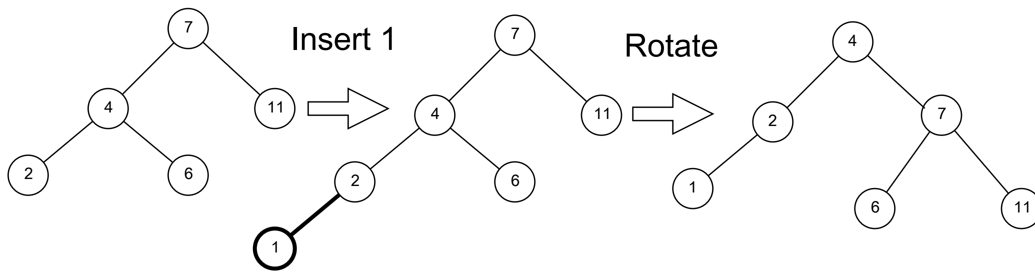
2-3 Trees and B-Trees

1 Recap

1.1 Balanced Binary Search Trees

Binary Search Trees that guarantee $O(log(n))$ height by rebalancing after Insert/Delete operations.

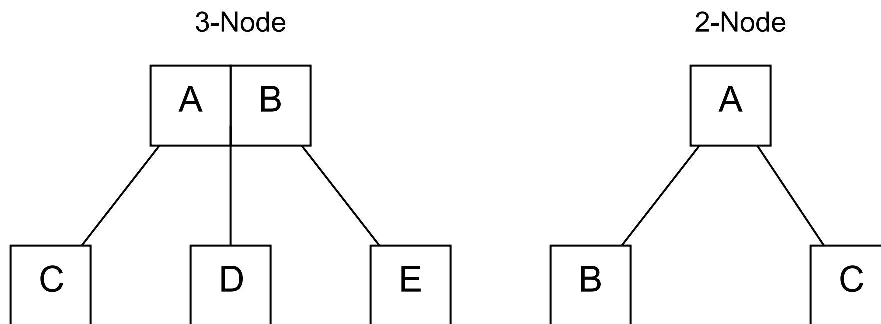
1.2 Example Insertion and Rotation



2 2-3 Trees

2.1 Properties

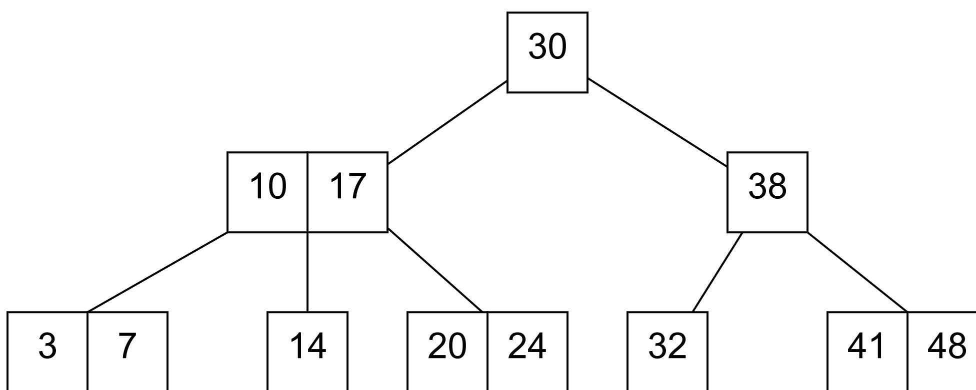
2-3 Trees are balanced search trees. Every node with children (non-leaf) has either two children (2-node) and consists of one piece of data, or has three children (3-node) and consists of 2 pieces of data.



- Every non-leaf is a 2-node or 3-node
- All leaves are at the same level
- All non-leaves branch

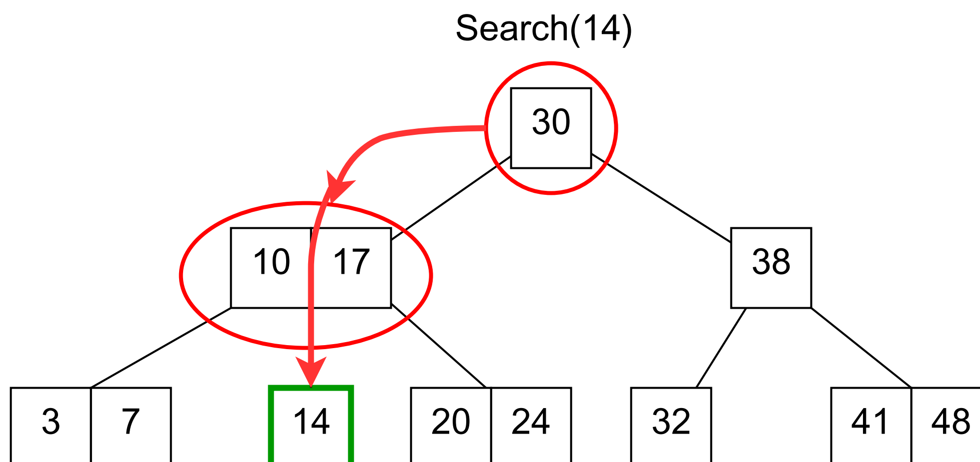
- All data is kept in sorted order
- Every leaf node will contain one or two fields
- Height $\leq lgn$ (More dense than BST)

2.2 Example 2-3 Tree



2.3 Search

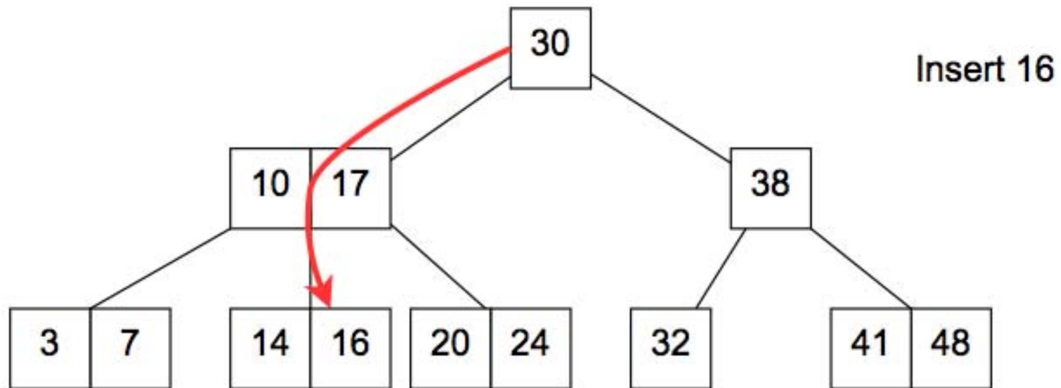
Very similar to Binary Search. Start at the root node, and traverse down tree in order. Tree is sorted so only need to look at one node for each level of tree. Because of this Search runtime is $O(lg(n))$

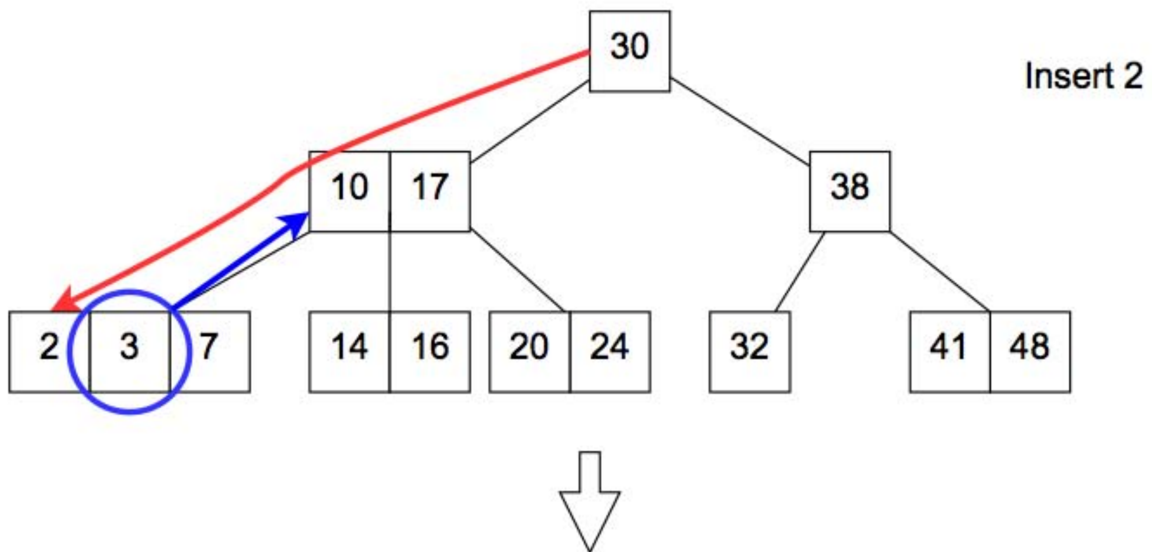


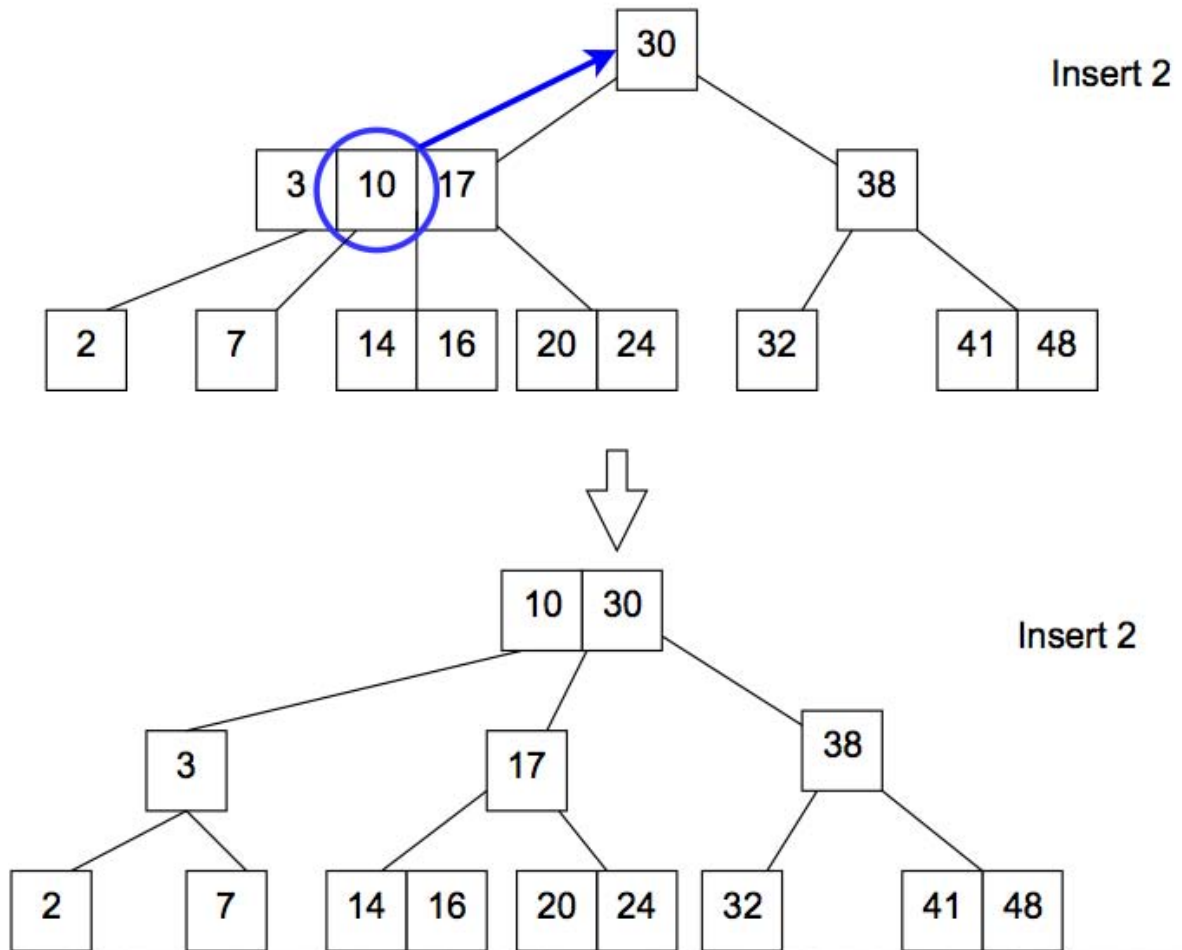
2.4 Insert

Insert(X) Steps:

- Search for element X for where it would go in a leaf of the tree.
- Insert element X into where it would go.
- While there is overflow, a node has more than 3 elements, Split node into left half, median, and right half Then promote median up a level. If there is a parent, add it to the node. If there is no parent, create a new node of just the Median node as the new root.
- Runtime is $O(lg(n))$





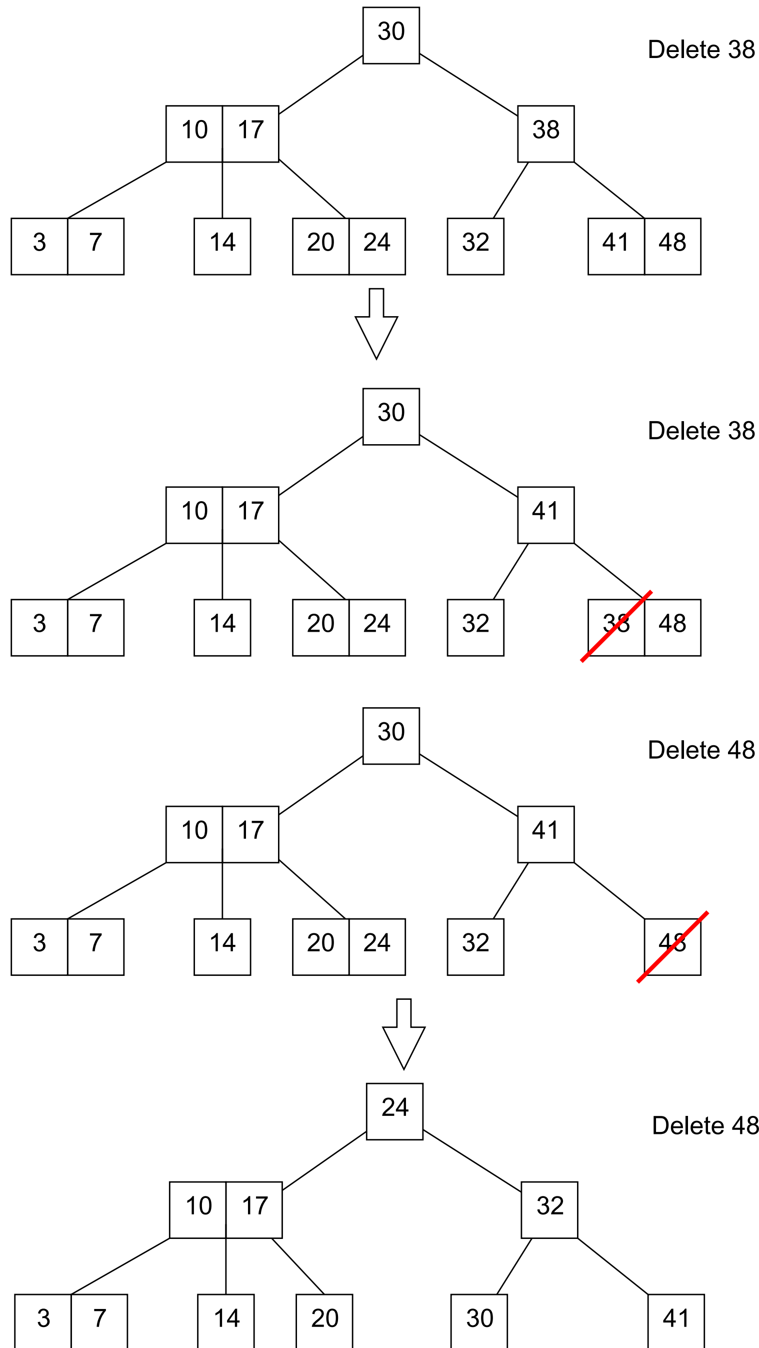


2.5 Delete

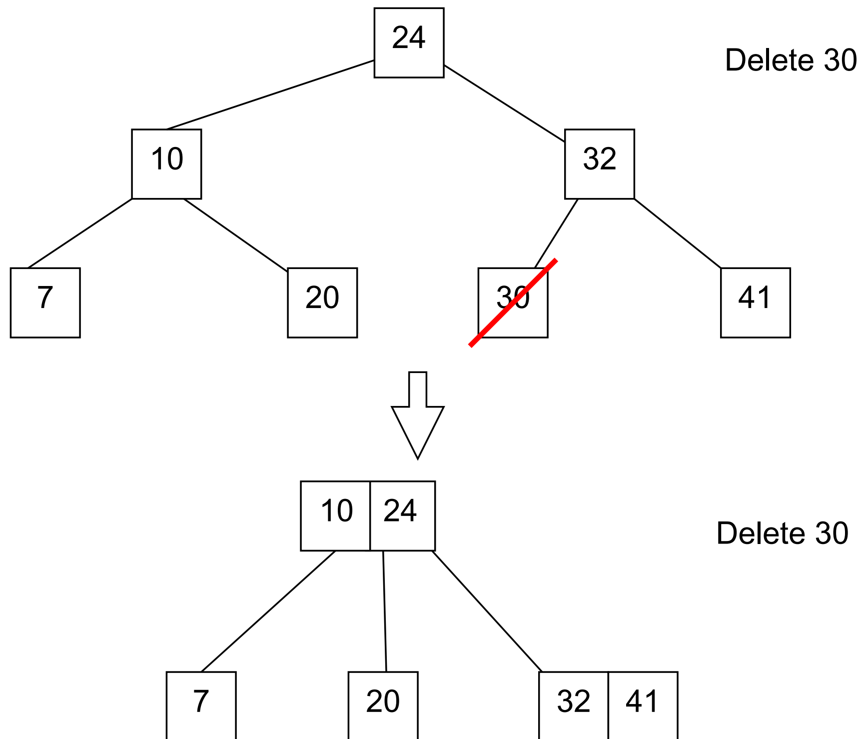
Delete(X) Steps:

- Swap item to delete with inorder successor if item is not already a leaf.
- Redistribute and merge nodes if there is underflowing in order to get back to a correct 2-3 tree
- Runtime is $O(lg(n))$

2.5.1 Redistribute Example



2.5.2 Merge Example



3 B-Trees

B-Trees are tree data structures that store sorted data. B-Trees can be seen as a generalization of Binary Search Trees where nodes can have more than one key/value and more than two children. Similar to *BST*s, they support *search*, *insertion* and *deletion* in logarithmic time.

3.1 Properties

A B-tree has a parameter called the *minimum degree* or *branching factor*. For the purposes of our discussion let the branching factor be B.

- For any non leaf node, the number of children is one greater than the number of keys in that node.
- Every non-root node contains at least $B - 1$ keys. Consequently, all internal (non-leaf and non-root) nodes have at least B children.
- Every node contains at most $2B - 1$ keys. Consequently, all nodes have at most $2B$ children.
- All the leaves are at the same depth.

The keys is a B-tree are sorted in a similar fashion to *BST*s. Consider a node x with C children. Let's say that x has keys $k_1 < k_2 < ... < k_C$. For ease of notation, we define $k_0 = \infty$ and $k_n + 1 = -\infty$. If K belongs to the $i^{th}(1 \leq i \leq n+1)$ sub-tree of x, then $k_{i-1} \leq K \leq k_i$.

- Search time is $O(lg(n))$
- Insert/Delete time is $O(lg(n))$ if B = $O(1)$

3.2 Why B-Trees

- Caches read whole blocks of data, and want entire block useful
- Set parameter B equal to block size
- $O(log_b(n))$ block reads per Search, Insert, Delete operations.

B-Trees are used by most databases and filesystems:
-Databases: Sleepycat/BerkelyDB, MySQL, SQLite
-Filesystems: MacOS HFS/HFS+, ReiserFS, Windows NTFS, Linux ext3, shmfs

MIT OpenCourseWare
http://ocw.mit.edu

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2015

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.