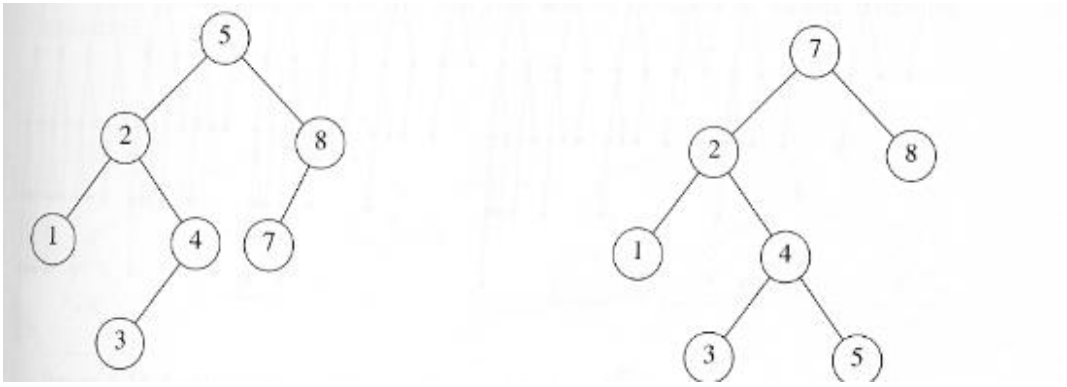
AVL Trees

Although the average depth of a binary search tree is $O(\log n)$, there exist important cases (e.g. when the data is mostly sorted) that cause the depth of a bst to approach $O(n)$, thus rendering the tree little more efficient than an array or linked list. On the other hand **Adelson-Velskii-Landis (AVL) trees** are binary search trees whose worst-case depths are $O(\log n)$. Moreover, an AVL tree is a binary search tree for which every node n satisfies

$$|\text{height(n.left)} - \text{height(n.right)}| \leq 1.$$

Example 1. Which if any of the following trees are AVL trees?



Lemma. The Fibonacci sequence of numbers $f_0, f_1, \ldots, f_n$ is recursively defined as $f_0 = 0$, $f_1 = 1$, and $f_n = f_{n-1} + f_{n-2}$ for all $n \geq 2$. This sequence grows exponentially.

Proof of Lemma. The recurrence relation

$$f_n = f_{n-1} + f_{n-2}$$

is called a **second-order linear homogeneous recurrence relation with constant cofficients**. It is second order because computing the current number f_n requires going back at most two units in history and computing f_{n-2}. It is linear with constant coefficients because the historical terms f_{n-1} and f_{n-2} are added as linear combinations, where the scalars are both equal to one. In other words $f_n = 1 \cdot f_{n-1} + 1 \cdot f_{n-2}$. Finally, it is homogeneous because f_n only depends on the linear combination of the historical terms, and nothing else. The general solution for a second order linear homogeneous recurrence relation with constant cofficients is given as

$$f_n = c_1 r_1^n + c_2 r_2^n,$$

where r_1 and r_2 are the solutions to the quadratic equation

$$r^2 + br + c = 0,$$

where $-b$ and $-c$ are the constant coffecients of the $n-1$ and $n-2$ historical terms respectively. Solving this equation for the Fibonacci sequence requires solving $r^2 - r - 1 = 0$ which has the solutions $r_1 = \frac{1+\sqrt{5}}{2}$ and $r_2 = \frac{1-\sqrt{5}}{2}$. It can be shown that $f_n = c_1 r_1^n + c_2 r_2^n = \Omega((\frac{1+\sqrt{5}}{2})^n)$, and hence grows exponentially.

Theorem. Let n be the number of nodes of an AVL tree T. Then the height h of T is $O(\log n)$.

Proof of Theorem. Let $N(h)$ denote the minimum number of nodes of a balanced binary search tree having height h. Then $N(0) = 1$, $N(1) = 2$, $N(3) = 7$, and, for $n \geq 2$,

$$N(h) = 1 + N(h-1) + N(h-2).$$

Thus, $N(h)$ grows faster than the Fibonacci sequence, and thus has exponential growth. Hence, there exists $a > 1$, and $C > 0$ such that $N(h) \geq Ca^h$, for sufficiently large h, implying that

$$h \leq \log_a N(h) - \log_a C \leq \log_a n - \log_a C = \mathrm{O}(\log \mathrm{n}).$$

For AVL trees we are interested in the same operations, as with other binary seach trees; namely find(), insert(), and delete(), and find_min(). The implementation of the find() and find_min() operations do not change from the bst implementation. However, insert() and delete() require new methods, since, e.g. the insertion of data into an AVL tree may lead to an unbalanced tree.

Observation. Assuming that insertion and deletion are performed as in the case of binary search trees, inserting or deleting a node from an AVL tree will cause a node n to be unbalanced by at most 2. In other words,

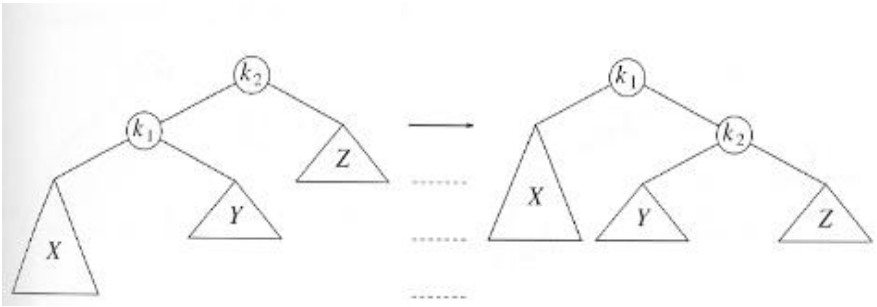
$$|\text{height(n.left)} - \text{height(n.right)}| \leq 2,$$

Moreover, the unbalanced nodes will exist on the same branch of the tree.

From the above observation we need only devise a method that re-balances trees which are unbalanced by at most 2, and for which imbalances occur on only one branch of the tree; namely the branch for which a node was inserted or deleted. Moving from the root down, let n be the last node on the branch which is unbalanced. n has four possible sub-trees emanating from it: ll, rr, rl, lr. For example, lr represents the right subtree of the left child of n. Of these four trees, the tallest one is responsible for creating the imbalance.

Single Rotations

If the tallest tree is either ll or rr, then the tree can be rebalanced by a **single rotation**, as is shown in the following diagram for the ll case (Here we assume that $n = k_2$ is the last unbalanced node).



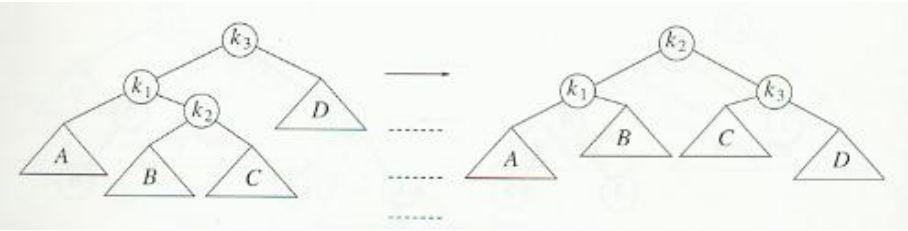
Proposition 1. The rotation shown in the above diagram rebalances the tree. Moreover, the resulting tree is a binary search tree.

Proof of Proposition 1. The resulting tree is clearly a binary search tree, by checking that all left (right) subtree key values are less than the root of the left (right) subtree of the rotated nodes k_1 and k_2. Moreover, Let h denote the height function for binary trees. Then we have $h(X) = h(Z) + 1$, and $h(Y) = h(X) - 1$. These equations tell us that the tree rooted at k_2 is balanced (since $h(Y) = h(Z)$), and from this the first equation implies that the tree rooted at k_1 is balanced.

Example 2. Insert the nodes $7, 6, 5, 4, 3, 2, 1$ (in that order) into an intially empty AVL tree.

Double Rotations

If the tallest tree is either lr or rl, then the tree can be rebalanced by a **double rotation**, as is shown in the following diagram for the lr case (Here we assume that $n = k_3$ is the last unbalanced node).



Proposition 2. The double rotation shown in the above diagram rebalances the tree. Moreover, the resulting tree is a binary search tree.

Proof of Proposition 2. Again, one can check each of the subtrees that their key values are appropriate for the given parent key in order to make a legal binary search tree. To see that the tree is balanced, it suffices to notice that the heights of both A and B, and C and D each differ by at most one. For example, assume that the item was added to B. Then we must have $h(B) = h(D)$, since the difference in height of two in the subtrees of k_3 is caused by the the ancestors k_1 and k_2 of B. Thus, the height of C must be one less than the height of A.

Example 3. Insert the nodes $12, 2, 6, 7, 9, 8$ (in that order) into an intially empty AVL tree.

Example 4. Insert the nodes $3, 2, 1, 4, 5, 6, 7, 16, 15, 14, 13, 12, 11, 10, 8, 9$ (in that order) into an intially empty AVL tree.

Exercises.

1. What is the minimum number of nodes that a balanced tree of height 15 can have?

2. Prove that if keys $1, 2, \ldots, 2^k - 1$ are inserted in order into an intially empty AVL tree, then the resulting tree is perfect. Hint: use mathematical induction.

3. Insert 2,1,4,5,9,3,6,7 into an intially empty AVL tree. Redraw the tree each time a rotation is required.

4. Consider the diagram on page 5 of the lecture notes on AVL trees that shows the double rotation. List all of the pointers that need to be updated after the rotation. Provide the new value for each pointer. Hint: every node has three pointers: parent, left, and right, and every pointer that needs updating can be expressed in terms of either k_1, k_2, or k_3. For example, if the parent pointer of k_2's right child needs updating, then we would write it as k_2.right.parent.

Exercise Hints and Answers.

1. 2583. Use recurrence $N(h)$.

2. Basis step: true for $k = 1$. Inductive step: assume that if keys $1, 2, \ldots, 2^j - 1$ are inserted in order into an intially empty AVL tree, then the resulting tree is perfect, for $j = 1, \ldots, k$. Show that it is also true when keys $1, 2, \ldots, 2^{k+1} - 1$ are inserted into an initially empty tree. After inserting keys $1, 2, \ldots, 2^k - 1$, the resulting tree is perfect, by the inductive assumption, with the root of the tree being 2^{k-1}. Furthermore, inserting the next 2^{k-1} keys produces a right subtree that has $2^k - 1$ nodes, and is perfect (again by the inductive assumption) with height k. At this point the left subtree has $2^{k-1} - 1$ nodes, and is perfect, while the right subtree has $2^k - 1$ nodes and is perfect. They have a height difference of one. Now, upon inserting the next key, the right tree will now have a height of $k+1$, which makes the entire tree unbalanced at the root. After a single rotation, the left tree will again be perfect with $2^k - 1$ in the left subtree, while the right subtree now has $2^k - 1 - (2^{k-1} - 1) = 2^{k-1}$ nodes (we subtract away 1 since the root of the right subtree is now the root of the entire tree, and also subtract away $2^{k-1} - 1$, since this is the number of nodes in the right-left subtree that moved over to the left subtree during the rotation). All that remains is to add the final $2^{k-1} - 1$ keys. Notice that the 2^{k-1} keys in the current right subtree are positioned exactly as they would have been had they been inserted in order into an initially empty tree. In other words, if we ignore the root and left subtree of the existing tree, and focus solely on the right subtree, then it is identical to the tree we would have obtained had we inserted $2^k + 1, \ldots, 2^k + 2^{k-1}$ into an intially empty tree. Therefore, by the inductive assumption, the insertion of the remaining keys will yield a perfect right subtree with $2^k - 1$ nodes. In other words, the entire resulting tree is perfect.

3. Before the last node 7 is inserted, the tree will be perfect with nodes 4,2,6,1,3,5,9 (here we are ordering the nodes according to their complete-tree indices).

4. Eight pointers must be updated:

```
k2.parent = k3.parent
k3.parent = k2
tmp = k2.right
k2.right = k3
tmp.parent = k3
k3.left = tmp
tmp = k2.left
tmp.parent = k1
k1.right = tmp
k1.parent = k2
```