AVL trees

Dynamic set ADT

A dynamic set ADT is a structure that stores a set of elements. Each element has a (unique) key and satellite data. The structure supports the following operations.

Search(S, k) Return the element whose key is k.

Insert(S, x) Add x to S.

Delete(S, x) Remove x from S (the operation receives a pointer to x).

Minimum(S) Return the element in S with smallest key.

Maximum(S) Return the element in S with largest key.

Successor(S, x) Return the element in S with smallest key that is larger than $x.\mathrm{key}$.

Predecessor(S, x) Return the element in S with largest key that is smaller than $x.\mathrm{key}$.

Motivation

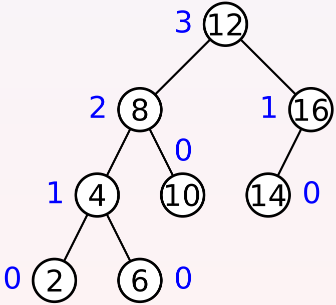
- In a binary search tree, all operation take $\Theta(h)$ time in the worst case, where h is the height of the tree.
- The optimal height of a binary search tree is $\lfloor \log n \rfloor$.
- Even if we start with a balanced tree, insertion/deletion operations can make the tree unbalanced.
- An AVL tree is a special kind of a binary search tree, which is always kept banalced.

AVL tree

An AVL tree is a binary search tree such that for every node x,

$$|\mathrm{height}(x.\mathrm{left}) - \mathrm{height}(x.\mathrm{right})| \leq 1$$

(we assume that $\mathrm{height}(\mathrm{NULL}) = -1$)



AVL trees are named after their inventors, Georgy Adelson-Velsky and Evgenii Landis.

The height of an AVL tree

Theorem

The height of an AVL tree is $\Theta(\log n)$.

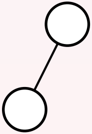
- Let n_k be the minimum number of nodes in an AVL tree with height k.
- $n_0 = 1$.

The height of an AVL tree

Theorem

The height of an AVL tree is $\Theta(\log n)$.

- Let n_k be the minimum number of nodes in an AVL tree with height k.
- $n_0 = 1$.
- $n_1 = 2$.

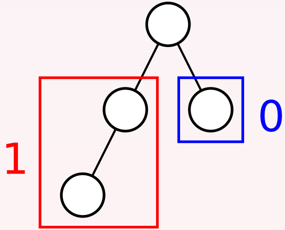


The height of an AVL tree

Theorem

The height of an AVL tree is $\Theta(\log n)$.

- Let n_k be the minimum number of nodes in an AVL tree with height k.
- $n_0 = 1$.
- $n_1 = 2$.
- $n_2 = 2 + 1 + 1 = 4$.

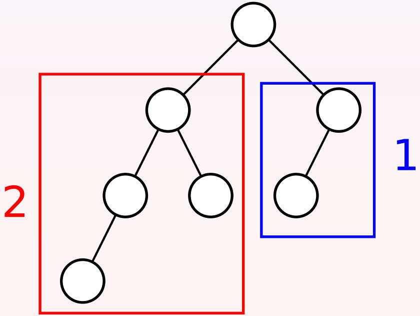


The height of an AVL tree

Theorem

The height of an AVL tree is $\Theta(\log n)$.

- Let n_k be the minimum number of nodes in an AVL tree with height k.
- $n_0 = 1$.
- $n_1 = 2$.
- $n_2 = 2 + 1 + 1 = 4$.
- $n_3 = 4 + 2 + 1 = 7$.
- $n_k = n_{k-1} + n_{k-2} + 1$.



The height of an AVL tree

- $n_k = F_{k+3} - 1$ where F_k is the k-th Fibonacci number.
  The proof is by induction:
  Base: $n_0 = 1 = F_3 - 1$, $n_1 = 2 = F_4 - 1$.
  $n_k = n_{k-1} + n_{k-2} + 1 = (F_{k+2} - 1) + (F_{k+1} - 1) + 1 = F_{k+3} - 1$
- It is known that
$$F_k = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^k - \left(\frac{1-\sqrt{5}}{2}\right)^k}{\sqrt{5}} \geq \frac{\left(\frac{1+\sqrt{5}}{2}\right)^k - 1}{\sqrt{5}}$$
- Therefore,
$$n_k \geq \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{k+3} - 1}{\sqrt{5}} - 1 \geq \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{k+3}}{\sqrt{5}} - 1.45$$
$$k \leq \log_{(1+\sqrt{5})/2}\left(\sqrt{5}(n_k + 1.45)\right) - 3 \leq 1.441 \log n_k .$$
- The height of an AVL tree with n nodes is $\leq 1.441 \log n$.

Implementation

- A node in an AVL tree has the same fields defined for binary search tree (key, left, right, p).
- Additionally, every node has a field height which stores the height of the node.

Searching

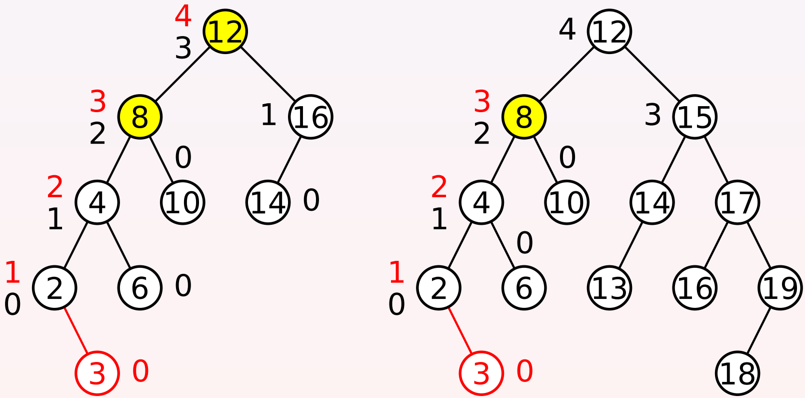
- The Search operation is handled exactly like in regular binary search trees.
- Time complexity: $\Theta(h) = \Theta(\log n)$.

Insertion/Deletion

- Insertion and deletion are done by first applying the insertion/deletion algorithm of binary search trees.
- After the insertion/deletion, the tree may not be balanced, so we need to correct it.
- The time complexity of insertion/deletion in AVL tree is $\Theta(\log n)$.

Insertion

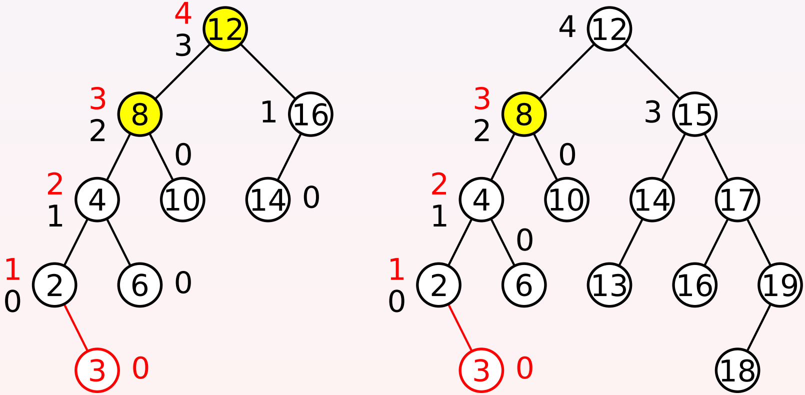
After a new leaf is inserted, the height of some of its ancestors increase by 1. The heights of the other nodes are unchanged.



Insertion

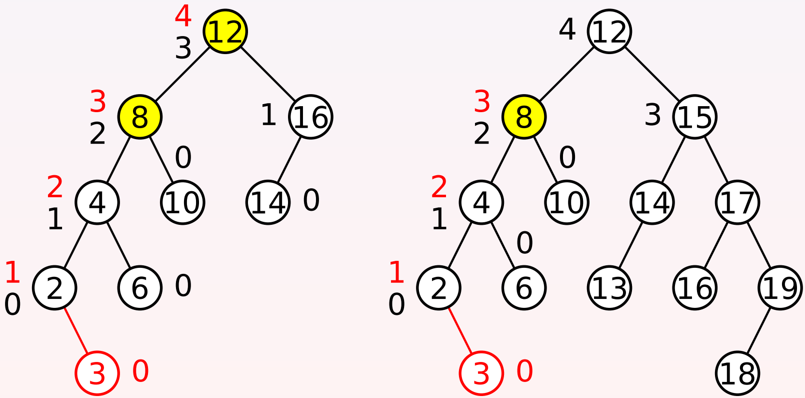
The height fields of the nodes can be updated by going up the tree from the inserted leaf, and for each ancestor v of the leaf perform

$$v.\text{height} = 1 + \max(v.\text{left.height}, v.\text{right.height})$$

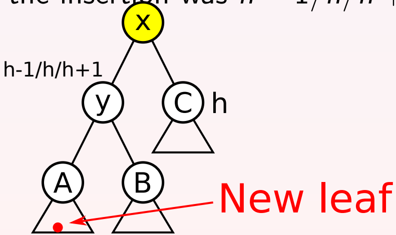


Insertion

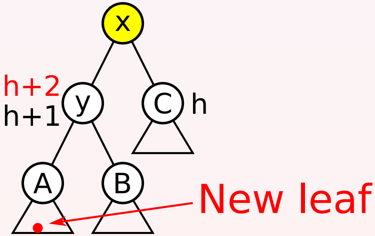
Some of the ancestors of the leaf may become unbalanced.



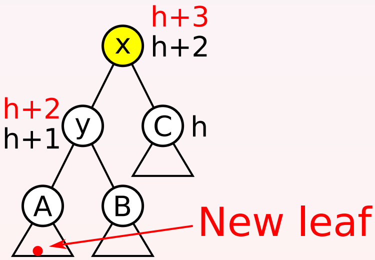
- Let x be the lowest node on the path from the new leaf to the root which is unbalanced (if x doesn't exist we are done).
- There are 4 cases. In Case 1 suppose that the new leaf is in the subtree of $x.\mathrm{left}.\mathrm{left}$.
- Let $y = x.\mathrm{left}$.
- Let h be the height of the right child of x.
- Since x was balanced before the insertion, the height of y before the insertion was $h - 1/h/h + 1$.



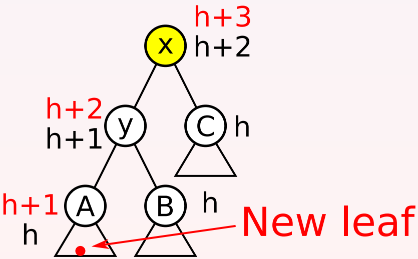
- The insertion either increases the height of a node or doesn't change the height. Since x is now unbalanced, the only possible case is that the height of y is $h+1$ before the insertion, and $h+2$ afterward.
- The height of x before the insertion is $h+2$, and $h+3$ afterward.



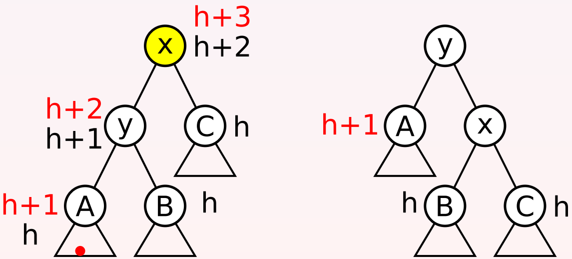
- The insertion either increases the height of a node or doesn't change the height. Since x is now unbalanced, the only possible case is that the height of y is $h+1$ before the insertion, and $h+2$ afterward.
- The height of x before the insertion is $h+2$, and $h+3$ afterward.



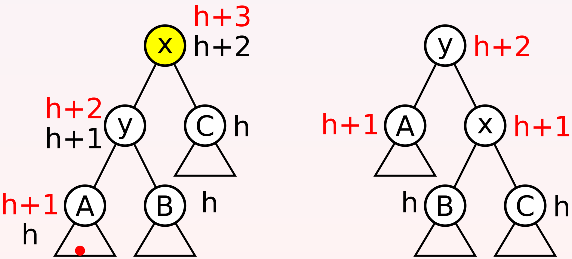
- After the insertion, y has a child with height $h+1$. This child must be the left child A. The height of A before the insertion is h.
- The height of the right child of y is h.



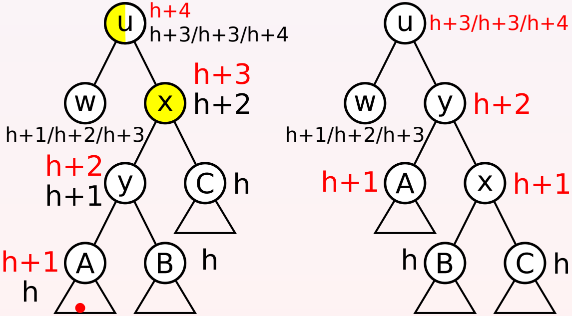
- To fix the imbalance of x, perform the following operation called right rotation.
- The rotation operation doesn't change the inorder of the nodes. Therefore, the new tree is a valid binary search tree.



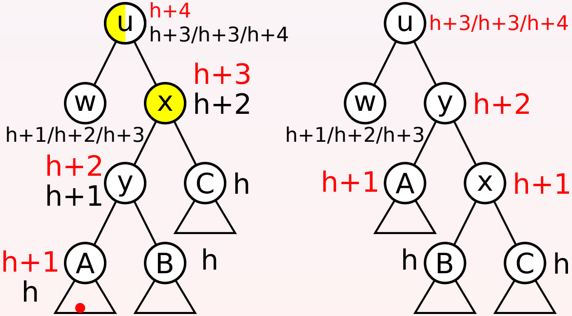
- The heights of x and y after the rotations are $h+1$ and $h+2$.
- After the rotation, x and y are balanced.



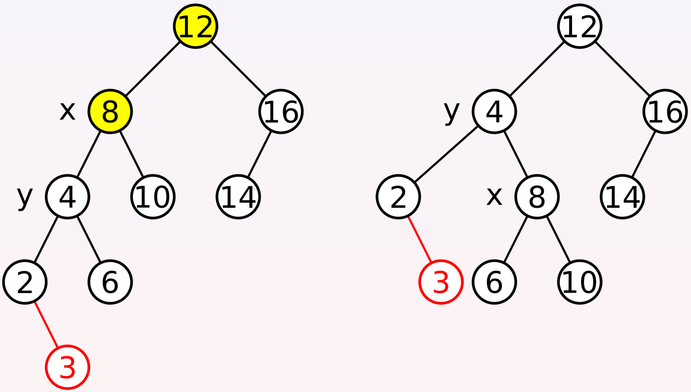
- Assume x had a parent the before the rotation, and denote it by u.
- Let w be the sibling of x before the rotation.
- The height of w is $h+1/h+2/h+3$.
- If the height of w is $h+1$, then u is unbalanced after the insertion (before the rotation).



- After the rotation, the height the sibling of w (node y) is $h+2$, which is equal to the height of the sibling of w before the rotation. Therefore, u is balanced.
- The height of u after the rotation is the same as the height before the insertion. Repeating these arguments, every ancestor of u is balanced and has same height as before the insertion.

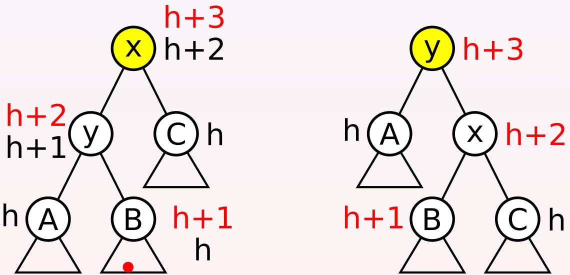


Example

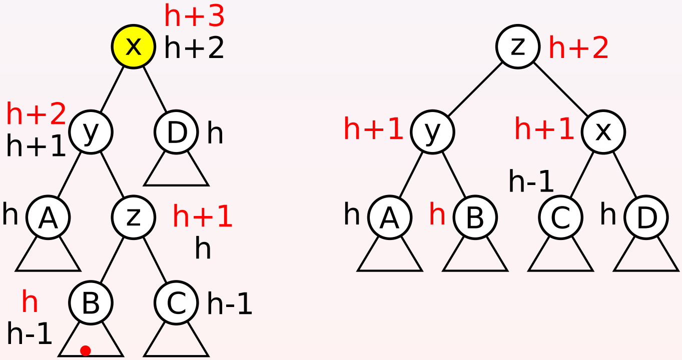


In this example, $h = 0$.

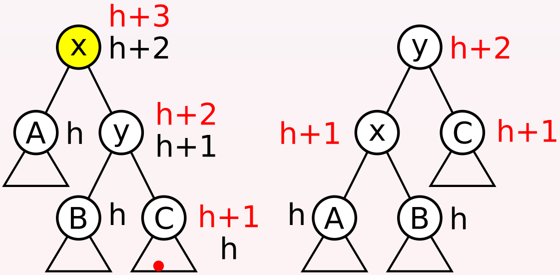
- In Case 2, suppose that the new leaf is in the subtree of $x.\mathrm{left}.\mathrm{right}$.
- Let $y = x.\mathrm{left}$.
- Performing a rotation on x and y does not work.



- Let $z = y.\mathrm{right}$. Perform a double rotation on x, y, z.
- The double rotation doesn't change the inorder of the nodes.
- After the double rotation, x, y, z are balanced. Moreover, the height of z is the same as the height of x before the insertion, and therefore all ancestors of z are balanced.

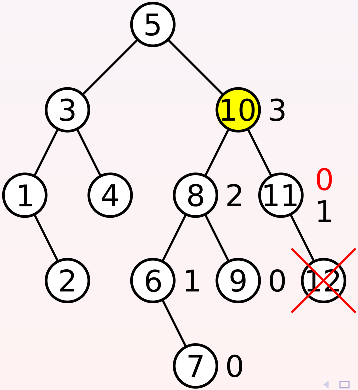


- Case 3 is when the new leaf is in the subtree of x.right.right, and Case 4 is when the new leaf is in the subtree of x.right.left.
- Case 3 and Case 4 are symmetric to Case 1 and Case 2.
- Case 3 is shown below.

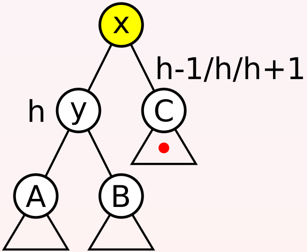


Deletion

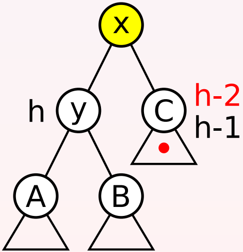
- After a node is deleted, the heights of some of its ancestors decrease by 1. The heights of the other nodes are unchanged.
- A single ancestor of the deleted node can become unbalanced.



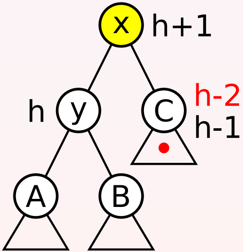
- Let x be the unbalanced node (if x doesn't exist we are done).
- Assume that the deleted node is in the subtree of $x.\mathrm{right}$.
- Let $y = x.\mathrm{left}$.
- Since x was balanced before the deletion, the height of $x.\mathrm{right}$ before the deletion was $h - 1/h/h + 1$.



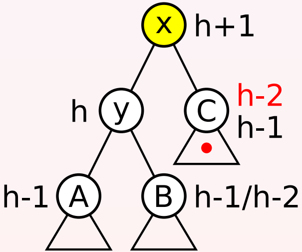
- The deletion either decreases the height of a node or doesn't change the height. Since x is now unbalanced, the only possible case is that the height of $x.\mathrm{right}$ is $h-1$ before the insertion, and $h-2$ afterward.
- The height of x is $h+1$ (the insertion doesn't change the height).



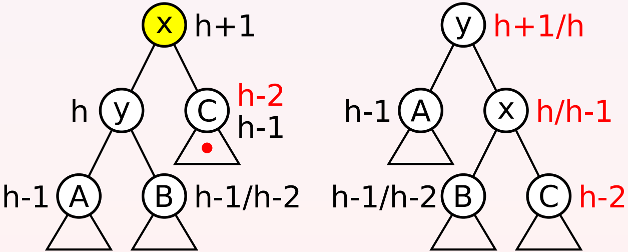
- The deletion either decreases the height of a node or doesn't change the height. Since x is now unbalanced, the only possible case is that the height of $x.\mathrm{right}$ is $h-1$ before the insertion, and $h-2$ afterward.
- The height of x is $h+1$ (the insertion doesn't change the height).



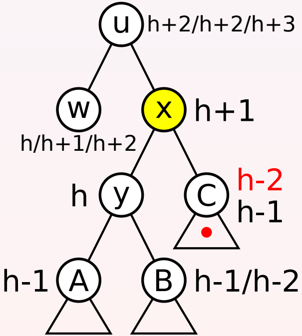
- Since y has height h and it is balanced, one of y's children has height $h-1$ and the other child has height $h-1/h-2$.
- In Case 1, assume that the height of $y.\mathrm{left}$ is $h-1$, and the height of $y.\mathrm{right}$ is $h-1/h-2$.

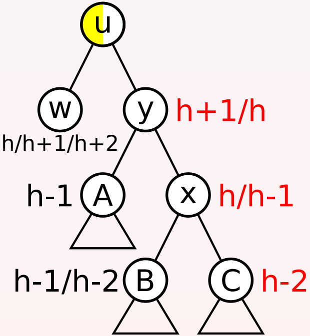


- In Case 1 we perform a right rotation on x and y.
- After the rotation, the height of x is $h/h-1$ and the height of y is $h+1/h$.
- After the rotation, x and y are balanced.

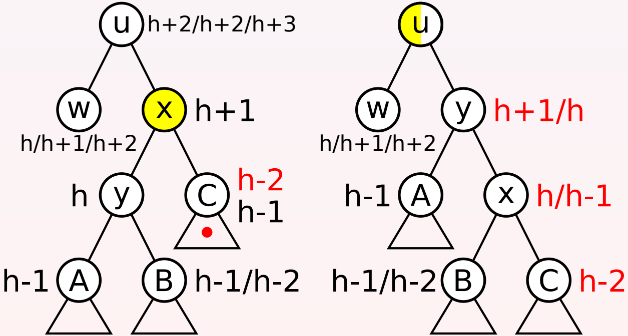


- Assume x had a parent the before the rotation, and denote it by u.
- Let w be the sibling of x before the rotation.
- The height of w is $h/h+1/h+3$.
- u is balanced after the insertion (before the rotation).





- After the rotation, u can become unbalanced. This occurs when the height of w is $h+2$, and the height of y after the rotation is h.
- If the height of w is h, and the height after the rotation is h, then the height of u is $h+2$ and $h+1$ afterwards. This can cause an imbalance in an ancestor of u.
- In the two cases above, additional rotation is needed.



- In Case 2, assume that the height of $y.\text{left}$ is $h-2$, and the height of $y.\text{right}$ is $h-1$.
- Let $z = y.\text{right}$.
- In this case we perform a double rotation of x, y, z.