Balanced binary search trees – AVL trees, 2-3 trees, B-trees

Professor Clark F. Olson (with edits by Carol Zander)

AVL trees

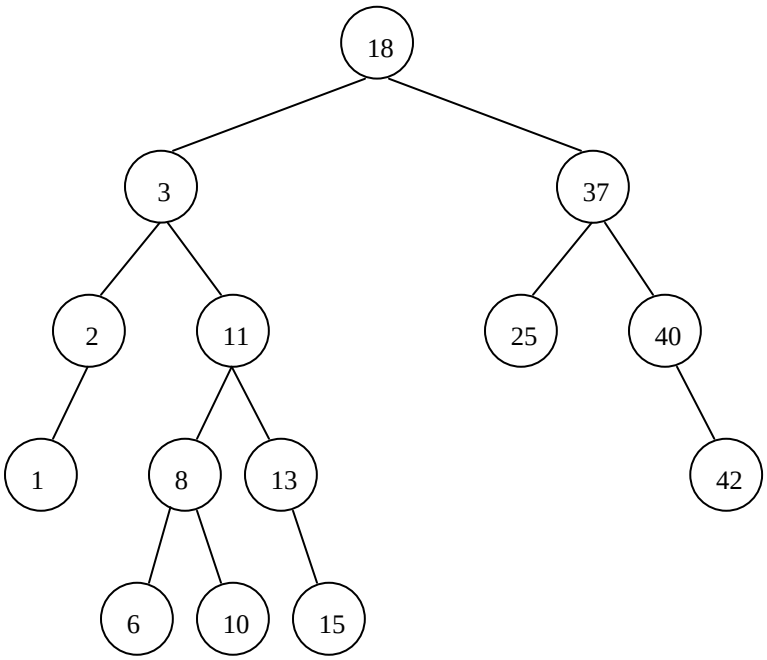
One potential problem with an ordinary binary search tree is that it can have a height that is O(n), where n is the number of items stored in the tree. This occurs when the items are inserted in (nearly) sorted order. We can fix this problem if we can enforce that the tree remains balanced while still inserting and deleting items in O(log n) time. The first (and simplest) data structure to be discovered for which this could be achieved is the AVL tree, which is names after the two Russians who discovered them, Georgy Adelson-Velskii and Yevgeniy Landis. It takes longer (on average) to insert and delete in an AVL tree, since the tree must remain balanced, but it is faster (on average) to retrieve.

An AVL tree must have the following properties:

- It is a binary search tree.
- For each node in the tree, the height of the left subtree and the height of the right subtree differ by at most one (the balance property).

The height of each node is stored in the node to facilitate determining whether this is the case. The height of an AVL tree is logarithmic in the number of nodes. This allows insert/delete/retrieve to all be performed in O(log n) time.

Here is an example of an AVL tree:



Inserting 0 or 5 or 16 or 43 would result in an unbalanced tree.

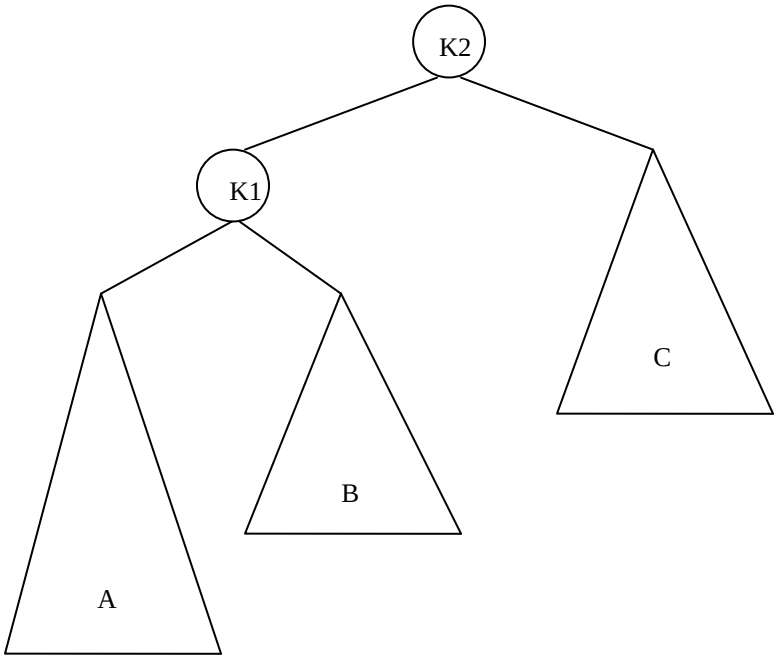
The key to an AVL tree is keeping it balanced when an insert or delete operation is performed. We will examine insertion in detail. If an insertion causes the balance property to be violated, then we must perform a tree "rotation" in order to fix the balance. If we start with an AVL tree, then what is needed is either a single rotation or a double rotation (which is two single rotations) on the unbalanced node and that will always restore the balance property in O(1) time. Note that the rotations are always applied at the lowest (deepest) unbalanced node and no nodes above it need to have rotations applied.

When a node becomes unbalanced, four cases need to be considered:

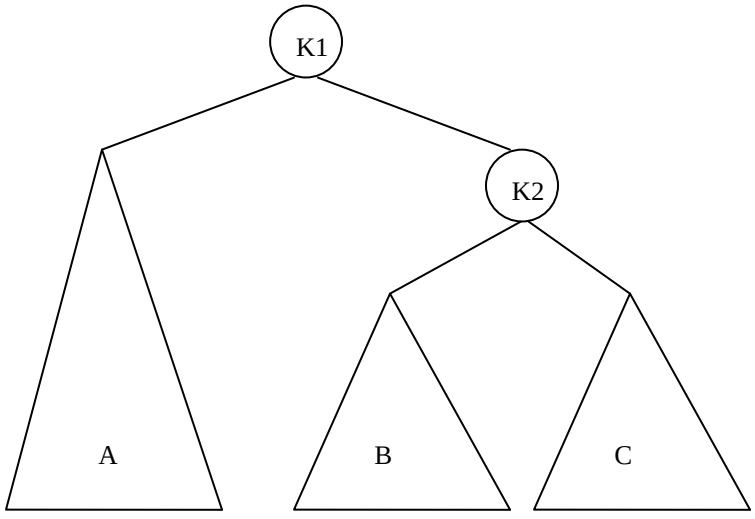
- Left-left: The insertion was in the left subtree of the left child of the unbalanced node.
- Right-right: The insertion was in the right subtree of the right child of the unbalanced node.
- Left-right: The insertion was in the right subtree of the left child of the unbalanced node.
- Right-left: The insertion was in the left subtree of the right child of the unbalanced node.

Single rotations

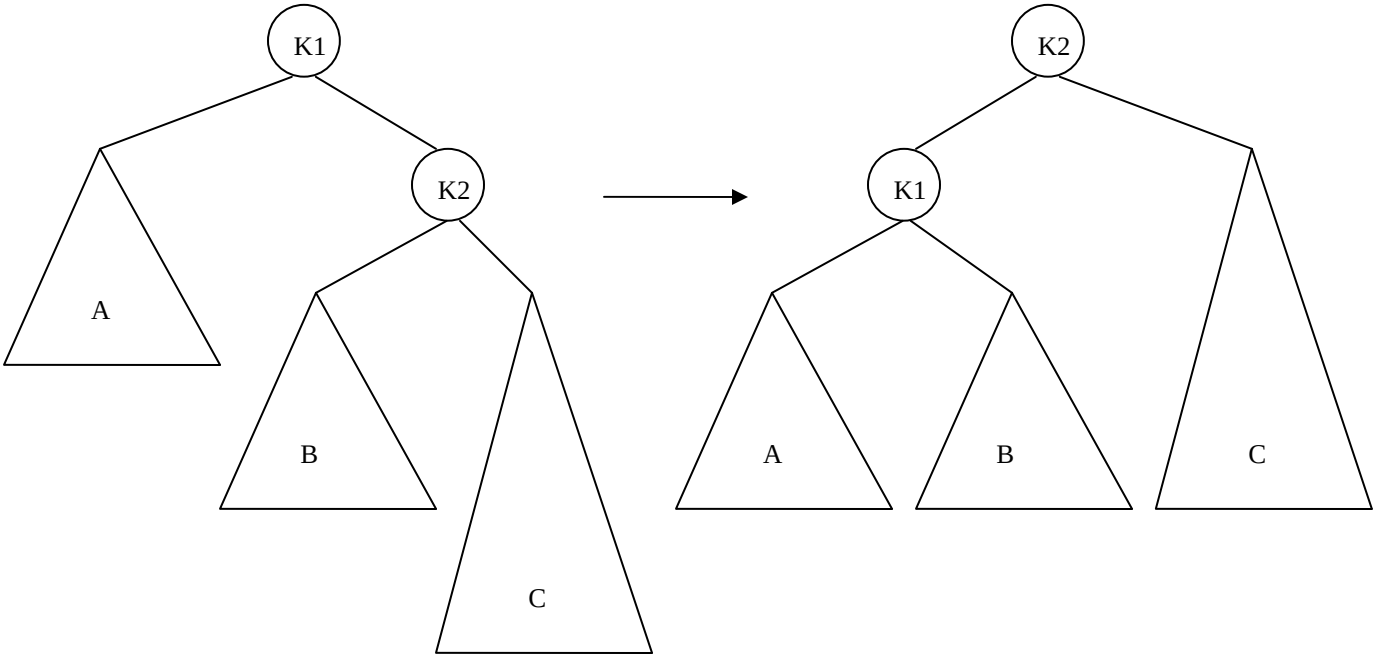
The first two require single rotations and the last two require double rotations to rebalance the tree. In general, a left-left looks like this:



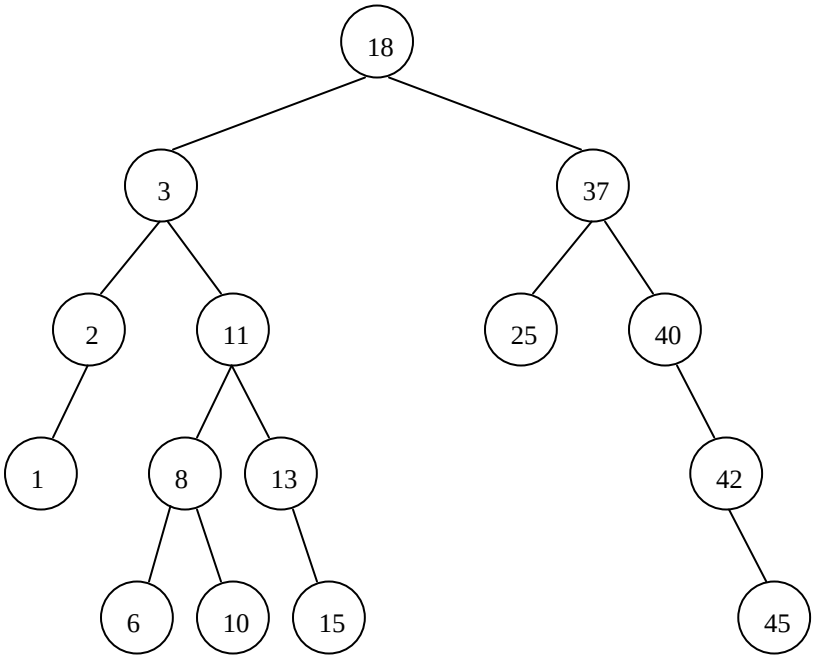
K2 is the unbalanced node. Nothing above it matters. The left-left subtree A always has height one greater than the right subtree C, which unbalances the tree. The solution is to make K1 the root of this (sub)tree with K2 as its' right child and B as the right-left subtree:



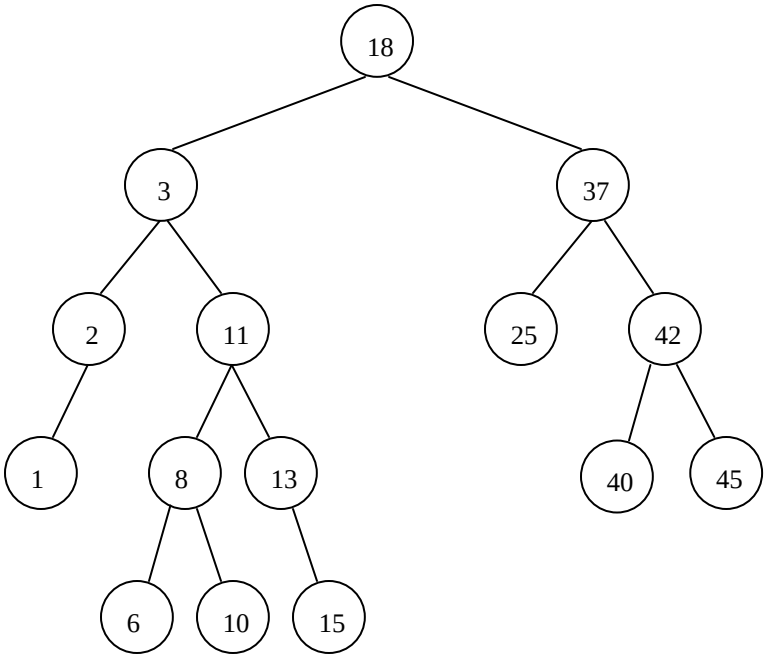
A right-right is symmetrical.



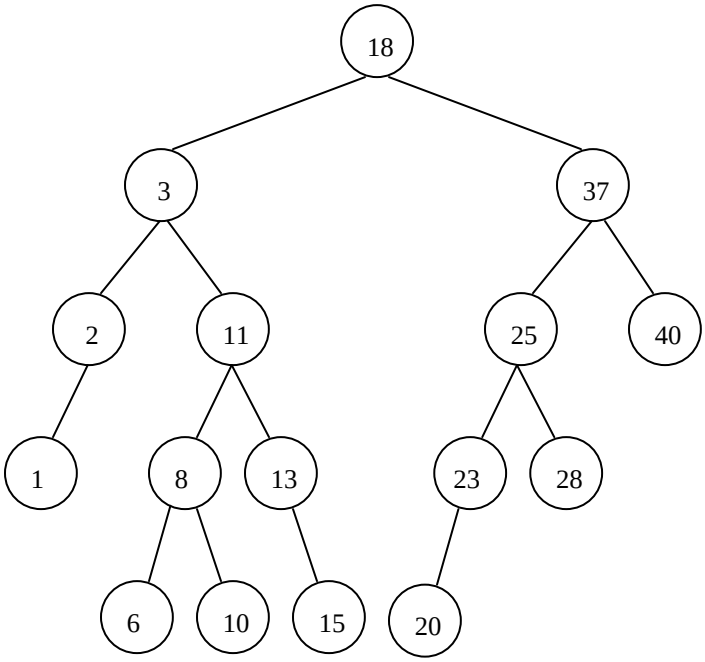
Here is a right-right example with actual data:



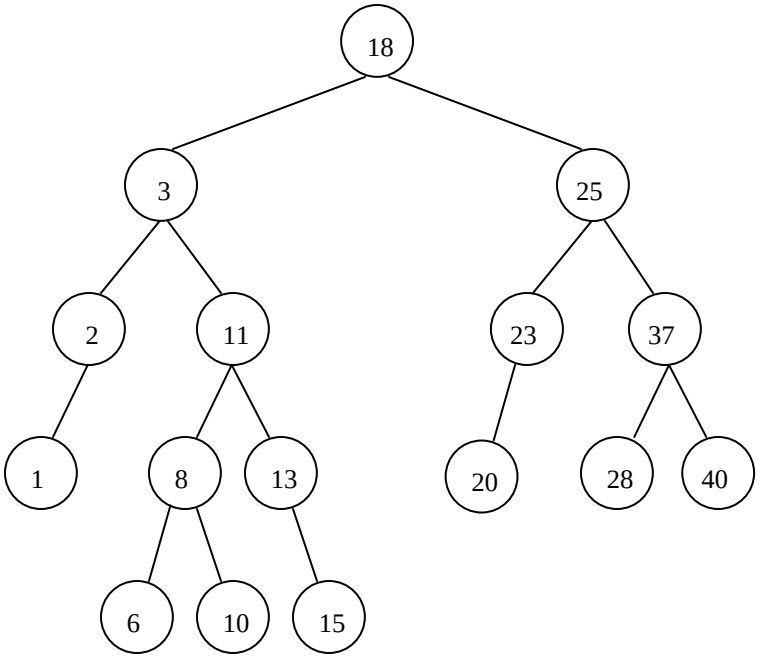
Nodes 37 and 40 are unbalanced. The lower node needs to be fixed. This is a right-right, since 42 is the right child of 40 and 45 is in the right subtree of 42. In this case, two of the subtrees, A and B, are empty. The subtree C has the single node 45. Rotating (with right child) yields:



Here is a left-left, where the unbalanced node is not as deep:

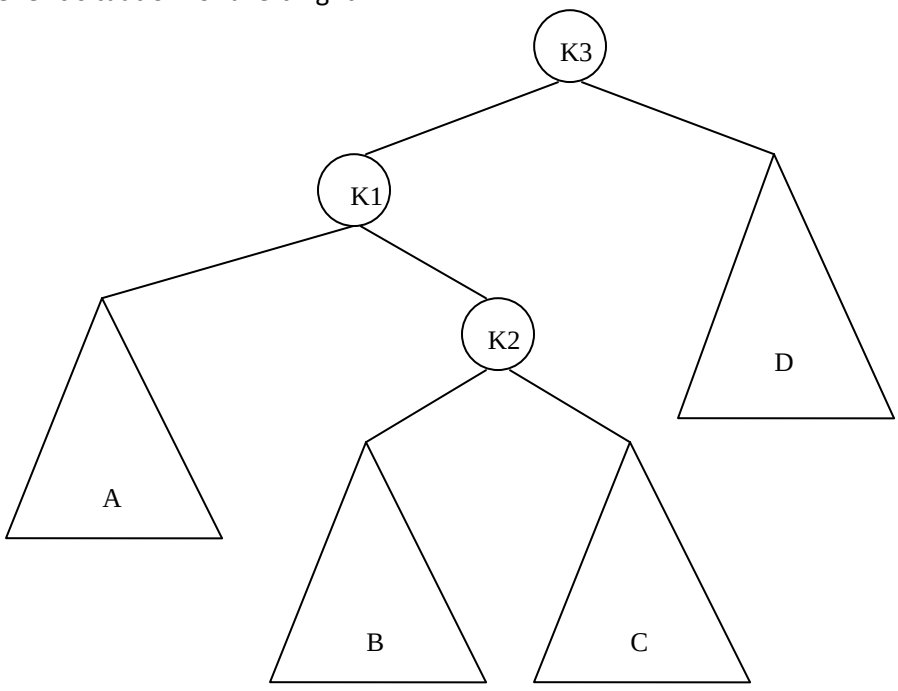


Node 37 is unbalanced. The inserted node is in the left subtree of the left child of 37. Therefore, we rotate 37 with its' left child:

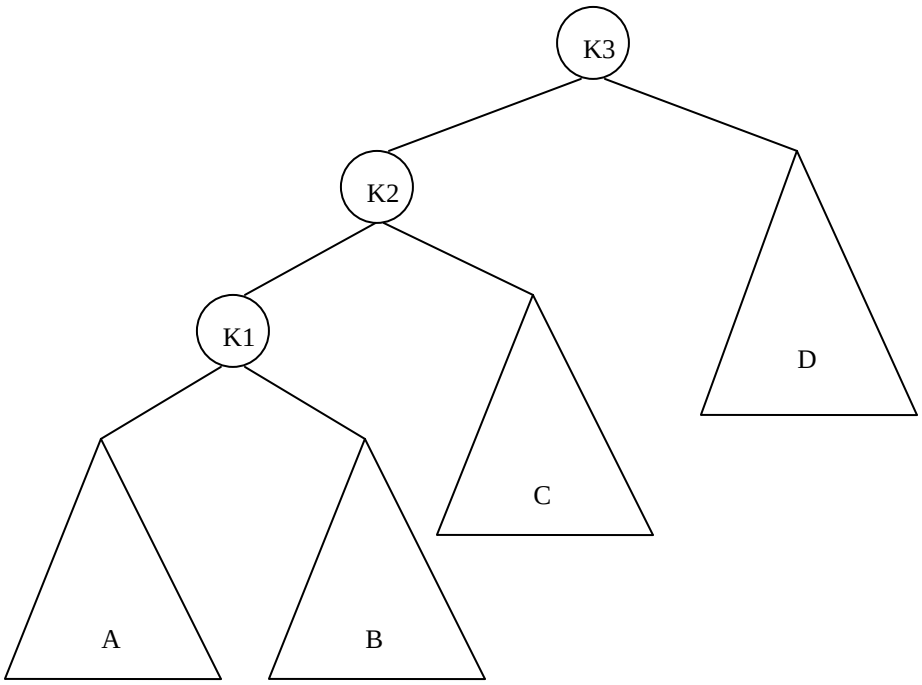


Double rotations

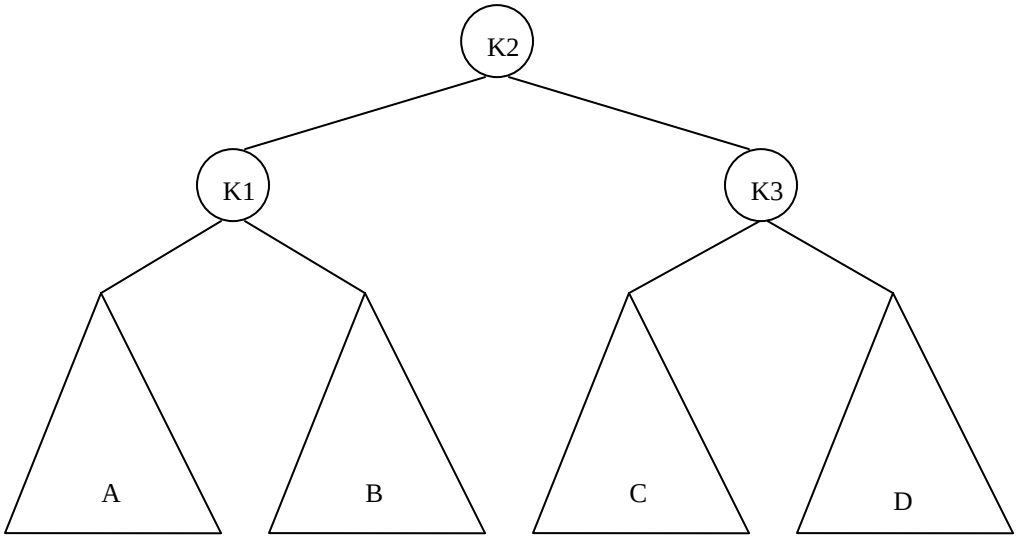
When we have the left-right and right-left cases, a single rotation is not sufficient to rebalance the tree. In this case, we need to perform a double rotation, which consists of two single rotations. Here is the generic situation for a left-right:



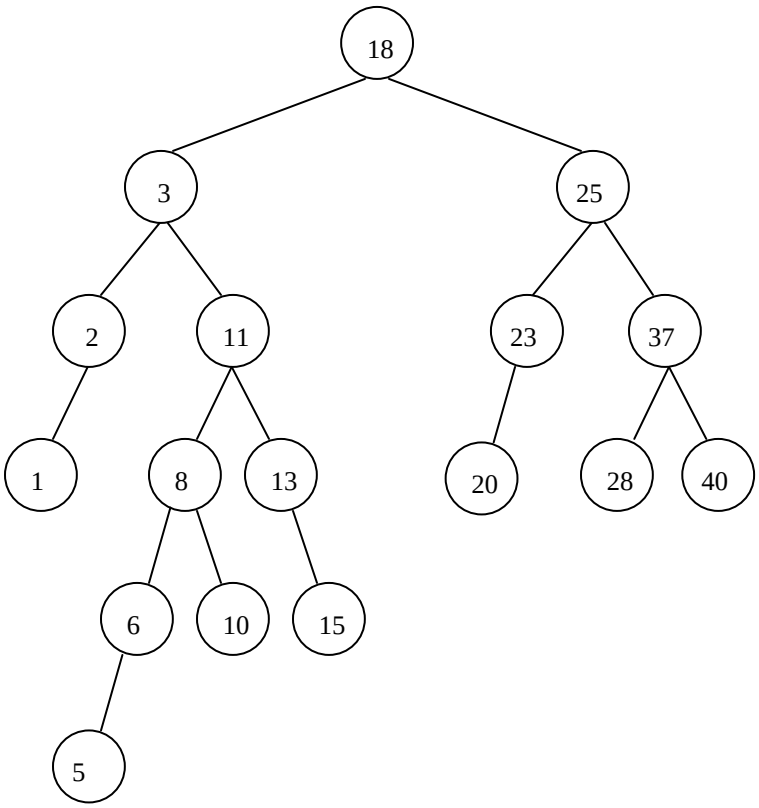
To fix this, we need to rotate K2 all the way to the top. First, we do a right-right rotation of K1 with K2 and then a left-left rotation of K3 with K2. The first rotation (K1/K2) yields:



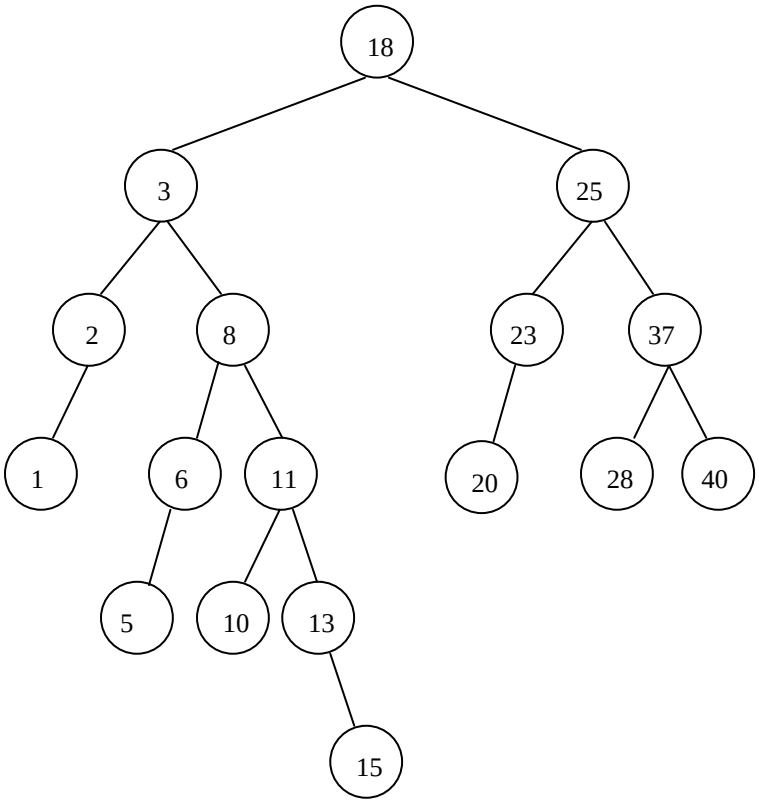
The second rotation (K2/K3) gives us:



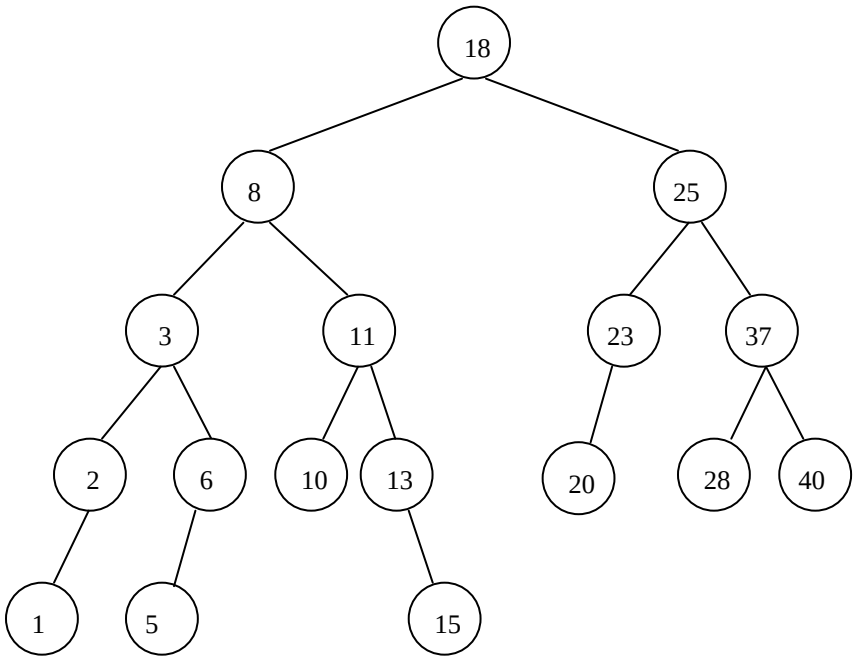
Conceptually, the K2 “splits” the K1 and the K3 and rises to the top, with the K1 and the K3 ending up being subchildren and A, B, C, and D being grandchild trees. Here is an example of a right-left with real data:



First rotation:



Second rotation:



It is worth noting that AVL trees are not commonly used in practice, since better balanced binary trees exist. However, these are more complex. The rotations in AVL trees demonstrate important concepts that are used in other balanced trees.

B-trees

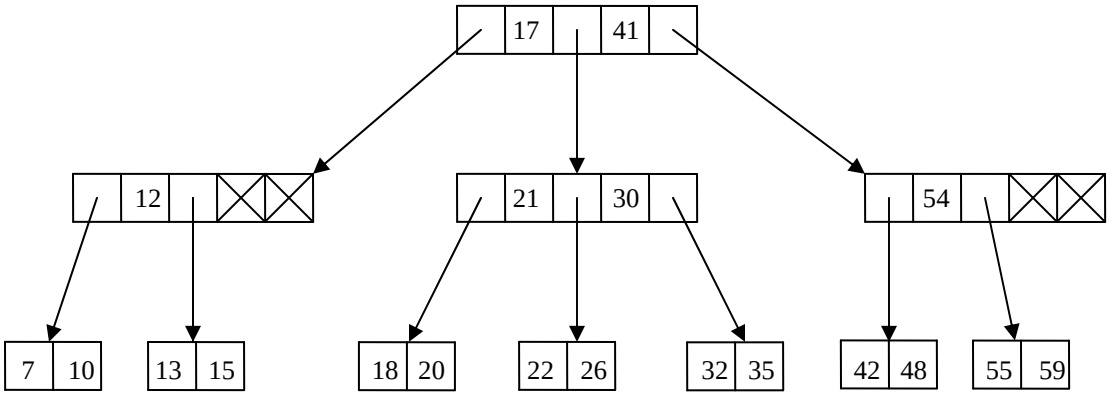
B-trees are balanced (non-binary) trees that are commonly used in practice. They allow much greater branching than in a binary tree (up to M children for a B-tree of order M) and have the following properties:

- Different implementations store data items at all nodes (our use) or at the leaves only.
- The root has between 2 and M children.
- All non-leaf nodes (except the root) have between $\lceil M/2 \rceil$ and M children.
- All leaves are at the same depth.
- If a non-leaf node has C children, then it stores C-1 keys. The i^{th} key is the smallest key in the *i +1* subtree. These keys guide the search process in the tree.

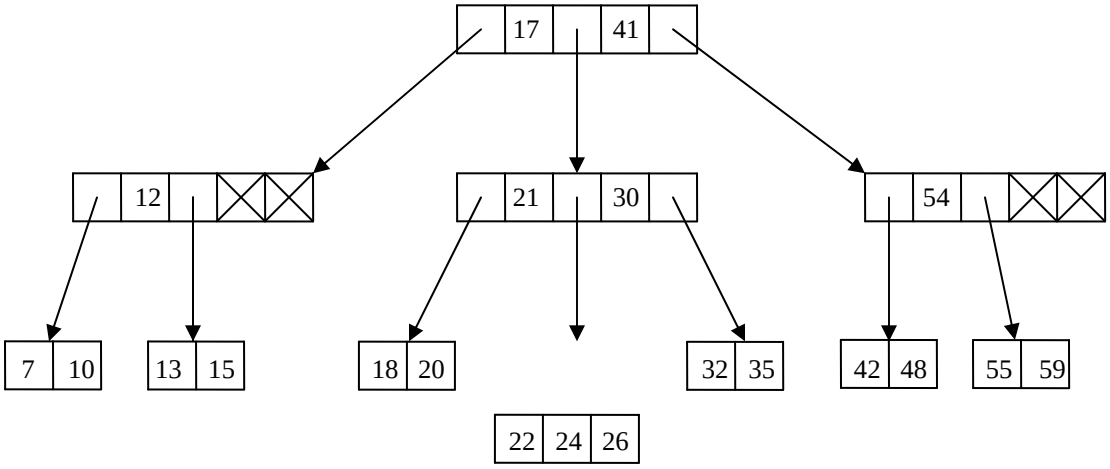
A 2-3 tree is a special case of a B-tree with M = 3. In a 2-3 tree, we might store data as follows:

```
struct node23 {
   node23* left;
   Object* smallItem;
   node23* middle;
   Object* largeItem;
   node23* right;
};
```

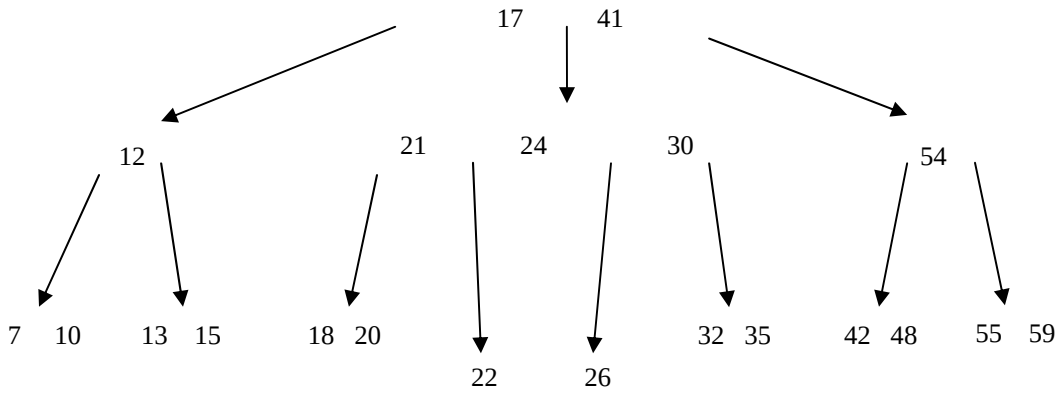
Here is an example of a 2-3 tree (only values are shown in leaf nodes, assume all pointers are NULL):



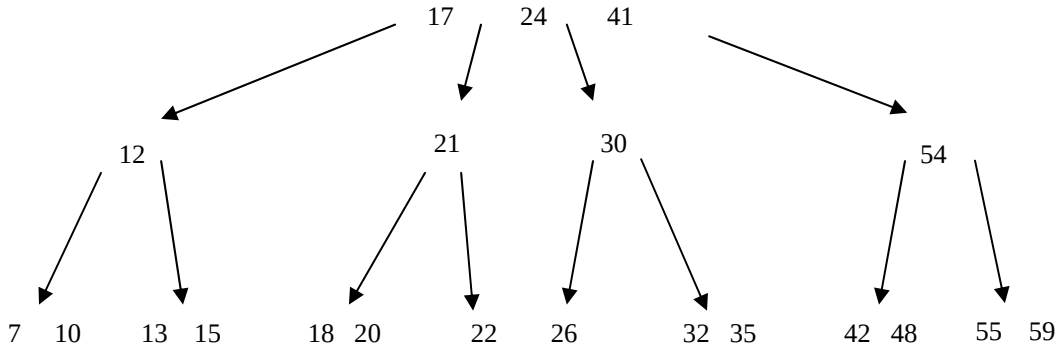
When we insert a value in a B-tree, we try to find a spot in the appropriate leaf. If none exists, then the leaf is split into two leaves and we propagate the process upwards. B-trees actually grow up rather than down, because if you insert enough time you have to split a leaf, which causes, the parent of the leaf to split, and, so on, all the way up to the root. For example, let's say we tried to insert 24. This would cause one of our leaves to have to be split since it cannot contain three values:



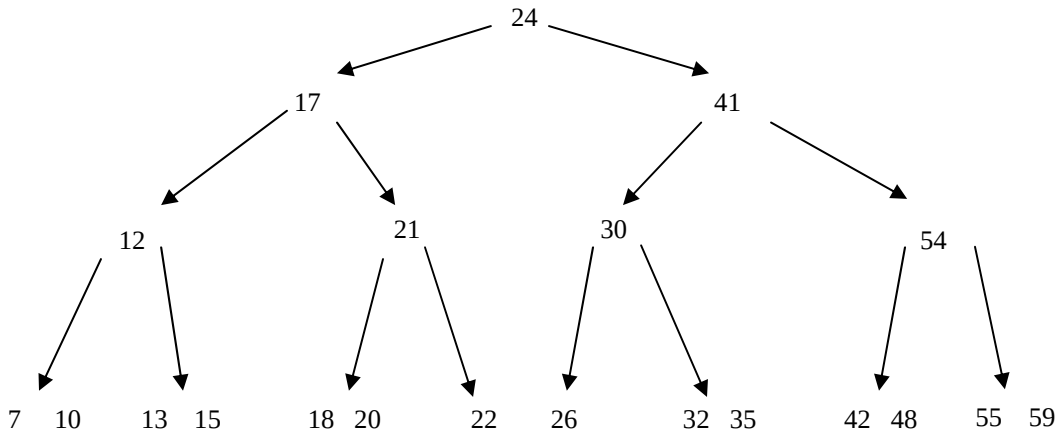
The middle value, the 24, moves up to its parent. However, there isn't room for another value in the parent node, which means we need to split it, too. The next diagrams only show values.



Again, the middle value, the 24, moves up to its parent, the tree root. Now there isn't enough space at the root and a new root must be created.



So we split the root and create a new root that has the old (split) root as children. Notice that whenever a node has four children in the interim process of inserting, a resulting parent is created with two children and those children will each point to half of the original four children.



Like AVL trees, 2-3 trees guarantee logarithmic time insert, delete, and retrieve. B-trees do also (since a 2-3 tree is a B-tree with M = 3). B-trees are commonly used in practice for applications such as databases. The reason for this is that the most expensive operation is accessing a disk. This is much more expensive than typically operations in memory. Therefore, we want to minimize the number of disk accesses and this is done by make the tree as short as possible. The more branches each node has, the shorter the tree will be. So, how large should we make M? The ideal case is to make a node in the B-tree large enough to fill an entire disk block (but no more), since an entire disk block is usually read at a time. Since the tree height is minimal, this minimizes the total number of disk accesses (swapping a node out) that are performed for operations on the tree, such as retrieving an item.

One optimization that can be used -- if an item doesn't fit in the current leaf, a leaf can put an item "up for adoption." In this case, we check to see if a sibling leaf can take that item without performing a split (note that parent values may have to be adjusted).