2-3 Trees

Contents

Introduction

Recall that, for binary-search trees, although the average-case times for the lookup, insert, and delete methods are all O(log N), where N is the number of nodes in the tree, the worst-case time is O(N). We can **guarantee** O(log N) time for all three methods by using a **balanced** tree -- a tree that always has height O(log N)-- instead of a binary-search tree.

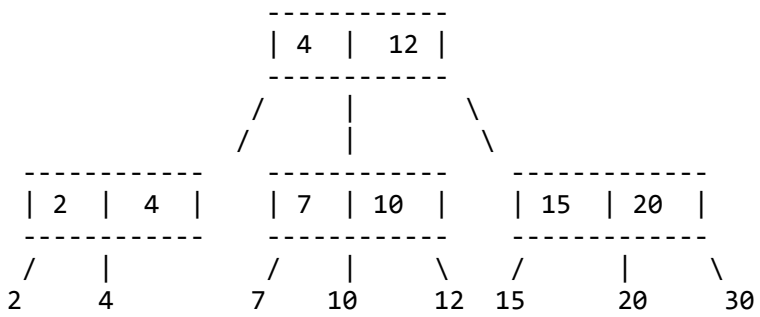
A number of different balanced trees have been defined, including **AVL trees**, **red-black trees**, and **B trees**. You might learn about the first two in an algorithms class, and the third in a database class. Here we will look at yet another kind of balanced tree called a **2-3 Tree**.

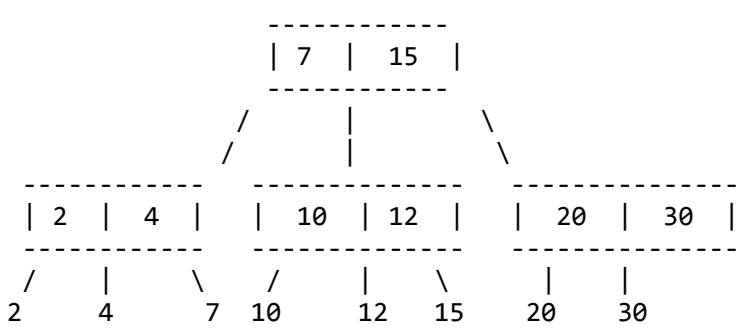
The important idea behind all of these trees is that the insert and delete operations may **restructure** the tree to keep it balanced. So lookup, insert, and delete will always be logarithmic in the number of nodes, but insert and delete may be more complicated than for binary-search trees.

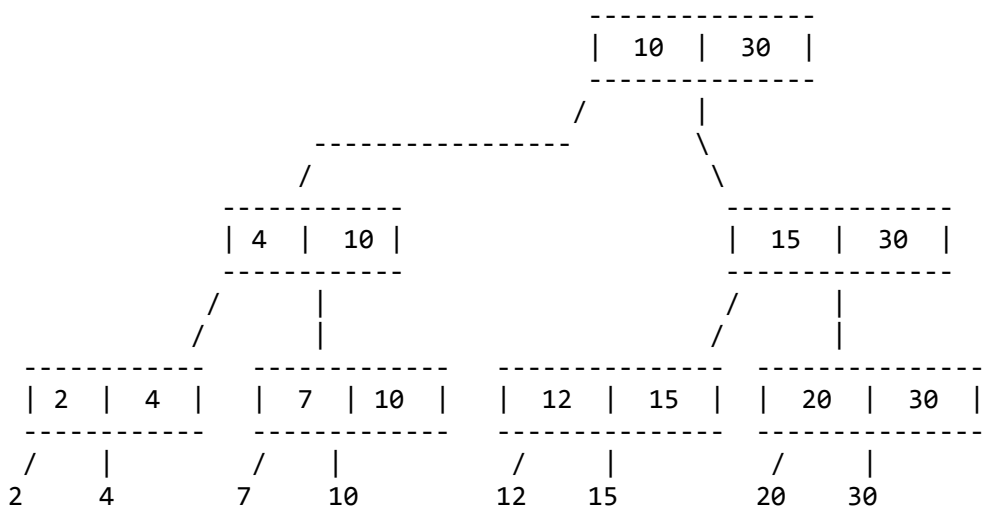
The important facts about a 2-3 tree are:

- Every non-leaf node has either 2 or 3 children.
- All leaves are at the same depth.
- Information (keys and associated data) is stored **only** at leaves (internal nodes are for organization only).
- Keys at leaves are ordered left to right.
- In addition to child pointers, each internal node stores:
  - the value of the max key in the **left** subtree (leftMax)
  - the value of the max key in the **middle** subtree (middleMax)
- If a node only has 2 children, they are left and middle (not left and right) children.

As for binary search trees, the same values can usually be represented by more than one tree. Here are three different 2-3 trees that all store the values 2,4,7,10,12,15,20,30:







TEST YOURSELF #1

Draw two different 2-3 trees, both containing the letters A through G as key values.

solution

Operations on a 2-3 Tree

The lookup operation

Recall that the lookup operation needs to determine whether key value k is in a 2-3 tree T. The lookup operation for a 2-3 tree is very similar to the lookup operation for a binary-search tree. There are 2 base cases:

1. T is empty: return false
2. T is a leaf node: return true iff the key value in T is k

And there are 3 recursive cases:

1. k <= T.leftMax: look up k in T's left subtree
2. T.leftMax < k <= T.middleMax: look up k in T's middle subtree
3. T.middleMax < k: look up k in T's right subtree

It should be clear that the time for lookup is proportional to the height of the tree. The height of the tree is O(log N) for N = the number of **nodes** in the tree. You may think this is a problem, since the actual values are only at the leaves. However, the number of leaves is always greater than N/2 (i.e., more than half the nodes in the tree are leaves). So the time for lookup is also O(log M), where M is the number of key values stored in the tree.

The insert operation

The goal of the insert operation is to insert key k into tree T, maintaining T's 2-3 tree properties. Special cases are required for empty trees and for trees with just a single (leaf) node. So the form of insert will be:

```
if T is empty replace it with a single node containing k
else if T is just 1 node m:
     (a) create a new leaf node n containing k
     (b) create a new internal node with m and n as its children,
         and with the appropriate values for leftMax and middleMax
else call auxiliary method insert(T, k)
```

The auxiliary insert method is the recursive method that handles all but the 2 special cases; as for binary-search trees, the first task of the auxiliary method is to find the (non-leaf) node that will be the **parent** of the newly inserted node.

The auxiliary insert method performs the following steps to find node n, the parent of the new node:

- base case: T's children are leaves - n is found! (T will be the parent of the new node)
- recursive cases:
    - k < T.leftMax: insert k into T's left subtree
    - T.leftMax < k < T.middleMax, or T only has 2 children: insert k into T's middle subtree
    - k > T.middleMax and T has 3 children: insert k into T's right subtree

Once n is found, there are two cases, depending on whether n has room for a new child:

Case 1: n has only 2 children

- Insert k as the appropriate child of n:
    1. if k < n.leftMax, then make k n's left child (move the others over), and fix the values of n.leftMax and n.middleMax. Note that all ancestors of n still have correct values for their leftMax and middleMax fields (because the new value is not the "max" child of n).

2. if k is between n.leftMax and n.middleMax, then make k n's middle child and fix the value of n.middleMax. Again, no ancestors of n need to have their fields changed.
3. if k > n.middleMax, then make k n's right child and fix the leftMax or middleMax fields of n's ancestors as needed.

Case 2: n already has 3 children

- Make k the appropriate new child of n, anyway (fixing the values of n.leftMax and/or n.middleMax as needed). Now n has 4 children.
- Create a new internal node m. Give m n's two rightmost children and set the values of m.leftMax and m.middleMax.
- Add m as the appropriate new child of n's parent (i.e., add m just to the right of n). If n's parent had only 2 children, then stop creating new nodes, just fix the values of the leftMax and middleMax fields of ancestors as needed. Otherwise, keep creating new nodes recursively up the tree. If the root is given 4 children, then create a new node m as above, and create a new root node with n and m as its children.

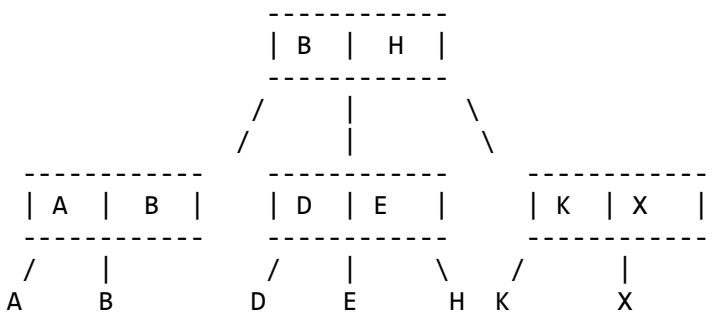
What is the time for insert? Finding node n (the parent of the new node) involves following a path from the root to a parent of leaves. That path is O(height of tree) = O(log N), where N is the number of nodes in the tree (recall that it is also log M, where M is the number of key values stored in the tree).

Once node n is found, finishing the insert, in the worst case, involves adding new nodes and/or fixing fields all the way back up from the leaf to the root, which is also O(log N).

So the total time is O(log N), which is also O(log M).

TEST YOURSELF #2

Question 1: Draw the 2-3 tree that results from inserting the value "C" into the following 2-3 tree:



Question 2: Now draw the tree that results from adding the value "F" to the tree you drew for question 1.

solution

The delete operation

Deleting key k is similar to inserting: there is a special case when T is just a single (leaf) node containing k (T is made empty); otherwise, the parent of the node to be deleted is found, then the tree is fixed up if necessary so that it is still a 2-3 tree.

Once node n (the parent of the node to be deleted) is found, there are two cases, depending on how many children n has:

case 1: n has 3 children

- Remove the child with value k, then fix n.leftMax, n.middleMax, and n's ancestors' leftMax and middleMax fields if necessary.

case 2: n has only 2 children

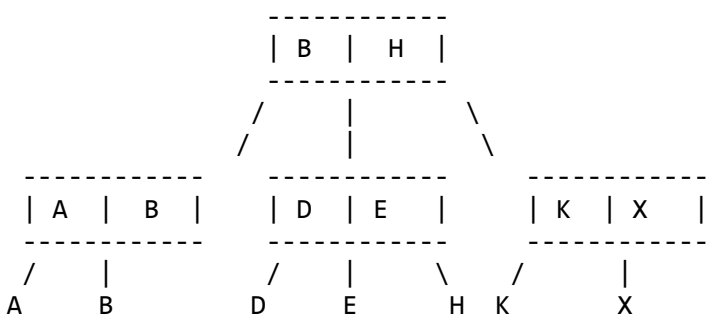
- If n is the root of the tree, then remove the node containing k. Replace the root node with the other child (so the final tree is just a single leaf node).
- If n has a left or right sibling with 3 kids, then:
  - remove the node containing k
  - "steal" one of the sibling's children
  - fix n.leftMax, n.middleMax, and the leftMax and middleMax fields of n's sibling and ancestors as needed.
- If n's sibling(s) have only 2 children, then:
  - remove the node containing k
  - make n's remaining child a child of n's sibling
  - fix leftMax and middleMax fields of n's sibling as needed
  - remove n as a child of its parent, using essentially the same two cases (depending on how many children n's parent has) as those just discussed

The time for delete is similar to insert; the worst case involves one traversal down the tree to find n, and another "traversal" up the tree, fixing leftMax and middleMax fields along the way (the traversal up is really actions that happen after the recursive call to delete has finished).

So the total time is 2 * height-of-tree = O(log N).

TEST YOURSELF #3

Question 1: Draw the 2-3 tree that results from deleting the value "X" from the following 2-3 tree:



Question 2: Now draw the tree that results from deleting the value "H" from the tree you drew for question 1.

solution

2-3 Tree Summary

In a 2-3 tree:

- keys are stored only at leaves, ordered left-to-right
- non-leaf nodes have 2 or 3 children (never 1)
- non-leaf nodes also have leftMax and middleMax values (as well as pointers to children)
- all leaves are at the same depth
- the height of the tree is O(log N), where N = # nodes in tree
- at least half the nodes are leaves, so the height of the tree is also O(log M) for M = # values stored in tree
- the lookup, insert, and delete methods can all be implemented to run in time O(log N), which is also O(log M)

Summary of Binary-Search Trees vs 2-3 Trees

	BST	2-3 Tree
where are values stored	every node	leaves only
extra info in non-leaf nodes	2 child ptrs	leftMax, middleMax, 3 child ptrs
worst-case time for lookup, insert, and delete (N = # values stored in tree)	O(N)	O(log N)
average-case time for lookup, insert, and delete (N = # values stored in tree)	O(log N)	O(log N)