Balanced Search Trees

What is a Balanced Binary Search Tree: A balanced binary search tree is a tree that automatically keeps its height small (guaranteed to be logarithmic) for a sequence of insertions and deletions. This structure provides efficient implementations for abstract data structures such as associative arrays.

Underlying Motivation: Most operations on a binary search tree (BST) take time proportional to the height of the tree. Desirable to keep the height small.

Approach to Balance: The primary step to get the flexibility needed to guarantee balance in binary search trees is to allow the nodes in our trees to hold more than one key. This may be achieved using 2–3 search trees (not binary, **but balanced**).

2-3 Search Trees: The 2-3 tree structure generalizes BSTs and provides the needed flexibility to guarantee fast operations. The 2-3 tree allows 1 or 2 keys per node – it allows for the possibility of a 3-node and a 2-node:

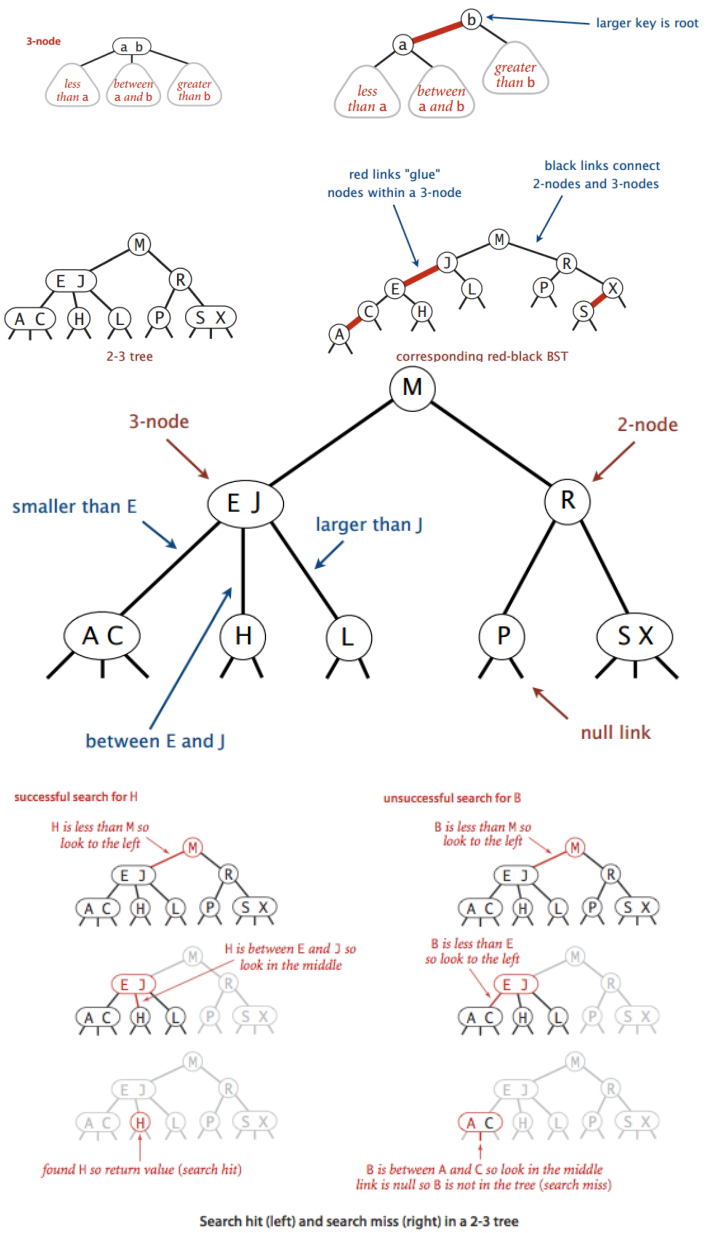
- **2-node:** one key, two children, left is less than key / right is greater than key
- **3-node:** two keys, three children, left is less than key / middle is between / right is greater than the two keys.

Properties of 2-3 Trees: 2-3 trees maintain:

- **Perfect Balance:** Every path from the root to the null link has the same length.
- **Symmetric Order:** Every node is larger than all the nodes on the left subtree, smaller than the keys on the right subtree, and in case of 3-node, all nodes in the middle are between the two keys of the 3-node. Thus, traversal of the nodes can take place in ascending order; In-order traversal.

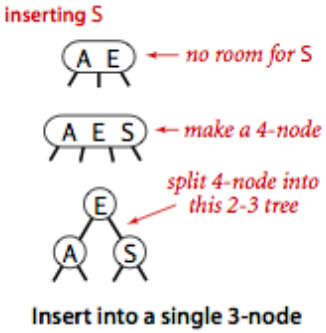
Operations Overview: Main operations are to search in the tree and add into it.

- **Search:** Searching for an item in a 2–3 tree is similar to searching for an item in a binary search tree since it maintains a symmetric order. In other terms, we compare between the given key against the key(s) in the node. If smaller than this key, go left. If between the two keys (of a 3-node), go to the middle link. If greater than this key, go right.
- **Insert (into a 2-node):** All insertion operations start with searching for the node (at the bottom) where one can insert the new node into it. If the node at which the search terminates is a 2-node, we just replace it with a 3-node containing its key and the new key to be inserted.
- **Insert (into a 3-node):** Suppose that we want to insert into a single 3-node. Such a node has no room for a new key. Thefore, to be able to perform this insertion, we temporarily



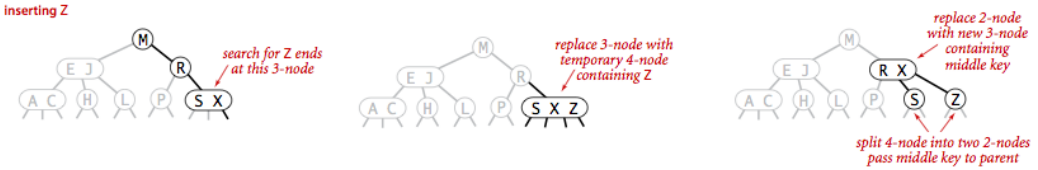
Search hit (left) and search miss (right) in a 2-3 tree

convert the 3-node into a 4-node (*a node with three keys, and four children*). Then, we split the 4-node into three 2-nodes, one with the middle key (at the root), one with the smallest of the three keys (pointed to by the left link of the root), and one with the largest of the three keys (pointed to by the right link of the root).



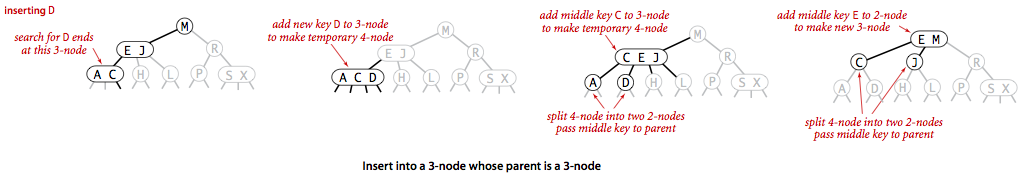
Insert into a single 3-node

- **Insert (into a 3-node whose parent is a 2-node):** Suppose that the search ends at a 3-node at the bottom whose parent is a 2-node. In this case, we follow the same steps as with a temporary 4-node, then splitting the 4-node, but then, instead of creating a new node to hold the middle key, we move the middle key to the parent node (2-node).



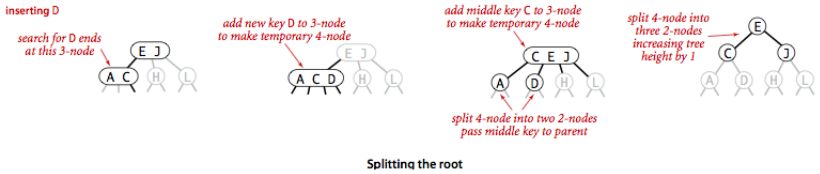
Insert into a 3-node whose parent is a 2-node

- **Insert (into a 3-node whose parent is a 3-node):** Suppose that the search ends at a 3-node at the bottom whose parent is a 3-node. Again, we make a temporary 4-node, afterwards split the 4-node, moving the middle key to the parent node (3-node). Since the parent node is a 3-node, we convert it into a temporary new 4-node. Then, we perform exactly the same transformation on that node. We continue doing this transformation as we go up the tree; splitting 4-nodes and moving the middle keys to their parents until reaching a 2-node, which we replace it with a 3-node that does not to be further split, or until reaching a 3-node at the root.

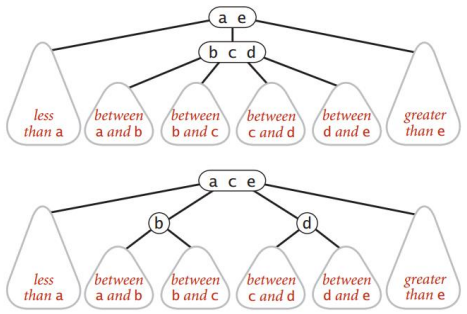


Insert into a 3-node whose parent is a 3-node

Operations in 2-3 Trees: Only constant number of operations are needed to do the transformation of splitting a 4-node, and also converting 3-node to 4-node and so on. These transformations preserve the properties of a 2–3 tree that the tree is in a *symmetric order* and *perfectly balanced*. This is because, when we insert or move keys around, we keep the keys in order; we maintain a symmetric order. Furthermore, we increase the height of the tree when we end up with a temporary 4-node at the root. In this case, we split the temporary 4-node into three 2-nodes. Therefoe, we can still split the root node (4-node) while maintaining perfect balance in the tree.

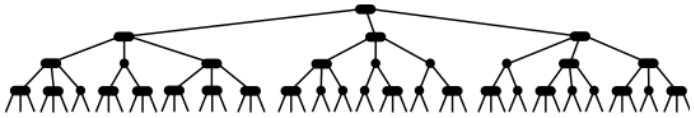


Splitting the root



Analysis: The cost of these operations is proportional to the height of the tree. Since it maintains a perfect black balance tree. It guarantees performance of $O(logN)$ in all operations.

- The worst case when all the nodes are 2-nodes; tree height is $logN$.
- The best case when all the nodes are 3-nodes; tree height is $logN$ (to the base of 3).


Typical 2-3 tree built from random keys

Sources:

[1] https://medium.com/omarelgabrys-blo
[2] http://algs4.cs.princeton.edu/home/