COMP2521 23T3
Balanced Binary Search Trees

Kevin Luxa

cs2521@cse.unsw.edu.au

balanced trees
avl trees

Balance

Balancing Operations

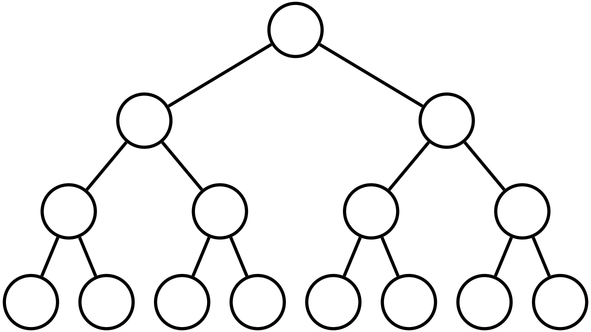
Balancing Methods

AVL Trees

The structure, height, and hence
performance
of a binary search tree
depends on the order of insertion.

Best case

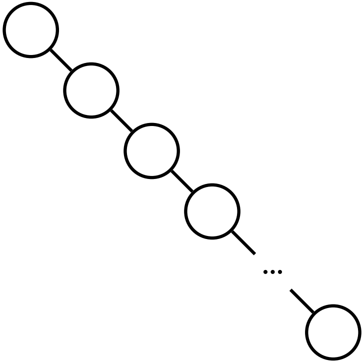
Items are inserted evenly on the left and right throughout the tree
Height of tree will be $O(\log n)$



Balance
Balancing Operations
Balancing Methods
AVL Trees

Worst case

Items are inserted in ascending or descending order
such that tree consists of a single branch
Height of tree will be $O(n)$



Balance

Balancing
Operations

Balancing
Methods

AVL Trees

A binary tree of n nodes is said to be
balanced if it has (close to) minimal height ($O(\log n)$), and
degenerate if it has (close to) maximal height ($O(n)$).

We want to build balanced trees.

Types of Balance

SIZE BALANCED

a *weight-balanced* or
size-balanced tree has,
for every node,

$$|\textsc{size}(l) - \textsc{size}(r)| \leq 1$$

HEIGHT BALANCED

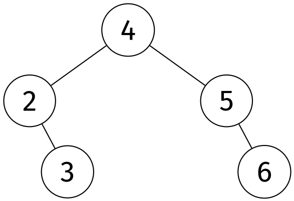
a *height-balanced* tree has,
for every node,

$$|\textsc{height}(l) - \textsc{height}(r)| \leq 1$$

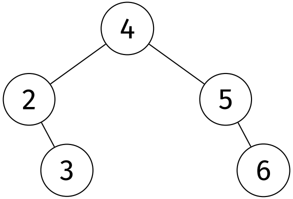
Balance
Examples
Balancing Operations
Balancing Methods
AVL Trees

Balanced or Not?

(I)



Balanced or Not?
(I)



$$\text{SIZE}\left(T_4\right) = 5$$
$$\text{SIZE}\left(T_2\right) = 2$$
$$\text{SIZE}\left(T_5\right) = 2$$
$$\text{SIZE}\left(T_3\right) = 1$$
$$\text{SIZE}\left(T_6\right) = 1$$
$$\text{SIZE}\left(T_\varnothing\right) = 0$$

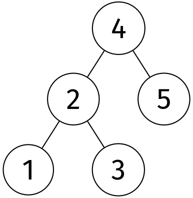
SIZE BALANCED

$$\text{HEIGHT}\left(T_4\right) = 2$$
$$\text{HEIGHT}\left(T_2\right) = 1$$
$$\text{HEIGHT}\left(T_5\right) = 1$$
$$\text{HEIGHT}\left(T_3\right) = 0$$
$$\text{HEIGHT}\left(T_6\right) = 0$$
$$\text{HEIGHT}\left(T_\varnothing\right) = -1$$

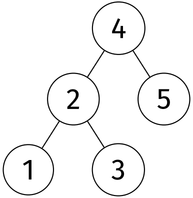
HEIGHT BALANCED

Balanced or Not?
(II)

Balance
Examples
Balancing Operations
Balancing Methods
AVL Trees



Balanced or Not? (II)



$$
\begin{aligned}
\text{SIZE}(\tau_4) &= 5 \\
\text{SIZE}(\tau_2) &= 3 \\
\text{SIZE}(\tau_5) &= 1 \\
\text{SIZE}(\tau_1) &= 1 \\
\text{SIZE}(\tau_3) &= 1 \\
|\text{SIZE}(\tau_2) - \text{SIZE}(\tau_5)| &= 2
\end{aligned}
$$

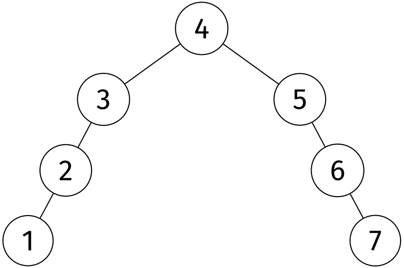
NOT SIZE BALANCED

$$
\begin{aligned}
\text{HEIGHT}(\tau_4) &= 2 \\
\text{HEIGHT}(\tau_2) &= 1 \\
\text{HEIGHT}(\tau_5) &= 0 \\
\text{HEIGHT}(\tau_1) &= 0 \\
\text{HEIGHT}(\tau_3) &= 0 \\
|\text{HEIGHT}(\tau_2) - \text{HEIGHT}(\tau_5)| &= 1
\end{aligned}
$$

HEIGHT BALANCED

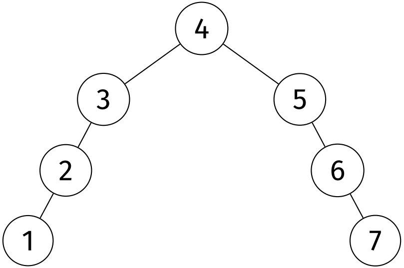
Balanced or Not?
(III)

Balance
Examples
Balancing Operations
Balancing Methods
AVL Trees



Balance
Examples
Balancing Operations
Balancing Methods
AVL Trees

Balanced or Not? (III)



Let's look at $\mathcal{T}_3$.

$$\text{SIZE}\left(\mathcal{T}_2\right) = 2$$
$$\text{SIZE}\left(\mathcal{T}_\varnothing\right) = 0$$
$$|2 - 0 = 2| > 1$$

NOT SIZE BALANCED

Let's look at $\mathcal{T}_5$.

$$\text{HEIGHT}\left(\mathcal{T}_\varnothing\right) = -1$$
$$\text{HEIGHT}\left(\mathcal{T}_6\right) = 1$$
$$|-1 - 1| = 2 > 1$$

NOT HEIGHT BALANCED

Challenge:

Prove that every size-balanced tree is height-balanced.

Balancing Operations

Balance
Balancing Operations
Rotations
Partition
Balancing Methods
AVL Trees

Rotation

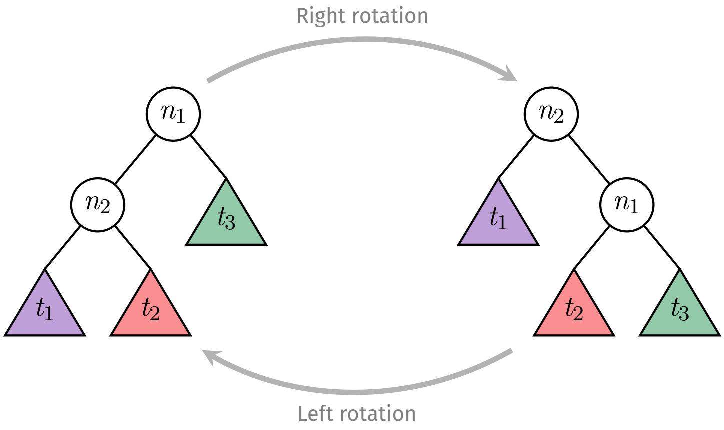
- Left rotation
  - Move right child to root, rearrange links to retain order
- Right rotation
  - Move left child to root, rearrange links to retain order

Partition

- Rearrange tree around a specified node by rotating it up to the root

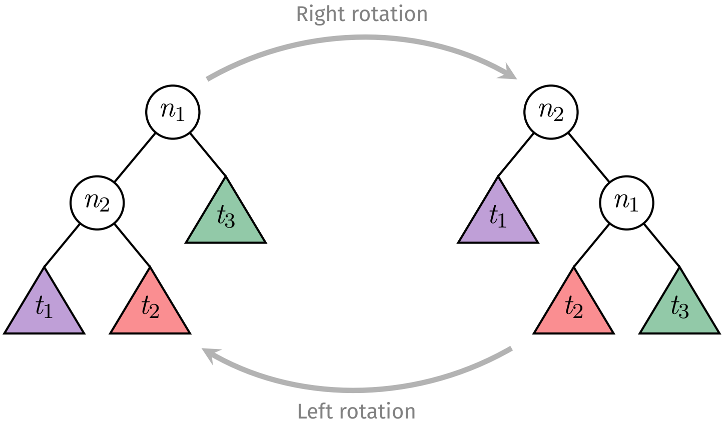
Rotations

LEFT ROTATION and RIGHT ROTATION:
a pair of operations
that change the balance of a tree



Balance
Balancing Operations
Rotations
Partition
Balancing Methods
AVL Trees

Rotations maintain the order of a search tree:

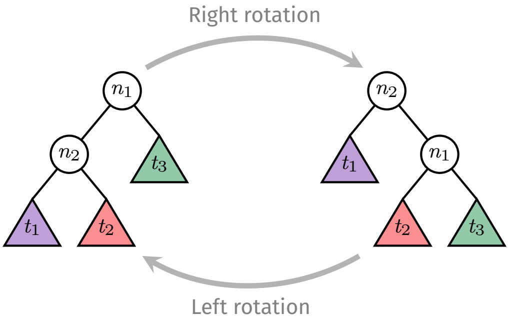


(all values in t_1) $< n_2 <$ (all values in t_2) $< n_1 <$ (all values in t_3)

Rotations

Method for right rotation:

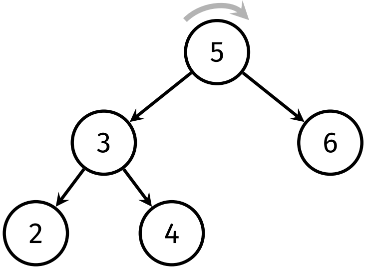
- before the rotation: n_1 is original root, n_2 is left child of root
- n_1's left subtree is now what was n_2's right subtree
- n_2's right child is now n_1
- n_2 is now the new root
- everything else is unchanged



Rotations

Example

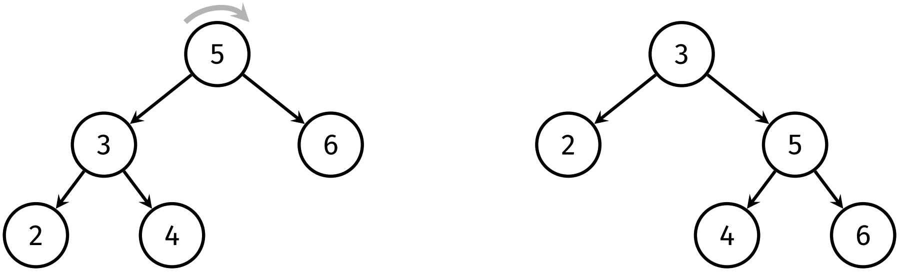
Rotate right at 5



Rotations

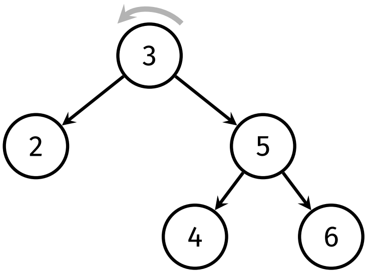
Example

Rotate right at 5



Balance
Balancing Operations
Rotations
Partition
Balancing Methods
AVL Trees

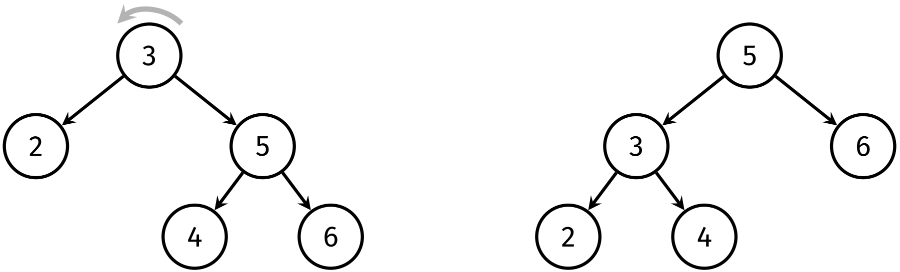
Rotate left at 3



Rotations

Example

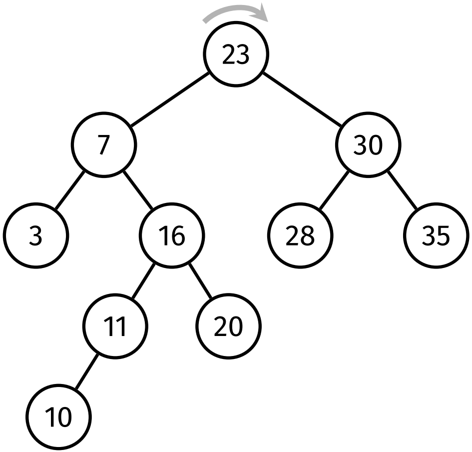
Rotate left at 3



Rotations

Example

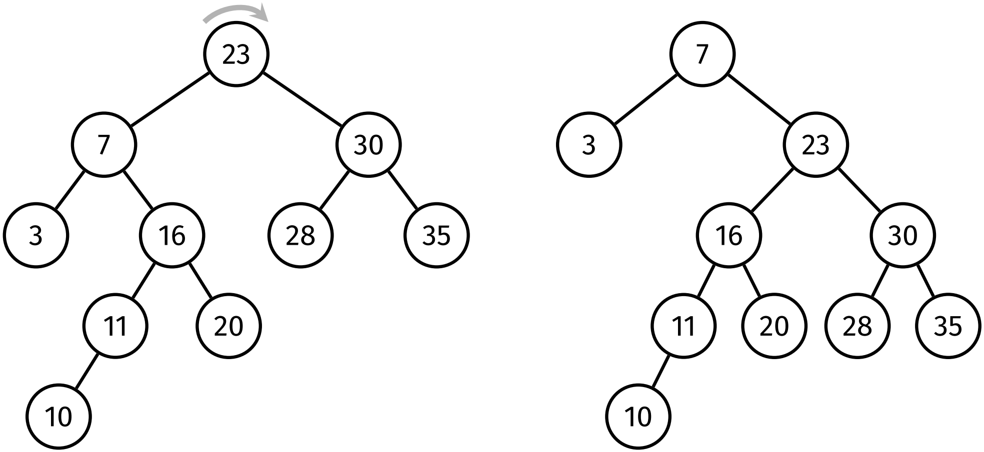
Rotate right at 23



Rotations

Example

Rotate right at 23



Rotations

Implementation

```
struct node *rotateRight(struct node *root) {
    if (root == NULL || root->left == NULL) return root;
    struct node *newRoot = root->left;
    root->left = newRoot->right;
    newRoot->right = root;
    return newRoot;
}


struct node *rotateLeft(struct node *root) {
    if (root == NULL || root->right == NULL) return root;
    struct node *newRoot = root->right;
    root->right = newRoot->left;
    newRoot->left = root;
    return newRoot;
}
```

Analysis:

- Rotation is cheap - $O(1)$
- Rotation requires simple, localised pointer re-arrangements

Sometimes, rotation is applied along one branch, from leaf to root

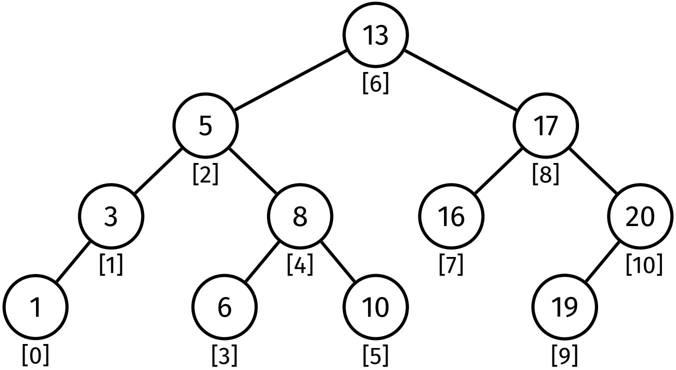
- Cost of this is $O(h)$ where h is the height of the tree

Balance
Balancing Operations
Rotations
Partition
Balancing Methods
AVL Trees

Partition

```
partition(tree, i)
```

Rearrange the tree so that the element with index i becomes the root



Partition

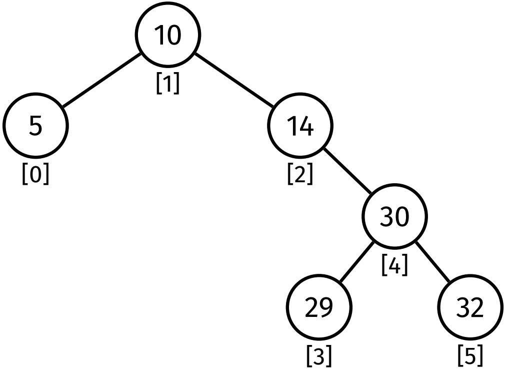
Method:

- Find element with index i
- Perform rotations to lift it to the root
  - If it is the left child of its parent, perform right rotation at its parent
  - If it is the right child of its parent, perform left rotation at its parent
  - Repeat until it is at the root of the tree

Partition

Example

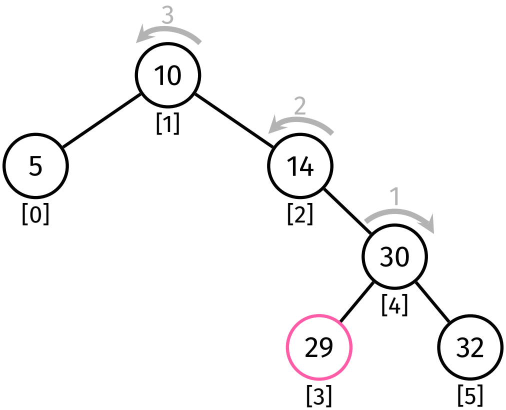
Partition this tree around index 3:



Partition

Example

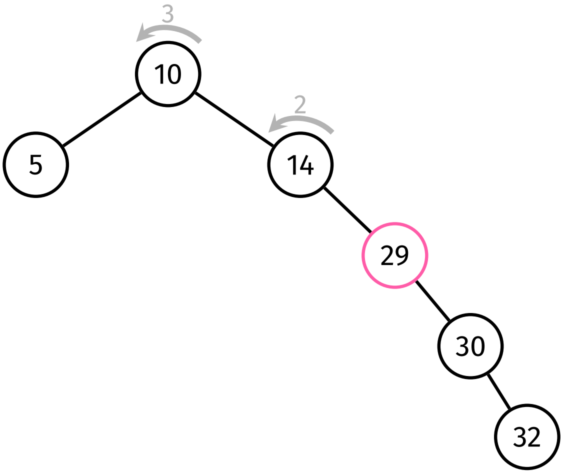
Partition this tree around index 3:



Partition

Example

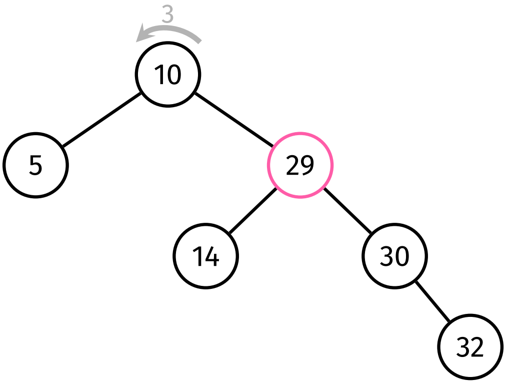
After right rotation at 30:



Partition

Example

After left rotation at 14:

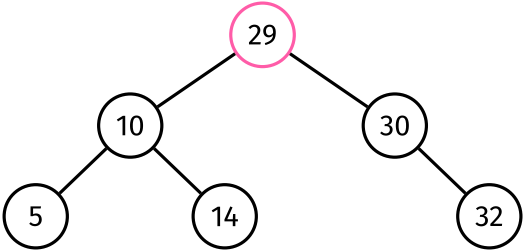


Balance
Balancing Operations
Rotations
Partition
Balancing Methods
AVL Trees

Partition

Example

After left rotation at 10:



Balance
Balancing Operations
Rotations
Partition
Balancing Methods
AVL Trees

Partition

Pseudocode

```
partition(t, i):
    Inputs: tree t, index i
    Output: tree with i-th item moved to root

    m = size(t->left)
    if i < m:
        t->left = partition(t->left, i)
        t = rotateRight(t)
    else if i > m:
        t->right = partition(t->right, i - m - 1)
        t = rotateLeft(t)

    return t
```

Balance
Balancing Operations
Rotations
Partition
Balancing Methods
AVL Trees

Analysis:

- size() operation is expensive
  - needs to traverse whole subtree
- can cause partition to be $O(n^2)$ in the worst case
- to improve efficiency, can change node structure so that each node stores the size of its subtree in the node itself
  - however, this will require extra work in other functions to maintain

```
struct node {
    int item;
    int size;
    struct node *left;
    struct node *right;
};
```

Balance
Balancing Operations
Balancing Methods
Global Rebalancing
Root Insertion
Randomised Insertion
AVL Trees

Balancing Methods

- Global Rebalancing
- Root Insertion
- Randomised Insertion

Global Rebalancing

Balance
Balancing Operations
Balancing Methods
Global Rebalancing
Root Insertion
Randomised Insertion
AVL Trees

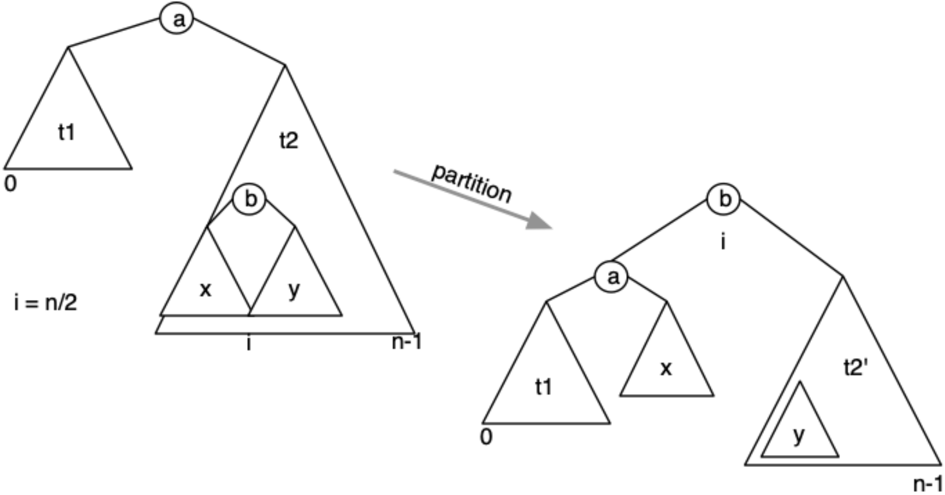
Idea:
Completely rebalance whole tree so it is size-balanced

Method:
Lift the median node to the root
by partitioning on $\text{SIZE}(t)/2$,
then rebalance both subtrees (recursively)

Balance
Balancing Operations
Balancing Methods
Global Rebalancing
Root Insertion
Randomised Insertion
AVL Trees

Global Rebalancing

First, partition on index $n/2$...



...then rebalance both subtrees

Balance
Balancing Operations
Balancing Methods
Global Rebalancing
Root Insertion
Randomised Insertion
AVL Trees

Global Rebalancing

Pseudocode

```
rebalance(t):
    Inputs: tree t
    Output: rebalanced t

    if size(t) < 3:
        return t

    t = partition(t, size(t) / 2)
    t->left = rebalance(t->left)
    t->right = rebalance(t->right)
    return t
```

Global Rebalancing

Analysis

Worst-case time complexity: $O(n \log n)$

- Assume nodes store the size of their subtrees
- First step: partition entire tree on index $n/2$
  - This takes at most n recursive calls, n rotations $\Rightarrow n$ steps
  - Result is two subtrees of size $\approx n/2$
- Then partition both subtrees
  - Partitioning these subtrees takes $n/2$ steps each $\Rightarrow n$ steps in total
  - Result is four subtrees of size $\approx n/4$
- ...and so on...
- About $\log_2 n$ levels of partitioning in total, each requiring n steps $\Rightarrow O(n \log n)$

Global Rebalancing

Problems

What if we insert more items?

- Options:
  - Rebalance on every insertion
    - Not feasible
  - Rebalance every k insertions; what k is good?
  - Rebalance when imbalance exceeds threshold.

- It's a tradeoff...
  - We either have more costly insertions
  - Or we have degraded performance for periods of time

Periodic Rebalancing

Balance
Balancing Operations
Balancing Methods
Global Rebalancing
Root Insertion
Randomised Insertion
AVL Trees

```
bstInsert(t, v):
    Inputs: tree t, value v
    Output: t with v inserted

    t = insertAtLeaf(t, v)

    if size(t) mod k = 0:
        t = rebalance(t)

    return t
```

Periodic Rebalancing

Remarks

Balance
Balancing Operations
Balancing Methods
Global Rebalancing
Root Insertion
Randomised Insertion
AVL Trees

- Good if tree is not modified very often
- Otherwise...
  - Insertion will be slow occasionally due to rebalancing
  - Performance will gradually degrade until next rebalance

Balance
Balancing Operations
Balancing Methods
Global Rebalancing
Root Insertion
Randomised Insertion
AVL Trees

Global vs Local Rebalancing

GLOBAL REBALANCING

walks every node, balances its subtree;
$\Rightarrow$ perfectly balanced tree — at cost.

LOCAL REBALANCING

do small, incremental operations
to improve the overall balance of the tree
... at the cost of imperfect balance

Balance

Balancing Operations

Balancing Methods

Global Rebalancing

Root Insertion

Randomised Insertion

AVL Trees

Idea:

Rotations change the structure of a tree

If we perform some rotations every time we insert, that may restructure the tree randomly enough such that it is more balanced

One systematic way to perform these rotations:
Insert new values at the root

Root Insertion

Method:

Insert new value normally (at the leaf) ...

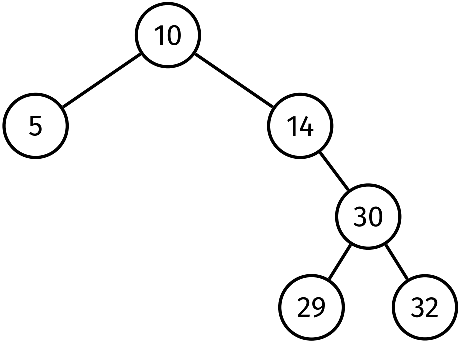
... and then rotate the new node up to the root.

Root Insertion

Example

Balance
Balancing Operations
Balancing Methods
Global Rebalancing
Root Insertion
Randomised Insertion
AVL Trees

Insert 24 at the root of this tree:

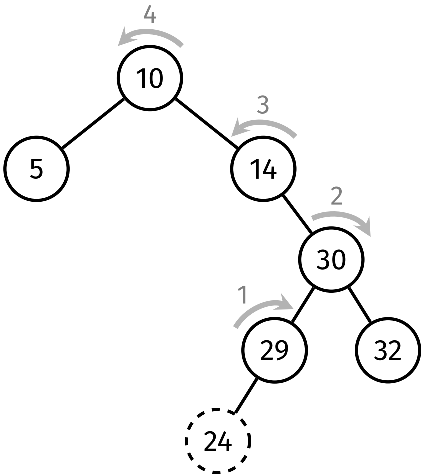


Balance
Balancing Operations
Balancing Methods
Global Rebalancing
Root Insertion
Randomised Insertion
AVL Trees

Root Insertion

Example

Insert 24 at the root of this tree:

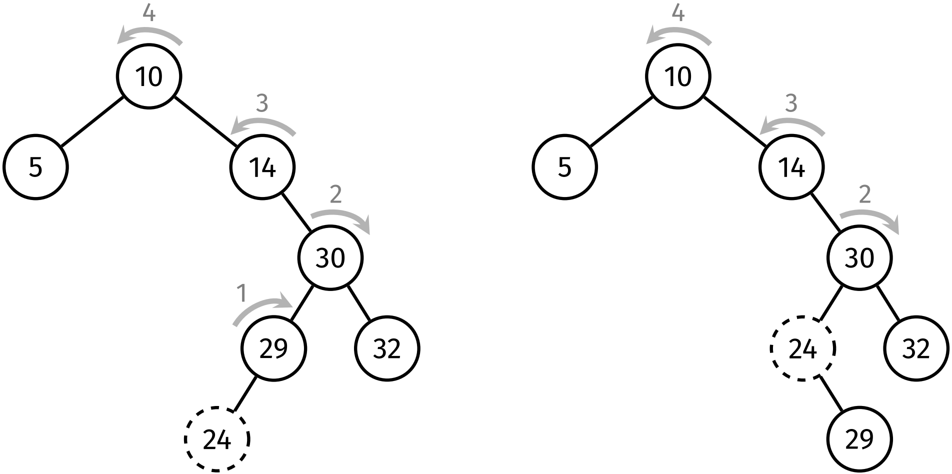


Balance
Balancing Operations
Balancing Methods
Global Rebalancing
Root Insertion
Randomised Insertion
AVL Trees

Root Insertion

Example

Rotate right at 29

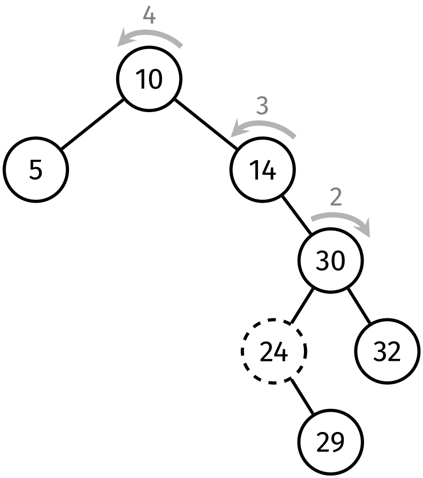


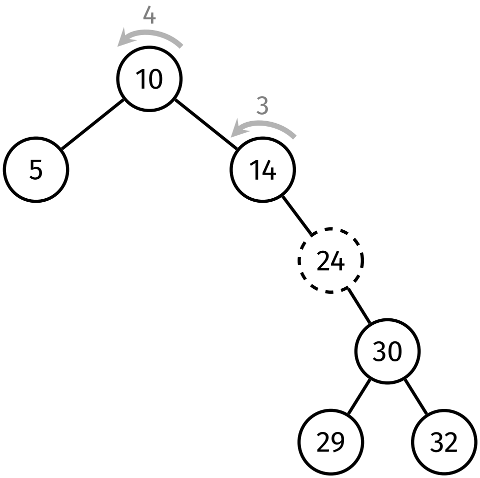
Balance
Balancing Operations
Balancing Methods
Global Rebalancing
Root Insertion
Randomised Insertion
AVL Trees

Root Insertion

Example

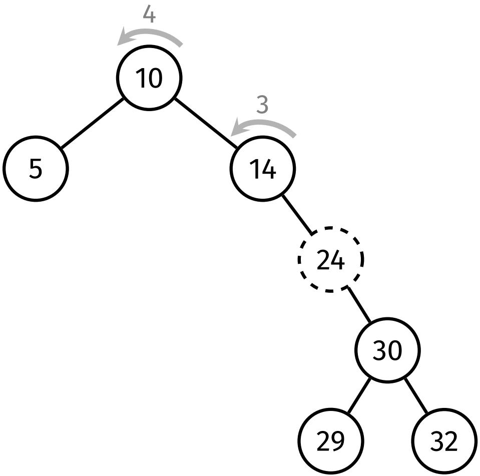
Rotate right at 30





Balance
Balancing Operations
Balancing Methods
Global Rebalancing
Root Insertion
Randomised Insertion
AVL Trees

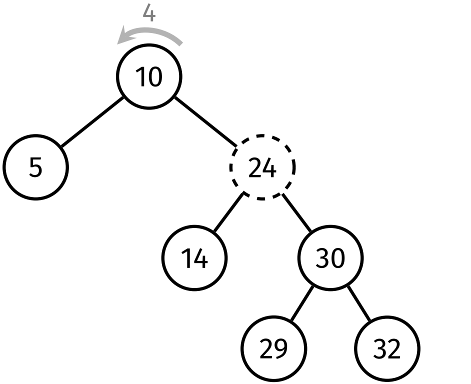
Rotate left at 14



Root Insertion

Example

Rotate left at 10



Root Insertion

Pseudocode

```
insertAtRoot(t, v):
    Inputs: tree t, value v
    Output: t with v inserted at the root

    if t is empty:
        return new node containing v
    else if v < t->item:
        t->left = insertAtRoot(t->left, v)
        t = rotateRight(t)
    else if v > t->item:
        t->right = insertAtRoot(t->right, v)
        t = rotateLeft(t)

    return t
```

Balance

Balancing Operations

Balancing Methods
Global Rebalancing
Root Insertion
Randomised Insertion

AVL Trees

Analysis:

- Same complexity as normal insertion: $O(h)$
    - In reality, cost is doubled, as you need to traverse down and rotate up
- Tree is more likely to be balanced, but no guarantee
- Insert at root ensures recently inserted items are close to the root
    - Useful for applications where recently added items are more likely to be searched
- Major problem: ascending-ordered and descending-ordered data is still a worst case for root insertion

Balance
Balancing Operations
Balancing Methods
Global Rebalancing
Root Insertion
Randomised Insertion
AVL Trees

Randomised Insertion

BSTs don't have control over insertion order.
worst cases — (partially) ordered data — are common.

Idea:
Introduce some randomness into insertion algorithm:
Randomly choose whether to insert normally or insert at root

Randomised Insertion

Pseudocode

```
insertRandom(t, v):
    Inputs: tree t, value v
    Output: t with v inserted

    if t is empty:
        return new node containing v

    // p/q chance of inserting at root
    if random() mod q < p:
        return insertAtRoot(t, v)
    else:
        return insertAtLeaf(t, v)
```

Note: `random()` is a pseudo-random number generator
30% chance of root insertion $\Rightarrow$ choose p = 3, q = 10

Balance
Balancing Operations
Balancing Methods
Global Rebalancing
Root Insertion
Randomised Insertion
AVL Trees

Randomised Insertion

Remarks

Randomised insertion creates similar results to inserting items in random order.

Tree is more likely to be balanced (but no guarantee)

Balance

Balancing Operations

Balancing Methods

AVL Trees

Insertion

Search

AVL Trees

Balance
Balancing Operations
Balancing Methods
AVL Trees
Insertion
Search

Motivation:

- Previous balancing methods are either inefficient, or don't guarantee a balanced tree ($O(\log n)$ height)

Balance
Balancing Operations
Balancing Methods
AVL Trees
Insertion
Search

Invented by Georgy Adelson-Velsky and Evgenii Landis (1962)

Approach:

- Keep tree height-balanced
- Repair balance as soon as imbalance occurs
  - During insertion or deletion
- Repairs are done locally, not by restructuring entire tree

AVL Tree Insertion

Balance
Balancing Operations
Balancing Methods
AVL Trees
Insertion
Search

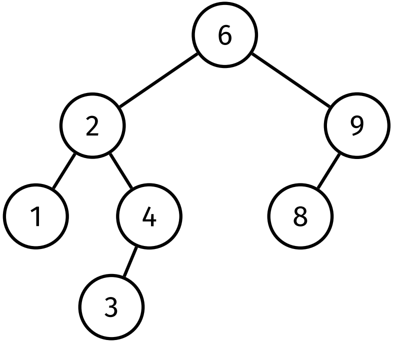
Method:

- Insert item recursively
- Check balance at each node along the insertion path *in reverse*
  - i.e., from bottom to top
- As soon as an imbalance is found, fix it

AVL Tree Insertion

Balance
Balancing Operations
Balancing Methods
AVL Trees
Insertion
Search

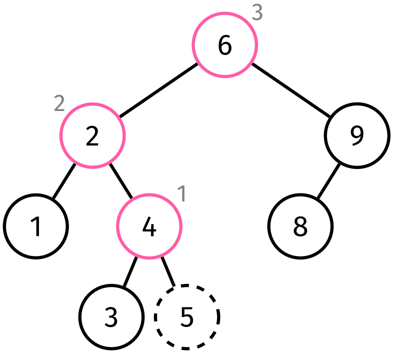
Example: Insert 5 into this tree



Balance
Balancing Operations
Balancing Methods
AVL Trees
Insertion
Search

AVL Tree Insertion

Example: Insert 5 into this tree



Balance must be checked at 4, then at 2, then at 6

AVL Tree Insertion

How to check balance along insertion path *in reverse*?

- Simple - perform balance checking as a *postorder* operation in the insertion function
  - In other words - insert balance checking code *below* recursive calls to insert

Outline of insertion process:

1. if the tree is empty:
   - return new node
2. insert recursively
3. check (and fix) balance
4. return root of updated tree

AVL Tree Insertion

Pseudocode

```
avlInsert(t, v):
    Inputs: AVL tree t, item v
    Output: t with v inserted

    if t is empty:
        return new node containing v
    else if v < t->item:
        t->left = avlInsert(t->left, v)
    else if v > t->item:
        t->right = avlInsert(t->right, v)
    else:
        return t

    ... continued on next slide ...
```

AVL Tree Insertion

Pseudocode

... continued from previous slide ...

```
leftHeight = height(t->left)
rightHeight = height(t->right)

if (leftHeight - rightHeight) > 1:
    if v > t->left->item:
        t->left = rotateLeft(t->left)
    t = rotateRight(t)
else if (rightHeight - leftHeight) > 1:
    if v < t->right->item:
        t->right = rotateRight(t->right)
    t = rotateLeft(t)

return t
```

Balance
Balancing Operations
Balancing Methods
AVL Trees
Insertion
Search

AVL Tree Insertion

There are 4 rebalancing cases:

Left Left

Left Right

Right Left

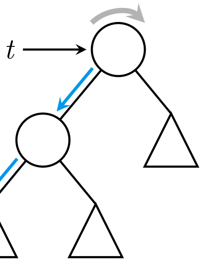
Right Right

AVL Tree Insertion

Balance
Balancing Operations
Balancing Methods
AVL Trees
Insertion
Search

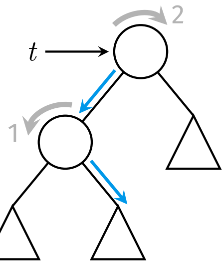
Left Left

```
if (leftHeight - rightHeight) > 1:
    if v > t->left->item:
        t->left = rotateLeft(t->left)
    t = rotateRight(t)
else if (rightHeight - leftHeight) > 1:
    if v < t->right->item:
        t->right = rotateRight(t->right)
    t = rotateLeft(t)
```



Left Right

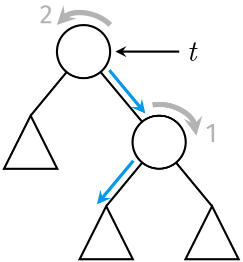
```
if (leftHeight - rightHeight) > 1:
    if v > t->left->item:
        t->left = rotateLeft(t->left)
    t = rotateRight(t)
else if (rightHeight - leftHeight) > 1:
    if v < t->right->item:
        t->right = rotateRight(t->right)
    t = rotateLeft(t)
```



Balance
Balancing Operations
Balancing Methods
AVL Trees
Insertion
Search

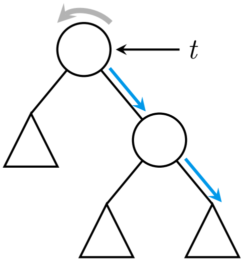
Right Left

```
if (leftHeight - rightHeight) > 1:
    if v > t->left->item:
        t->left = rotateLeft(t->left)
    t = rotateRight(t)
else if (rightHeight - leftHeight) > 1:
    if v < t->right->item:
        t->right = rotateRight(t->right)
    t = rotateLeft(t)
```



Right Right

```
if (leftHeight - rightHeight) > 1:
    if v > t->left->item:
        t->left = rotateLeft(t->left)
    t = rotateRight(t)
else if (rightHeight - leftHeight) > 1:
    if v < t->right->item:
        t->right = rotateRight(t->right)
    t = rotateLeft(t)
```

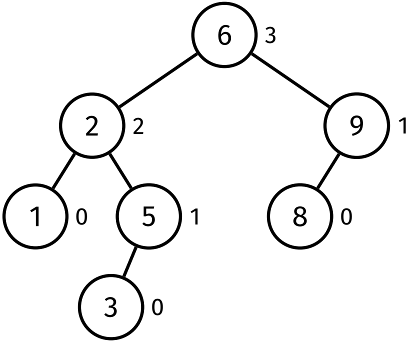


Balance
Balancing Operations
Balancing Methods
AVL Trees
Insertion
Search

AVL Tree Insertion

Rebalancing - Left Left

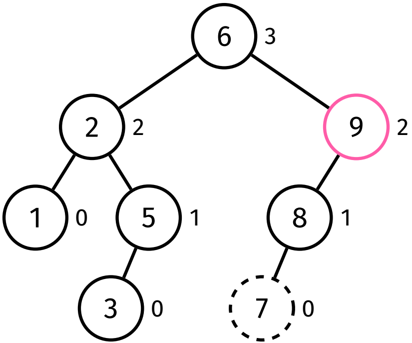
Insert 7 into this tree:



Balance
Balancing Operations
Balancing Methods
AVL Trees
Insertion
Search

AVL Tree Insertion

Rebalancing - Left Left



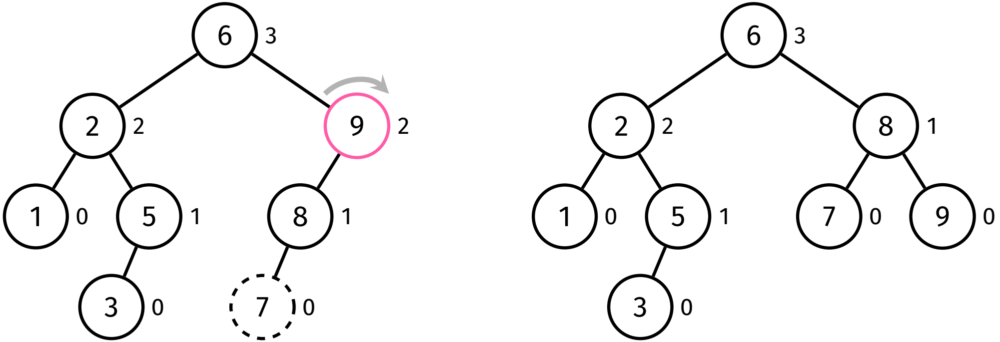
Check for balance at 8, then at 9, then at 6.

9 is unbalanced.

AVL Tree Insertion

Rebalancing - Left Left

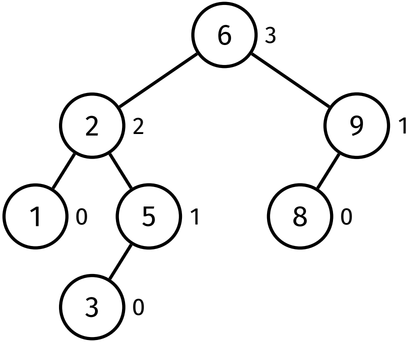
Balance
Balancing Operations
Balancing Methods
AVL Trees
Insertion
Search



AVL Tree Insertion

Rebalancing - Left Right

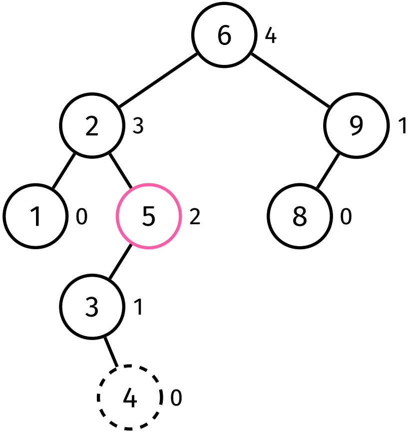
Insert 4 into this tree:



Balance
Balancing Operations
Balancing Methods
AVL Trees
Insertion
Search

AVL Tree Insertion

Rebalancing - Left Right

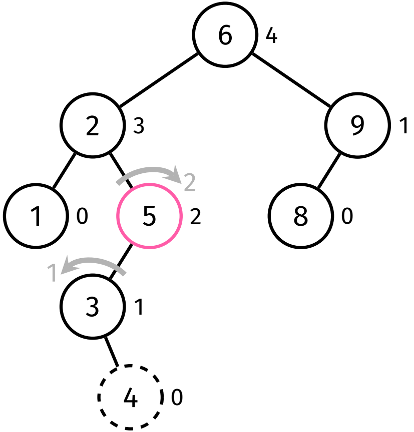


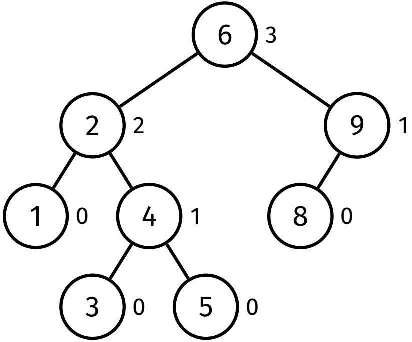
Check for balance at 3, then at 5, then at 2, then at 6.

5 is unbalanced.

AVL Tree Insertion

Rebalancing - Left Right





AVL Tree Insertion

Issues

AVL tree insertion requires balance checking
at each node on the insertion path...

...which requires the height of many subtrees to be computed

In an ordinary binary search tree, computing the height is $O(n)$!
(need to traverse whole (sub)tree)

AVL Tree Insertion

Issues

Solution:

For each node, store the height of its subtree in the node itself:

```
struct node {
    int item;
    int height;
    struct node *left;
    struct node *right;
};
```

Extra effort is required to maintain this data whenever the tree is modified.

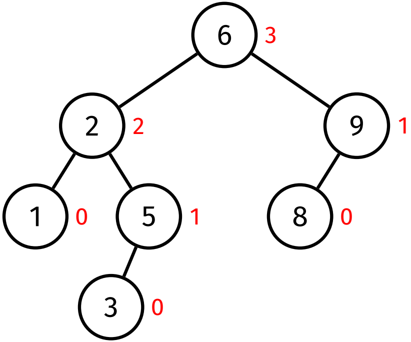
Balance

Balancing Operations

Balancing Methods

AVL Trees
Insertion
Search

Height of each node's subtree is stored in the node itself



AVL Tree Insertion

Maintaining Height Data

When does height data need to be maintained?

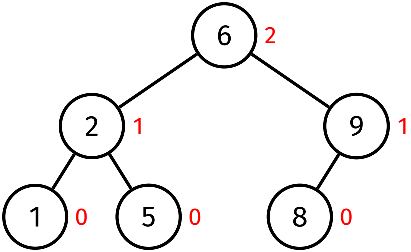
- Whenever a node is inserted
    - Heights of all ancestors may be affected
- Whenever a rotation is performed
    - Heights of original root and new root may be affected

AVL Tree Insertion

Maintaining Height Data - Insertions

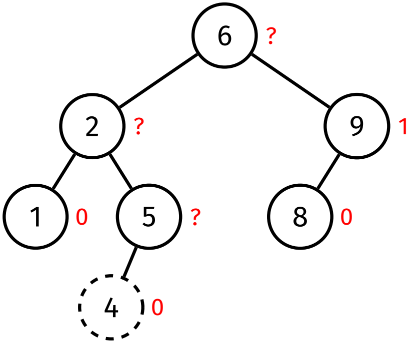
Whenever a node is inserted...
...heights of all ancestors may be affected

Example: Insert 4 into this tree



AVL Tree Insertion

Maintaining Height Data - Insertions

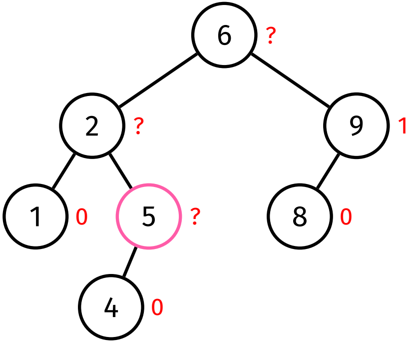


Recompute height of each ancestor (from bottom to top) using the heights stored in its children.

Balance
Balancing Operations
Balancing Methods
AVL Trees
Insertion
Search

AVL Tree Insertion

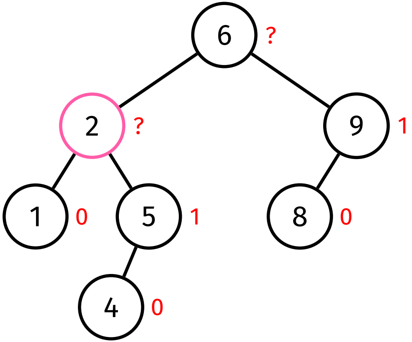
Maintaining Height Data - Insertions



The heights of 5's children are 0 and -1 (empty tree).

Thus, the height of 5 is $\max(0, -1) + 1 = 1$.

Balance
Balancing Operations
Balancing Methods
AVL Trees
Insertion
Search



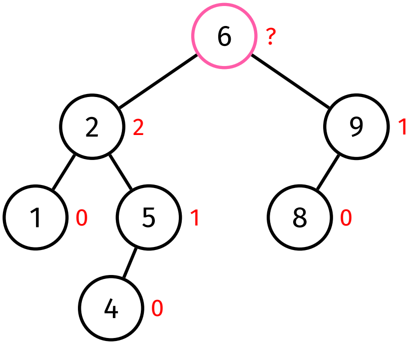
The heights of 2's children are 0 and 1.

Thus, the height of 2 is $\max(0, 1) + 1 = 2$.

Balance
Balancing Operations
Balancing Methods
AVL Trees
Insertion
Search

AVL Tree Insertion

Maintaining Height Data - Insertions

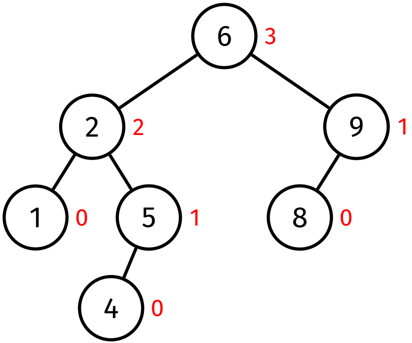


The heights of 6's children are 2 and 1.

Thus, the height of 6 is $\max(2, 1) + 1 = 3$.

AVL Tree Insertion

Maintaining Height Data - Insertions

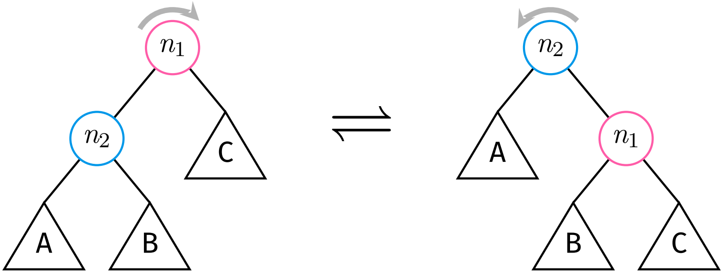


Done.

Note that recomputing the height of each node was done in $O(1)$ time.

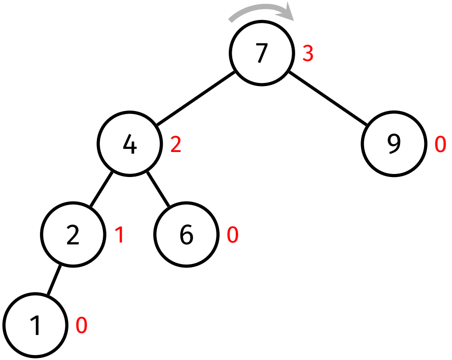
Balance
Balancing Operations
Balancing Methods
AVL Trees
Insertion
Search

Whenever a rotation is performed...
...heights of original root and new root may be affected



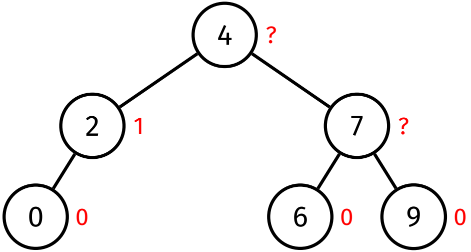
Balance
Balancing Operations
Balancing Methods
AVL Trees
Insertion
Search

Example: Perform a right rotation at 7



AVL Tree Insertion

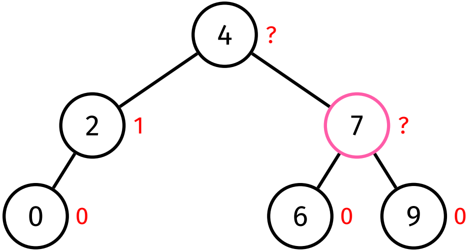
Maintaining Height Data - Rotations



Recompute height of original root
then recompute height of new root
using the heights stored in their children.

AVL Tree Insertion

Maintaining Height Data - Rotations



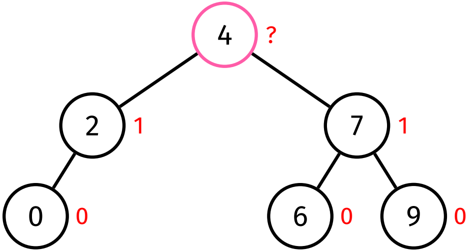
The height of 7's children are 0 and 0.

Thus, the height of 7 is $\max(0, 0) + 1 = 1$.

Balance
Balancing Operations
Balancing Methods
AVL Trees
Insertion
Search

AVL Tree Insertion

Maintaining Height Data - Rotations



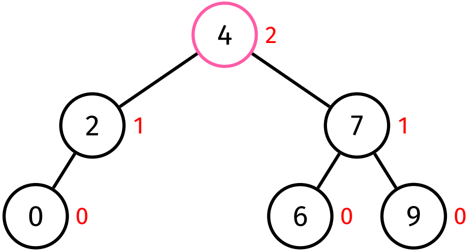
The height of 4's children are 1 and 1.

Thus, the height of 4 is $\max(1, 1) + 1 = 2$.

Balance
Balancing Operations
Balancing Methods
AVL Trees
Insertion
Search

AVL Tree Insertion

Maintaining Height Data - Rotations



Done.

Every rotation, two height updates are performed, each in $O(1)$ time.

Balance
Balancing Operations
Balancing Methods
AVL Trees
Insertion
Search

Analysis:

- An AVL tree is always height balanced
  - So height of an AVL tree is $O(\log n)$
- Checking/fixing balance and maintaining height data is $O(1)$
- So checking/fixing balance adds $O(1)$ extra work for each node on insertion path
- Therefore, worst-case time complexity of AVL tree insertion is $O(\log n)$

Balance
Balancing Operations
Balancing Methods
AVL Trees
Insertion
Search

AVL Tree Search

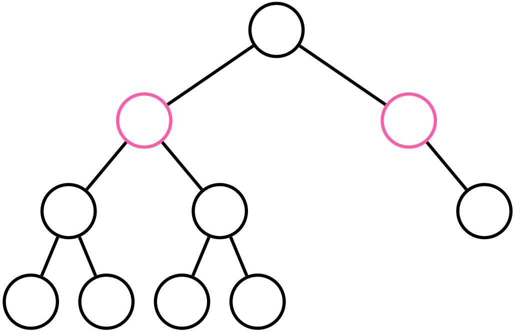
Exactly the same as for regular BSTs.

Worst-case time complexity is $O(\log n)$, since AVL trees are height-balanced

AVL Trees

Final Remarks

- AVL trees are always height-balanced
- Worst-case time complexity of $O(\log n)$ for insertion, search, deletion
- AVL trees are not necessarily weight-balanced, for example:



Feedback

https://forms.office.com/r/aPF09YHZ3X