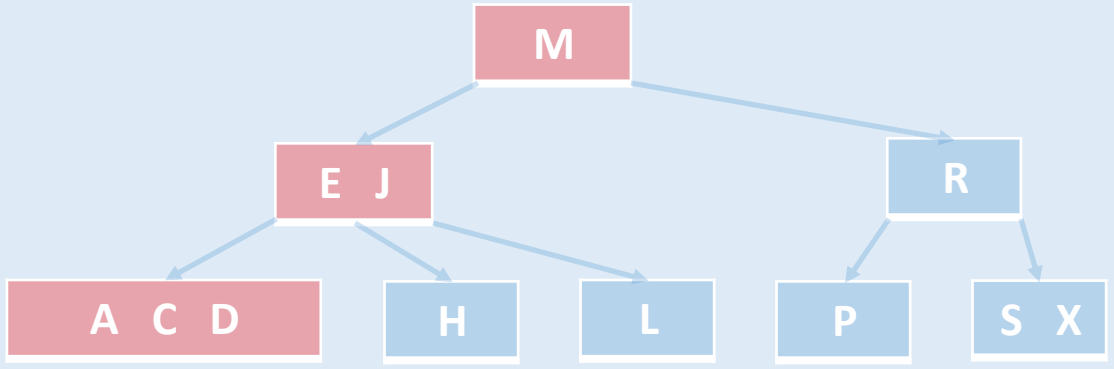


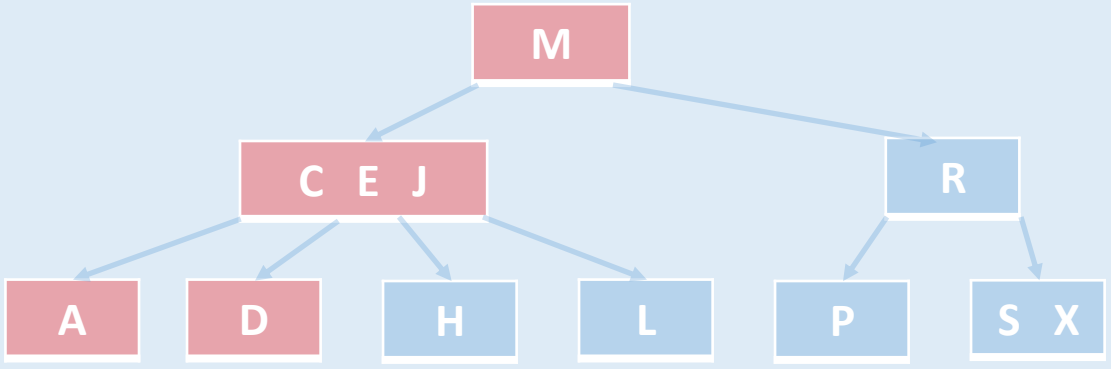


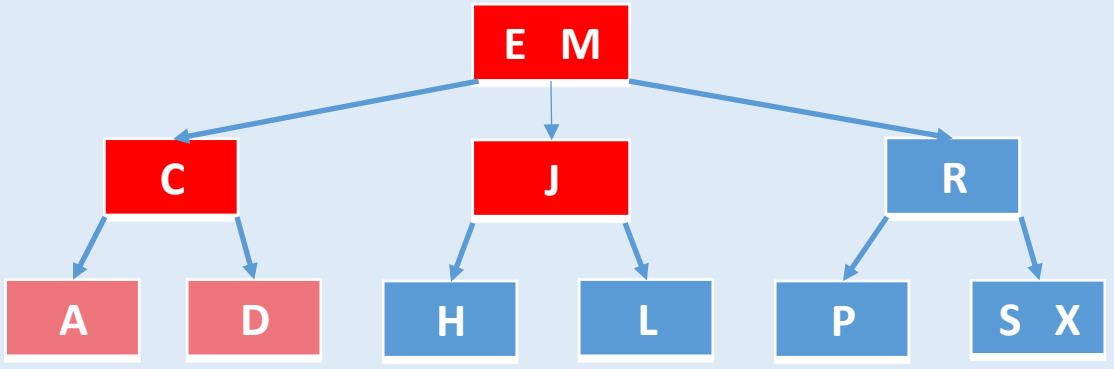
2-3-trees and red-black trees (a reminder)

Inserting an item into a 2-node:



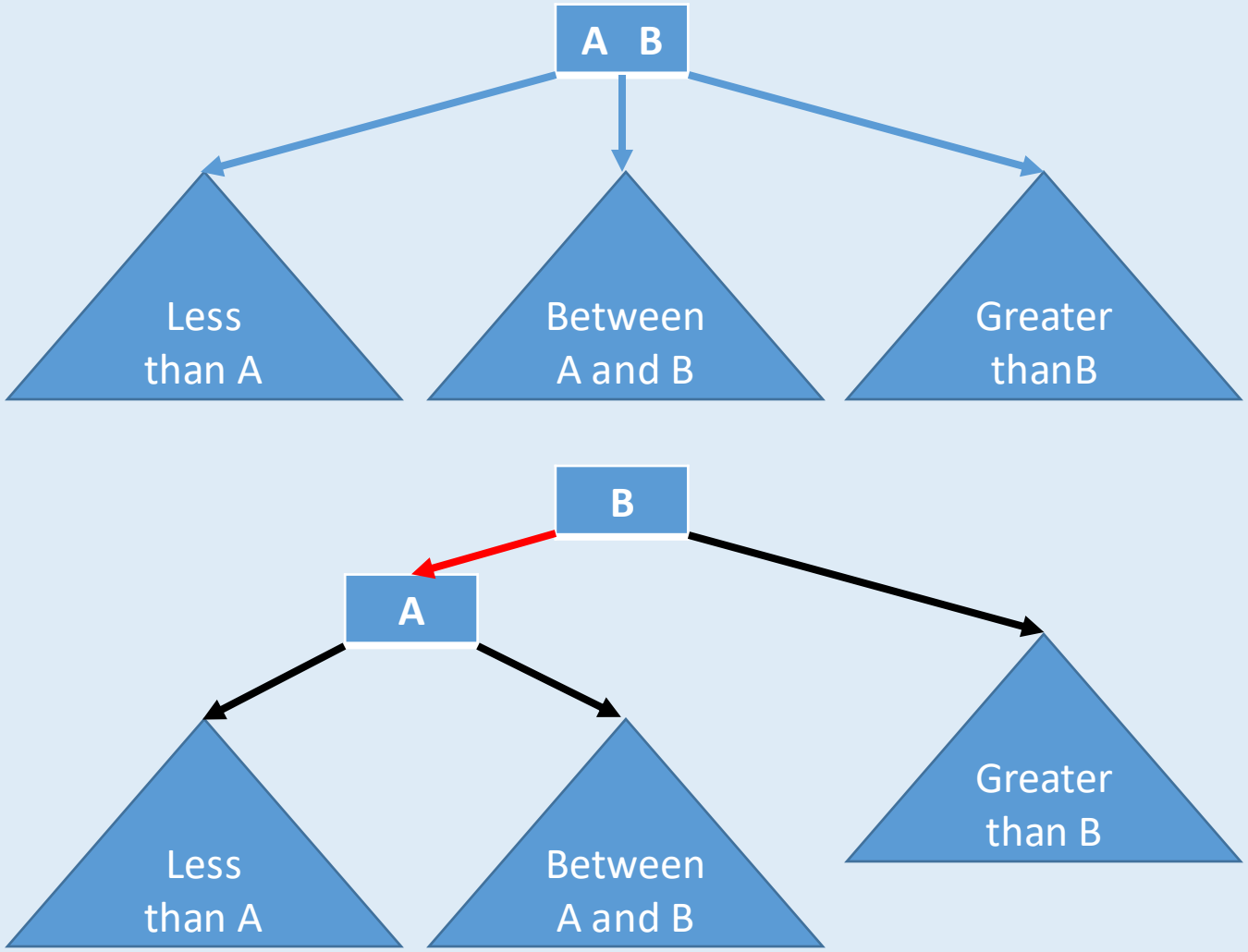






2-3-trees and red-black trees (a reminder)

2-3-trees bijectively map to red-black trees:



See the details here:
https://algs4.cs.princeton.edu/33balanced/

B-trees: a bit of motivation

2-3-trees have a higher branching factor than BST, hence their height is lower.

1. Lower height is beneficial because it implies fewer random IOs when moving from the root to a leaf.
2. However, nodes need more storage space and take more time to read and write.

#2 is not an issue.

B-trees: a bit of motivation

2-3-trees have a higher branching factor than BST, hence their height is lower.

1. Lower height is beneficial because it implies fewer random IOs when moving from the root to a leaf.
2. However, nodes need more storage space and take more time to read and write.

#2 is not an issue. Consider a disk with 4k sectors. Reading 16 bytes, 128 bytes or 4096 bytes takes the same time and amount of RAM.

B-trees: a bit of motivation

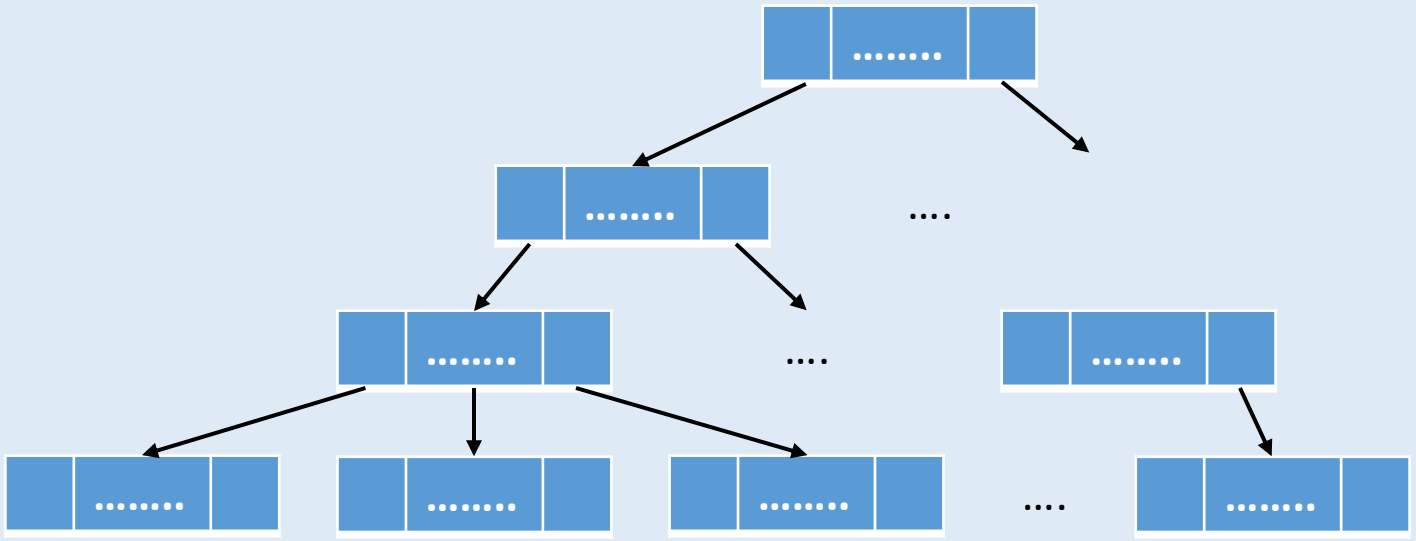
2-3-trees have a higher branching factor than BST, hence their height is lower.

1. Lower height is beneficial because it implies fewer random IOs when moving from the root to a leaf.
2. However, nodes need more storage space and take more time to read and write.

#2 is not an issue. Consider a disk with 4k sectors. Reading 16 bytes, 128 bytes or 4096 bytes takes the same time and amount of RAM.

The RAM behaves in a very similar way. A processor reads and writes only whole cache lines which turns the RAM into a block device with the sector size of 64 bytes on x86-64.

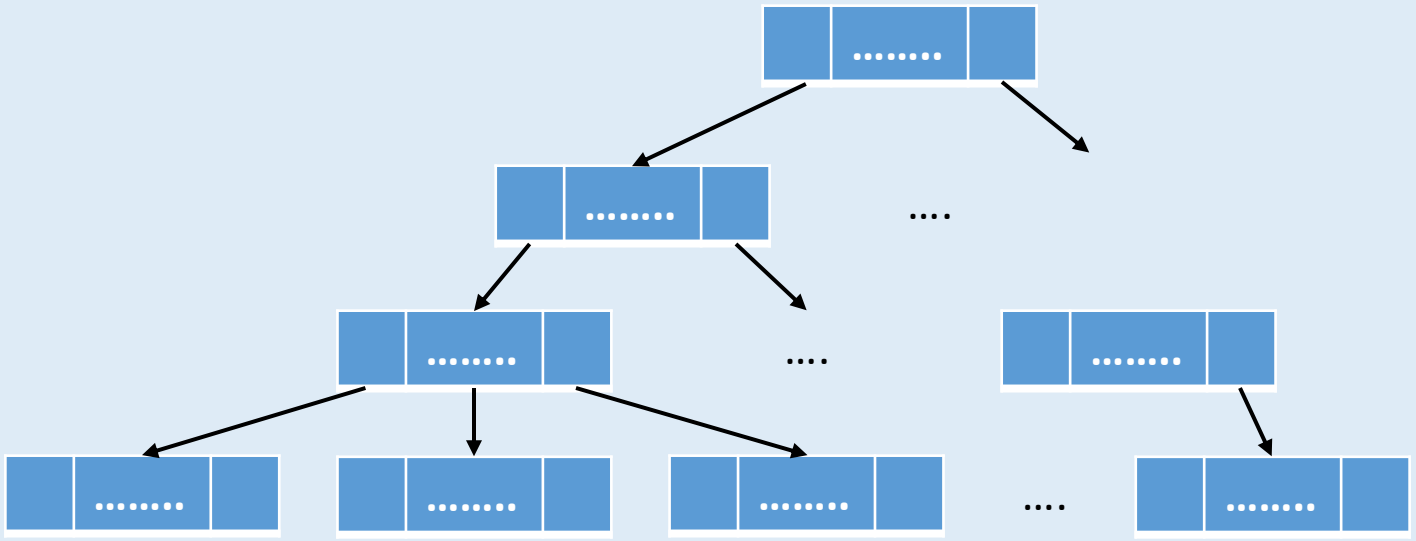
B-trees



The definition of a B-tree the same as that of a 2-3-tree. However, we allow more elements in a node.

- Nodes contain arrays of pairs (k_i, p_i) that are ordered by their key,
- Pointers in leaf nodes point at user data, pointers in internal nodes point to tree nodes in the next level,
- Nodes contain L elements at least and 2L elements at most,
- All leaf nodes are at the same tree level,
- The pointer p_i in an internal node points to a subtree with keys in the range $[k_{i-1}, k_i)$ (be careful with k_0!),
- If insertion produces a node with 2L+1 elements, then the head L items and tail L items are split into their nodes, and the median item is moved to the parent.

B-trees

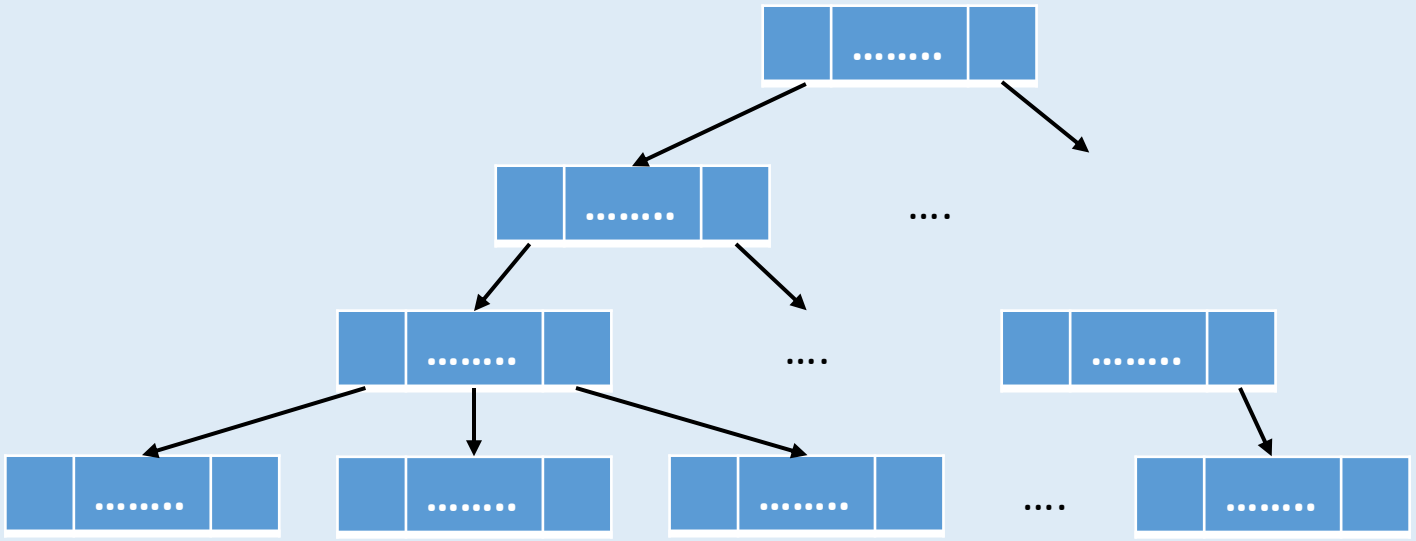


The definition of a B-tree the same as that of a 2-3-tree. However, we allow more elements in a node.

- Nodes contain arrays of pairs (k_i, p_i) that are ordered by their key,
- Pointers in leaf nodes point at user data, pointers in internal nodes point to tree nodes in the next level,
- Nodes contain L elements at least and 2L elements at most,
- All leaf nodes are at the same tree level,
- The pointer p_i in an internal node points to a subtree with keys in the range $[k_{i-1}, k_i)$ (be careful with k_0!),
- If insertion produces a node with 2L+1 elements, then the head L items and tail L items are split into their nodes, and the median item is moved to the parent.

Deciding which subtree to descend into is naturally implemented with a binary search because keys in a node are sorted. However, implementations of B-trees often use linear search instead.
Quiz: why is that often a good idea?

B-trees



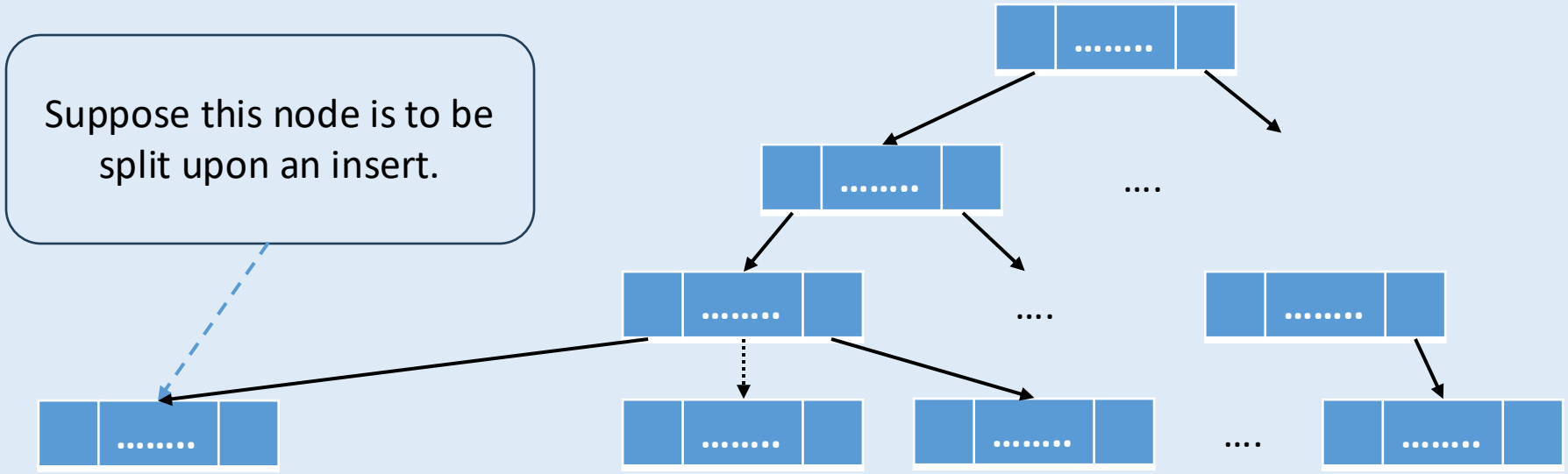
The definition of a B-tree the same as that of a 2-3-tree. However, we allow more elements in a node.

- Nodes contain arrays of pairs (k_i, p_i) that are ordered by their key,
- Pointers in leaf nodes point at user data, pointers in internal nodes point to tree nodes in the next level,
- Nodes contain L elements at least and 2L elements at most,
- All leaf nodes are at the same tree level,
- The pointer p_i in an internal node points to a subtree with keys in the range [k_{i-1}, k_i) (be careful with k_0!),
- If insertion produces a node with 2L+1 elements, then the head L items and tail L items are split into their nodes, and the median item is moved to the parent.

Some obvious problems to deal with:

- Inserting an element produces random IO if nodes need to be split.
- Removing an item needs complex rebalancing, or produces garbage.
- A multithreaded writer to a B-tree needs to lock all nodes going from the root to the leaf. Essentially, it a global lock.

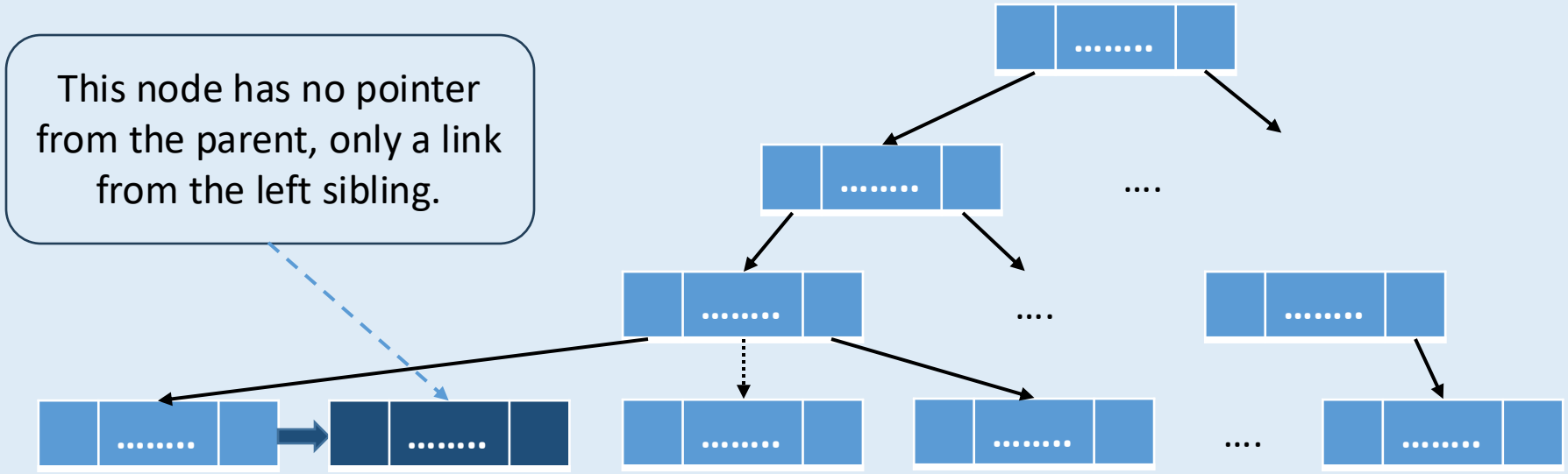
B^{link}-trees (Lehman, Yao)*



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

** Postgres uses these.*

B^{link}-trees (Lehman, Yao)*



When splitting a node, one need not modify the parent immediately. We can augment a node with a pointer to the sibling on the right. Essentially, it implements nodes have more than 2L elements as a list of properly sized nodes.

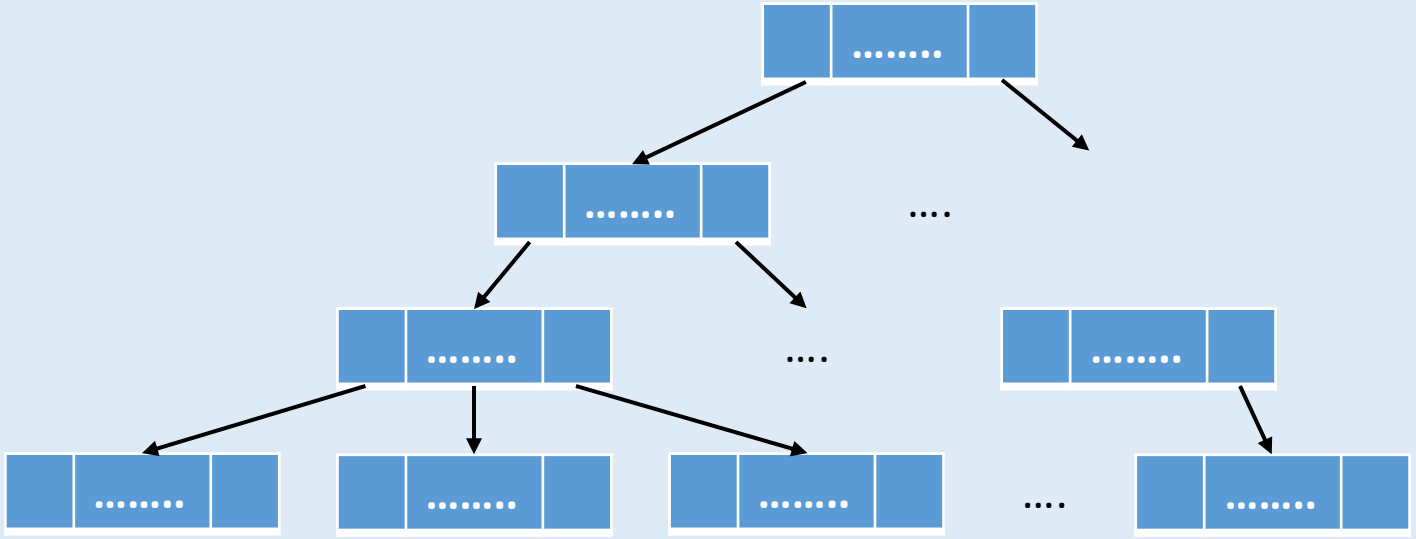
A background process later updates parent nodes to add pointers to all nodes produced by splitting.

This way it suffices to write-lock only one node when inserting a value.

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

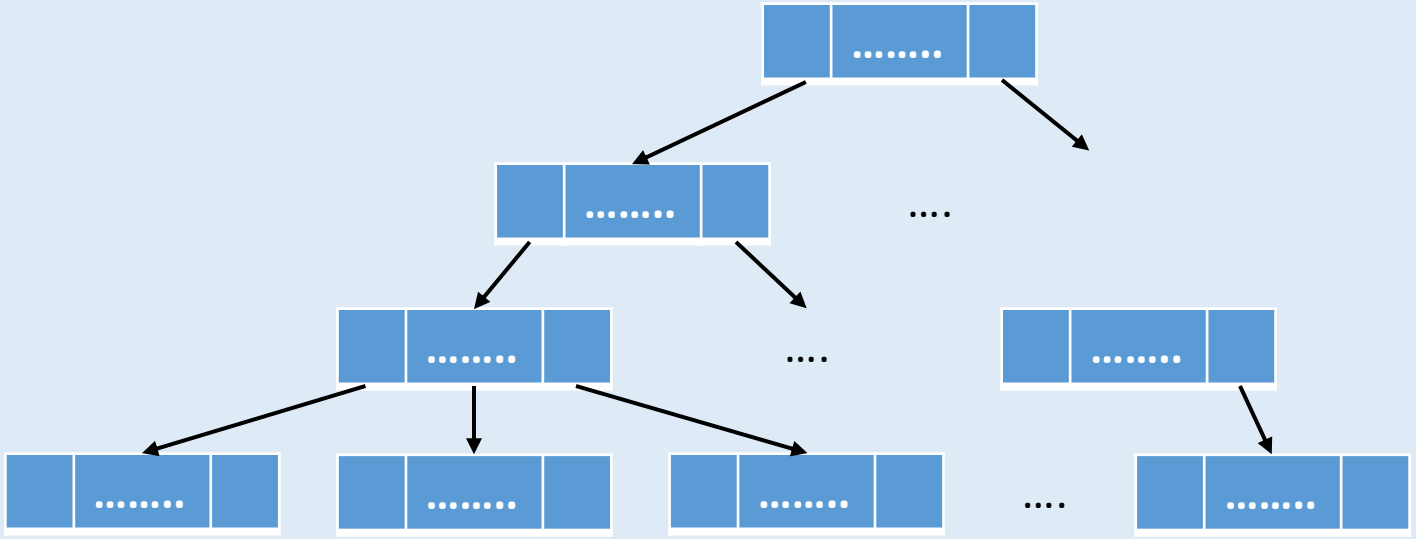
** Postgres uses these.*

B-trees: different designs



There may be many definitions of B-trees tuned to particular use cases.

B-trees: different designs



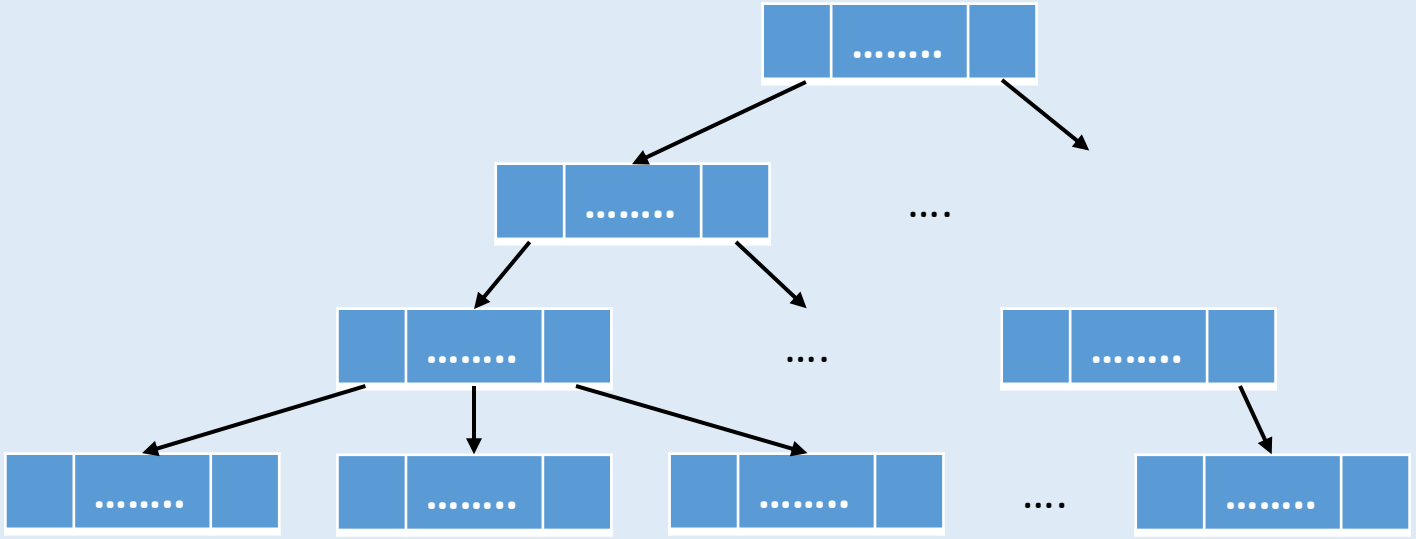
There may be many definitions of B-trees tuned to particular use cases.

Keys and values may be

- fixed-length (a DB index over a fixed-width column),
- variable-length (a list of directory entries).

When a tree has variable-length keys and values, it is fine to disregard the limits on the number of items in a node.

B-trees: different designs

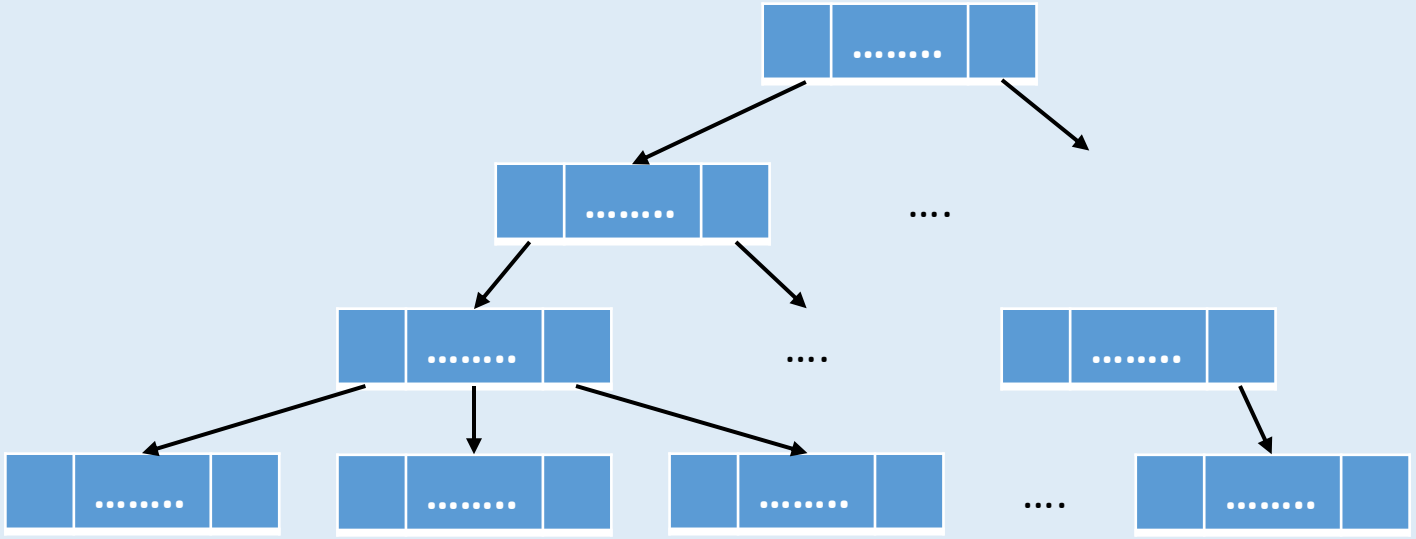


There may be many definitions of B-trees tuned to particular use cases.

Red-black trees store user data both in leaf and internal nodes. Our definition of B-trees has pointers to user data only in leaves.

When implementing a map, we may define B-tree nodes to contain triples `(key, value, pointer)`. In this case internal nodes may also be used to store user data.

B-trees: different designs



There may be many definitions of B-trees tuned to particular use cases.

Red-black trees store user data both in leaf and internal nodes. Our definition of B-trees has pointers to user data only in leaves.

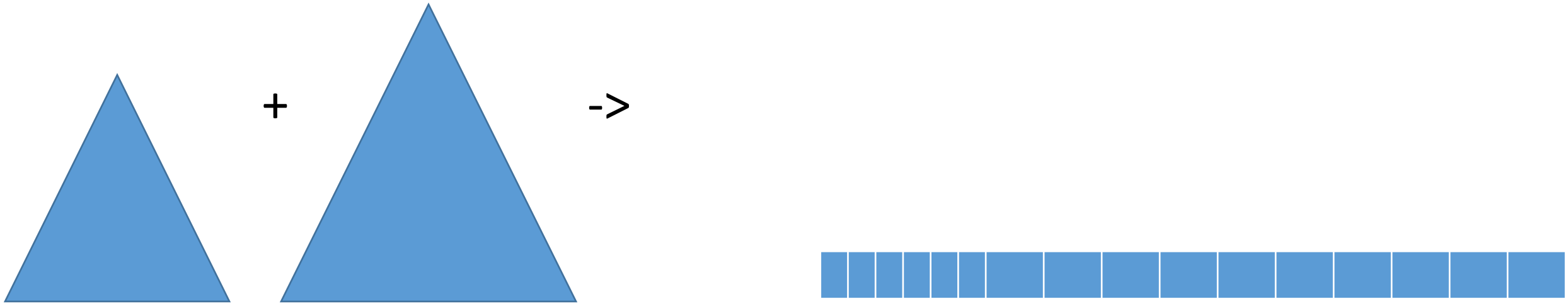
Reiserfs 3 has a unified tree to store all file system objects. Some are stored in the tree, and some outside:

- file metadata, directory entries and data blocks with packed tails are stored in leaf nodes directly,
- non-tail parts of files only have pointers to blocks in leaf nodes.

How to merge two B-trees

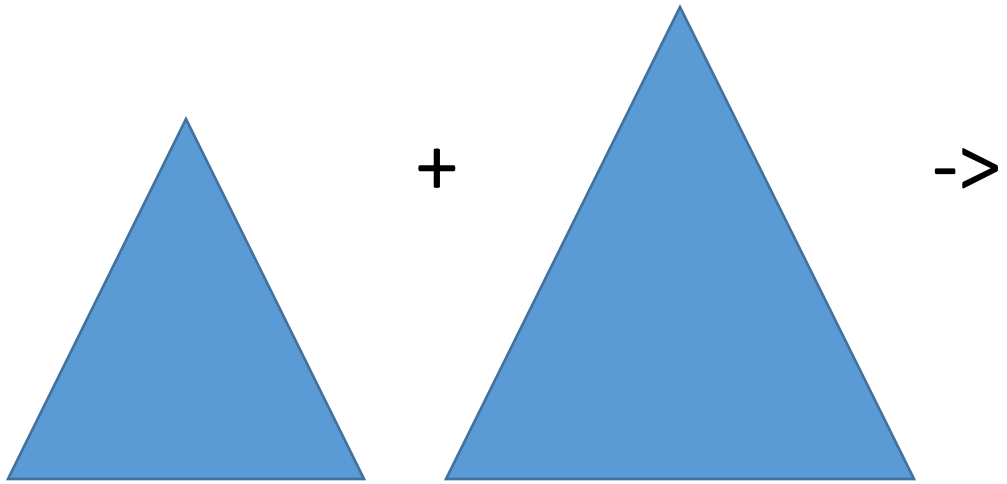
Suppose we have two sets represented as B-trees. We can construct their union in linear time.

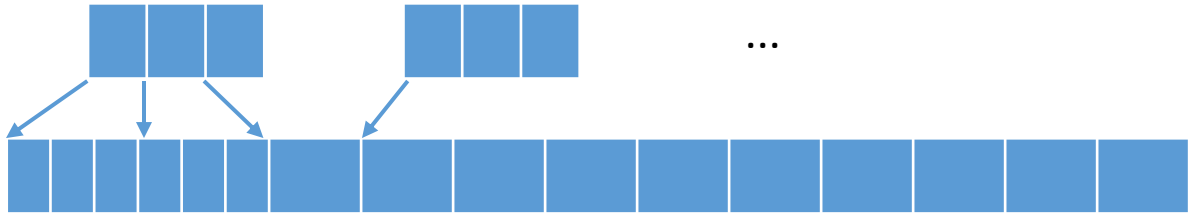
How to merge two B-trees



1. Iterating over keys in leaf nodes sequentially yields them in ascending order. As we have two ascending sequences of keys, we can find their union like the merge sort does.
2. The output of “merge sort” can be sequentially written into pages with 2L elements each. These will be the leaf nodes of the merged tree.
3. For every 2L-1 adjacent leaf nodes we make an internal node pointing to them. We write such internal nodes sequentially, too.
4. We create upper levels of internal nodes in a similar way.
5. Once we reach a level that has only one internal node, we call it the root of the merged tree.

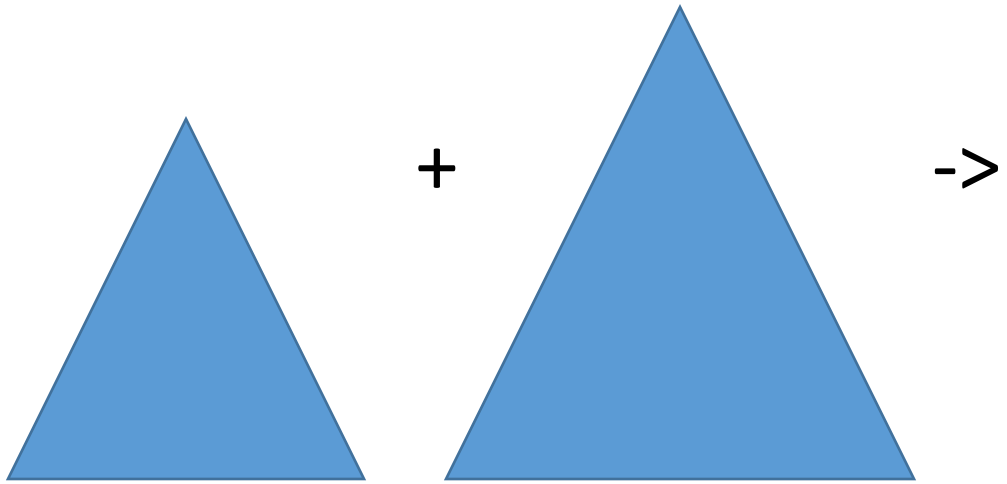
How to merge two B-trees

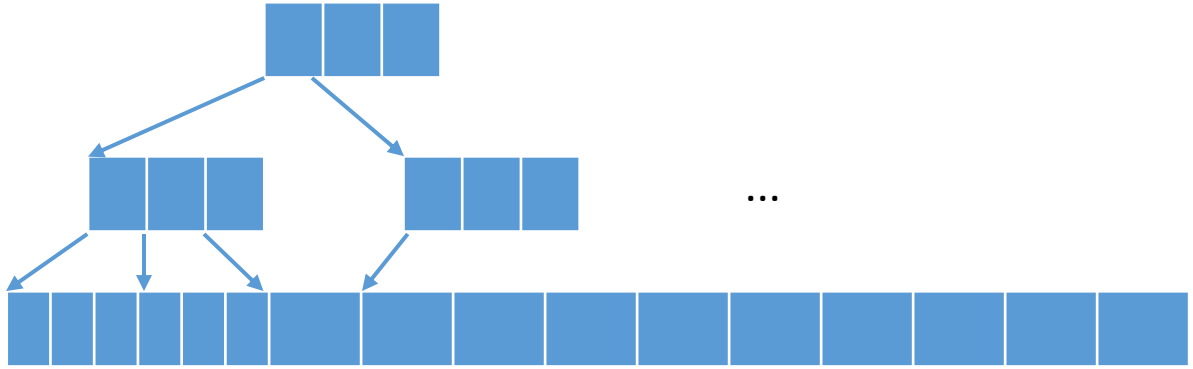




1. Iterating over keys in leaf nodes sequentially yields them in ascending order. As we have two ascending sequences of keys, we can find their union like the merge sort does.
2. The output of “merge sort” can be sequentially written into pages with 2L elements each. These will be the leaf nodes of the merged tree.
3. For every 2L-1 adjacent leaf nodes we make an internal node pointing to them. We write such internal nodes sequentially, too.
4. We create upper levels of internal nodes in a similar way.
5. Once we reach a level that has only one internal node, we call it the root of the merged tree.

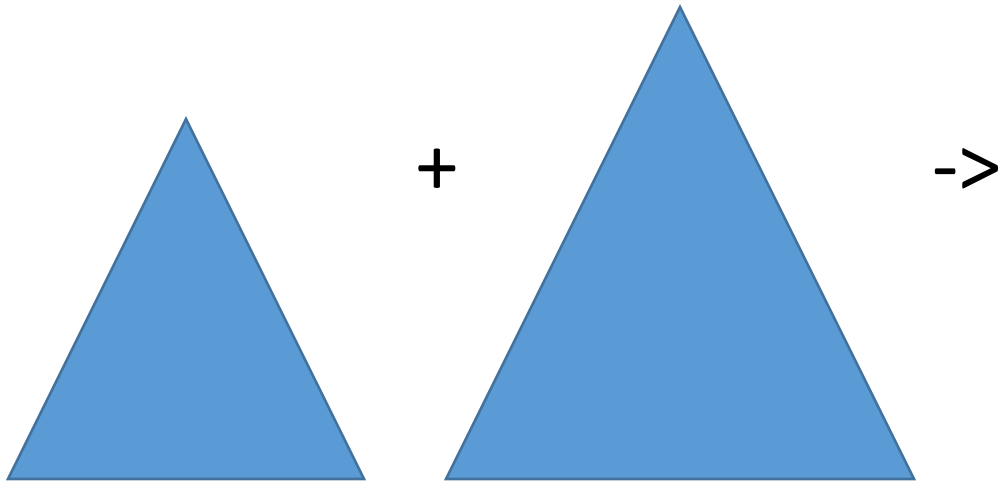
How to merge two B-trees

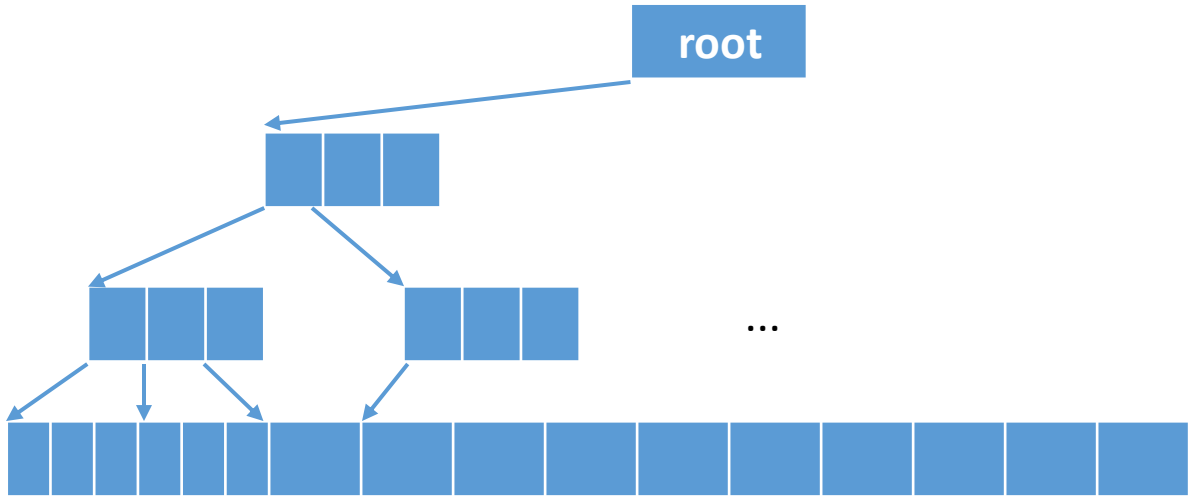




1. Iterating over keys in leaf nodes sequentially yields them in ascending order. As we have two ascending sequences of keys, we can find their union like the merge sort does.
2. The output of “merge sort” can be sequentially written into pages with 2L elements each. These will be the leaf nodes of the merged tree.
3. For every 2L-1 adjacent leaf nodes we make an internal node pointing to them. We write such internal nodes sequentially, too.
4. We create upper levels of internal nodes in a similar way.
5. Once we reach a level that has only one internal node, we call it the root of the merged tree.

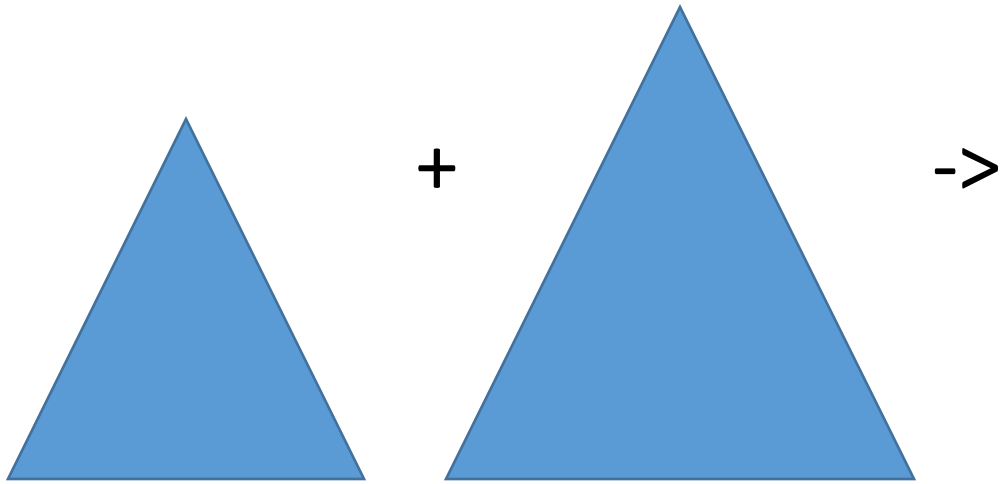
How to merge two B-trees





1. Iterating over keys in leaf nodes sequentially yields them in ascending order. As we have two ascending sequences of keys, we can find their union like the merge sort does.
2. The output of “merge sort” can be sequentially written into pages with 2L elements each. These will be the leaf nodes of the merged tree.
3. For every 2L-1 adjacent leaf nodes we make an internal node pointing to them. We write such internal nodes sequentially, too.
4. We create upper levels of internal nodes in a similar way.
5. Once we reach a level that has only one internal node, we call it the root of the merged tree.

How to merge two B-trees



Key properties:

- Reading leaf nodes of input trees issues mostly sequential reads from input trees.
- Nodes of the output tree are written sequentially.
- If the input trees were produced by merging smaller trees, then reads from leaf nodes are purely sequential.

Log-Structured Merge Tree

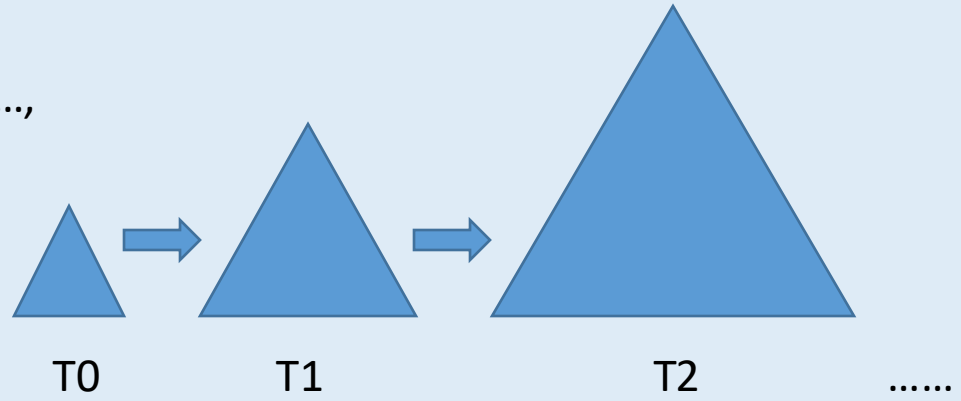
The algorithm for merging B-trees may be used to create a data structure that has very fast inserts, and avoids random IO needed to rebalance B-trees.

It is friendly towards HDDs and SSDs, but it is especially well-suited to cloud object storages.

Log-Structured Merge Tree

An LSM tree is a sequence of B-trees.

- One searches for an item in a LSM tree by searching all member trees T_0, T_1, …,
- Inserts go to T_0,
- T_0 (possibly also a few smaller-numbered ones) are stored in the RAM which guarantees fast inserts,
- When a tree T_i grows "too big", it is merged into T_{i+1} and the old T_i is removed,
- Deleting an item is implemented by inserting a deletion marker.*. Both an item and a deletion marker will be erased from the disk during a subsequent merge.



Fact: the overhead of merges is minimised if the max sizes of member trees are a geometric progression.

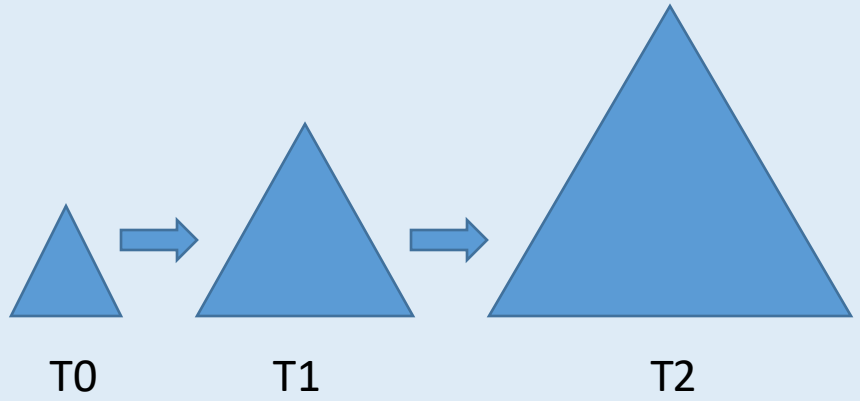
https://www.cs.umb.edu/~poneil/lsmtree.pdf
https://queue.acm.org/detail.cfm?id=3220266
https://arxiv.org/pdf/1812.07527

** Such deletion markers are called "tombstones".*

Log-Structured Merge Tree

Inserts to an LSM tree are fast because they affect only the in-RAM component T_0, and tree merges produce only sequential IO.

However, the need to merge trees may introduce noticeable delays.



So, we have two problems to work around:

1. Searching for an item must search many member trees.
2. Write amplification and unpredictable insert latencies: the amortised time of inserts is low, but the run time of inserts that trigger merges may be orders of magnitude longer than the run time of inserts that modify only T0. Such inserts also create bursts of IO.

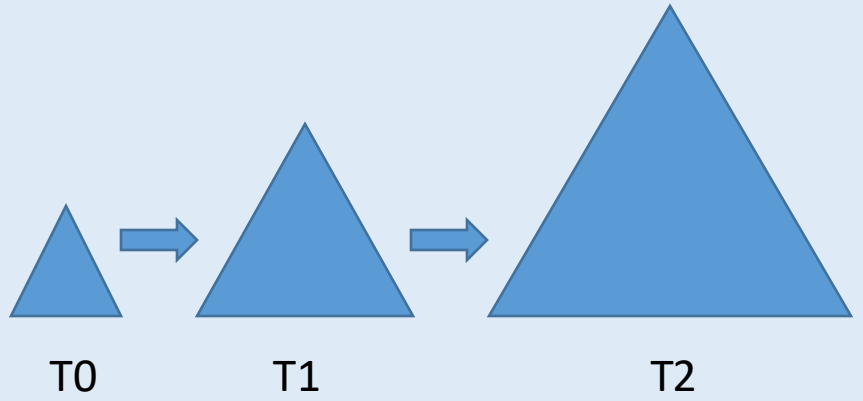
Log-Structured Merge Tree

Inserts to an LSM tree are fast because they affect only the in-RAM component T_0, and tree merges produce only sequential IO.

However, the need to merge trees may introduce noticeable delays.

So, we have two problems to work around:

1. Searching for an item must search many member trees.
2. Write amplification and unpredictable insert latencies: the amortised time of inserts is low, but the run time of inserts that trigger merges may be orders of magnitude longer than the run time of inserts that modify only T0. Such inserts also create bursts of IO.



- Trees may be merged in a background process.
- Components are immutable so original trees may be searched during a merge.

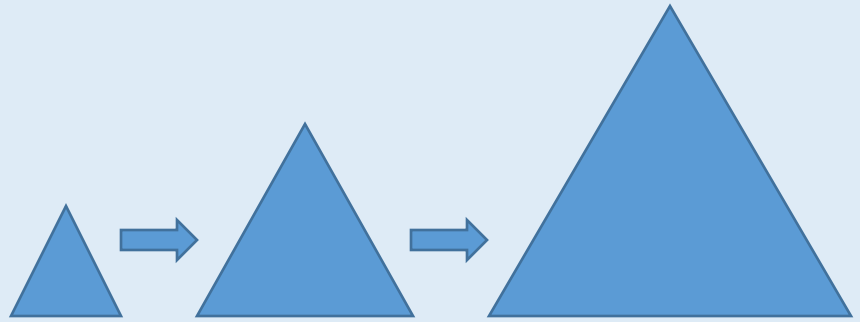
Log-Structured Merge Tree

Inserts to an LSM tree are fast because they affect only the in-RAM component T_0, and tree merges produce only sequential IO.

However, the need to merge trees may introduce noticeable delays.

So, we have two problems to work around:

1. Searching for an item must search many member trees.
2. Write amplification and unpredictable insert latencies: the amortised time of inserts is low, but the run time of inserts that trigger merges may be orders of magnitude longer than the run time of inserts that modify only T0. Such inserts also create bursts of IO.



- Bloom filters

Bloom filters

Searching an item in an LSM tree needs to search every component of the LSM tree.

We may often skip most components T_i if we come up with an efficient way to detect that T_i does not contain a key that we are looking for. This can be achieved with a Bloom filter constructed from the keys in T_i. A Bloom filter is a probabilistic data structure. Like a set, it implements methods `Add()` and `Contains()`. However, `s.Contains(x)` returns the following values:

- `s` does not contain `x`,
- `s` maybe contains `x`.

Allowing `Contains()` to produce wrong allows us to use much less RAM for a Bloom filter than for a set. The probability of wrong answers may be made arbitrarily low by tuning the parameters of a Bloom filter.

Bloom filters

Searching an item in an LSM tree needs to search every component of the LSM tree.

We may often skip most components T_i if we come up with an efficient way to detect that T_i does not contain a key that we are looking for. This can be achieved with a Bloom filter constructed from the keys in T_i. A Bloom filter is a probabilistic data structure. Like a set, it implements methods `Add()` and `Contains()`. However, `s.Contains(x)` returns the following values:

- `s` does not contain `x`,
- `s` maybe contains `x`.

The implementation: let us have a bit array with **m** entries and **k** independent hash functions $\mathbf{f_i}$ that produce values in the range [0, m-1).

- To insert an item **x**, set bits in positions $f_1(x)$, $f_2(x)$, … $f_k(x)$ to 1,
- To verify whether **y** belongs to the set, verify whether bits in positions $f_1(y)$, $f_2(y)$, …, $f_k(y)$ are set to 1.

Bloom filters

Searching an item in an LSM tree needs to search every component of the LSM tree.

We may often skip most components T_i if we come up with an efficient way to detect that T_i does not contain a key that we are looking for. This can be achieved with a Bloom filter constructed from the keys in T_i. A Bloom filter is a probabilistic data structure. Like a set, it implements methods `Add()` and `Contains()`. However, `s.Contains(x)` returns the following values:

- `s` does not contain x,
- `s` maybe contains x.

The implementation: let us have a bit array with **m** entries and **k** independent hash functions $\mathbf{f_i}$ that produce values in the range [0, m-1).

- To insert an item **x**, set bits in positions $f_1(x)$, $f_2(x)$, … $f_k(x)$ to 1,
- To verify whether **y** belongs to the set, verify whether bits in positions $f_1(y)$, $f_2(y)$, …, $f_k(y)$ are set to 1.

If items **x** are uniformly sampled from a set that has **N** elements and the probability of a wrong answer «**y** is present» must be lower than **p**, then we need to use the following Bloom filter parameters:

$$k >= -\log_2(p),\ m >= k*N / \ln(2).$$

The power of two choices

Why do Bloom filters need multiple independent hashes?

The power of two choices

Why do Bloom filters need multiple independent hashes?

Fact 1: Suppose we have a hash table with N buckets, and let us insert N random elements into it.
With the probability >= 1 – O(1/N) the cardinality of the busiest bucket will be

$$\frac{\log N}{\log \log N} + O(1)$$

The power of two choices

Why do Bloom filters need multiple independent hashes?

Fact 1: Suppose we have a hash table with N buckets, and let us insert N random elements into it.
With the probability >= 1 – O(1/N) the cardinality of the busiest bucket will be

$$\frac{\log N}{\log \log N} + O(1)$$

Fact 2: Suppose we have a hash table with N buckets, but inserts into the table use d independent hashes. Insert(x) computes d hashes of x to pick the candidate buckets and inserts x into the least populated bucket.

Let us insert N random elements into such hash table. With the probability >= 1 - O(1/N) the cardinality of the busiest bucket will be

$$\frac{\log \log N}{\log d} + O(1)$$

The use of two independent hash functions instead of one improves the asymptotic of the cardinality of the busiest bucket. Adding more hashes no longer improves it.

The power of two choices

Applications:

- Hash tables,
- Bloom filters,
- ?

The power of two choices

Applications:

- Hash tables,
- Bloom filters,
- Load balancing in distributed systems,
- Controlling tail latencies in distributed systems.

The power of two choices

Applications:

- Hash tables,
- Bloom filters,
- Load balancing in distributed systems,
- Controlling tail latencies in distributed systems.

Naïve approach: Suppose there are multiple HTTP servers that can serve the same file download. A client may choose a less loaded server this way:

1. Pick two random servers to try,
2. Send a download request to both servers,
3. Wait for the first byte from any of the servers,
4. Use the one that replied first to complete the download, and cancel the other request.

No one does this because it doubles the number of requests, but there is a modification to this algorithm that makes it useful both for load balancing and controlling tail latencies.

The power of two choices

Applications:

- Hash tables,
- Bloom filters,
- Load balancing in distributed systems,
- Controlling tail latencies in distributed systems.

Problem: let us have N identical servers that handle 99% requests in < 10ms, but the remaining 1% (random) requests take 1s. Suppose it takes 10 requests to those servers to handle a single request from a client. What is the share of client requests that are processed in ≈10ms?

The power of two choices

Applications:

- Hash tables,
- Bloom filters,
- Load balancing in distributed systems,
- Controlling tail latencies in distributed systems.

A better approach: Suppose there are multiple servers that can serve the same request. A client may proceed this way:

1. Pick two random servers to try,
2. Send a request to one of them,
3. If the processing time exceeds the 95-th percentile of request durations, then re-issue the request to the second server.

This addresses both problems and creates at most 5% of duplicated requests.

In fact, many systems aim to have >=99.9% or >=99.99% requests that run fast. In that case the slow request logic needs to kick in when 99-th percentile is exceeded which means 1% duplicated requests.

See also: https://barroso.org/publications/TheTailAtScale.pdf
See also: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9167399

To do at home

1. Implement `std::map<int64_t, int64_t>` as a B-tree. Remove items by inserting tombstones.
2. Write a function that merges two B-trees.
3. Measure the latency distribution of insertions into your tree (make this question precise before answering it).