02-613 Week 8

Algorithms and Advanced Data Structures

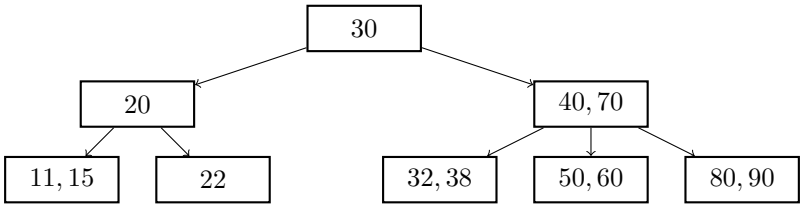
Aidan Jan

October 10, 2025

2, 3 - Trees

A 2, 3-Tree is a search tree where all internal nodes have between 2 and 3 children.

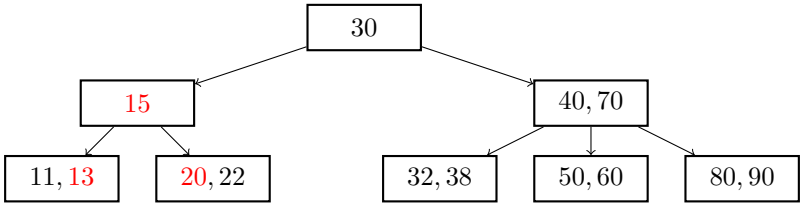
- If there are two children on the node, then the node has **1 key**, and the left child is smaller than the key, the right child is larger.
- If there are three children on the node, then the node has **2 keys**. The left child is smaller than both, the middle child is smaller than the second key but larger than the first, and the right child is larger than both.
- This tree is always 'balanced'.



Insertion

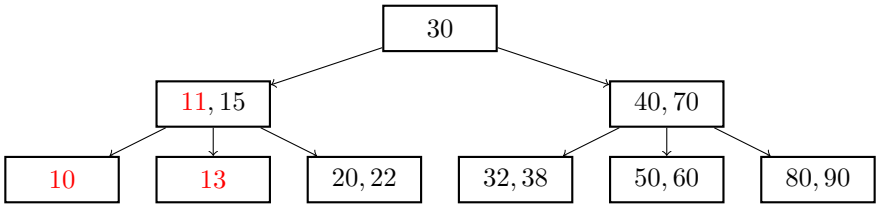
- If we want to insert, we would bubble down from the top to the bottom node, and attempt to add the number as the second key to a node.
- If the chosen node already has two keys, we attempt to push one of the keys to a neighboring node. The 'pushing to neighbor' operation is called a **rotation**.
- If there are no neighbors to **overflow** to, then we do a **split** operation, which a key is pushed up to the parent, and a new node is created. (We can push up multiple layers!)
- If the entire tree is full, then we split the root and add a new node above it, effectively creating a new root.

Suppose we wanted to insert 13. However, our node is full (and there are neighbors that are not)! Instead, since we are in a left child, we move the largest value to the root, and push the parent into the right child.

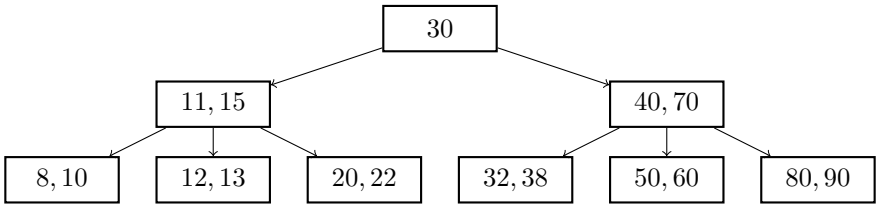


Suppose we now wanted to add 10 to the tree. There is not enough space any of the sister nodes, but there is in the parent. We therefore need to do a split.

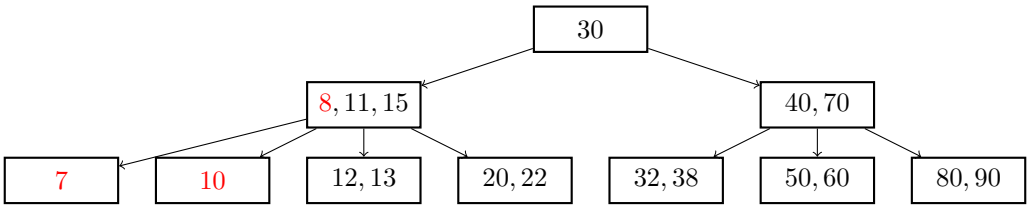
- The middle number (11) gets pushed up to the parent node, and 10 and 13 become their own, separate nodes.



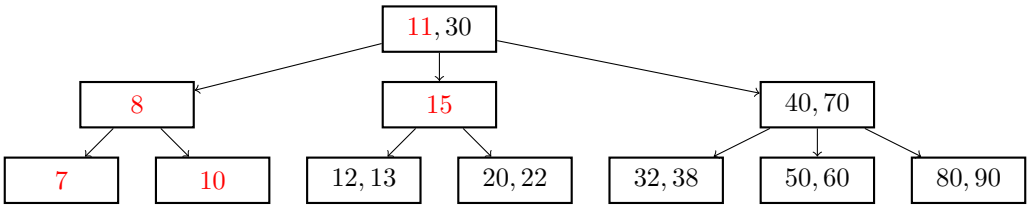
Suppose we added a few more numbers so the left subtree of the root is now saturated.



Now, what happens if, say, we wanted to add a node with 7? None of the leaf nodes or parents have room, so we have to push up to the root. First, add 7 to the left most node in the bottom level. Now we push the middle number up to the parent and split.



Now the middle number of the parent node is pushed to the root, and we split the tree.



- Alternatively, if the right side of the tree happened to have space, we could do a rotation at the parent level instead of splitting the entire tree at the root.
- Also, if the parent is full, we would create a new (empty) node above the parent, and do a split. This creates a new root with two branches.

a, b - Trees

The more general case for the 2, 3-Tree is the a, b-Tree

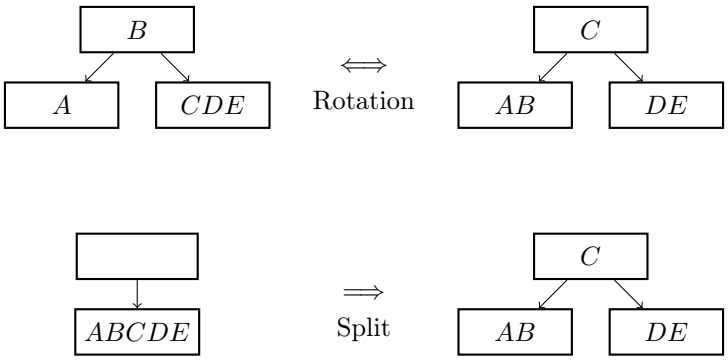
- All nodes (except the root and leaves) have between $a - b$ children.
- Root has between $2 - b$ children.

- $a \geq 2$, $b \geq 2a - 1$.
  - $a \geq 2$ because otherwise we can't have branches.
  - $b \geq 2a - 1$ because otherwise split does not work.
- Note that there is also a **B-Tree**, which is a subset of a, b-Trees where $a = \frac{b+1}{2}$, where b is odd. It is basically a tree that perfectly matches the requirements for the a, b-Tree.

Suppose we have a node with b keys and $b + 1$ children, and we need to split. What do we do?

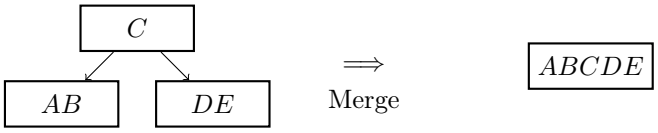
- We would push the middle number up to the parent, and create a new branch on0 the parent.
- Now, one node will have $\lceil \frac{b+1}{2} \rceil$ children, and the other would have $\lfloor \frac{b+1}{2} \rfloor$ children.

In summary:



Merge

What if we wish to delete nodes? We can't just leave empty nodes, so we need to add a `merge` function.



Delete Algorithm

As just alluded to, our delete algorithm relies on the merge function.

[INSERT IMAGE]

Variants of a, b-Trees

B-Trees

B-Trees are a subset of a, b-Trees where $a = \frac{b+1}{2}$, where b is odd. It is basically a tree that perfectly matches the requirements for the a, b-Tree.

B$^+$-Trees

The B$^+$-Tree is a subset of the B-Tree, where each node also stores the parent's keys.

- We can define $M = 2a = b + 1$.
- All nodes have between a and $m - 1$ keys, except the root (minimum 2).

- We can add an additional property that all client (child) nodes contain copies of parent key.
- Leaves are "external" (as in, external storage).

[INSERT IMAGE]

An real-world example of a B^+-Tree is a network attached storage (NAS). A NAS stores directories in a B^+-Tree. The tree stores directory entries in the tree (e.g., directory file), and the actual files in other parts of storage.

Runtime Analysis

Suppose our B^+-Tree has n elements. If we write the height of the tree as a function of n, we get

$$\log_{\frac{M}{2}}(n) \geq \#\text{levels} \geq \log_{m-1}(n)$$

This means that our search time must be bounded by $\log_{m-1}(n)$ and $\log_{\frac{m}{2}}(n)$, since at worst we must traverse the height of the tree. Search runs in $\Theta(\log(n))$.