COM1009
Introduction to Algorithms and Data Structures

Topic 10: Binary Search Trees; AVL trees

Essential Reading:
Sections 12.1-12.3

Aims of this lecture

- We've seen a lot of binary trees already
  - Recurrence tree for visualising runtime in recursive calls; HeapSort uses imaginary trees; decision trees in the lower bound for comparison sorts
- This topic: binary trees in more detail, including
  - how to prove **inductive statements about trees**.
- **Binary search trees** and their typical operations.
- **AVL trees**
  - **self-balancing trees** with operations in time $O(\log n)$.
  - **How to rebalance a tree** using **rotations**

Recall

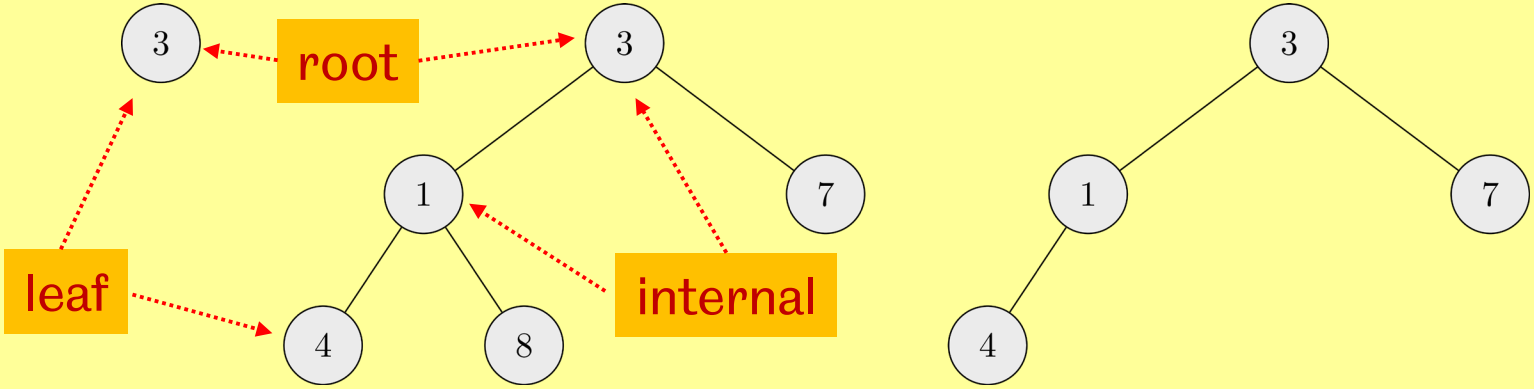
- Elements can contain **satellite data**
  - a **key** is used to identify the element.
- Operations on dynamic sets S:
  - **Search(S, k):** returns element x with key k, or NIL
  - **Insert(S, x):** adds element x to S
  - **Delete(S, x):** removes element x from S
  - **Minimum(S), Maximum(S):** only for totally ordered sets
  - **Successor(S, x), Predecessor(S, x):** next or previous element
- **Time** often measured using n, the number of elements in S.

Binary trees

- Trees where every node has at most two children.
- We can also define binary trees recursively: A **binary tree** is a structure defined using a finite set of nodes, where
  - Either: The tree is empty (no nodes)
  - Or: It comprises a root node, a left subtree and a right subtree
- This view is very handy for proving statements about trees by induction (see later).
- The root of the left subtree of a node is called the **left child**, that of the right subtree is called the **right child**.

Definitions for binary trees

- We'll assume that all nodes are labelled by numbers.

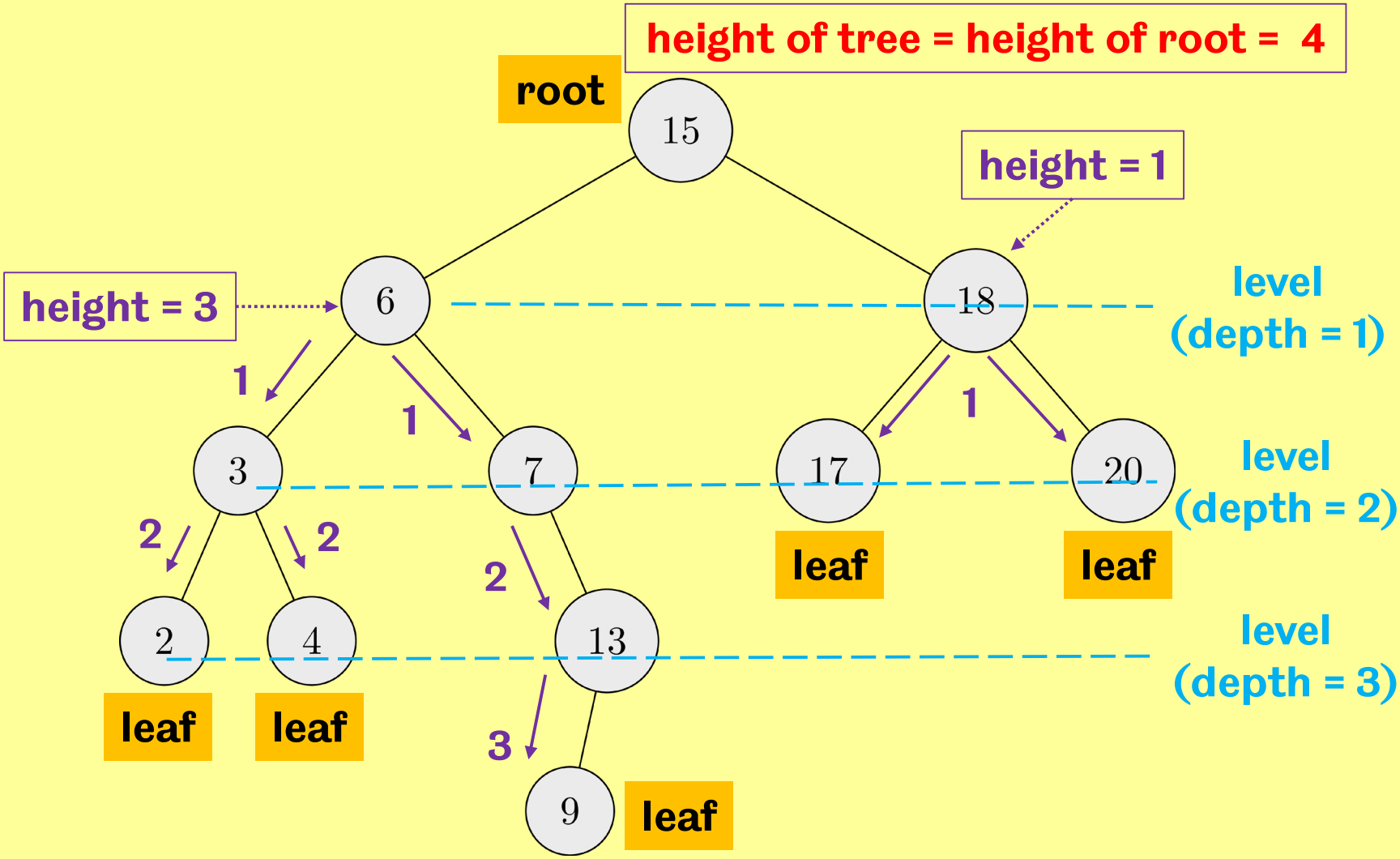


- A **path** in a tree is a sequence of nodes linked by edges. The **length** of a path is the number of edges.
- A **leaf** is a node that has no children; otherwise it is **internal**.
- Also: **root, siblings, parents, ancestor, descendant, etc.**

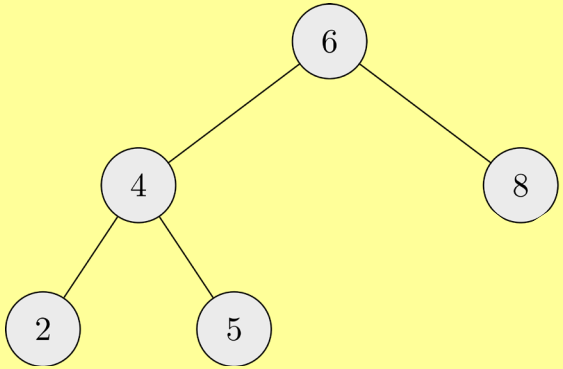
Depth and height

- The **depth** of a node in a tree is the length of a (simple) path from that node to the root.
- A **level** of a tree is a set of nodes of the same depth.
- The **height** of a node in a tree is the length of the longest path from that node to a leaf.
- The **height of a tree** is the height of its root.
- A binary tree is **full** if each node is either a leaf or has exactly two children.
- It is **complete** if it is full and all leaves have the same level.

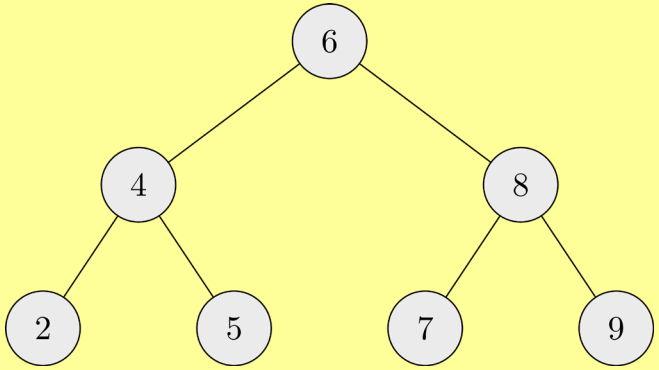
Depth, height and levels



Full vs complete



Full – each node is either a leaf or has exactly two children



Complete – full + all leaves have the same depth

Principle of induction for binary trees

- We can use the recursive definition to prove statements about trees using a generalised form of induction. The general recipe is this:
  - **Base case**: show that the statement holds for the "smallest" trees, e.g. an empty tree or just the root node (depending on the statement).
  - **Induction step**: any larger tree has a root and two subtrees (possibly empty). Assume that the statement holds for both subtrees and show that it then holds for the whole tree.
- Caveat: if a statement reads "for all non-empty trees", in the induction step we may need to watch out for empty subtrees.

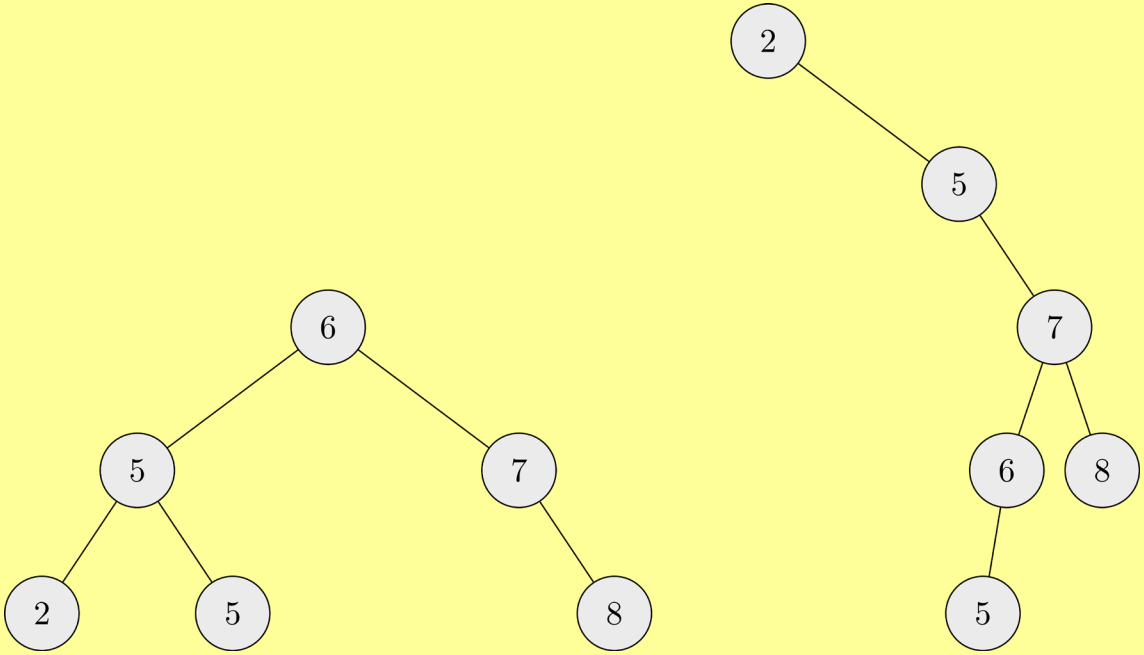
Inductive proofs on trees: example

- **Theorem**: A non-empty binary tree of height h has no more than 2^h leaves.
  - We used this statement in the lower bound for comparison sorts. Now we prove it.
- **Proof**:
  - **Base case**: A tree of height 0 is an isolated root node. It has exactly $2^0=1$ leaves (the root is itself a leaf in this case).
  - **Induction step**: a tree of height $h > 0$ has a root and two subtrees, at least one of which has height h-1. The other subtree is either empty or has height $\leq h$-1. Assume that the statement holds for both subtrees. Then the subtrees have at most 2^{h-1} leaves, so the whole tree has at most $2 \cdot 2^{h-1}=2^h$ leaves.

  That completes the proof. QED.

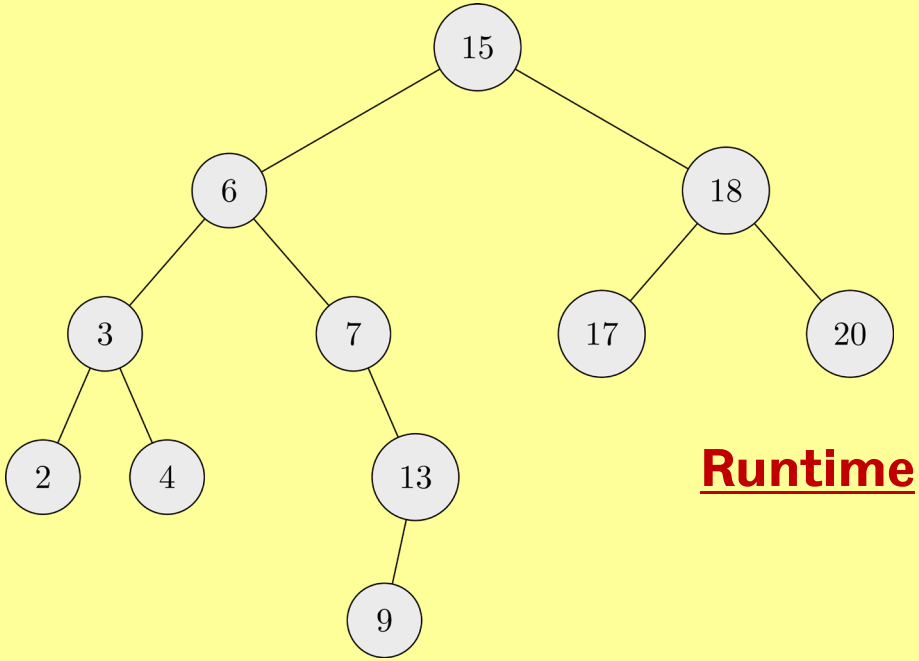
Binary search trees

- A binary search tree is a binary tree (BST) where all labels (keys) satisfy the **binary search tree property**:
  - If y is a node in the left subtree of x, then y.key $\leq$ x.key.
  - If y is a node in the right subtree of x, then y.key $\geq$ x.key.



Searching in a BST

- **Search(S, k):** returns element *x* with key *k*, or NIL if no such element exists
- **Idea**: compare against current key: stop or go down left or right.



ITERATIVE-TREE-SEARCH(x, k)

```
1  while x ≠ NIL and k ≠ x.key
2        if k < x.key
3              x = x.left
4        else  x = x.right
5  return x
```

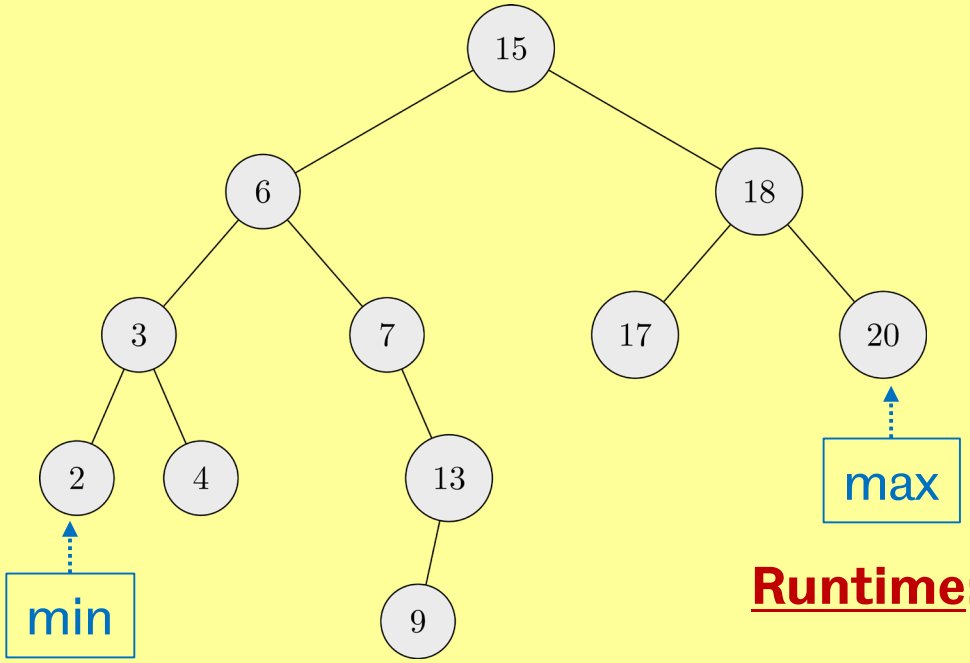
Runtime: *O(h), h* the height of the tree

Minimum, Maximum, Successor in a BST

Minimum: starting from the root, go left until the left child is NIL.

Maximum: starting from the root, go right until the right child is NIL.

Successor: minimum in right subtree (if it exists, e.g. Successor(15)=17) or lowest ancestor of *x* whose left child is also an ancestor of *x* (first larger ancestor, e.g. Successor(13)=15).



SUCCESSOR(x)

```
1: if x.right ≠ NIL then
2:         return MINIMUM(x.right)
3: y = x.parent
4: while y ≠ NIL and x == y.right do
5:         x = y
6:         y = y.parent
7: return  y
```

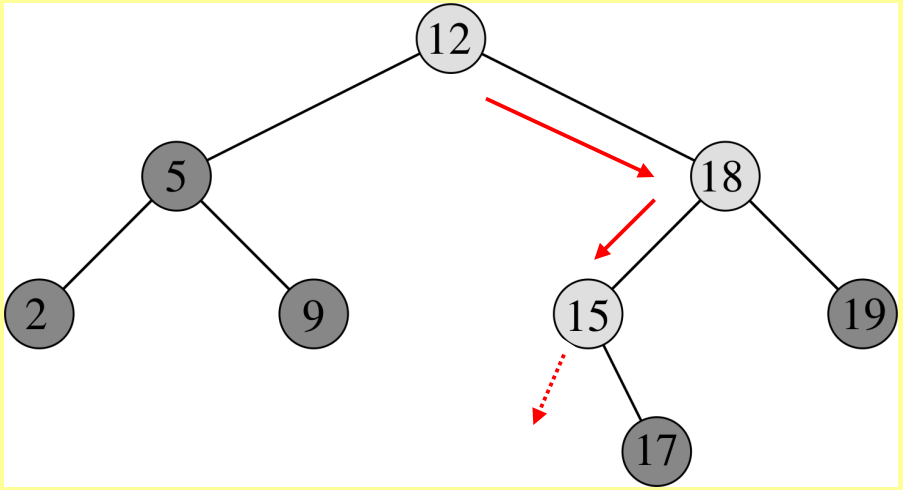
Runtime: *O(h)*, *h* the height of the tree

Insert(z)

Idea (pseudocode in the book)

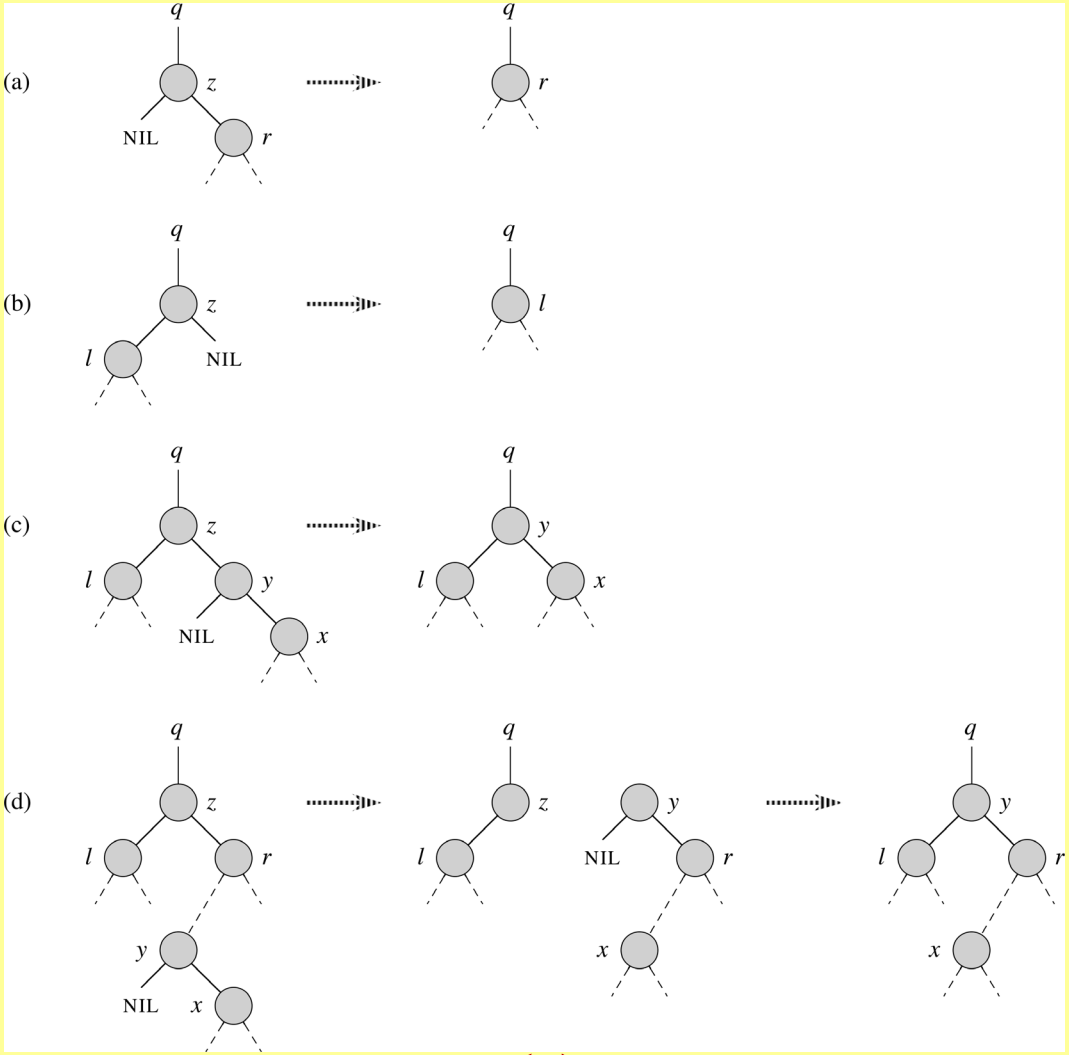
1. Go down the tree as in SEARCH to find where the new element needs to go.
2. The search will end in NIL, hence we traverse the **search path** (e.g. 12, 18, 15, NIL).
3. Add the element as a left or right subtree to last non-NIL node.

Example: Insert(13)



Runtime: $O(h)$, h the height of the tree

Delete(z)



(a) Node has no children or only a right child

(b) Node has only a left child

(c) Special case where right child is the successor.

(d) Successor *y* is the minimum in right subtree; *y*'s left child is NIL. Swap z and y.

Runtime: $O(h)$, h the height of the tree

Tree walks

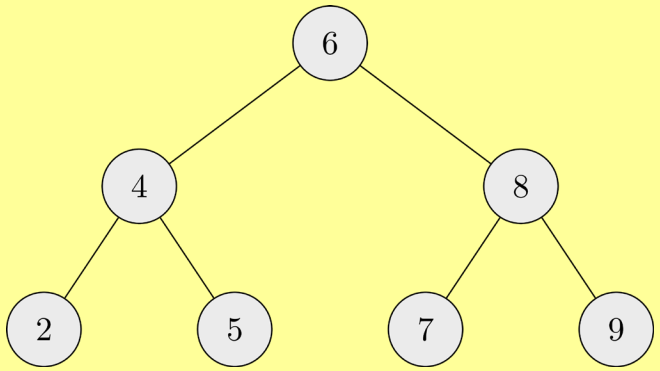
- We can print out the keys of a BST by a **tree walk**:

```
PREORDER(x)
1: if x ≠ NIL then
2:     print x.key
3:     PREORDER(x.left)
4:     PREORDER(x.right)
```

```
INORDER(x)
1: if x ≠ NIL then
2:     INORDER(x.left)
3:     print x.key
4:     INORDER(x.right)
```

```
POSTORDER(x)
1: if x ≠ NIL then
2:     POSTORDER(x.left)
3:     POSTORDER(x.right)
4:     print x.key
```

- Inorder tree walk outputs sorted sequence.



Inorder: 2, 4, 5, 6, 7, 8, 9
Preorder: 6, 4, 2, 5, 8, 7, 9
Postorder: 2, 5, 4, 7, 9, 8, 6

Tree walks: runtime

Theorem: Inorder (Preorder/Postorder) tree walk of the root of an n-node tree takes time $\Theta(n)$.

```
PREORDER(x)
1: if x ≠ NIL then
2:     print x.key
3:     PREORDER(x.left)
4:     PREORDER(x.right)
```

```
INORDER(x)
1: if x ≠ NIL then
2:     INORDER(x.left)
3:     print x.key
4:     INORDER(x.right)
```

```
POSTORDER(x)
1: if x ≠ NIL then
2:     POSTORDER(x.left)
3:     POSTORDER(x.right)
4:     print x.key
```

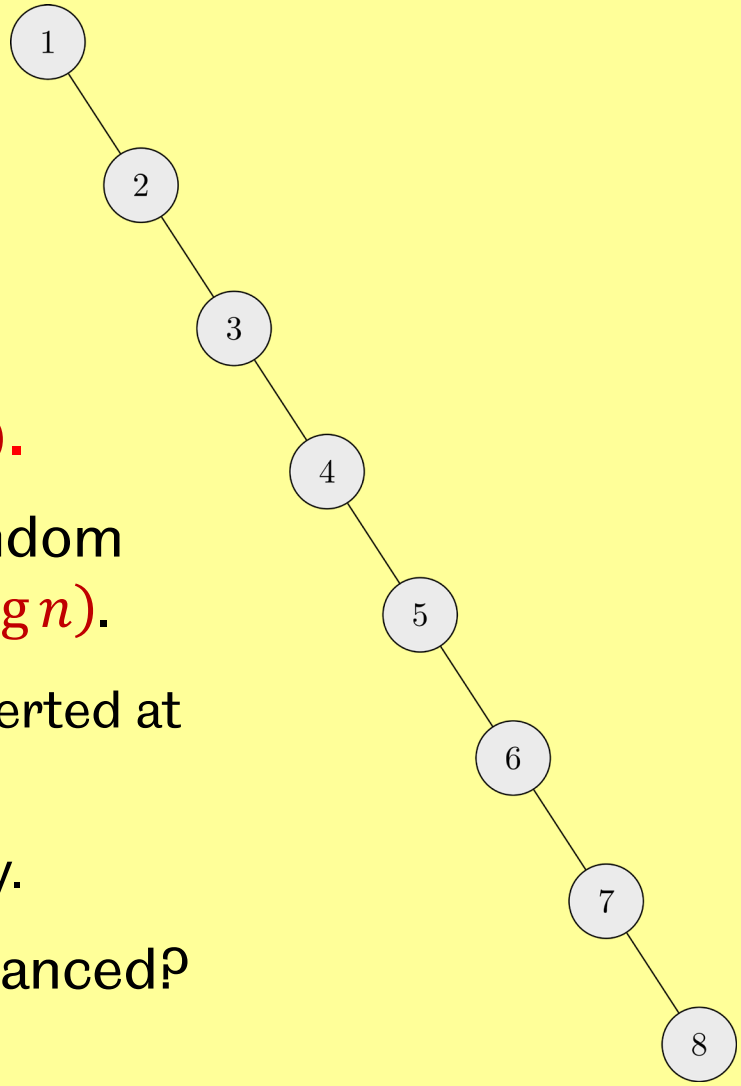
- Book gives a rather dull proof based on recurrences.
- A simpler proof:
  - Assign costs (time) for operations made at x to node *x*.
  - Cost at each node is $\Theta(1)$, and all costs are accounted for.
  - Sum of costs = runtime is $n \cdot \Theta(1) = \Theta(n)$.
- NB: This kind of argument is called **accounting method**.

Summary so far

- Binary trees have at most 2 children and can be defined recursively:
  - A tree is either empty or it contains a root and two subtrees (=trees).
  - Very useful for inductive proofs for trees.
- Binary search trees store data such that
  - smaller keys are in the left subtree
  - larger keys are in the right subtree.
- Inorder/preorder/postorder walks output all elements in time $\Theta(n)$.
- BSTs of height h execute the following operations in time $O(h)$
  - Searching, Minimum, Maximum, Successor, Insertion, Deletion
- But is O(h) good or bad? It depends on how the height h is related to the number of nodes n.

How good is O(*h*)?

- BSTs can be unbalanced and even degenerate to a single path
  - Height can be as bad as *n-1*
  - So the worst-case runtime is $\theta(n)$.
- If keys are inserted in uniform random order, the expected height is $O(\log n)$.
  - Can we rely on our data being inserted at random?
  - Such inputs might be very unlikely.
- Can we ensure that the tree is balanced?



Balanced trees

- Intuitively: the left and right subtrees are roughly the same height
- Let *v* be a node and T_l, T_r be its left and right subtrees, respectively. Then $bal(v) := h(T_l) - h(T_r)$ is the **balance factor** of *v*, where $h(\,)$ denotes the height of a tree.
- A balanced tree is one where balance factors for all nodes are close to zero.

Self-balancing trees

- Whether a tree ends up balanced or not depends entirely how you decide to insert/delete values. Different approaches give rise to different classes of tree.
- There are many types of binary search trees that are guaranteed to have depth $O(\log n)$.
  - AVL Trees
  - 2-3 Trees
  - B-Trees
  - Red-Black Trees, Splay Trees, Van Emde Boas Trees
- We'll look at one of these: AVL trees

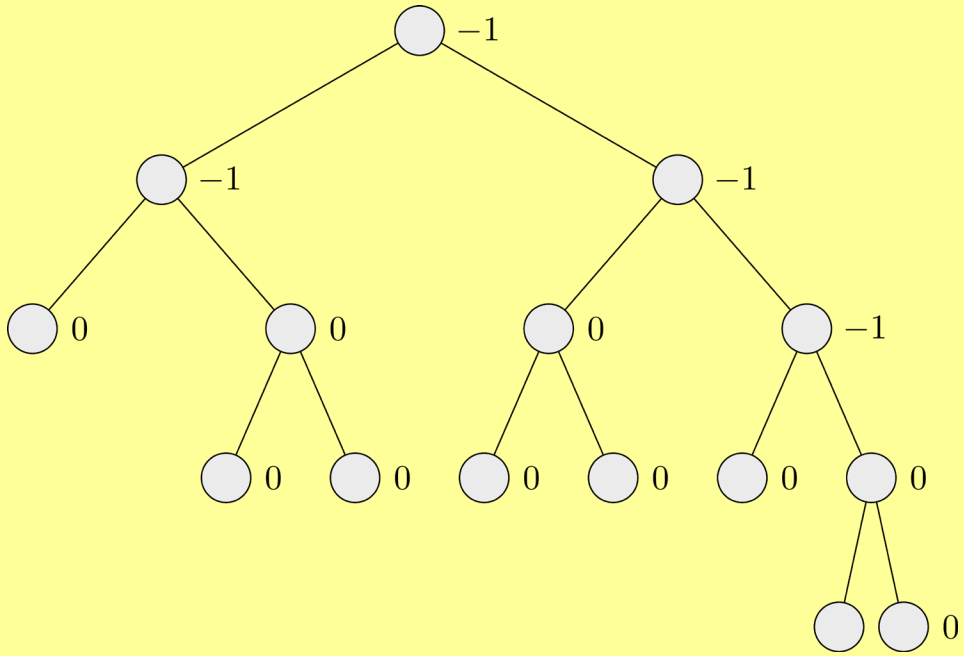
AVL Trees

- Invented by and named after Adelson-Velskii and Landis.
- A binary tree is called an AVL tree if all the nodes are locally balanced, i.e. for every node the following holds:
  - the height of the left subtree and the height of the right subtree only differ by at most 1.
  - Or in other words, for every node *v* we have $bal(v) \in \{-1, 0, +1\}$
- The insertion and deletion functions are specially designed to ensure that all nodes remain locally balanced when we add or remove data from an AVL tree.
  - "being locally balanced" is an invariant property of AVL trees

Balance properties

- Beware: AVL trees can still be a bit lopsided. For example:

The numbers show the balance factors at each node



- However, overall the tree is still pretty balanced.

Estimating the depth of an AVL tree

Theorem: the height of an AVL tree with n nodes is at most

$$h \leq \frac{1}{\log((\sqrt{5}+1)/2)} \log n \approx 1.44 \log n.$$

- This is only around 44% deeper than a perfectly balanced tree.

Proof outline:

- Consider the minimum number of nodes in any AVL tree of height h and call it $A(h)$.
  - This means that any AVL tree of height h will have $n \geq A(h)$ nodes.
- Show that $A(h)$ (and thus n) is exponentially large in h.
  - We'll show that $A(h)$ is similar to the Fibonacci numbers.
- Take logarithms (+maths) to get the claimed bound.

Minimum number of nodes in an AVL tree

- Let $A(h)$ be the minimum number of nodes in any AVL tree of height h.
  - An AVL tree with height 0 consists of the root only, hence $A(0) = 1$.
  - The smallest AVL tree of height 1 has two nodes, hence $A(1) = 2$.
  - An AVL tree of height h has to have a root with one subtree of height $h - 1$, and the other subtree of height at least $h - 2$. Hence $A(h) = 1 + A(h - 1) + A(h - 2)$.
- This is similar to the Fibonacci numbers (apart from the "$1 +$"):
  - $Fib(0) = Fib(1) = 1$ and
  - $Fib(h) = Fib(h - 1) + Fib(h - 2)$.

Link to Fibonacci numbers

We prove by induction that $A(h) = Fib(h+2) - 1$.

Base cases

$$A(0) = 1 = 2 - 1 = Fib(2) - 1$$

$$A(1) = 2 = 3 - 1 = Fib(3) - 1.$$

Step case

Assume that the claim holds for $\mathrm{A}(h-1)$ and $\mathrm{A}(h-2)$. Then

$$\begin{aligned} A(h) &= 1 + A(h-1) + A(h-2) && \text{(by recurrence)} \\ &= 1 + \mathrm{Fib}(h+1) - 1 + \mathrm{Fib}(h) - 1 && \text{(2x induction hypothesis)} \\ &= \mathrm{Fib}(h+1) + \mathrm{Fib}(h) - 1 \\ &= \mathrm{Fib}(h+2) - 1 && \text{(by definition of Fib}(h+2)\text{).} \end{aligned}$$

QED

Proof of the theorem

- We've shown that $A(h) = Fib(h+2) - 1$.
  - So, every AVL tree with n nodes and height h has

$$n \geq A(h) = Fib(h+2) - 1.$$

- Recall: $Fib(k) \geq \frac{1}{\sqrt{5}}[\gamma^{k+1} - 1]$ where $\gamma = \frac{\sqrt{5}+1}{2}$
  - So: $n \geq \frac{1}{\sqrt{5}}[\gamma^{h+3} - 1] - 1$
- Rearranging this gives $\gamma^{h+3} \leq \sqrt{5}n + 1 + \sqrt{5}$
- Finally, taking logs gives:

$$(h+3)\log\gamma \leq \log(\sqrt{5}n + 1 + \sqrt{5})$$

Some mathematical magic…

- We've just seen that $(h+3)\log\gamma \leq \log(\sqrt{5}n+1+\sqrt{5})$
- We'll simplify the expression on the right:
  - General formula: $\log(a+c) = \log(a) + \log(1+{}^{c}/_{a})$. So:
  - $\log(\sqrt{5}n+1+\sqrt{5}) = \log(\sqrt{5}n) + \log\left(1+\frac{1+\sqrt{5}}{\sqrt{5}n}\right)$

$$\leq \log(\sqrt{5}n) + \log(1+\tfrac{\gamma}{\sqrt{5}}) \quad \textbf{for } \boldsymbol{n \geq 2}$$

$$= \log n + \log(\sqrt{5}+\gamma) = \log n + 1.95$$

$$< \log n + 3\log\gamma \qquad \text{(because } \log\gamma \approx 0.69\text{)}$$

- Thus: $(h+3)\log\gamma < \log n + 3\log\gamma$ when $n \geq 2$.

Finishing the proof

- We've shown that:

$$(h+3)\log\gamma < \log n + 3\log\gamma \qquad \text{if} \quad n \geq 2$$

- It follows that $h < \frac{\log n}{\log \gamma}$ when $n \geq 2$.

- Finally, we just need to check the formula for $n = 1$. In this case the tree is an isolated root, so $h = 0 = \log n$. So we have $h = \frac{\log n}{\log \gamma}$ when $n = 1$.

- Therefore: $h \leq \frac{\log n}{\log \gamma} \approx 1.44 \log n$ for all $n \geq 1$.

QED

Search in an AVL Tree

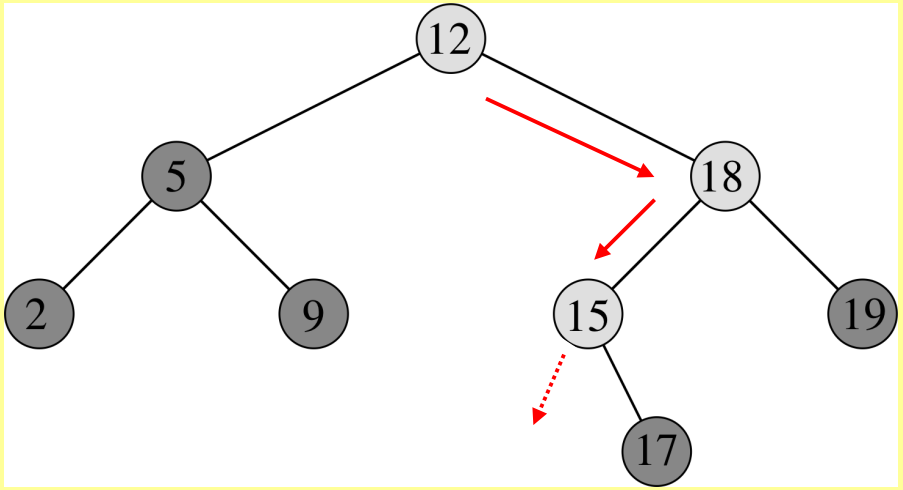
- An AVL tree is a binary search tree, so
  - We can use the same search technique as before
  - Gives runtime = O(h)
- The key here is not that we're searching in a different way, but that we can guarantee that *h* is reasonably small.

Our original Insert(z) algorithm

Idea (pseudocode in the book)

1. Go down the tree as in SEARCH to find where the new element needs to go.
2. The search will end in NIL, hence we traverse the **search path** (12, 18, 15, NIL).
3. Add the element as a left or right subtree to last non-NIL node.

Example: Insert(13)

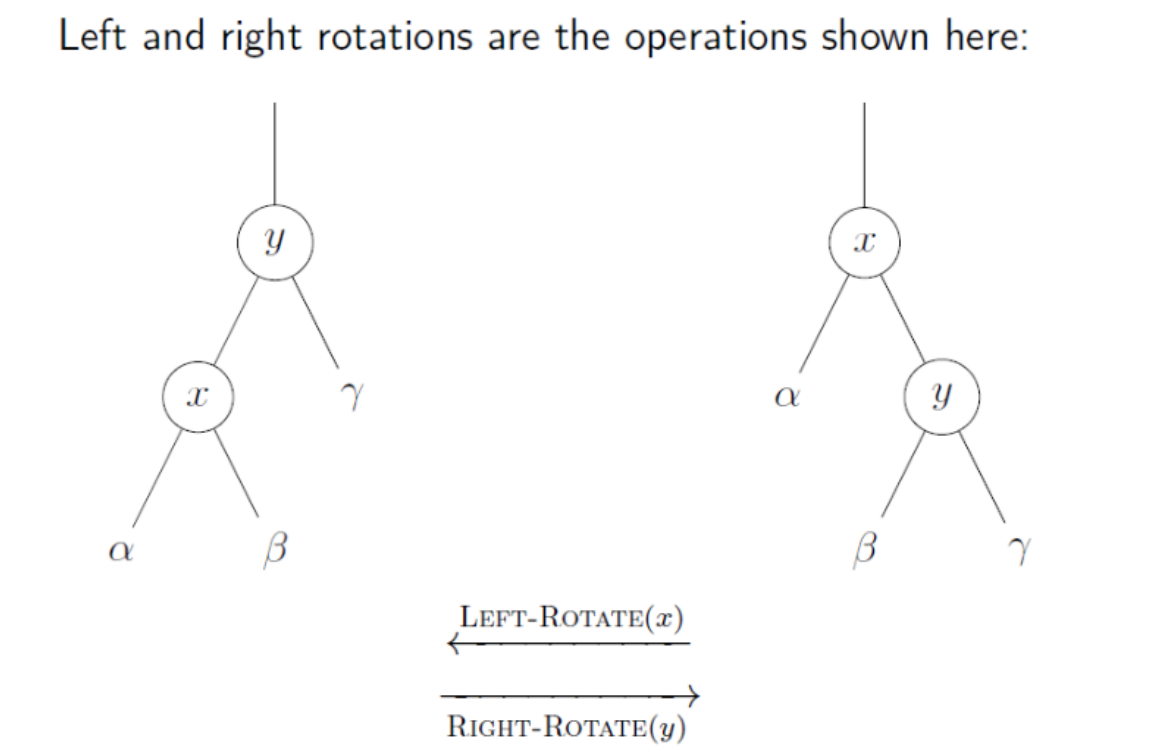


Runtime: $O(h)$, h the height of the tree

Inserting values into an AVL Tree

- Works like in an ordinary binary search tree.
- But the tree may become unbalanced, hence we need to **rebalance**.
  - Remember: the balance factor at each node is initially equal to 1, 0 or -1
  - We'll show how to rebalance a node using rotations.

Rotations



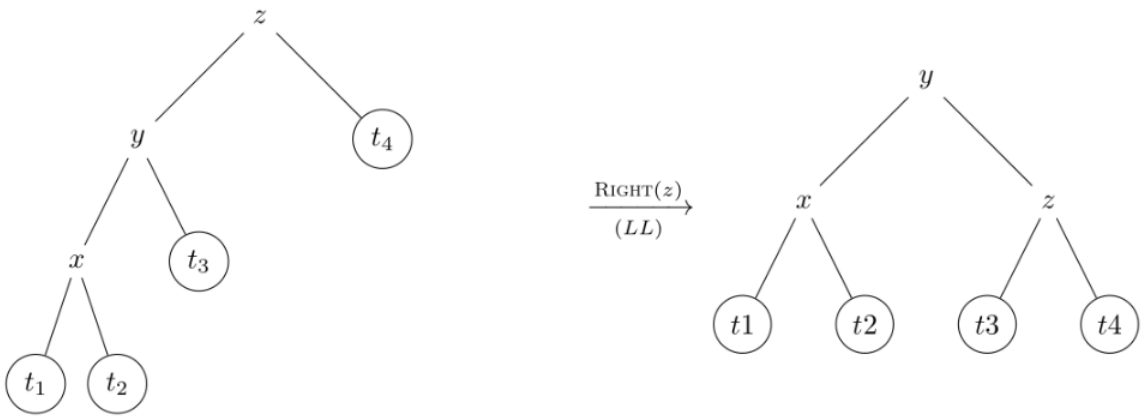
Both run in constant time

Which nodes need rebalancing?

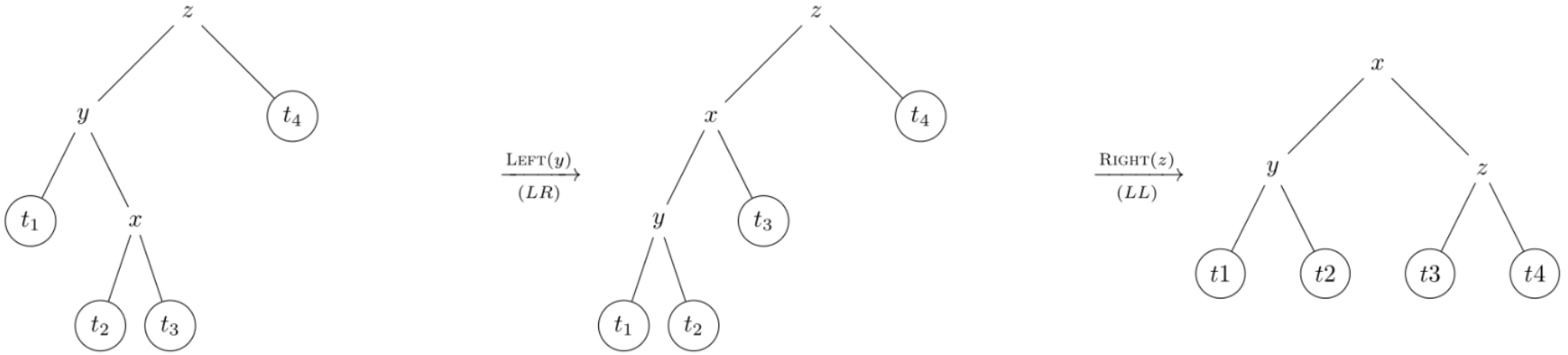
- We record the **search path** to the new element, and then **work back up the search path** rebalancing the nodes.
  - Only nodes on the search path are relevant
  - Only needed if the height of the current subtree has increased
  - The newly inserted value is a leaf, so it's already balanced
  - If a node's only children are leaves, it's also already balanced.
- Therefore: we only need to worry about nodes on the search path that have grandchildren
  - Look at the routes to the relevant grandchildren
  - We can go Left or Right at each step down the search path

If we go left first...

LL: Rotate right

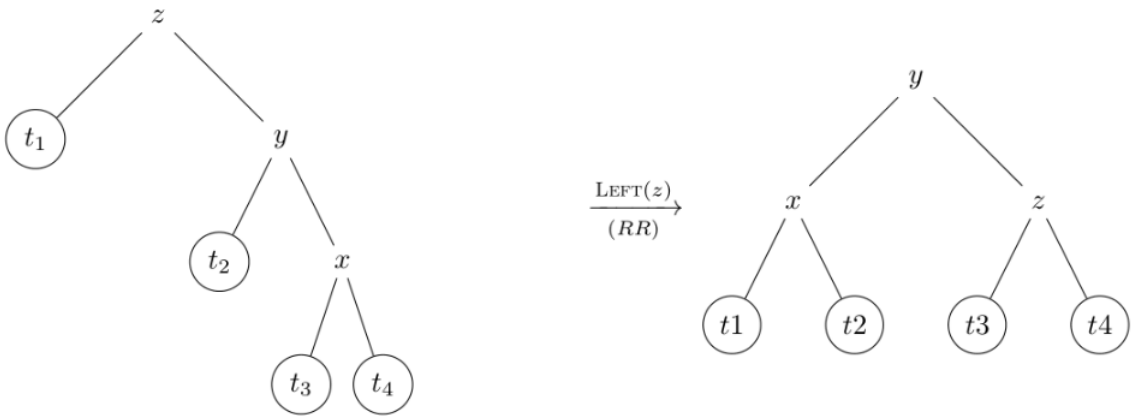


LR: Rotate left then right

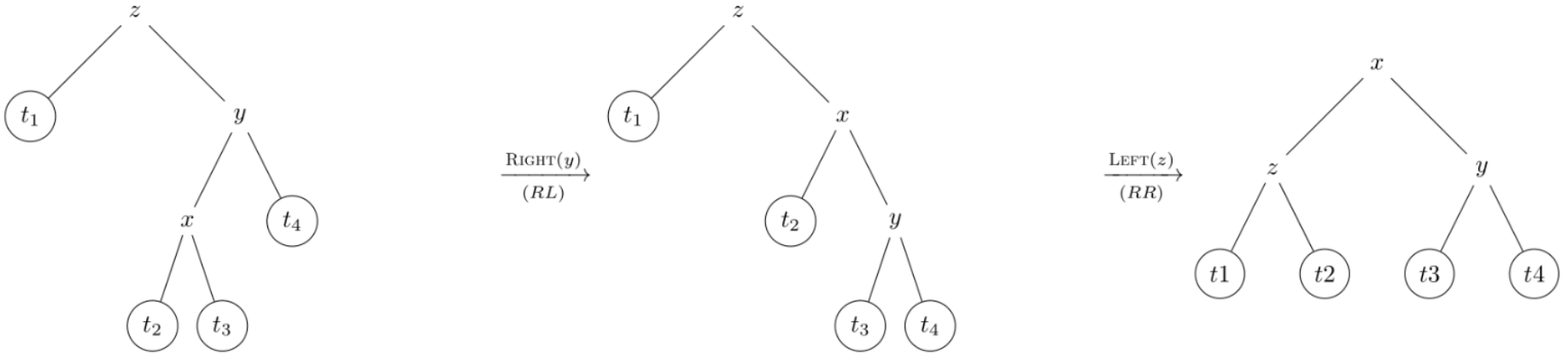


If we go right first

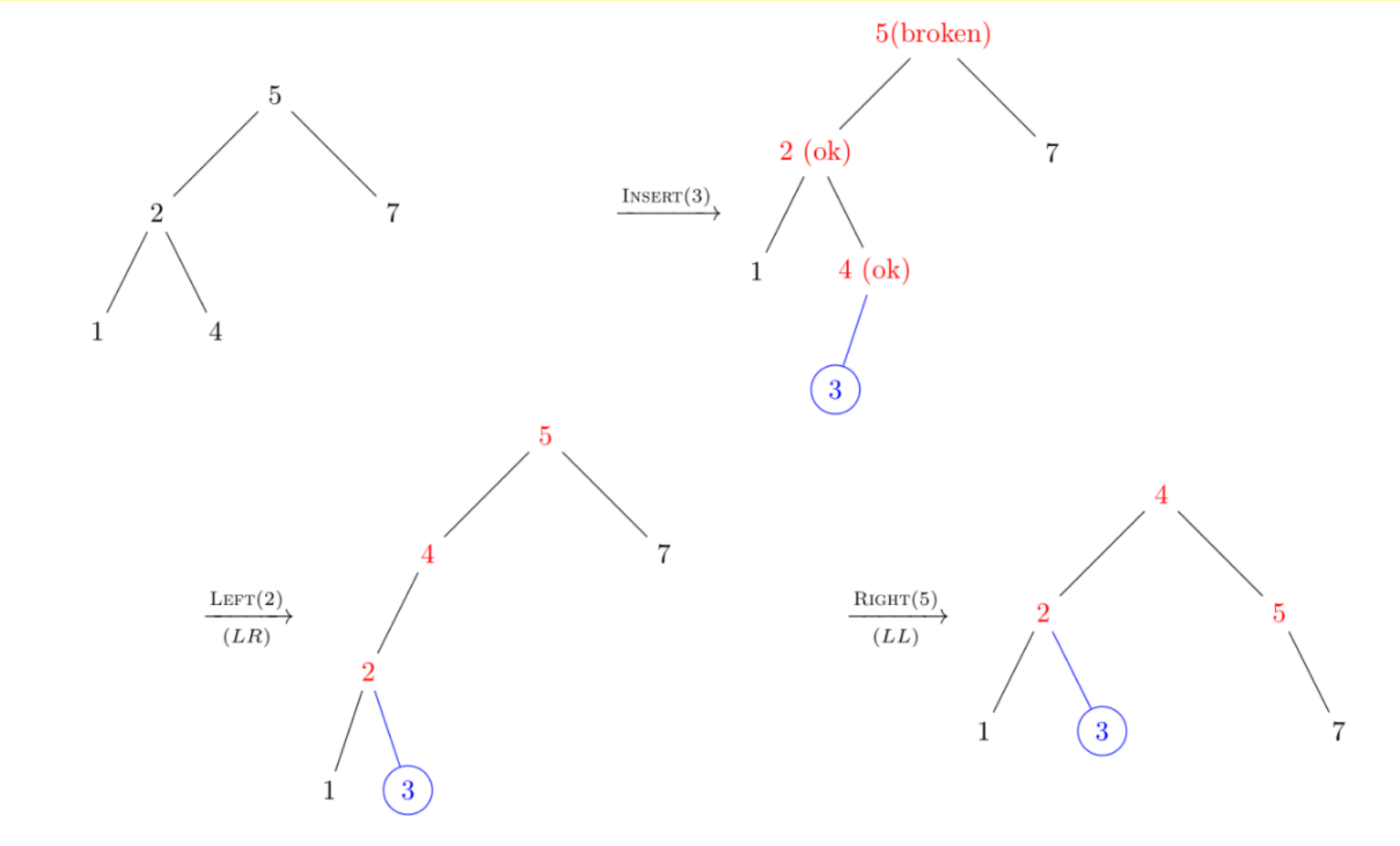
RR: Rotate left



RL: Rotate right then left



Insertion example

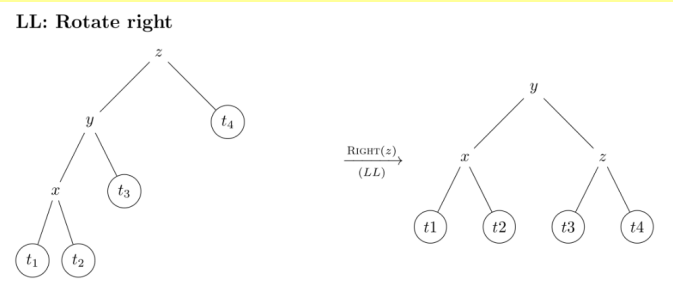


Does rebalancing work?

EXERCISE

Prove that the tree is correctly balanced again after these rebalancing rules have been applied.

You only need to consider cases LL and LR. The other two follow "by symmetry".



See the online notes by Dirk Sudholt

Getting started

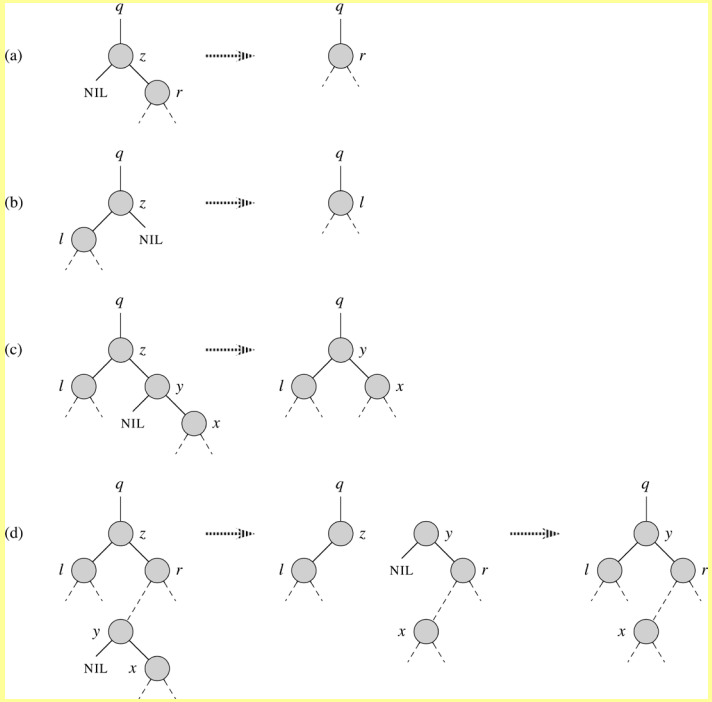
Case LL: *By assumption all the relevant nodes below z are ok, so we can say something about how the heights of subtrees were related before insertion happened. Suppose the value was inserted into t1 (insertion in t2 is analogous). Since the tree is now unbalanced at z, the following must have happened during insertion ... Therefore ..., and hence ...*

Runtime of Insert

- Inserting an element takes time $O(h)$.
- Rebalancing:
  - Moving up the search path takes us through $O(h)$ nodes
  - At each node we do at most 2 rotations
  - Each rotation takes $O(1)$ time
  - Therefore: total time to rebalance is also $O(h)$
- Total runtime of Insert: $O(h) = O(\log n)$.

Deleting in an AVL Tree

- As with Insert, we work backwards up the search path to rebalance (only needed if the height of the current subtree has decreased). Exercise: Prove this works.



(a) We know r was balanced before deletion happened, therefore ...

(b) We know l was balanced ...

(c) We know various subtrees were balanced, therefore ...

(d) ...

See the online notes by Dirk Sudholt

Summary

- AVL trees with n elements have height $O(\log n)$.
- AVL trees with n nodes execute the following operations in time $O(\log n)$
  - **Searching, Minimum, Maximum, Successor**
    - This is because AVL trees are binary search trees whose height is always $h = O(\log n)$.
  - **Insertion**, **Deletion**
    - Extra work may be needed to rebalance the tree, but it can be done quickly