A Symmetric Concurrent B-Tree Algorithm

Article: Vladimir Lanin and Dennis Shasha

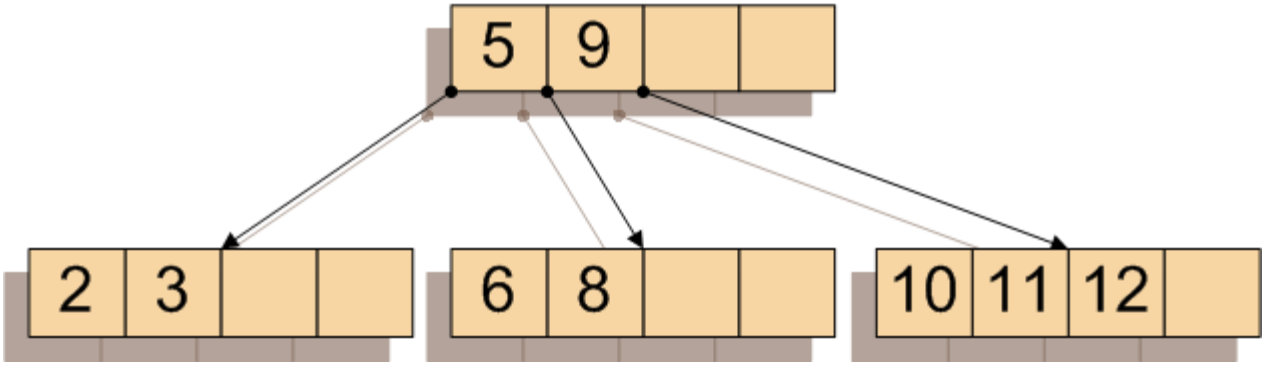
Presentation: Elise Cormie

Introduction: B-trees

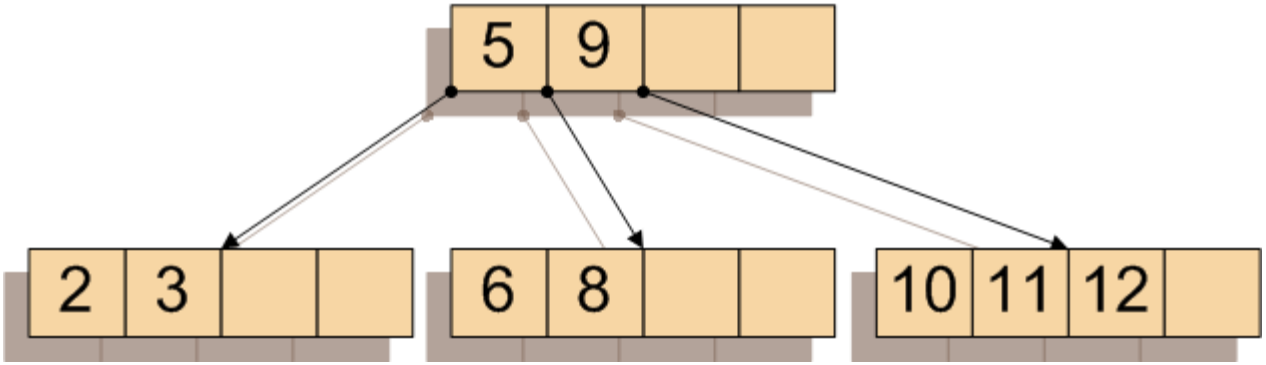
- Search trees with order t.
- Support *search*, *insert*, and *delete*.
- Except root, all nodes contain between t and $2t$ keys.
- Each node has between 0 and $2t+1$ children. Links to children are located to either side of each key.
- All leaves are at the same depth.
- Use: Minimize disk accesses by making leaves store a many keys, which are all loaded together into memory.
- Generally used to store large amounts of data.

Example: Tree with order 2

- Minimum keys per non-root node: 2.
- Maximum: 4



- Each key can have a child to its right and to its left.
- The left subtree contains only keys smaller than the parent key. The right contains only larger keys.

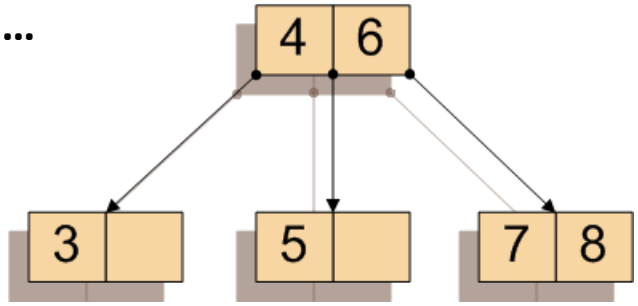


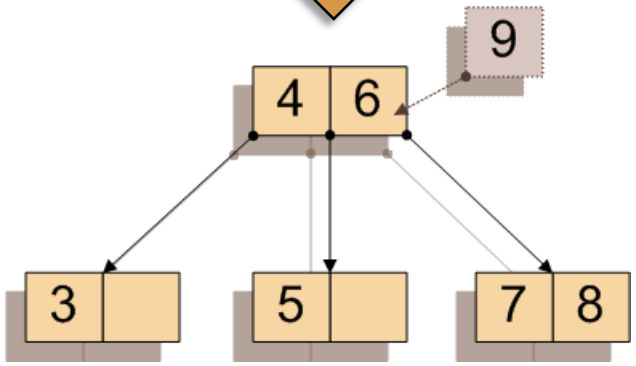
Insertion and Split

- Insertion finds the leaf node where a key should be located, and splits full nodes as necessary to allow new elements to be inserted.
- *Split - Ascending version*: Starts at a full leaf node where a new element should be placed, and propagates up the tree.
  - Most common algorithm (?).
- *Split - Descending version*: Executed on any full node insertion encounters while descending the tree, guaranteeing it will be able to insert the new element once it finds the correct leaf.
  - Does not need to follow any child→parent links. More efficient.
  - Maximum number of keys must be odd instead of even (min t-1, max 2t-1) so each full node has a median to split around.

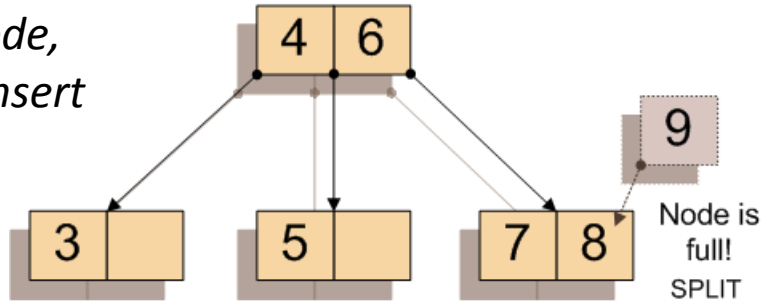
Insertion/Split (Ascending)

Insert 9…

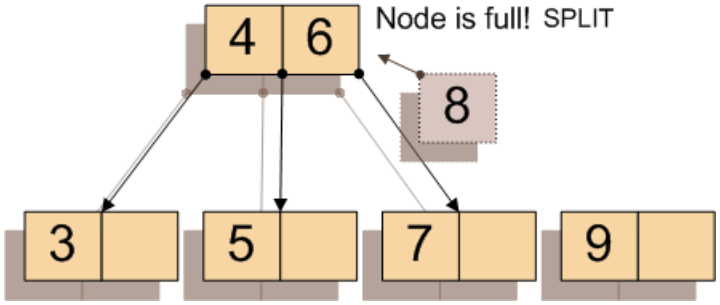


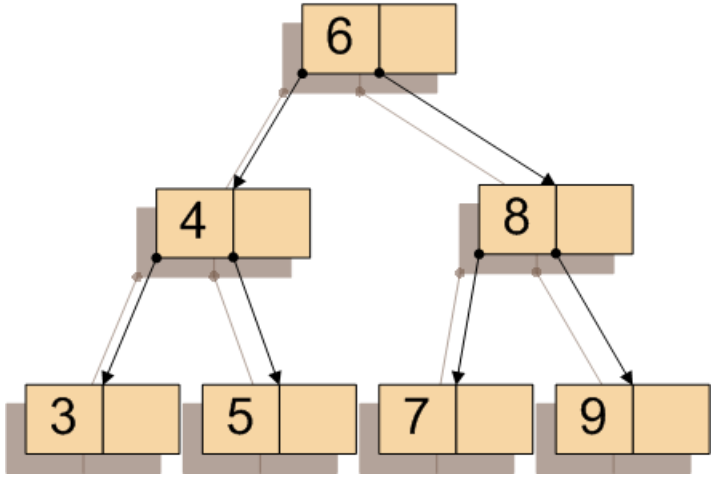


Have reached leaf node, must insert here.



Split propagates upwards through full nodes.



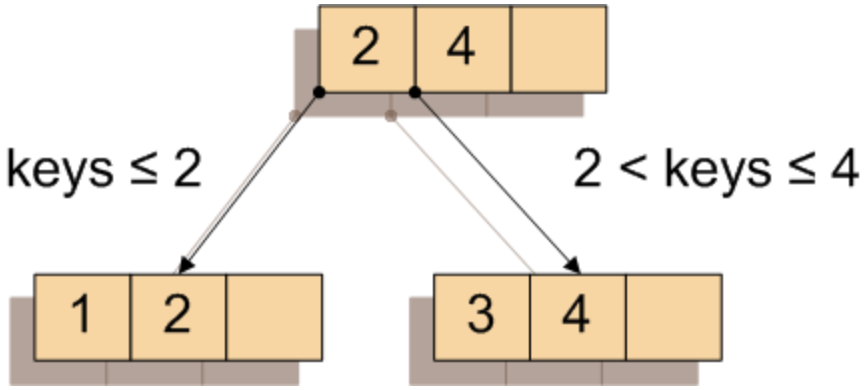


Deletion

- Similar to insertion, a bit more complicated.
- Different cases for internal/leaf nodes, deleting a key that separates two children, deleting a key in a node that already has the minimum number of keys, merging nodes, etc.
- Can be done either while ascending (from the node that a key has been deleted from), or descending (to find that node).

B+ tree

- Like a B-tree, except all actual keys and data are stored in leaves.
- Interior nodes only contain keys, to index the keys in the leaf nodes.



Approaches to Parallelization

1. Writers exclusively lock entire subtrees, starting at the highest node that might be modified.

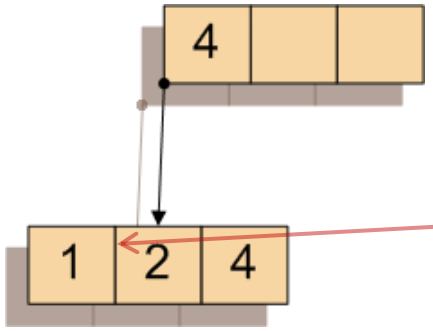
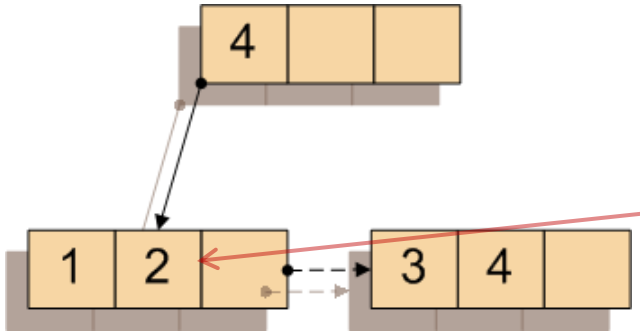
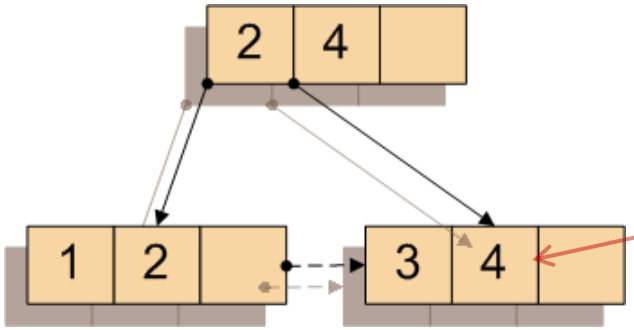
2. “Optimistic” writers place read-locks on the subtrees they descend, assuming they will be able to insert/delete without propagating change up the tree. They place an exclusive lock only on one node they want to modify. If they end up having to propagate changes upwards to other nodes, they go back and re-do the descent using exclusive locks.

3. Using proactive, descending approach to insert/delete, writers can exclusively lock nodes they need to modify and then release them as they descend further, knowing they won't have to modify them again. (Less of the tree is locked at once.)

4. Add extra links to allow data to be found even when the tree is in an invalid or changed state. Fewer locks then need to be used.

B-link Trees

- The algorithm in this article is based on one by Lehman & Yao, where “rightlinks” are introduced to B+ trees, creating “B-link” trees.
- Data should only move to the right in a B-link tree.
- Data being searched for can end up to the right of where it is expected while the tree is being modified.
- Rightlinks link each node in a level, from left to right. So if an element is not found where it is expected, the search can continue in the next node of the level.
- This allows fewer locks to be used, and removes the need to lock entire subtrees.

Split leaf to insert 3.	Concurrently, search for 4.
	4 ≤ 4, so go to left child and search for 4.
	Meanwhile, the node has been split. This node no longer contains 4!
	Follow rightlink and continue searching. 4 is found.

The Article

- Uses the concept of adding extra links from Lehman & Yao.
- Lehman & Yao's algorithm did not include concurrent merge operations, so their tree would end up unbalanced unless the entire tree was locked while exclusive balancing procedures were performed.
- In this article, Lanin & Shasha use the linking idea, but add another type of link ("outlink") and concurrent merging procedures. Their algorithm thus keeps the b-tree balanced.
- Their simulations show it to be an improvement on the previous algorithms mentioned.