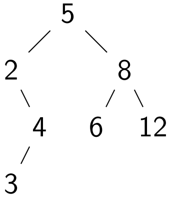
Balancing binary trees

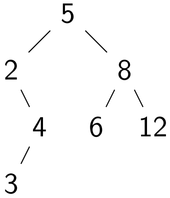
Binary search trees – refresher

- Each node stores a value.
- Values in left subtree are smaller; right, bigger.



Binary search trees – refresher

- Each node stores a value.
- Values in left subtree are smaller; right, bigger.



- Insertion, deletion, membership queries all take time $O(\mathrm{height})$.
- In a typical binary search tree, $\mathrm{height} \approx \log n$.
- In the worse case, $\mathrm{height} = n - 1$.

Balanced binary trees

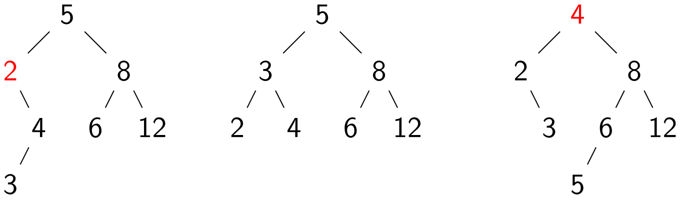
A binary search tree is **balanced** if:

- number of nodes in the left and right subtrees of every node differ by at most 1.

Balanced binary trees

A binary search tree is **balanced** if:

- number of nodes in the left and right subtrees of every node differ by at most 1.

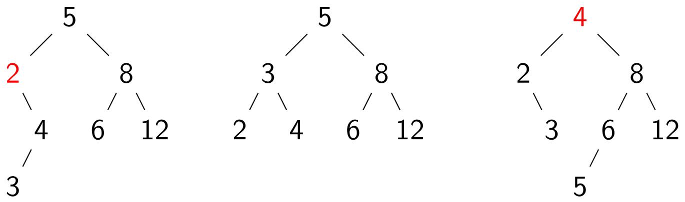


unbalanced   balanced   unbalanced

Balanced binary trees

A binary search tree is **balanced** if:

- number of nodes in the left and right subtrees of every node differ by at most 1.

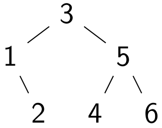


unbalanced | balanced | unbalanced

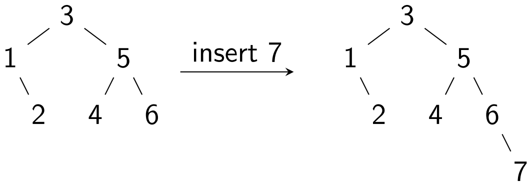
A balanced binary tree with n nodes has height exactly $\lfloor \log_2 n \rfloor$.

The cost of keeping BSTs balanced

Inserting into a balanced BST may unbalance it.

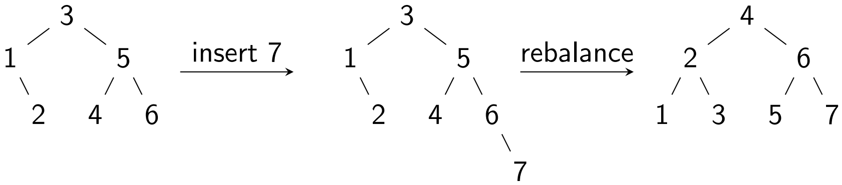


Inserting into a balanced BST may unbalance it.



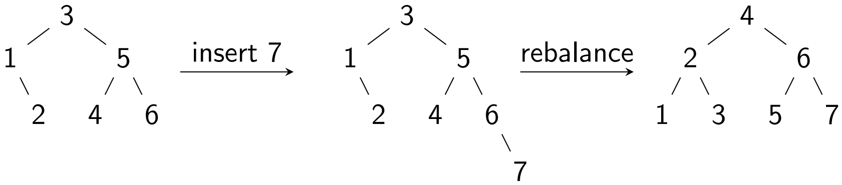
The cost of keeping BSTs balanced

Inserting into a balanced BST may unbalance it.



The cost of keeping BSTs balanced

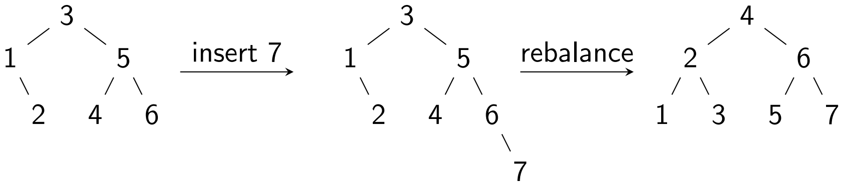
Inserting into a balanced BST may unbalance it.



- This is the only balanced BST storing 1–7.

The cost of keeping BSTs balanced

Inserting into a balanced BST may unbalance it.



- This is the only balanced BST storing 1–7.
- Every vertex has moved and every vertex except 7 now has a different parent!
- This shows rebalancing takes time $\Theta(n)$ – too expensive.

AVL-balance

A binary search tree is **AVL-balanced** if:

- the heights of every node's left and right subtrees differ by at most 1.

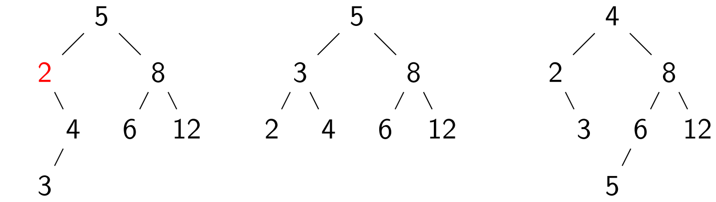
Convention: the height of a subtree with no vertices is -1.

AVL-balance

A binary search tree is **AVL-balanced** if:

- the heights of every node's left and right subtrees differ by at most 1.

Convention: the height of a subtree with no vertices is -1.



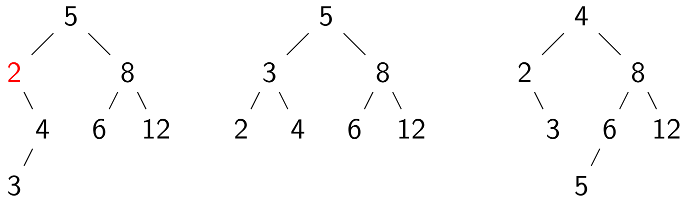
(unbalanced) AVL-unbalanced | (balanced) AVL-balanced | (unbalanced) AVL-balanced

AVL-balance

A binary search tree is **AVL-balanced** if:

- the heights of every node's left and right subtrees differ by at most 1.

Convention: the height of a subtree with no vertices is -1.



(unbalanced) AVL-unbalanced | (balanced) AVL-balanced | (unbalanced) AVL-balanced

AVL = Adelson-Velski and Landis, the two inventors.

Balance factors

To track whether a BST is AVL-balanced, define each node's **balance factor** as

$$\mathrm{BF}(x) = \mathrm{height}(x.\mathrm{right}) - \mathrm{height}(x.\mathrm{left})\,.$$

- For AVL-balanced trees, $\mathrm{BF}(x) \in \{-1, 0, +1\}$ for every node x.

Balance factors

To track whether a BST is AVL-balanced, define each node's **balance factor** as

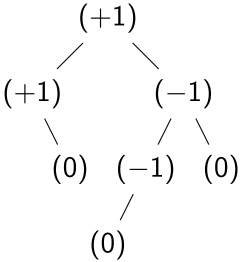
$$\mathrm{BF}(x) = \mathrm{height}(x.\mathrm{right}) - \mathrm{height}(x.\mathrm{left})\,.$$

- For AVL-balanced trees, $\mathrm{BF}(x) \in \{-1, 0, +1\}$ for every node x.
- If $\mathrm{BF}(x) < 0$, we say x is **left-heavy** – its left subtree is taller.
- If $\mathrm{BF}(x) > 0$, we say x is **right-heavy** – its right subtree is taller.

Updating balance factors

Balance factors can be stored in tree nodes and updated after inserting.

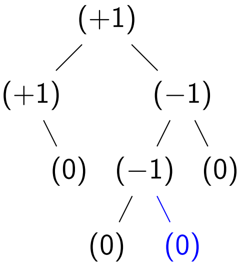
In this diagram, the numbers are balance factors, not node values.



Updating balance factors

Balance factors can be stored in tree nodes and updated after inserting.

In this diagram, the numbers are balance factors, not node values.



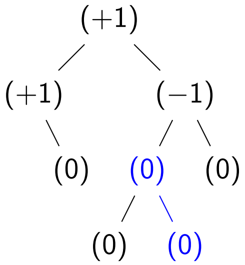
Updates only needed on the path up to the root.

Updates stop when we set a balance factor to zero.

Updating balance factors

Balance factors can be stored in tree nodes and updated after inserting.

In this diagram, the numbers are balance factors, not node values.



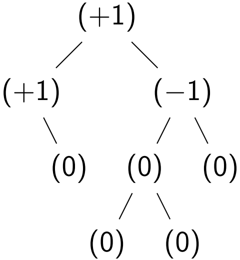
Updates only needed on the path up to the root.

Updates stop when we set a balance factor to zero.

Updating balance factors

Balance factors can be stored in tree nodes and updated after inserting.

In this diagram, the numbers are balance factors, not node values.



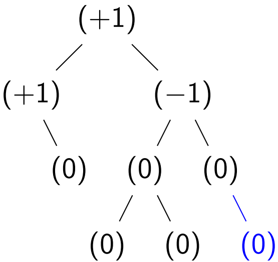
Updates only needed on the path up to the root.

Updates stop when we set a balance factor to zero.

Updating balance factors

Balance factors can be stored in tree nodes and updated after inserting.

In this diagram, the numbers are balance factors, not node values.



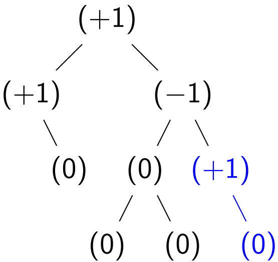
Updates only needed on the path up to the root.

Updates stop when we set a balance factor to zero.

Updating balance factors

Balance factors can be stored in tree nodes and updated after inserting.

In this diagram, the numbers are balance factors, not node values.



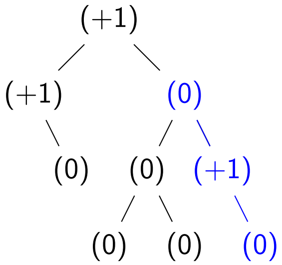
Updates only needed on the path up to the root.

Updates stop when we set a balance factor to zero.

Updating balance factors

Balance factors can be stored in tree nodes and updated after inserting.

In this diagram, the numbers are balance factors, not node values.



Updates only needed on the path up to the root.

Updates stop when we set a balance factor to zero.

When the tree stops being AVL-balanced

The insertions we just saw kept every node's balance factor in $\{-1, 0, +1\}$.

How can an insertion produce a balance factor outside $\{-1, 0, +1\}$?

When the tree stops being AVL-balanced

The insertions we just saw kept every node's balance factor in $\{-1, 0, +1\}$.

How can an insertion produce a balance factor outside $\{-1, 0, +1\}$?

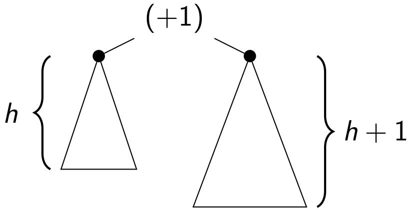
- Either the left subtree of a left-heavy node became taller;
- or the right subtree of a right-heavy node became taller.

When the tree stops being AVL-balanced

The insertions we just saw kept every node's balance factor in $\{-1, 0, +1\}$.

How can an insertion produce a balance factor outside $\{-1, 0, +1\}$?

- Either the left subtree of a left-heavy node became taller;
- or the right subtree of a right-heavy node became taller.

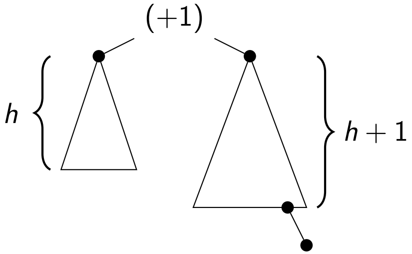


When the tree stops being AVL-balanced

The insertions we just saw kept every node's balance factor in $\{-1, 0, +1\}$.

How can an insertion produce a balance factor outside $\{-1, 0, +1\}$?

- Either the left subtree of a left-heavy node became taller;
- or the right subtree of a right-heavy node became taller.

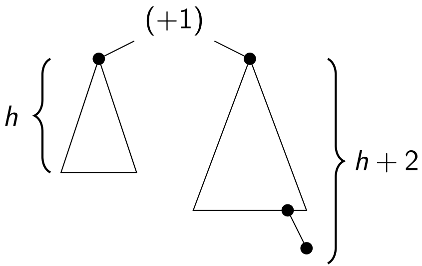


When the tree stops being AVL-balanced

The insertions we just saw kept every node's balance factor in $\{-1, 0, +1\}$.

How can an insertion produce a balance factor outside $\{-1, 0, +1\}$?

- Either the left subtree of a left-heavy node became taller;
- or the right subtree of a right-heavy node became taller.

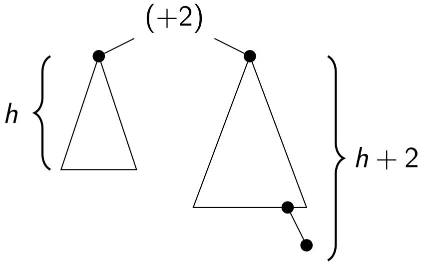


When the tree stops being AVL-balanced

The insertions we just saw kept every node's balance factor in $\{-1, 0, +1\}$.

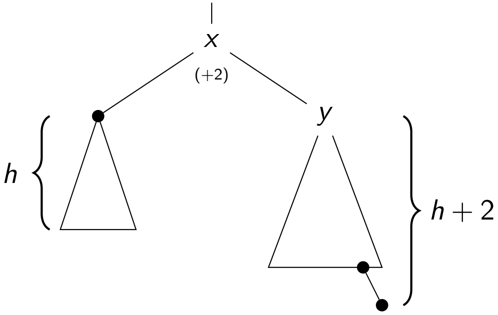
How can an insertion produce a balance factor outside $\{-1, 0, +1\}$?

- Either the left subtree of a left-heavy node became taller;
- or the right subtree of a right-heavy node became taller.



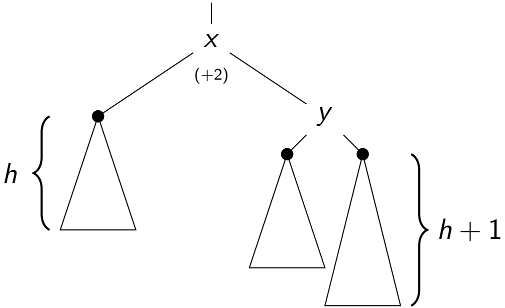
Rebalancing unbalanced trees (1)

Suppose we're updating balance factors and x is the first node where we set $\mathrm{BF} = +2$.



Rebalancing unbalanced trees (1)

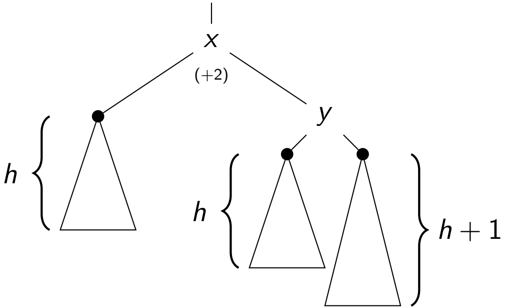
Suppose we're updating balance factors and x is the first node where we set $\mathrm{BF} = +2$.



Look more closely at y's subtrees.
We inserted into the right subtree: it must now have height $h + 1$.

Rebalancing unbalanced trees (1)

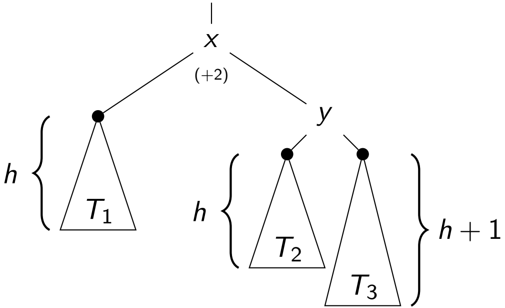
Suppose we're updating balance factors and x is the first node where we set $\mathrm{BF} = +2$.



y's left subtree must have height h. If not, we would have set $\mathrm{BF}(y) = 0$ and stopped updating.

Rebalancing unbalanced trees (1)

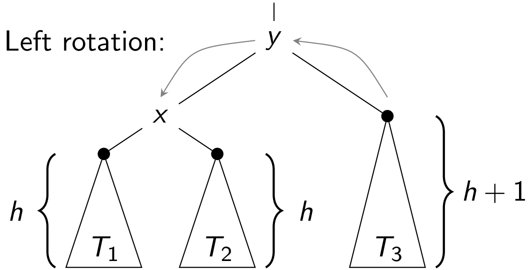
Suppose we're updating balance factors and x is the first node where we set $\mathrm{BF} = +2$.



Nodes in T_1 have value v with $v < x$
Nodes in T_2 have value v with $x < v < y$.
Nodes in T_3 have value v with $v > y$

Rebalancing unbalanced trees (1)

Suppose we're updating balance factors and x is the first node where we set $\mathrm{BF} = +2$.

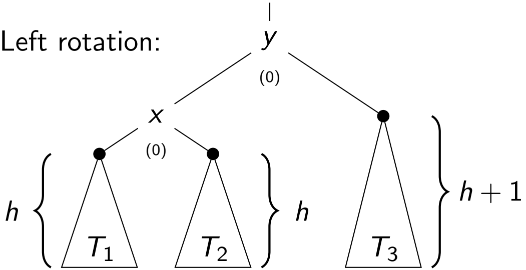


Nodes in T_1 have value v with $v < x$
Nodes in T_2 have value v with $x < v < y$.
Nodes in T_3 have value v with $v > y$

Rebalancing unbalanced trees (1)

Suppose we're updating balance factors and x is the first node where we set $\mathrm{BF} = +2$.

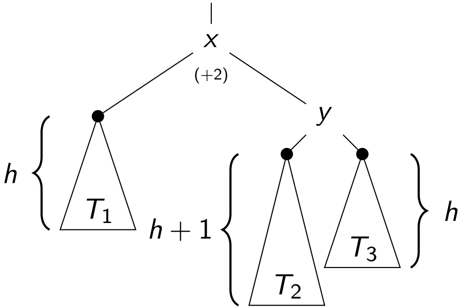
Left rotation:



We have a valid BST and $\mathrm{BF}(y) = 0$ so the tree is AVL-balanced again.

Rebalancing unbalanced trees (2)

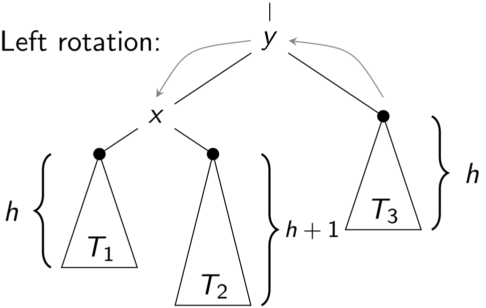
But what if it was y's left subtree that had height $h+1$?



Rebalancing unbalanced trees (2)

But what if it was y's left subtree that had height $h+1$?

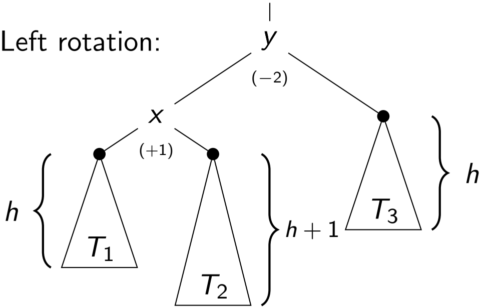
Left rotation:



Rebalancing unbalanced trees (2)

But what if it was y's left subtree that had height $h+1$?

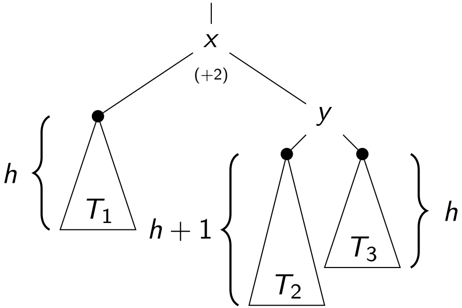
Left rotation:



Left rotation has failed: now y's balance factor is bad instead of x's.

Rebalancing unbalanced trees (2)

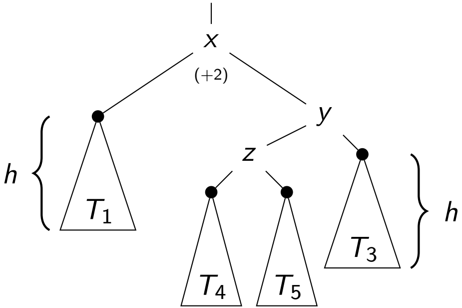
But what if it was y's left subtree that had height $h+1$?



We must look at y's left subtree T_2 more closely.

Rebalancing unbalanced trees (2)

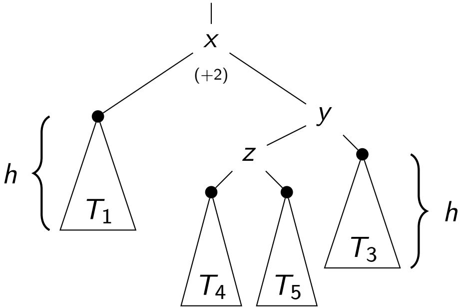
But what if it was y's left subtree that had height $h+1$?



One of T_4 and T_5 has height h; the other has height $h-1$.
(They both used to have height $h-1$ but we just inserted into one of them to cause the imbalance.)

Rebalancing unbalanced trees (2)

But what if it was y's left subtree that had height $h+1$?

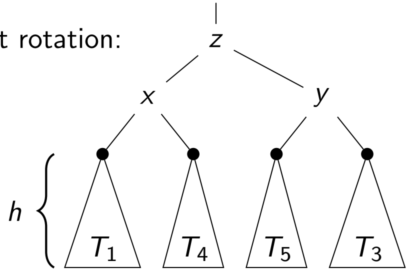


T_1 contains values v with $v < x$.
T_3 contains values v with $v > y$.
T_4 contains values v with $x < v < z$.
T_5 contains values v with $z < v < y$.

Rebalancing unbalanced trees (2)

But what if it was y's left subtree that had height $h+1$?

Right-left rotation:

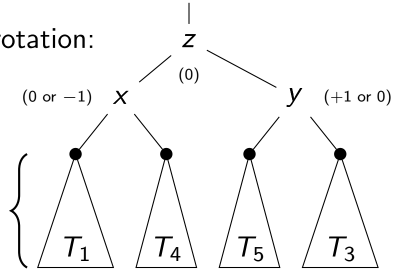


T_1 contains values v with $v < x$.
T_3 contains values v with $v > y$.
T_4 contains values v with $x < v < z$.
T_5 contains values v with $z < v < y$.

Rebalancing unbalanced trees (2)

But what if it was y's left subtree that had height $h+1$?

Right-left rotation:



One of T_4 and T_5 has height $h-1$.
Either $\mathrm{BF}(x)=0$ and $\mathrm{BF}(y)=+1$ or $\mathrm{BF}(x)=-1$ and $\mathrm{BF}(y)=0$.
In both cases, $\mathrm{BF}(z)=0$ so the tree is AVL-balanced again.

AVL rebalancing

The $+2/-2$ node is	Its taller subtree is	You rotate
left-heavy (-2)	left	right
	right	left-right
right-heavy $(+2)$	left	right-left
	right	left

Right-rotation and left-right-rotation are the mirror-images of the previous slides.

AVL tree insertion

- Insert as in an ordinary BST.
- Recalculate balance factors up path from the insertion point towards the root.
- Stop if a node's new balance factor is 0.
- If a balance factor becomes ± 2, rotate and stop.

AVL tree insertion

- Insert as in an ordinary BST.
- Recalculate balance factors up path from the insertion point towards the root.
- Stop if a node's new balance factor is 0.
- If a balance factor becomes ± 2, rotate and stop.

Running time:

- Insert takes time $O(\mathrm{height})$.

AVL tree insertion

- Insert as in an ordinary BST.
- Recalculate balance factors up path from the insertion point towards the root.
- Stop if a node's new balance factor is 0.
- If a balance factor becomes ± 2, rotate and stop.

Running time:

- Insert takes time $O(\mathrm{height})$.
- Retracing takes time $O(\mathrm{height})$.

AVL tree insertion

- Insert as in an ordinary BST.
- Recalculate balance factors up path from the insertion point towards the root.
- Stop if a node's new balance factor is 0.
- If a balance factor becomes ± 2, rotate and stop.

Running time:

- Insert takes time $O(\text{height})$.
- Retracing takes time $O(\text{height})$.
- Rotation requires adjusting at most three nodes' references: takes time $O(1)$.

Height of AVL trees

- It can be shown that an AVL-balanced tree has height $\Theta(\log n)$.
- The height $\Theta(n)$ worst-case of ordinary BSTs is impossible in an AVL tree.

Height of AVL trees

- It can be shown that an AVL-balanced tree has height $\Theta(\log n)$.
- The height $\Theta(n)$ worst-case of ordinary BSTs is impossible in an AVL tree.
- Therefore, insert runs in time $O(\log n)$.

Height of AVL trees

- It can be shown that an AVL-balanced tree has height $\Theta(\log n)$.
- The height $\Theta(n)$ worst-case of ordinary BSTs is impossible in an AVL tree.
- Therefore, insert runs in time $O(\log n)$.
- Search is also $O(\text{height}) = O(\log n)$.
- Deletion can also be done in time $O(\log n)$, using rotations appropriately.

AVL trees – summary

- Balance factor = (height of right) – (height of left).
- AVL trees are BSTs where $\mathrm{BF}(x) \in \{-1, 0, +1\}$ for all nodes x.

AVL trees – summary

- Balance factor = (height of right) – (height of left).
- AVL trees are BSTs where $\mathrm{BF}(x) \in \{-1, 0, +1\}$ for all nodes x.
- Balance factor is stored in nodes and updated on inserts.
- Unbalanced nodes require rotations.

AVL trees – summary

- Balance factor $=$ (height of right) $-$ (height of left).
- AVL trees are BSTs where $\mathrm{BF}(x) \in \{-1, 0, +1\}$ for all nodes x.
- Balance factor is stored in nodes and updated on inserts.
- Unbalanced nodes require rotations.
- Insert, search, delete in time $O(\log n)$.
- Avoids the $\Theta(n)$ worst-case of BSTs.