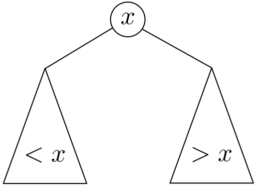
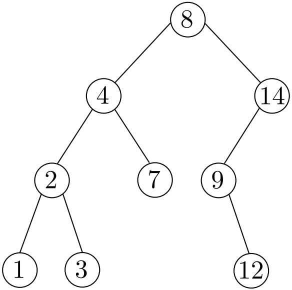
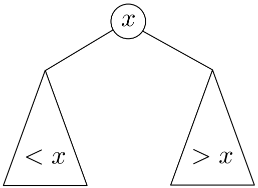
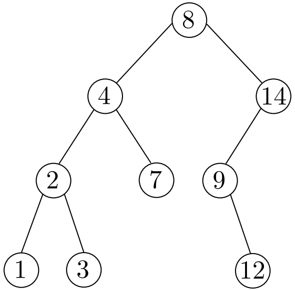
AVL Trees

Version of April 4, 2019


x
$< x$
$> x$


8
4
14
2
7
9
1
3
12



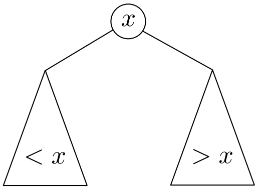


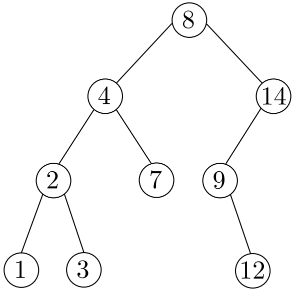
Binary-search-tree property

For every node x

- All keys in its left subtree are smaller than the key value in x

Binary Search Trees





Binary-search-tree property

For every node x

- All keys in its left subtree are smaller than the key value in x
- All keys in its right subtree are larger than the key value in x

Height

The **height** of a node in a tree is the number of edges on the longest downward path from the node to a leaf

Height

The **height** of a node in a tree is the number of edges on the longest downward path from the node to a leaf

- Node height
  $=$ max(children height) $+1$

Height

The **height** of a node in a tree is the number of edges on the longest downward path from the node to a leaf

- Node height
  $= \max(\text{children height}) + 1$
- Leaves: height $= 0$

Height

The **height** of a node in a tree is the number of edges on the longest downward path from the node to a leaf

- Node height
  $=$ max(children height) $+1$
- Leaves: height $= 0$
- Tree height $=$ root height

Height

The **height** of a node in a tree is the number of edges on the longest downward path from the node to a leaf

- Node height
  $= \max(\text{children height}) + 1$
- Leaves: height $= 0$
- Tree height $=$ root height
- Empty tree: height $= -1$

Height

The **height** of a node in a tree is the number of edges on the longest downward path from the node to a leaf

- Node height $= \max(\text{children height}) + 1$
- Leaves: height $= 0$
- Tree height $=$ root height
- Empty tree: height $= -1$

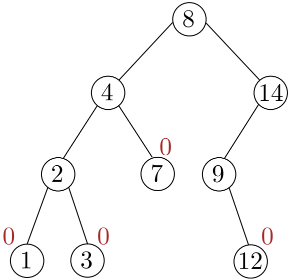
- Tree operations typically take $O(\text{height})$ time

Height

The **height** of a node in a tree is the number of edges on the longest downward path from the node to a leaf

- Node height
  $=$ max(children height) $+1$
- Leaves: height $= 0$
- Tree height $=$ root height
- Empty tree: height $= -1$

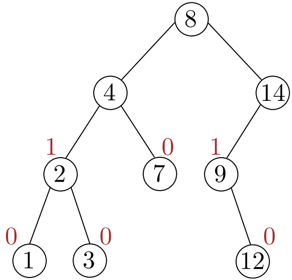
- Tree operations typically take $O(\text{height})$ time



Height

The **height** of a node in a tree is the number of edges on the longest downward path from the node to a leaf

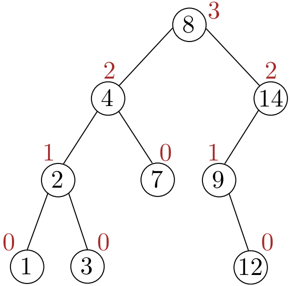
- Node height $= \max(\text{children height}) + 1$
- Leaves: height $= 0$
- Tree height $=$ root height
- Empty tree: height $= -1$
- Tree operations typically take $O(\text{height})$ time



Height

The **height** of a node in a tree is the number of edges on the longest downward path from the node to a leaf

- Node height $=$ max(children height) $+1$
- Leaves: height $= 0$
- Tree height $=$ root height
- Empty tree: height $= -1$
- Tree operations typically take $O(\text{height})$ time

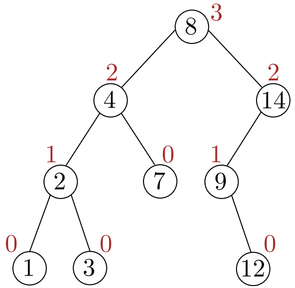


Height

The **height** of a node in a tree is the number of edges on the longest downward path from the node to a leaf

- Node height
  $=$ max(children height) $+1$
- Leaves: height $= 0$
- Tree height $=$ root height
- Empty tree: height $= -1$

- Tree operations typically take $O(\text{height})$ time



Question

Let n be the size of a binary search tree. How can we keep its height $O(\log n)$ under insertion and deletion?

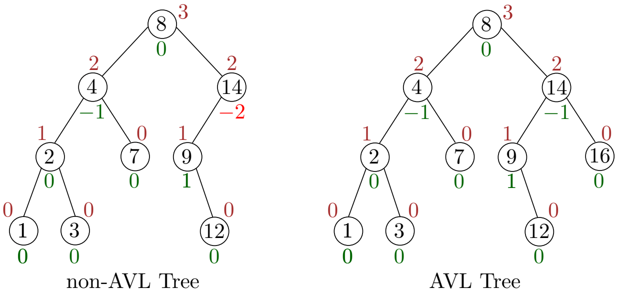
Balanced Binary Search Tree: AVL Tree

An **AVL** [Adelson-Velskii & Landis, 1962] tree is a binary search tree in which

- for every node in the tree, heights of its left and right subtrees differ by at most 1.

An **AVL** [Adelson-Velskii & Landis, 1962] tree is a binary search tree in which

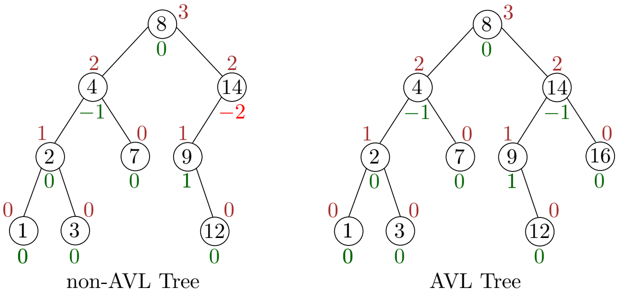
- for every node in the tree, heights of its left and right subtrees differ by at most 1.



- The *balance factor* of a node is the height of its right subtree minus the height of its left subtree.

An **AVL** [Adelson-Velskii & Landis, 1962] tree is a binary search tree in which

- for every node in the tree, heights of its left and right subtrees differ by at most 1.



- The *balance factor* of a node is the height of its right subtree minus the height of its left subtree.
- A node with balance factor 1, 0 or -1 is considered *balanced*.

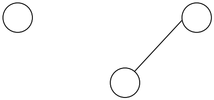
AVL Trees with Minimum Number of Nodes

Let n_h be the minimum number of nodes in an AVL tree of height h

$n_0 = 1$

AVL Trees with Minimum Number of Nodes

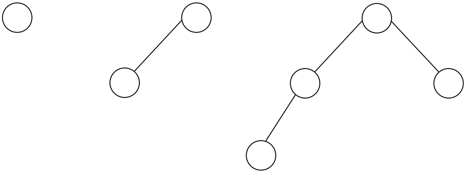
Let n_h be the minimum number of nodes in an AVL tree of height h



$n_0 = 1$ $n_1 = 2$

AVL Trees with Minimum Number of Nodes

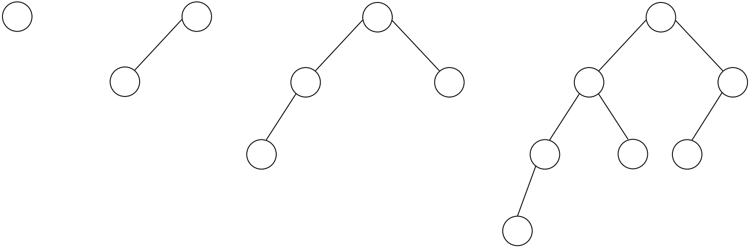
Let n_h be the minimum number of nodes in an AVL tree of height h



$n_0 = 1$ $\quad n_1 = 2$ $\quad n_2 = n_1 + n_0 + 1 = 4$

AVL Trees with Minimum Number of Nodes

Let n_h be the minimum number of nodes in an AVL tree of height h



$n_0 = 1$ $n_1 = 2$ $n_2 = n_1 + n_0 + 1 = 4$ $n_3 = n_2 + n_1 + 1 = 7$

Height of AVL Tree

- Let n_h denote the minimum number of nodes in an AVL tree of height h
- $n_0 = 1, n_1 = 2$ (base)
- $n_h = n_{h-1} + n_{h-2} + 1$

Height of AVL Tree

- Let n_h denote the minimum number of nodes in an AVL tree of height h
- $n_0 = 1, n_1 = 2$ (base)
- $n_h = n_{h-1} + n_{h-2} + 1$
- For any AVL tree of height h and size n,

$$n \geq n_h = n_{h-1} + n_{h-2} + 1 > 2n_{h-2} > 4n_{h-4} > \cdots > 2^i n_{h-2i}$$

Height of AVL Tree

- Let n_h denote the minimum number of nodes in an AVL tree of height h
- $n_0 = 1, n_1 = 2$ (base)
- $n_h = n_{h-1} + n_{h-2} + 1$
- For any AVL tree of height h and size n,

$$n \geq n_h = n_{h-1} + n_{h-2} + 1 > 2n_{h-2} > 4n_{h-4} > \cdots > 2^i n_{h-2i}$$

- If h is even, let $i = h/2$. $\Rightarrow n > 2^{h/2} n_0 = 2^{h/2} \cdot 1 \Rightarrow$ $h = O(\log n)$

Height of AVL Tree

- Let n_h denote the minimum number of nodes in an AVL tree of height h
- $n_0 = 1, n_1 = 2$ (base)
- $n_h = n_{h-1} + n_{h-2} + 1$
- For any AVL tree of height h and size n,

$$n \geq n_h = n_{h-1} + n_{h-2} + 1 > 2n_{h-2} > 4n_{h-4} > \cdots > 2^i n_{h-2i}$$

- If h is even, let $i = h/2$. $\Rightarrow n > 2^{h/2} n_0 = 2^{h/2} \cdot 1 \Rightarrow h = O(\log n)$
- If h is odd, let $i = (h-1)/2$. $\Rightarrow n > 2^{(h-1)/2} n_1 = 2^{(h-1)/2} \cdot 2 \Rightarrow h = O(\log n)$

Height of AVL Tree

- Let n_h denote the minimum number of nodes in an AVL tree of height h
- $n_0 = 1, n_1 = 2$ (base)
- $n_h = n_{h-1} + n_{h-2} + 1$
- For any AVL tree of height h and size n,

$$n \geq n_h = n_{h-1} + n_{h-2} + 1 > 2n_{h-2} > 4n_{h-4} > \cdots > 2^i n_{h-2i}$$

- If h is even, let $i = h/2$. $\Rightarrow n > 2^{h/2} n_0 = 2^{h/2} \cdot 1 \Rightarrow h = O(\log n)$
- If h is odd, let $i = (h-1)/2$. $\Rightarrow n > 2^{(h-1)/2} n_1 = 2^{(h-1)/2} \cdot 2 \Rightarrow h = O(\log n)$
- Thus, many operations (e.g., insertion, deletion, and search) on an AVL tree will take $O(logn)$ time

AVL Trees and Fibonacci Numbers

We saw that $n_h = n_{h-1} + n_{h-2} + 1$.
Recall Fibonacci numbers satisfy $f_h = f_{h-1} + f_{h-2}$. Now compare

AVL Trees and Fibonacci Numbers

We saw that $n_h = n_{h-1} + n_{h-2} + 1$.
Recall Fibonacci numbers satisfy $f_h = f_{h-1} + f_{h-2}$. Now compare

h	0	1	2	3	4	5	6	7	8
n_h	1	2	4	7	12	20	33	54	88
f_h	1	1	2	3	5	8	13	21	34

AVL Trees and Fibonacci Numbers

We saw that $n_h = n_{h-1} + n_{h-2} + 1$.
Recall Fibonacci numbers satisfy $f_h = f_{h-1} + f_{h-2}$. Now compare

h	0	1	2	3	4	5	6	7	8
n_h	1	2	4	7	12	20	33	54	88
f_h	1	1	2	3	5	8	13	21	34

Lemma:

$n_h = f_{h+2} - 1$

AVL Trees and Fibonacci Numbers

We saw that $n_h = n_{h-1} + n_{h-2} + 1$.
Recall Fibonacci numbers satisfy $f_h = f_{h-1} + f_{h-2}$. Now compare

h	0	1	2	3	4	5	6	7	8
n_h	1	2	4	7	12	20	33	54	88
f_h	1	1	2	3	5	8	13	21	34

Lemma:

$n_h = f_{h+2} - 1$

Proof: by induction

$$n_{h+1} \quad = \quad 1+n_h+n_{h-1} \quad = \quad 1+f_{h+2}-1+f_{h+1}-1 \quad = \quad f_{h+3}-1$$

Since $f_h \sim c\phi^h$ for golden ratio $\phi = \frac{1+\sqrt{5}}{2}$, this also immediately provides alternative derivation that $h = O(\log n)$.

Insertion

- Basically follows insertion strategy of binary search tree

Insertion

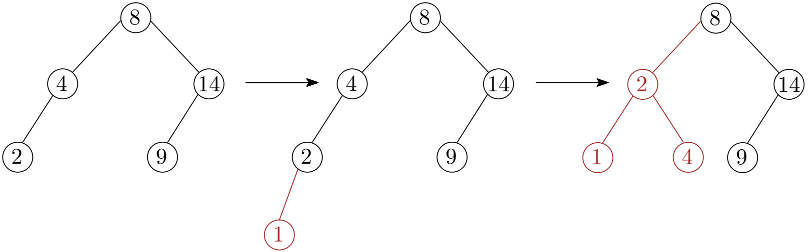
- Basically follows insertion strategy of binary search tree
  - But may cause violation of AVL tree balance property

Insertion

- Basically follows insertion strategy of binary search tree
    - But may cause violation of AVL tree balance property
- Rebalance tree nodes if needed

Insertion

- Basically follows insertion strategy of binary search tree
  - But may cause violation of AVL tree balance property
- Rebalance tree nodes if needed



Insert 1

Restore height balance

Insertion: Observation

After an insertion, only nodes that are on the path from the insertion node to the root might have their balance altered

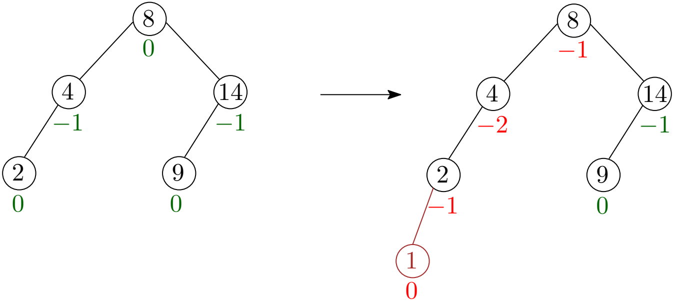
Insertion: Observation

After an insertion, only nodes that are on the path from the insertion node to the root might have their balance altered

- Because only those nodes have their subtrees altered

After an insertion, only nodes that are on the path from the insertion node to the root might have their balance altered

- Because only those nodes have their subtrees altered



Insert 1

Insertion: Four Cases

- Let A denote the *lowest node* violating AVL tree property

Insertion: Four Cases

- Let A denote the *lowest node* violating AVL tree property
  - Case 1 (Left-Left case)
    — insert into the left subtree of the left child of A

Insertion: Four Cases

- Let A denote the *lowest node* violating AVL tree property
  - Case 1 (Left-Left case)
    — insert into the left subtree of the left child of A
  - Case 2 (Left-Right case)
    — insert into the right subtree of the left child of A

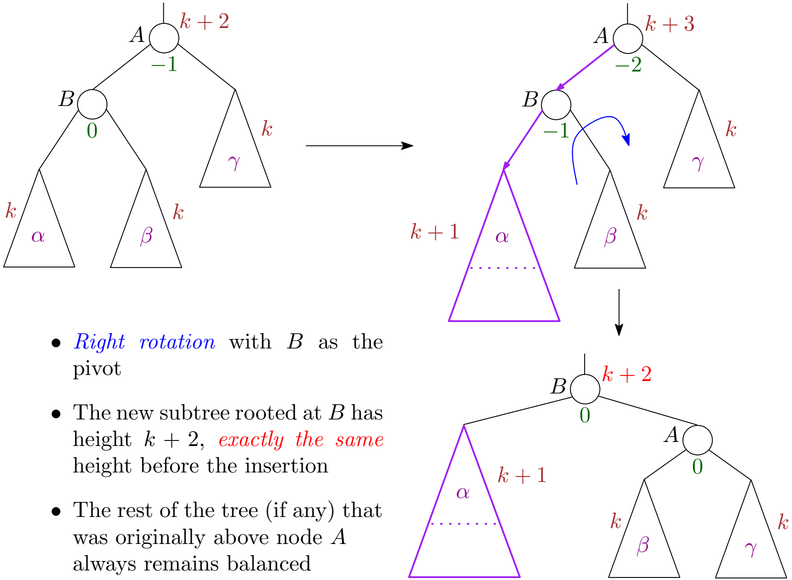
- Let A denote the *lowest node* violating AVL tree property
  - Case 1 (Left-Left case)
    — insert into the left subtree of the left child of A
  - Case 2 (Left-Right case)
    — insert into the right subtree of the left child of A
  - Case 3 (Right-Left case)
    — insert into the left subtree of the right child of A

Insertion: Four Cases

- Let A denote the *lowest node* violating AVL tree property
  - Case 1 (Left-Left case)
    — insert into the left subtree of the left child of A
  - Case 2 (Left-Right case)
    — insert into the right subtree of the left child of A
  - Case 3 (Right-Left case)
    — insert into the left subtree of the right child of A
  - Case 4 (Right-Right case)
    — insert into the right subtree of the right child of A

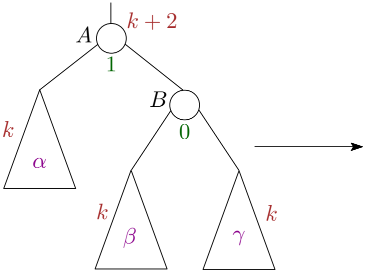
Insertion: Four Cases

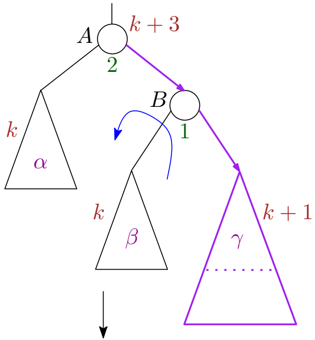
- Let A denote the *lowest node* violating AVL tree property
  - Case 1 (Left-Left case)
    — insert into the left subtree of the left child of A
  - Case 2 (Left-Right case)
    — insert into the right subtree of the left child of A
  - Case 3 (Right-Left case)
    — insert into the left subtree of the right child of A
  - Case 4 (Right-Right case)
    — insert into the right subtree of the right child of A
- Cases 1 and 4 are mirror image symmetries with respect to A, as are cases 2 and 3



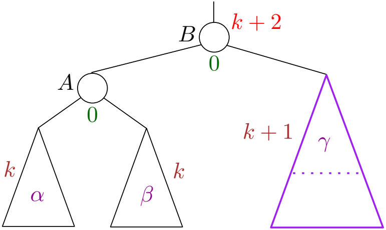
- *Right rotation* with B as the pivot
- The new subtree rooted at B has height $k+2$, *exactly the same* height before the insertion
- The rest of the tree (if any) that was originally above node A always remains balanced

Insertion: Right-Right Case

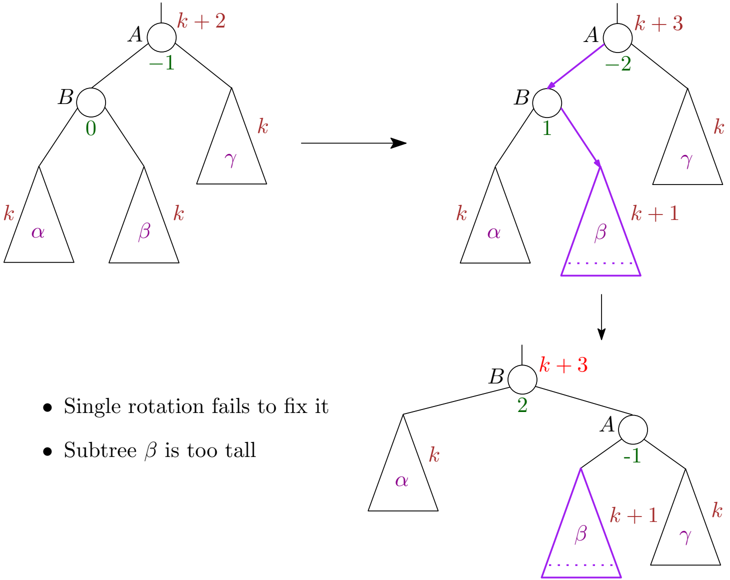




- *Left rotation* with B as the pivot
- The new subtree rooted at B has height $k+2$, *exactly the same* height before the insertion
- The rest of the tree (if any) that was originally above node A always remains balanced

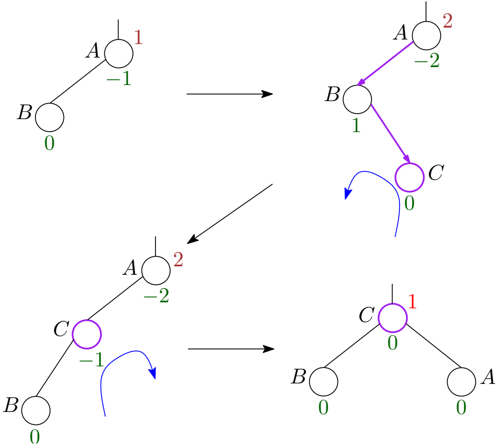


Insertion: Left-Right Case



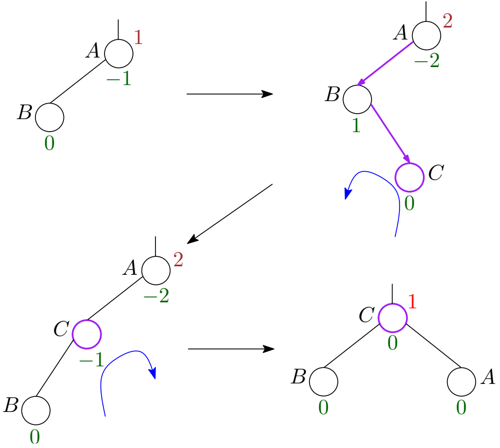
- Single rotation fails to fix it
- Subtree β is too tall

When subtree α, β and γ are empty, $k = -1$. Insert C:



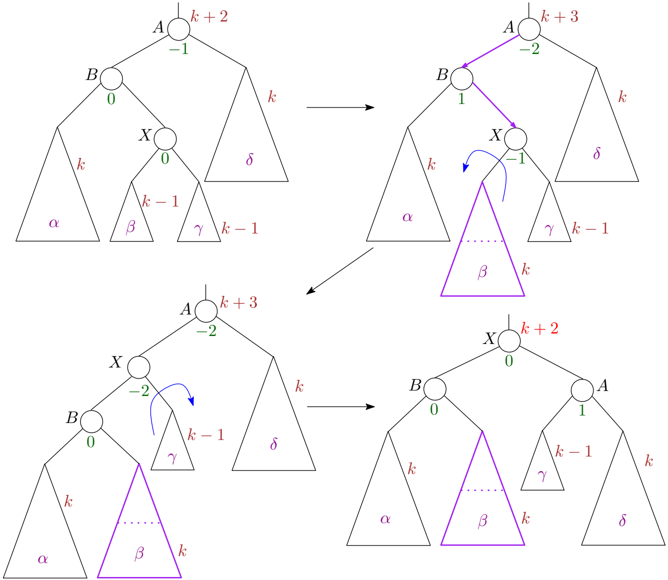
Left-Right Case: Special Case

When subtree α, β and γ are empty, $k = -1$. Insert C:

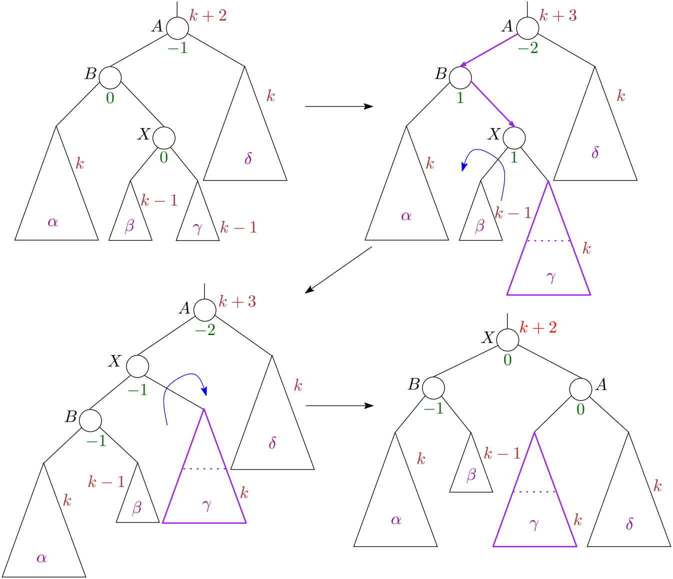


- Left rotation and then right rotation with C as the pivot. Done!

Left-Right Case: General Case 1

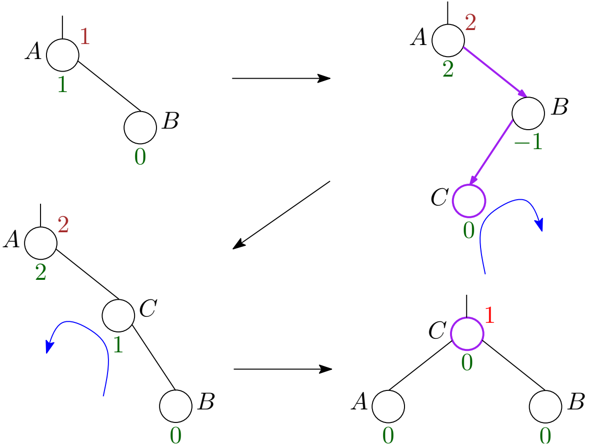


Left-Right Case: General Case 2



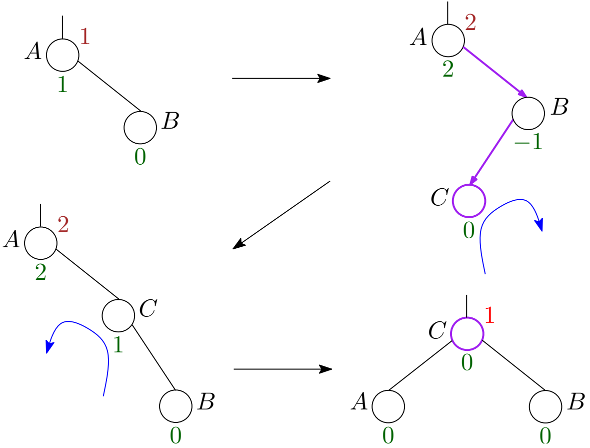
Right-Left Case: Special Case

When subtree α, β and γ are empty, $k = -1$. Insert C:



Right-Left Case: Special Case

When subtree α, β and γ are empty, $k = -1$. Insert C:



- Right rotation and then left rotation with C as the pivot. Done!

Insertion: Summary

- Left-Left case: single rotation
  — Right rotation

Insertion: Summary

- Left-Left case: single rotation
  — Right rotation
- Left-Right case: double rotation
  — Left rotation and then right rotation

Insertion: Summary

- Left-Left case: single rotation
  — Right rotation
- Left-Right case: double rotation
  — Left rotation and then right rotation
- Right-Left case: double rotation
  — Right rotation and then left rotation

Insertion: Summary

- Left-Left case: single rotation
  — Right rotation
- Left-Right case: double rotation
  — Left rotation and then right rotation
- Right-Left case: double rotation
  — Right rotation and then left rotation
- Right-Right case: single rotation
  — Left rotation

Insertion: Summary

- Left-Left case: single rotation
  — Right rotation
- Left-Right case: double rotation
  — Left rotation and then right rotation
- Right-Left case: double rotation
  — Right rotation and then left rotation
- Right-Right case: single rotation
  — Left rotation

For each insertion, at most two rotations are needed to restore the height balance of the entire tree.

Insertion: Summary

- Left-Left case: single rotation
  — Right rotation
- Left-Right case: double rotation
  — Left rotation and then right rotation
- Right-Left case: double rotation
  — Right rotation and then left rotation
- Right-Right case: single rotation
  — Left rotation

For each insertion, at most two rotations are needed to restore the height balance of the entire tree.

Note that in all cases, height of rebalanced subtree is unchanged! This means no further tree modifications are needed.

Deletion

Delete a node as in ordinary binary search tree

Deletion

Delete a node as in ordinary binary search tree

- If the node is a leaf, remove it

Delete a node as in ordinary binary search tree

- If the node is a leaf, remove it
- If not, replace it with either the largest in its left subtree or the smallest in its right subtree, and remove that node

Deletion

Delete a node as in ordinary binary search tree

- If the node is a leaf, remove it
- If not, replace it with either the largest in its left subtree or the smallest in its right subtree, and remove that node
  Note: The replacement node has at most one subtree

Deletion

Delete a node as in ordinary binary search tree

- If the node is a leaf, remove it
- If not, replace it with either the largest in its left subtree or the smallest in its right subtree, and remove that node
  Note: The replacement node has at most one subtree
- Trace the path from the parent of removed node to root

Deletion

Delete a node as in ordinary binary search tree

- If the node is a leaf, remove it
- If not, replace it with either the largest in its left subtree or the smallest in its right subtree, and remove that node
  Note: The replacement node has at most one subtree
- Trace the path from the parent of removed node to root
- For each node along path, restore the height balance if needed by doing single or double rotations

Deletion

Delete a node as in ordinary binary search tree

- If the node is a leaf, remove it
- If not, replace it with either the largest in its left subtree or the smallest in its right subtree, and remove that node
  Note: The replacement node has at most one subtree
- Trace the path from the parent of removed node to root
- For each node along path, restore the height balance if needed by doing single or double rotations

Deletion (continued)

For each node along the path, restore the height balance if needed by doing single or double rotations

Deletion (continued)

For each node along the path, restore the height balance if needed by doing single or double rotations

- Very similar to insertion with one major caveat

Deletion (continued)

For each node along the path, restore the height balance if needed by doing single or double rotations

- Very similar to insertion with one major caveat
  - In insertion a (single/double) rotation restored balance *and* kept height of rebalanced subtree unchanged.
    $\Rightarrow$ Only *one* rotation needed.
  - In deletion, rotation restores *balance* but final *height* of rotated subtree might decrease by one. If this occurs, need to continue walking up path towards root, continuing to restoring balance by rotations when necessary.

Deletion (continued)

For each node along the path, restore the height balance if needed by doing single or double rotations

- Very similar to insertion with one major caveat
  - In insertion a (single/double) rotation restored balance *and* kept height of rebalanced subtree unchanged.
    $\Rightarrow$ Only *one* rotation needed.
  - In deletion, rotation restores *balance* but final *height* of rotated subtree might decrease by one. If this occurs, need to continue walking up path towards root, continuing to restoring balance by rotations when necessary.
  - Since path has length $O(h)$ this might require $O(h) = O(\log n)$ rotations.

Deletion (continued)

For each node along the path, restore the height balance if needed by doing single or double rotations

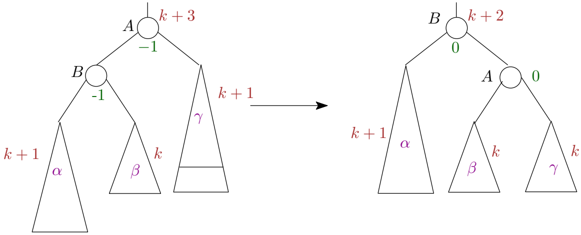
- Very similar to insertion with one major caveat
  - In insertion a (single/double) rotation restored balance *and* kept height of rebalanced subtree unchanged.
    $\Rightarrow$ Only *one* rotation needed.
  - In deletion, rotation restores *balance* but final *height* of rotated subtree might decrease by one. If this occurs, need to continue walking up path towards root, continuing to restoring balance by rotations when necessary.
  - Since path has length $O(h)$ this might require $O(h) = O(\log n)$ rotations.

$\Rightarrow$ Deletion can also be done in $O(\log n)$ time.

Deletion Example

Diagram below illustrates example in which subtree rooted at A has height $k+3$. An item is deleted from subtree γ, reducing its height from $k+1$ to k, leading to an imbalance.

After a single rotation, the subtree is now rooted at B with no imbalance. But, B has height $k+2$. This might cause an imbalance further up the tree, so the algorithm might need to continue walking upwards, correcting that imbalance.



Going Further I

AVL trees can be used to implement other operations in addition to `Insert` and `Delete`. Two particularly useful ones are

- `Split(T, x)` : Splits AVL tree T into two AVL trees, L and R. L contains all nodes in T with key values $\leq x$ and R contains all nodes with key values $> x$.
- `Join(L, R)` : Takes as input two AVL trees L, R such that all key values in L are less than all key values in R. Combines them into one output AVL tree containing all of the nodes.

Both of these operations can be implemented in $O(\log n)$ time where, in `Split(T, x)`, n is the number of nodes in T and, in `Join(L, R)`, n is the total number of nodes in L and R together.

Going Further II

AVL trees are one particular type of *Balanced* Search trees, yielding $O(\log n)$ behavior for dictionary operations.

There are many other types of Balanced Search Trees, e.g.

- red-black trees
- B-trees
- (a, b) trees
  $(2, 3)$ and $(2, 3, 4)$ trees are special cases
- treaps (randomized BSTs)
- splay trees (only $O(\log n)$ in amortized sense)
- skip lists (randomized; not trees, but behave similarly)