Foundations of Computer Science

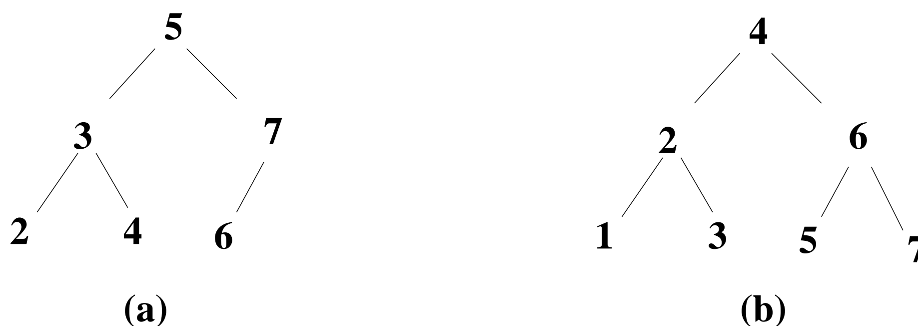
Manfred Kerber

2-3 Trees

Note: The following section is based on [Aho, Hopcroft, Ullman, 1983, Data Structures and Algorithms, Addison-Wesley] (abbreviated [AHU83] in the following).

Binary search trees allow efficient search of elements according to a key assumed they are balanced. In order to generate balanced binary search trees we can sort the elements, pick the middle most element and choose it as top element. However, when the structure changes, for instance, since further elements are to be inserted later this approach is impracticable. Assume we have a tree of size n. We may be lucky and the element can be inserted in the usual way and the tree is still balanced, that is, the time complexity of insert is $O(\log_2(n))$. We may, however, be unlucky and have to touch every node and rebuild the whole tree, that is, in this case the time complexity is $O(n)$.

An example is the following prefectly balanced binary search tree in (a). If we add now the new element `1`, we need to rearrange the tree in a way that every element changes its position.



(Example taken from [AHU83, p.170].)

The example can be generalized by taking the numbers $2, 3, 4, 5, \ldots, 2^h$ and arrange them analogously to (a) and then add 1 as in (b). That is, the complexity of inserting a single element in a perfectly balanced binary search tree (and maintaining the property being of a perfectly balanced binary search tree) is in the worst case n (with n the size of the tree).

2-3 trees have been invented to avoid this problem. We will first introduce them, see how to find elements in them, and how to insert elements.

Definition. A *2-3 tree* is a tree with the following two properties:

(1) Each interior node has two or three children.

(2) Each path from the root to a leaf has the same length.

[AHU83, p.169].

As with binary search trees we assume that the elements of he tree has a key which can be compared with respect to size (e.g., `<`, `==`, `>` on numbers, or lexicographical order on strings). The

elements all occur at leaves, inner nodes consist of one or two keys as follows: we record the smallest element that is a descendant of the second child, and if there is a third child, we record the key of the smallest element descending from the third child as well. An example is given in Figure 1 (see [AHU83, p.170]).

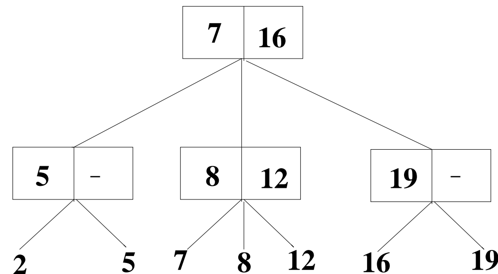


Figure 1: Example of a 2-3 tree

This tree has a height of $h = 2$ and 7 leaves. The number of leaves is in general in the range from 2^h to 3^h.

Insertion of new nodes into a 2-3 tree If we want to insert a new node `x` into a 2-3 tree we always have first to find a subtree into which it is to go and secondly to check whether it still fits in the appropriate place. Then two cases must be distinguished: If it fits we directly insert it, if not, we have to increase the tree. In any case we have to adjust the labels at the inner nodes from the newly inserted node to the root.

Let us first look at an example for the first (simple) case. For instance, if we want to insert a node with key 18 in the tree in Figure 1 we first compare 18 with 7. Since $18 > 7$ we know that it will NOT be inserted in the left subtree. Next we compare it with 16. Since $18 > 16$ we know that it will have to go into the right subtree. (We would recursively continue to compare until we come to an inner node which has only leaves as children.) The next inner node has only leaves as children. Since there are only two, it is possible to fit the 18 in. That is, the leaves will be 16, 18, and 19. The labels have to be readjusted to 18 and 19. We get the tree in Figure 2.

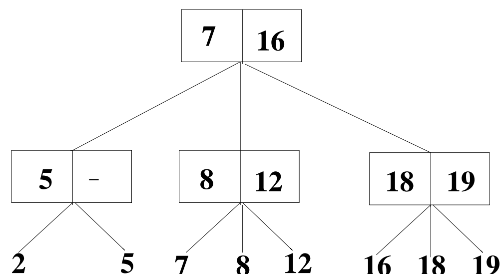


Figure 2: 2-3 tree with 18 inserted

Now, let us consider the second, more complicated case. We want to insert an element with key `10` into the tree in Figure 2. Making the same comparisons as before, we see that it has to go into the middle part. We first generate an (ill-formed) intermediate state, 10 would become a fourth child in the middle block, as displayed in Figure 3.

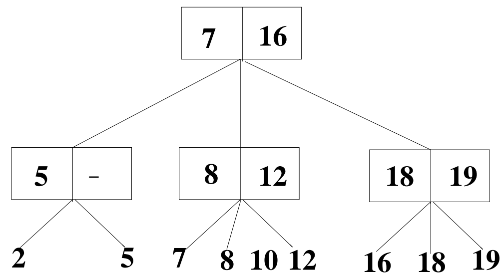


Figure 3: 2-3 tree with 10 to insert

Since we cannot fit the element in, we have to split the middle node and get another (ill-formed) structure, as displayed in Figure 4.

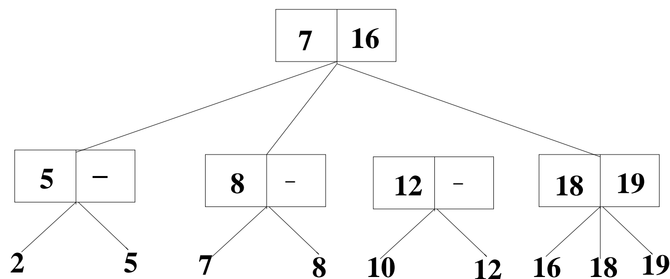


Figure 4: 2-3 tree with subtree to insert

We have the same problem as before, however, at a level higher (and we may have to go up to the root of the tree this way), namely that there is no room to insert the new intermediate node, since the root would have 4 children. In order to make room we split the top level node into two and insert it. The two nodes have to be kept together by a new root node. All in all we get the tree in Figure 5.

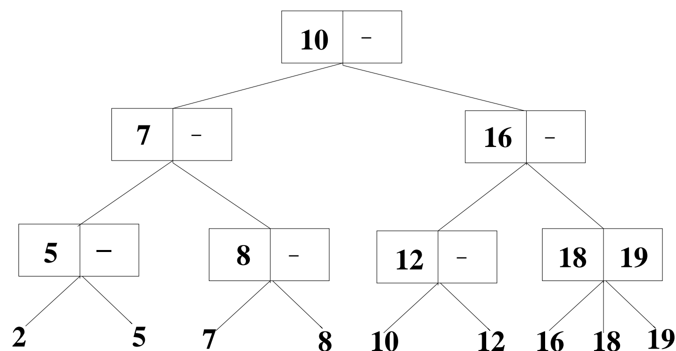


Figure 5: 2-3 tree with 10 inserted

Note that – although easy in principle – a single insertion step of a node is a bit complicated in detail (for further information, see [AHU83]. However, when we insert a new node then we have to insert additional inner nodes maximally on a path from the new element to the root of the tree, that is, the worst case time complexity of `insert` is $O(\log(n))$.

2-3 trees have been generalized to allow for more children per node (between floor$(m/2)$ and m children) to so-called B-trees. For details see [AHU83, p.369 ff].