Topics: Balanced Search Trees

1 Balanced Search Trees

In the worst case, binary search trees can run as slowly as in linear time due to imbalance introduced by insertions and removals. We can guarantee logarithmic time algorithms by forcing our search trees to be balanced.

1.1 Balanced BSTs

There are examples of BSTs that are required to be balanced by the introduction of more constraints on the tree, such as AVL trees and red-black trees. Both structures use *rotations* when an item is added or removed in order to restore balance. We won't discuss either of them, but you are required to know red-black trees.

1.2 B-Trees

Another approach to balanced trees is to grow a tree *upward*. In order for this to make any sense, we must modify our tree structure. We examine *B-trees* as an example.

The major modification B-trees make is that each node may contain multiple elements. B-trees have an *order*, which is the maximum number of children a node can have. All internal nodes except the root node in an order n tree must have at least $\lceil \frac{n}{2} \rceil$ children. A node that has k children must contain $k-1$ elements. Leaf nodes must be all on the last level and are always empty. For simplicity, we will not draw them in, but they are required to maintain the above properties.

B-trees also have an ordering property similar to that of BSTs. Within a single level, all elements must be less than the element to its right. The children and elements in a node are arranged in an alternating pattern, so between any two children is an element. The subtree to the left of a key may only contain elements less than it, and the subtree to the right only elements greater than it.

Searching a B-tree is almost exactly like searching a BST. Insertion also starts out the same way, with a position in the last internal level located for the new element, and the element placed there. However, this may cause *overflow*, in which the node the element was inserted in may now contain more than n elements. In such a case, the node is split, with each side getting half the elements, but one element promoted to the next level. So the parent node gets another child due to the split, but it also gets another element, so the property that there is one more child than elements is preserved. This may cause the parent node to overflow, but that can be dealt with in the same manner.

Removal is also done as in a BST. The element to be removed is swapped to the last internal level as in a BST and then deleted. This deletion may cause an *underflow*, in which the affected node now contains fewer than $\lceil \frac{n}{2} \rceil$ elements. In such a case, the node is merged with one if its siblings, and the element between them in the parent node is demoted to the newly combined node. Thus the parent node loses a child but also loses an element, so the child to element relationship is preserved. This may cause either overflow of the combined node or underflow of the parent node, but either case can be dealt with as before.

1.3 Skiplists

One structure that is not much of a tree at all is a *skiplist*. It is actually a two-dimensional linked list, but with elements duplicated in the vertical dimension. This duplication is probabilistic and results in logarithmic operations for the structure.

The bottom level of a skiplist always contains all the elements in the skiplist, as well as two more items representing $-\infty$ and $+\infty$. The elements in a particular level are ordered, so $-\infty$ is always all the way

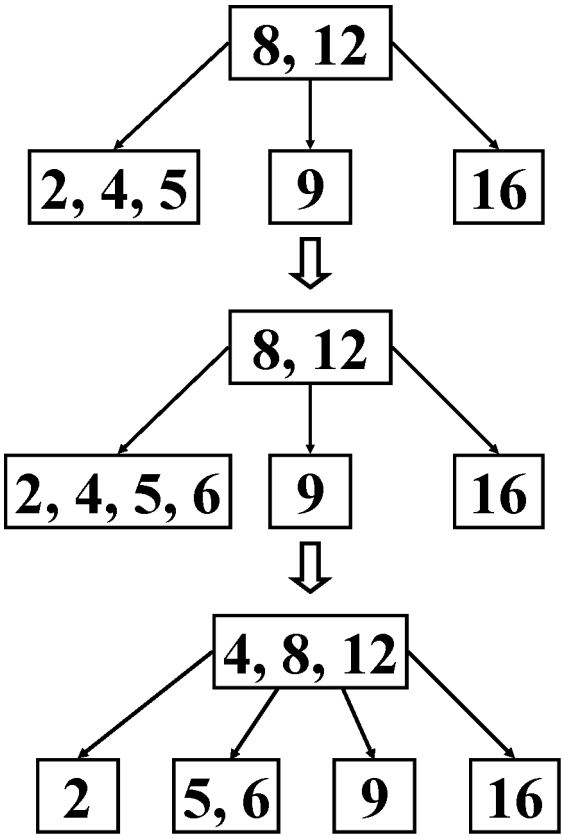

Figure 1: Insertion into an order 4 B-tree. When an overflow occurs, the overflowed node is split, and one key is promoted.

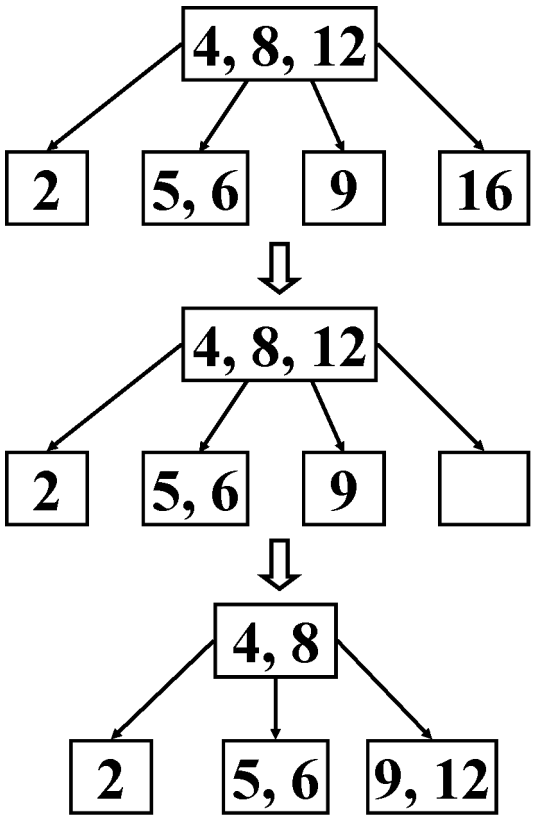

Figure 2: Removal from an order 4 B-tree. When an underflow occurs, the underflowed node is merged with a sibling, and the parent key between them is demoted. A resulting overflow is dealt with as in insertion.

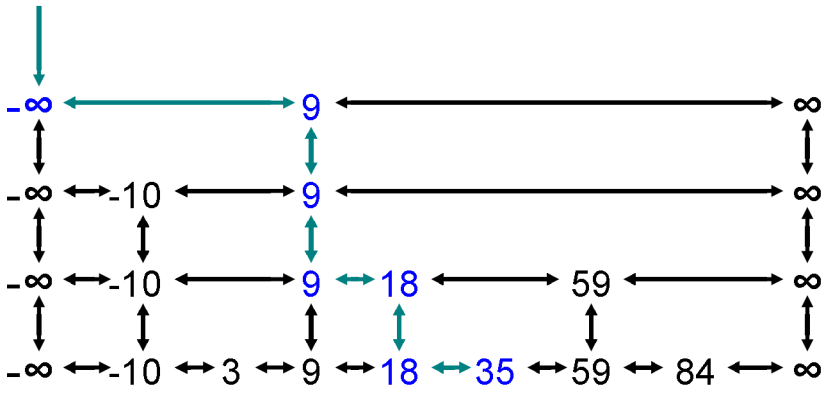

Figure 3: Searching for an element in a skiplist.

to the left and $+\infty$ all the way to the right. Now each subsequent level contains probabilistically half the elements of the previous level. This is accomplished by flipping a coin when an item is inserted to see if it should be added to the next level, and repeating until the coin orders us to stop. Each level must contain $-\infty$ and $+\infty$. The start of the skiplist is the top left corner.

To search for an element in a skiplist, we start at the top left. If the element to our right is larger than the element we are looking for, we go down a level. If it is smaller, we go right. If it is the element, well then we're done. We repeat this procedure from the next position, until we've located the element. Insertion consists of searching for the proper location to insert the new element, placing it there, and repeatedly flipping coins and adding the element to subsequent levels until the coin tells us to stop. Removal is just a search for the element in question, and removing it from each level in which it is.

It is not a trivial task to prove that the above operations take logarithmic time in a skiplist. We will not attempt such a proof here, but just state it as a fact.