Computer Science 385
Design and Analysis of Algorithms
Siena College
Spring 2019

Topic Notes: 2-3 Trees

You are familiar with the idea of a *binary search tree* and with the importance of maintaining a *balance condition* in those binary search trees to maintain the $\Theta(\log n)$ behavior of many operations on those trees.

Popular ways to maintain balance in binary search trees include *AVL Trees* and *red-black Trees*, each of which places rules when parts of the tree will need to be "rotated" to restore balance after an operation that modifies the tree has caused it to violate its balance condition.

Here, we will consider another variation on the search tree that maintains balance during construction by allowing nodes of the tree to contain multiple keys. Therefore, these are not **binary** search trees, but they are still **balanced** search trees. The specific construct will will study is called a *2-3 Tree*.

Key features and properties of a 2-3 tree include:

- Each node in the tree stores either 1 or 2 key values
- Nodes that are storing 1 key value are called *2-nodes* and have either 0 (when a leaf node) or 2 (when an interior node) children
- Nodes that are storing 2 key values are called *3-nodes* and have either 0 (when a leaf node) or 3 (when an interior node) children
- A valid 2-3 tree is always *perfectly balanced*, in the sense that every node in the tree's bottom level is a leaf, and every other node in the tree is an interior node

In class, we will work through examples of building 2-3 trees. The idea is very similar to that of inserting into an AVL tree or other type of balanced tree:

- We insert the value by visiting the interior nodes and deciding whether the new value should be added to its left or right subtree (for a 2-node) or its left, middle, or right child (for a 3-node). 2-nodes work just like nodes in a traditional binary search tree in that values smaller than its key go to the left subtree, and values larger than its key go to the right subtree. In 3-nodes, we have 3 choices. Values smaller than both keys go to the left subtree, values between the two keys go to the middle subtree, and values larger than both keys go to the right subtree.
- Once a leaf node is encountered, the value is added to that leaf. If the leaf was a 2-node, it becomes a 3-node, and we are done. If the leaf was a 3-node, it would then be a 4-node (one containing 3 keys and with potentially 4 children). This is not permitted in a 2-3 tree,

so the tree must now be reconfigured to be a 2-3 tree once again. This is accomplished by splitting the node into two new 2-nodes, one with the node's smallest value and one with the node's largest. The middle value is promoted to the parent. If the node was already the root of the entire tree, the promoted value will become the new root. Otherwise, the middle value is added as a new key to the parent. If it was a 2-node, it is now a 3-node and we are done. Otherwise, it is now a 4-node, and must again be split. This process continues back up the tree until either a 2-node becomes a 3-node and we can stop, or the entire tree gets a new root.

We will run through one or two examples of the construction of 2-3 trees in class.

It is fairly straightforward to see that the height of a 2-3 tree containing n keys falls between $\log_3 n$ (for a tree consisting of only 3-nodes) and $\log_2 n$ (for a tree consisting only of 2-nodes).

From this, we can see that an insertion is $\Theta(\log n)$, because finding the leaf for insertion will involve either 1 or 2 comparisons at each level, of which there are no more than $\log_2 n$. And then if the tree needs configuration, the changes can only propagate back up through those same number of levels, possibly introducing a new level in the case of a new root being created.

Let's consider an algorithm to check if a 2-3 tree contains a given value:

```
ALGORITHM CONTAINS23TREE($T, k$)
    //Input: a 2-3 tree node $T$, a search key $k$
    //Output: a boolean indicating whether $k$ is stored anywhere in the 2-3 tree rooted at $T$
    if $T$ is a 2-node then
        if $T.key1 = k$ then
            return true
        if $T$ is a leaf then
            return false
        // it's not a leaf, we need to search in a child
        if $k < T.key1$ then
            return $Contains23Tree(T.left, k)$
        else
            return $Contains23Tree(T.right, k)$
    else
        // $T$ is a 3-node
        if $T.key1 = k$ or $T.key2 = k$ then
            return true
        if $T$ is a leaf then
            return false
        // it's not a leaf, we need to search in a child
        if $k < T.key1$ then
            return $Contains23Tree(T.left, k)$
        else if $k > T.key2$ then
            return $Contains23Tree(T.right, k)$
        else
            return $Contains23Tree(T.middle, k)$
```

We can quickly note that this algorithm has no loops, and in the worst case it needs to recurse to a leaf node. Since the height of the tree is $\Theta(\log n)$, the entire algorithm is $\Theta(\log n)$.

We will write additional algorithms to operate on 2-3 trees.