CS 161: Design and Analysis of Algorithms

Mark Zhandry

Announcements

- Office Hours:
  - Mark: Wednesday 12-2PM in Gates 494
  - Tarun: Monday, Thursday 4-6PM in Gates B24A
  - Jun: Wednesday 4-8PM in Gates B24B
- Updated homework policies
  - SCPD students only: scan of written work is okay
- Updated problem 8 on HW 1

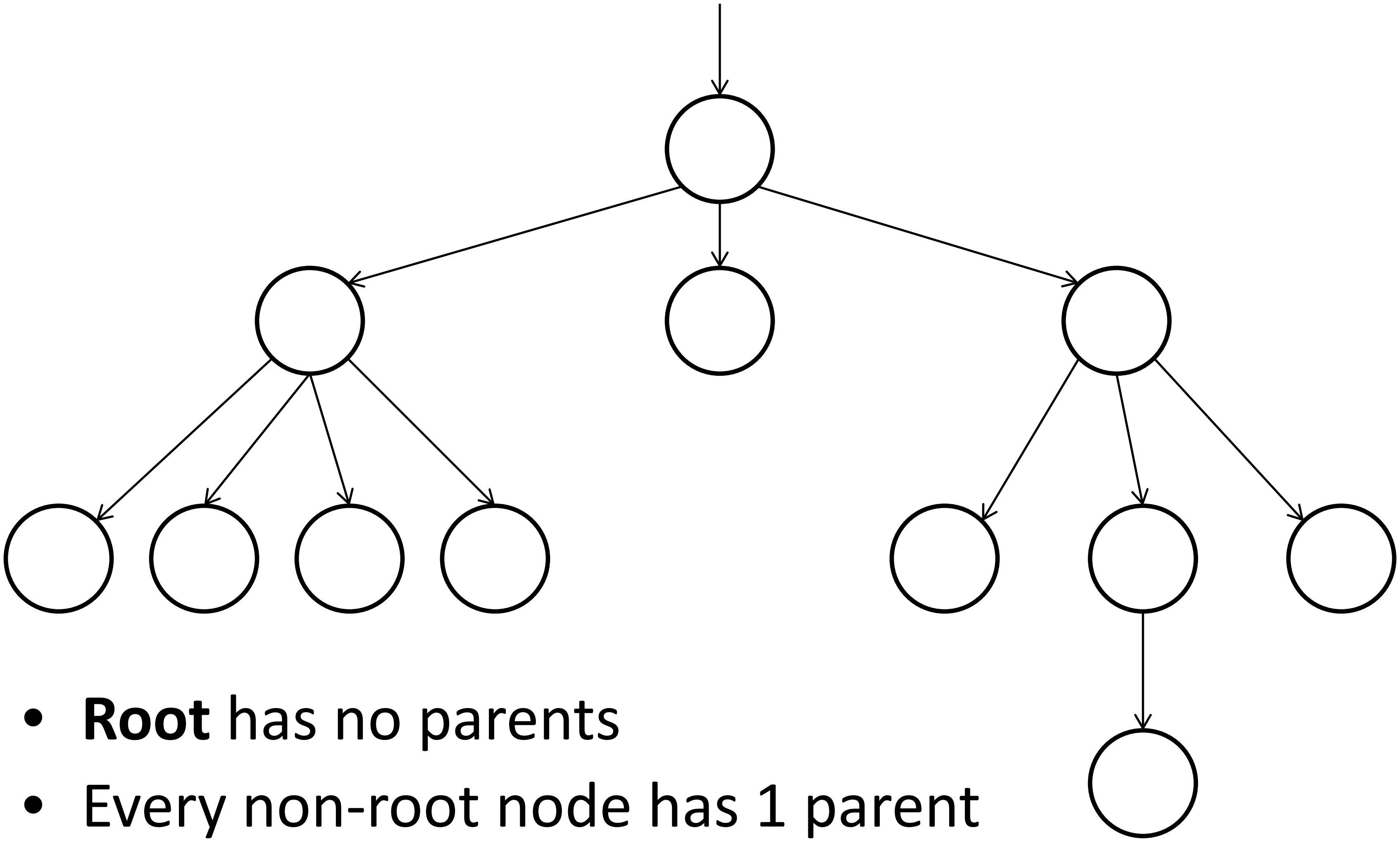
Data Structures 2:
Storing Ordered Data

- Binary Search Trees
- Self-Balancing Trees
- Heaps

The Goal

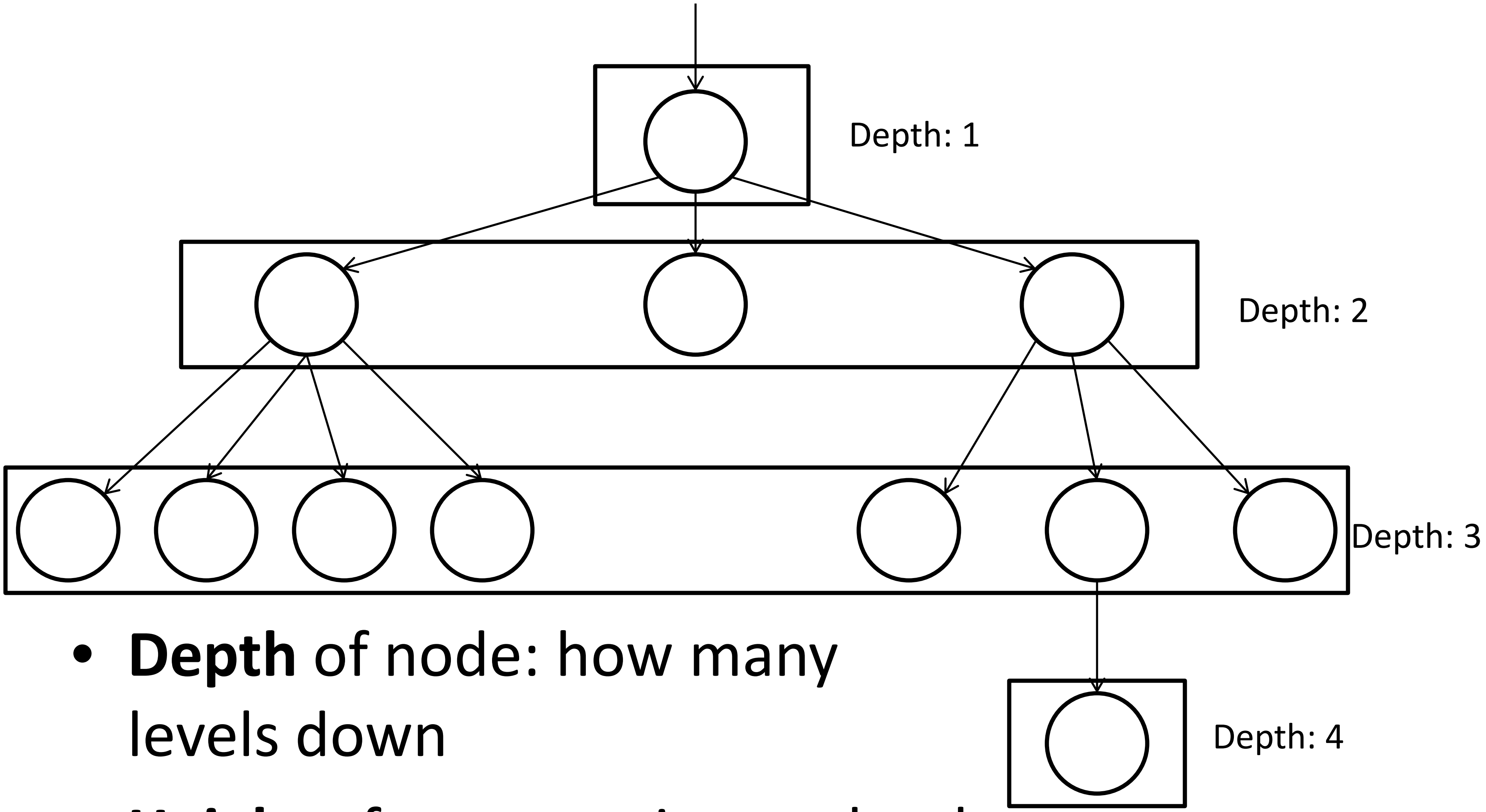
- We would like to store data that is ordered (i.e. we can tell if x < y)
- We want to answer questions relating to the order:
  - What is the maximum?
  - What is the minimum?
  - What are all the elements, in order?

Trees



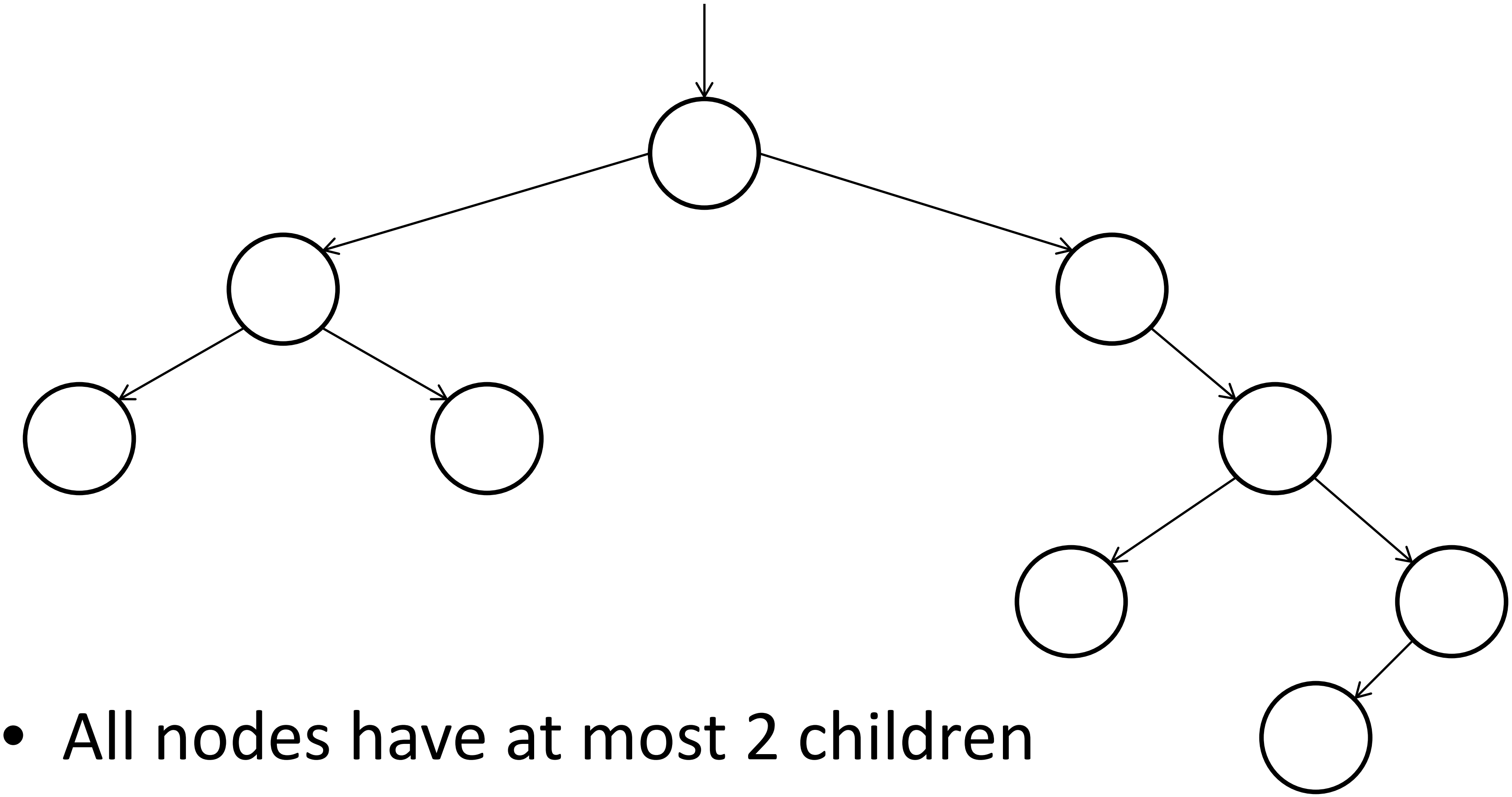
- **Root** has no parents
- Every non-root node has 1 parent
- Nodes without children: **leaf nodes**

Trees



- **Depth** of node: how many levels down
- **Height** of tree: maximum depth

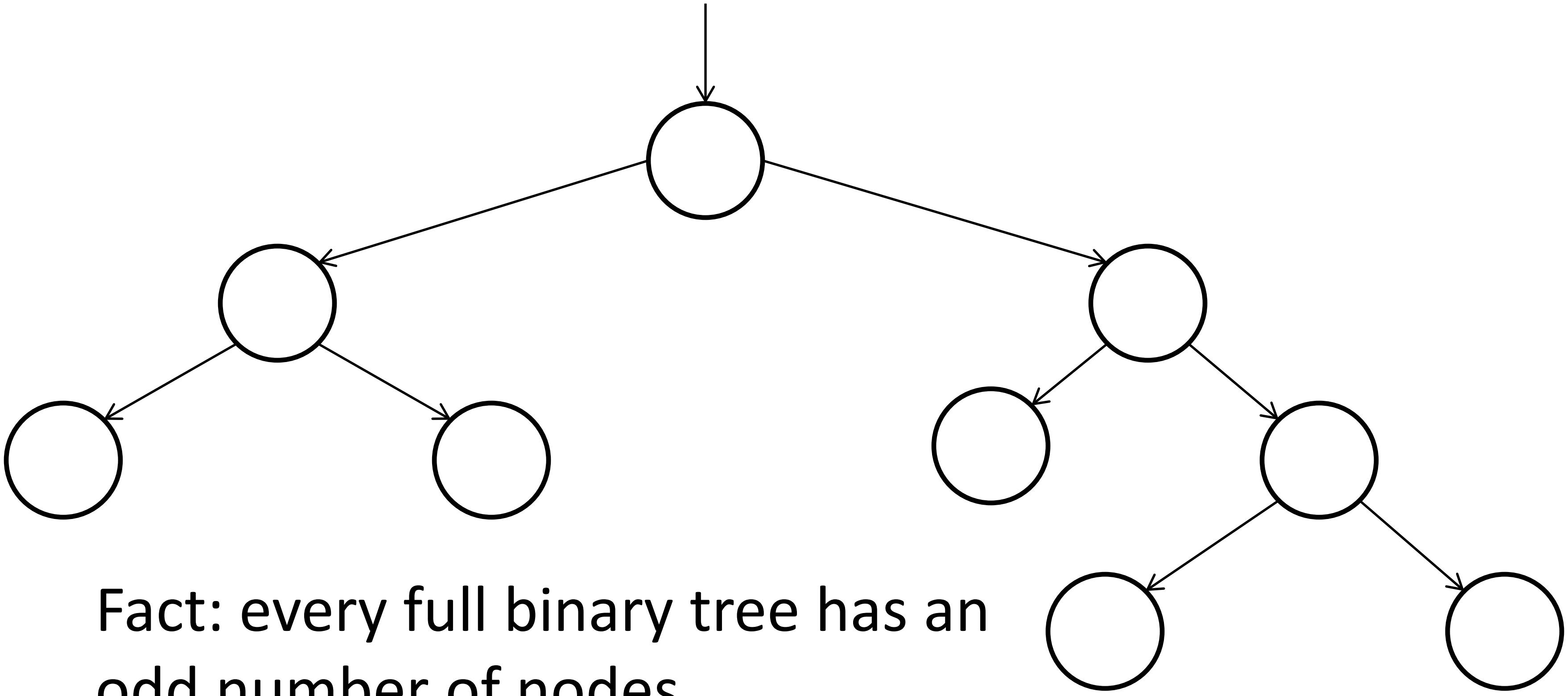
Binary Trees



- All nodes have at most 2 children

Types of Binary Trees

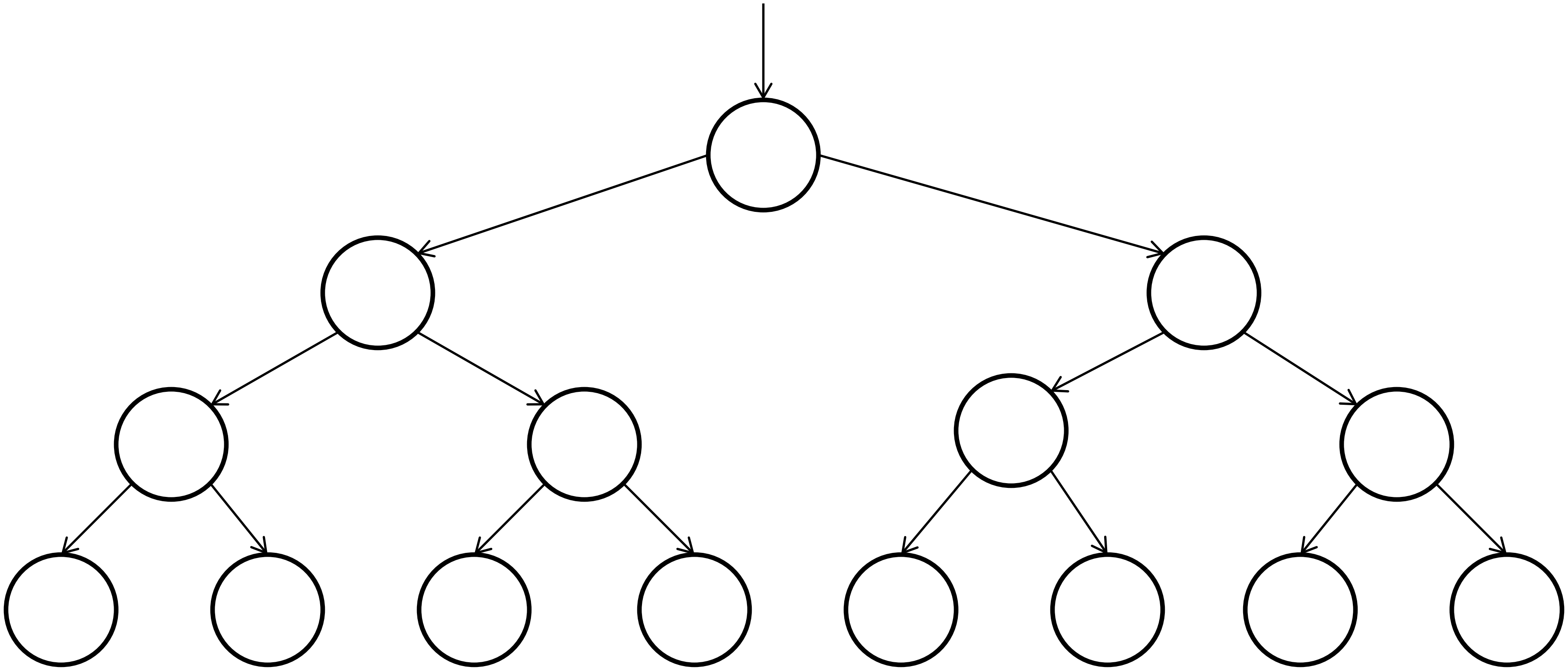
- **Full**: All non-leaf nodes have 2 children



Fact: every full binary tree has an odd number of nodes

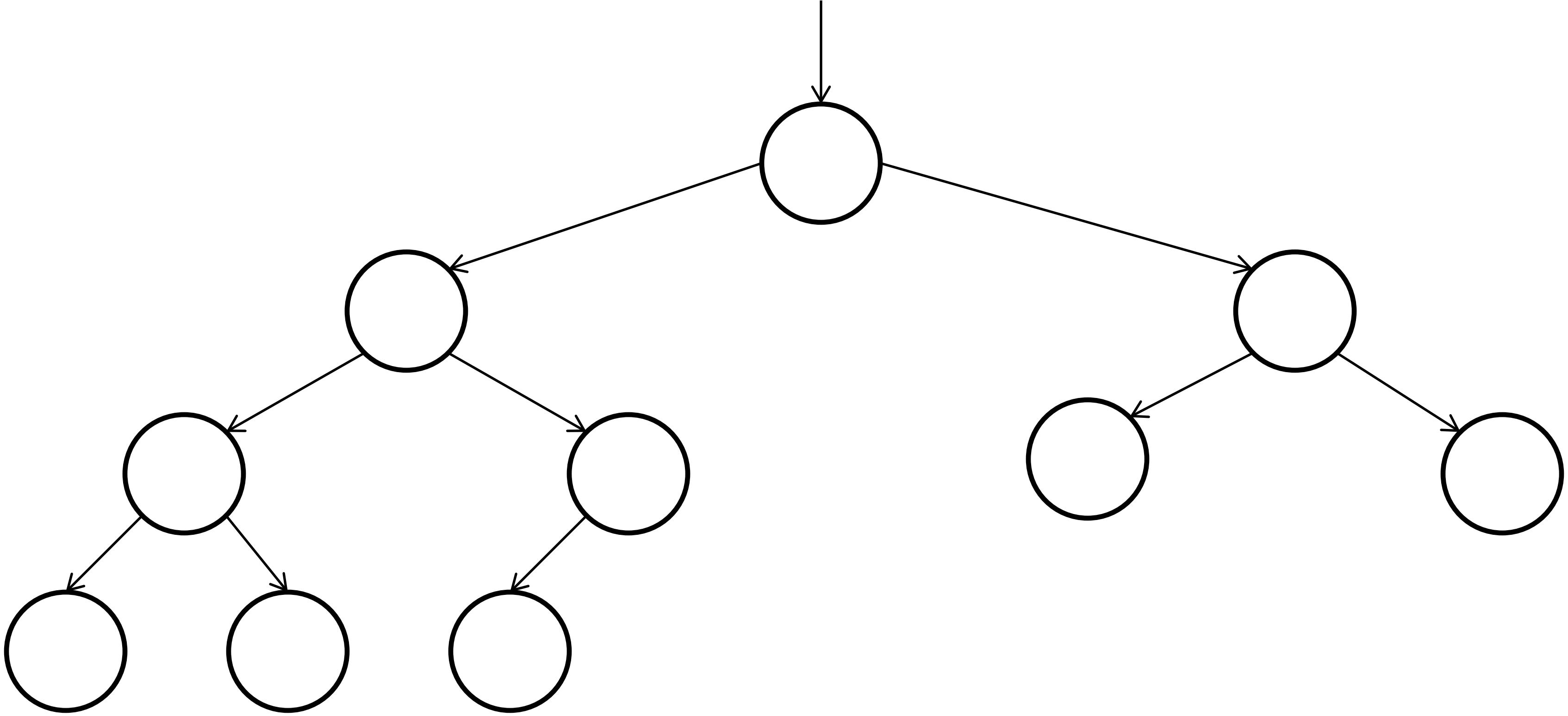
Types of Binary Trees

- **Perfect**: Full, all leaf nodes at same level



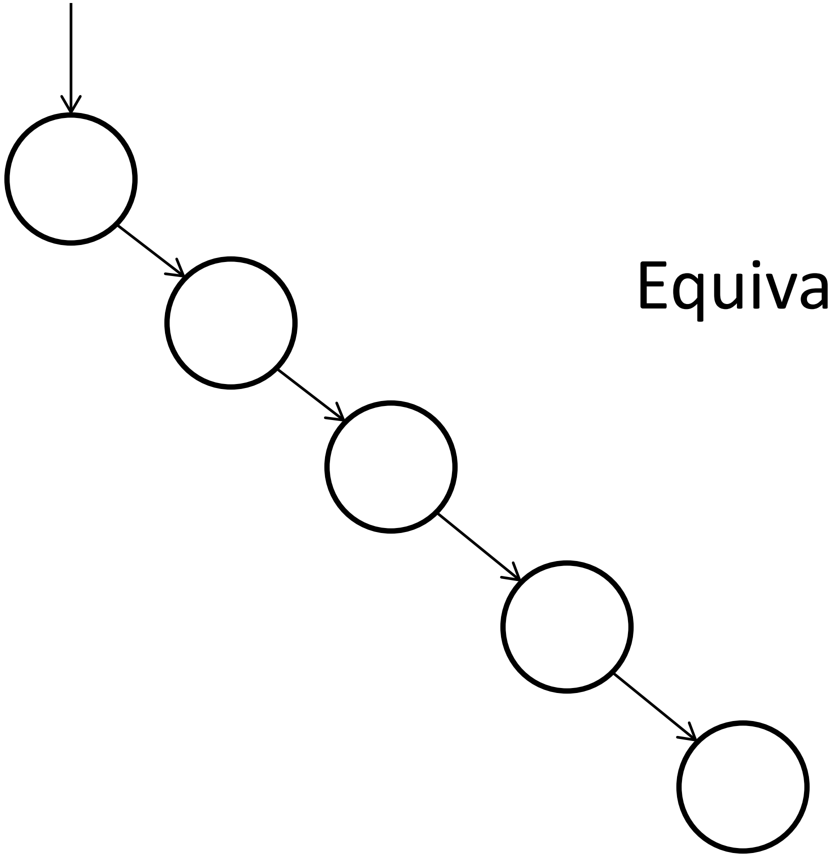
Types of Binary Trees

- **Complete**: all levels completely full, except possibly last. All nodes as far left as possible



Types of Binary Trees

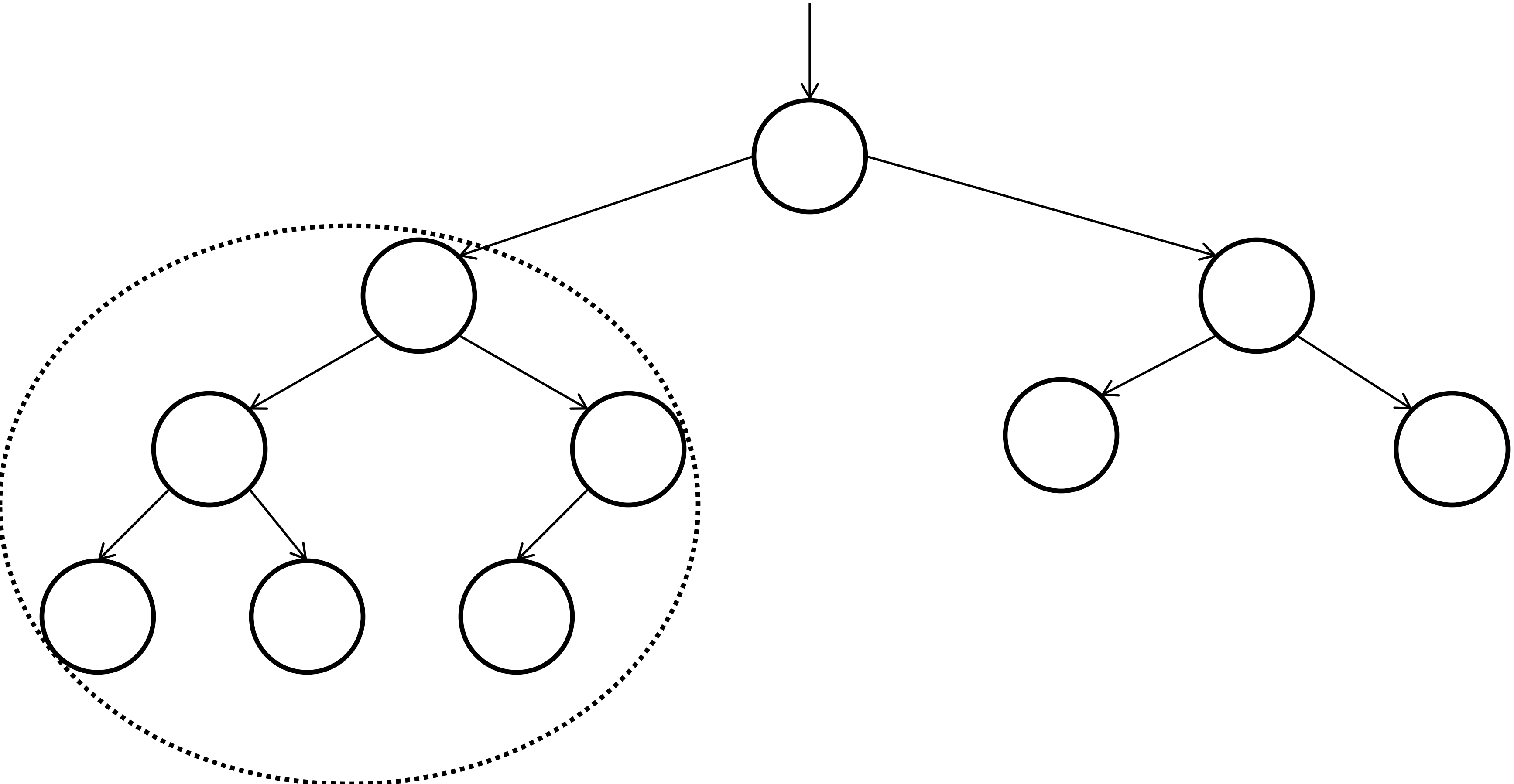
- **Degenerate**: All nodes have at most 1 child



Equivalent to linked list

Subtrees

- **Subtree**: tree formed by looking at descendant of a node



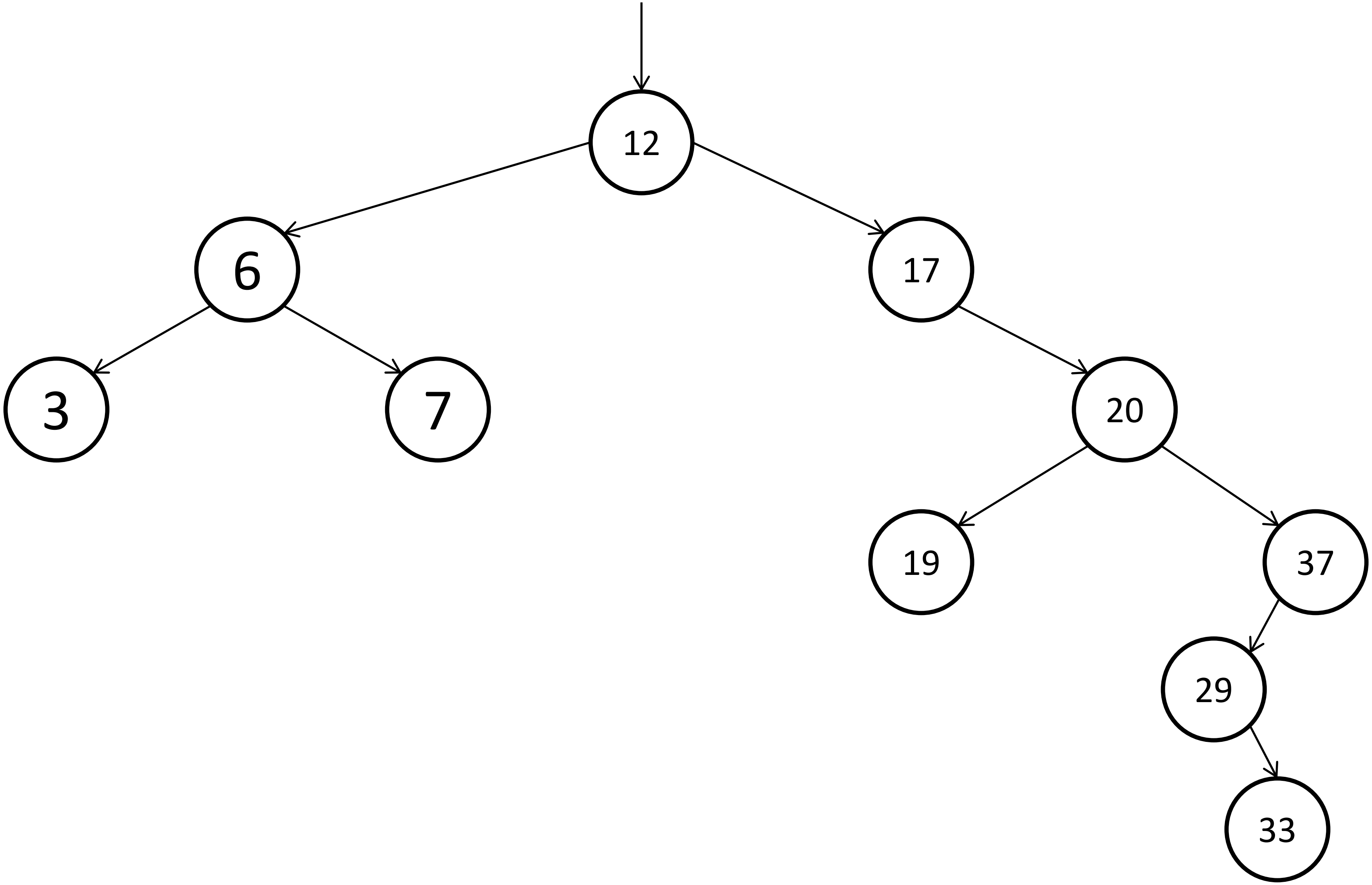
Subtree Facts

- Every subtree of a full binary tree is full
- Every subtree of a perfect binary tree is perfect
- Every subtree of a complete binary tree is complete
- Every subtree of a degenerate tree is degenerate

Binary Search Trees (BSTs)

- Every node stores some value.
- For each node with value v, the values stored in the left subtree are smaller than v, and the values stored in the right subtree are at least as large as v

Binary Search Trees (BSTs)



Traversing BST

- Simple algorithm to list all values in order
- traverse(root) =
  - If left left child L exists, traverse(L)
  - Output root
  - If right child R exists, traverse(R)
- Called inorder traversal
  - Opposed to preorder and postorder

Traversing a BST

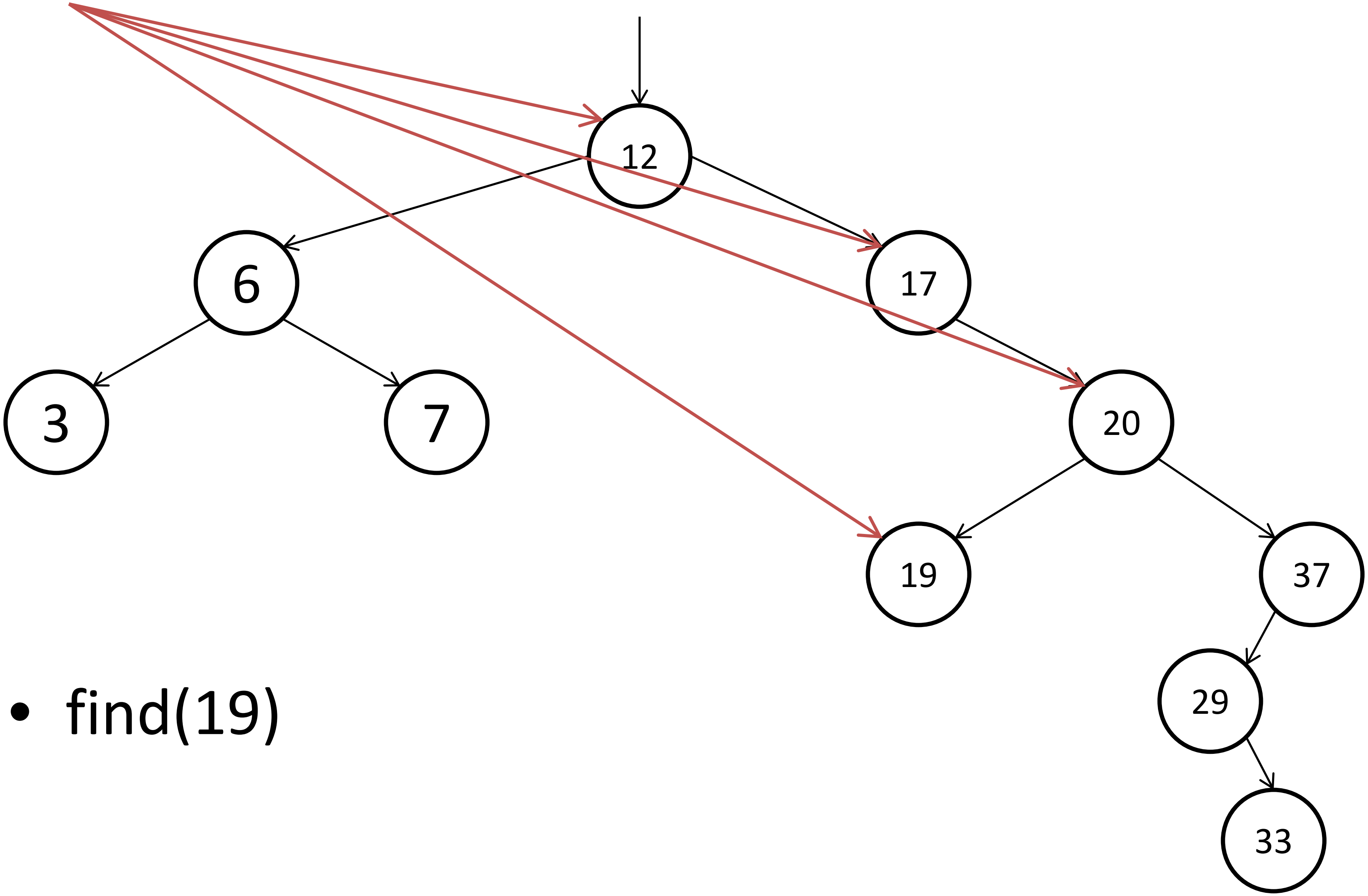
- Running time?
- Constant work per call to traverse
- Call traverse once on each node
- O(|V|) for entire traversal

Searching a BST

- Given a value v, and a root r, find v in the tree rooted at r
- search(r,v) = {
  - If r has value v, return r
  - If v < r and left child L exists, call search(L,v)
  - If v > r and right child R exists, call search(R,v)
  - Otherwise, report that v isn’t found

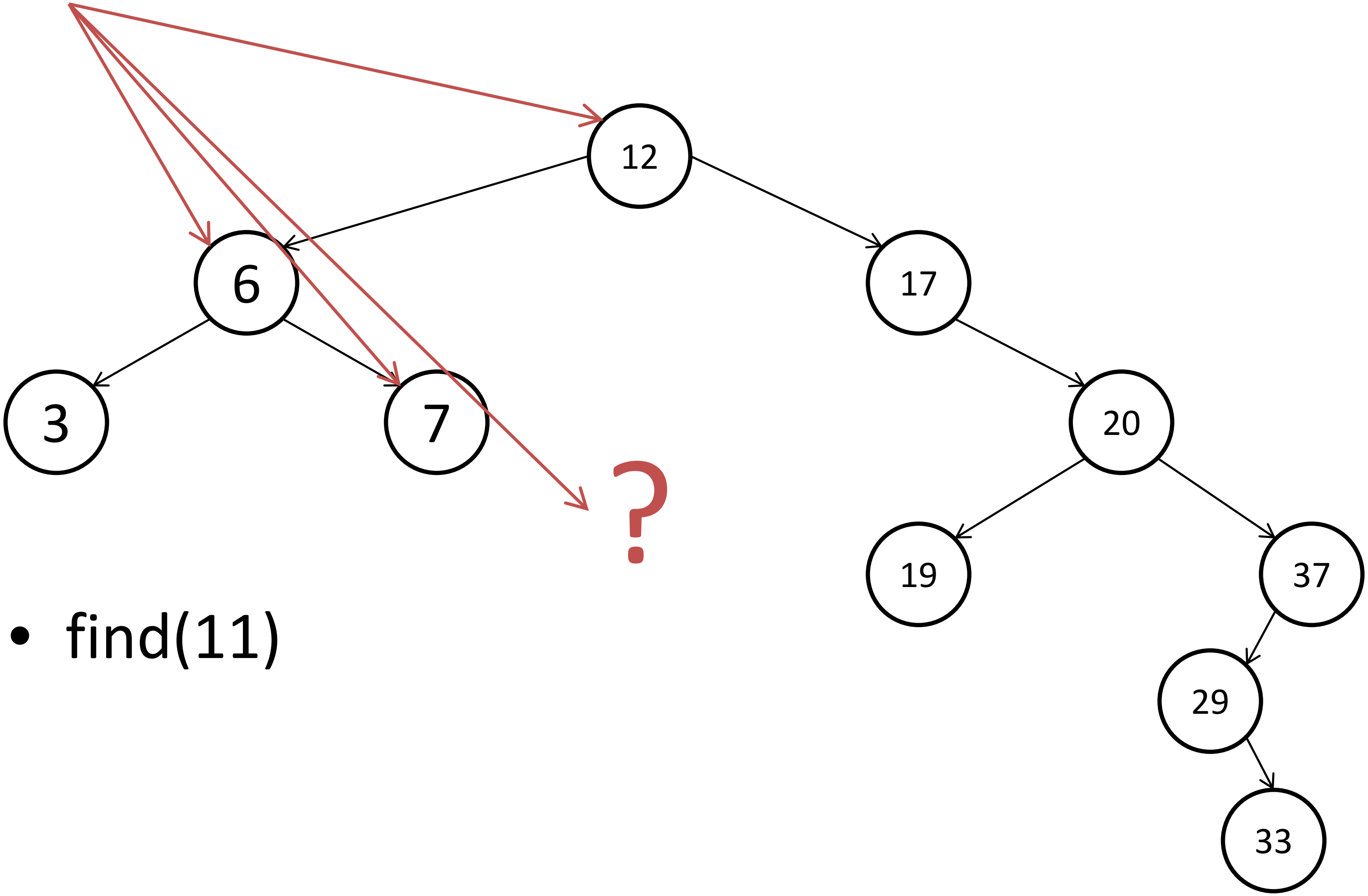
  }

Searching a BST



- find(19)

Searching a BST



- find(11)

Time to Search

- Constant number of operations per call to search.
- If we find v, we make one call per ancestor of v
- If we do not find v, the number of calls is at most the height of the tree
- Time to search: O(h) where h is height of tree
- If we have |V| nodes, we would like to bound the maximum height

Height of Binary Trees

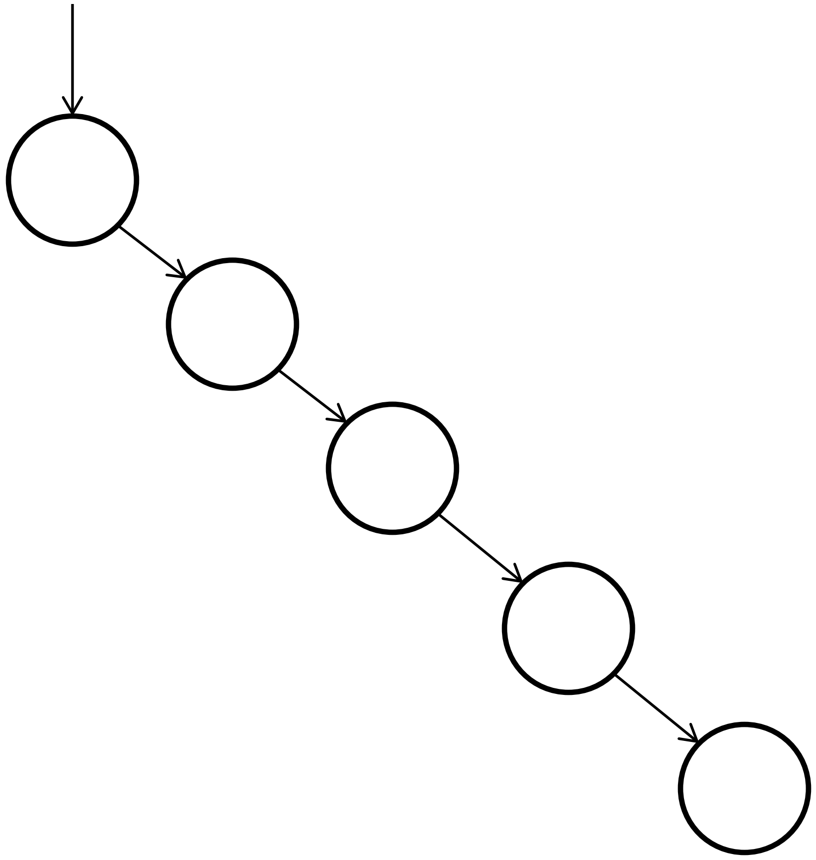
- For any binary tree: there are at most 2^{i-1} nodes at level i
- A binary tree with height h and n nodes satisfies

$$|V| \leq \sum_{i=1}^{h} 2^{i-1} = 2^{h} - 1$$

- Therefore, $h \geq \log(|V| + 1)$
- What kind of upper bounds can we get?

Height of Binary Trees

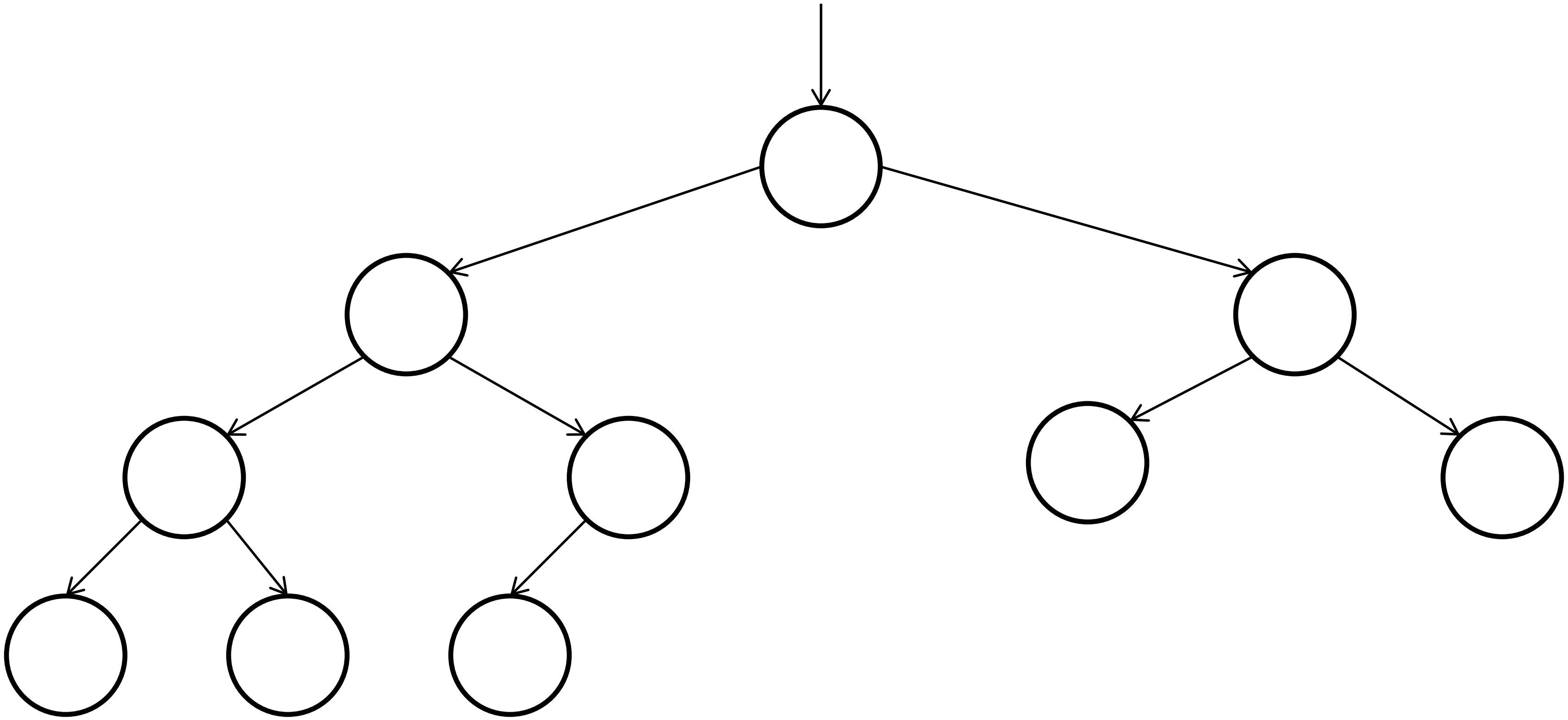
- Degenerate:



$h = |V|$

Hight of Binary Trees

- Complete



Height of Complete Binary Tree

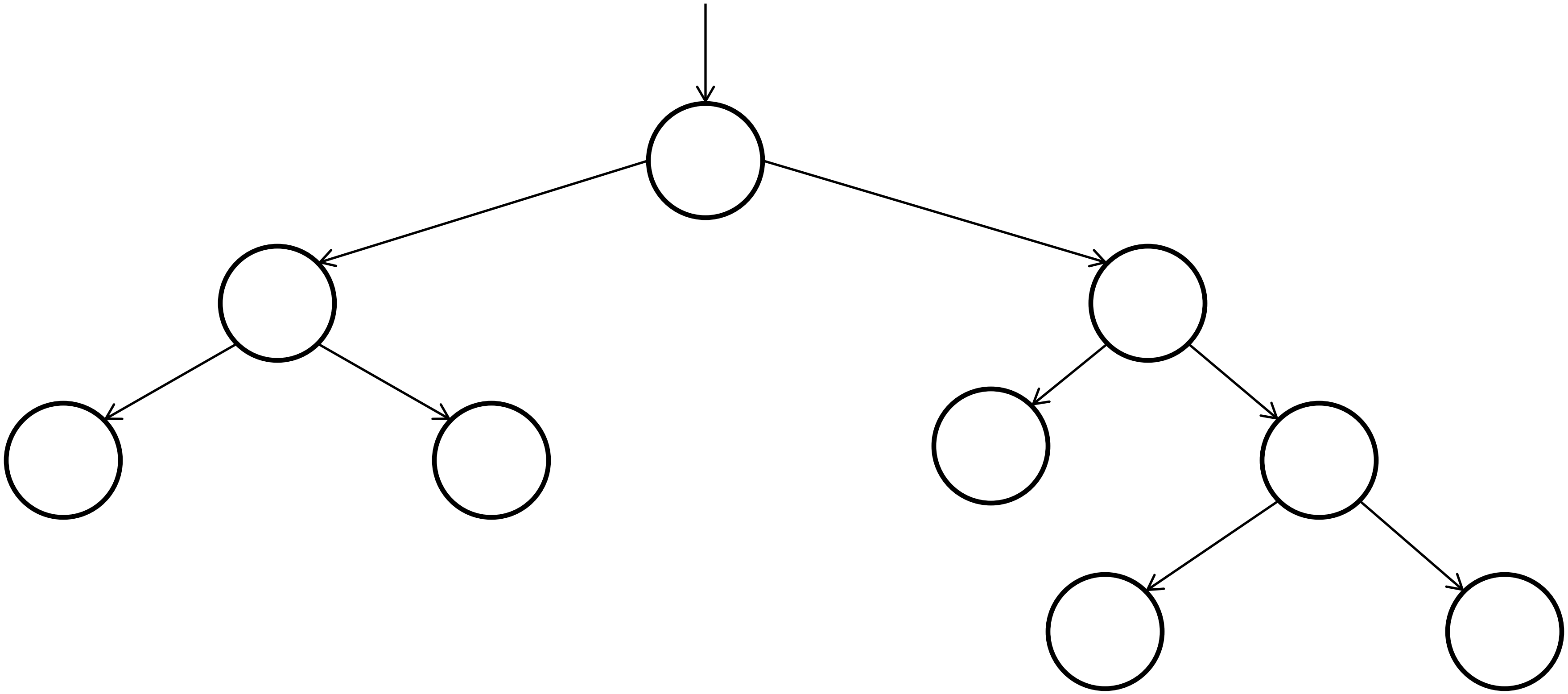
- There are exactly 2^{i-1} nodes at every level i, except for i = d, which has at least 1
- Therefore, a complete tree satisfies

$$|V| \geq 1 + \sum_{i=1}^{d-1} 2^{i-1} = 2^{d-1}$$

- Therefore, $d \leq \log |V| + 1$

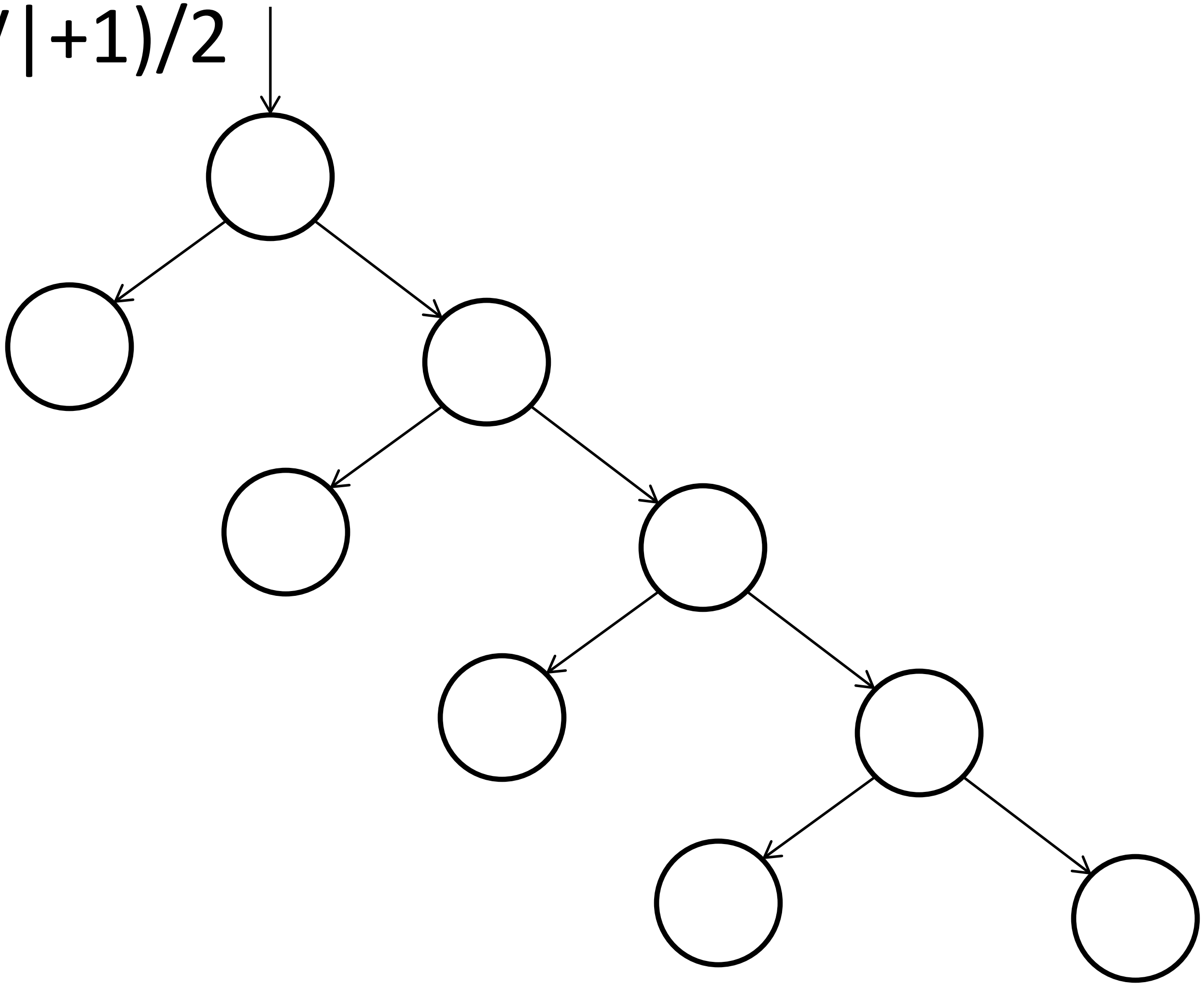
Height of Binary Trees

- Full?



Height of Binary Trees

- Full: (|V|+1)/2



Balanced Binary Trees

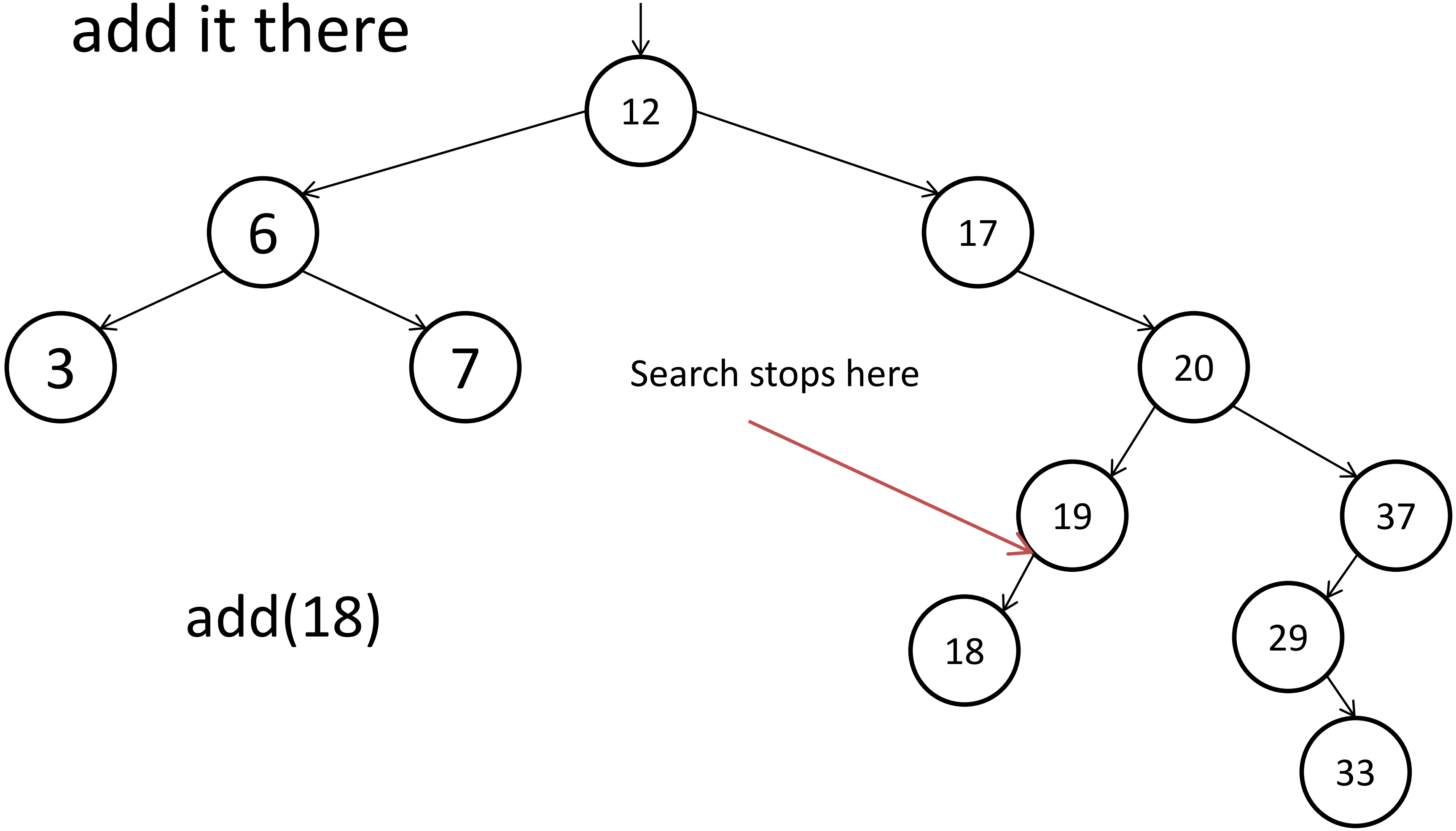
- Relaxation of complete binary tree
- h = O(log |V|)
- Makes searching asymptotically optimal
- Intuition: no leaf is much deeper than any other leaf

Modifying BSTs

- In general, BST will change over time.
- Would like a BST to stay balanced so operations stay efficient

Inserting into a BST

- Find where value should go using a search, add it there



add(18)

Inserting into a BST

- Same time as search: O(h)
- What do we do if we are inserting a value v that already exists in the tree?
  - In doing search, we'll find v
  - The right subtree of v contains values at least v
  - Therefore, insert v into right subtree

Inserting into a BST

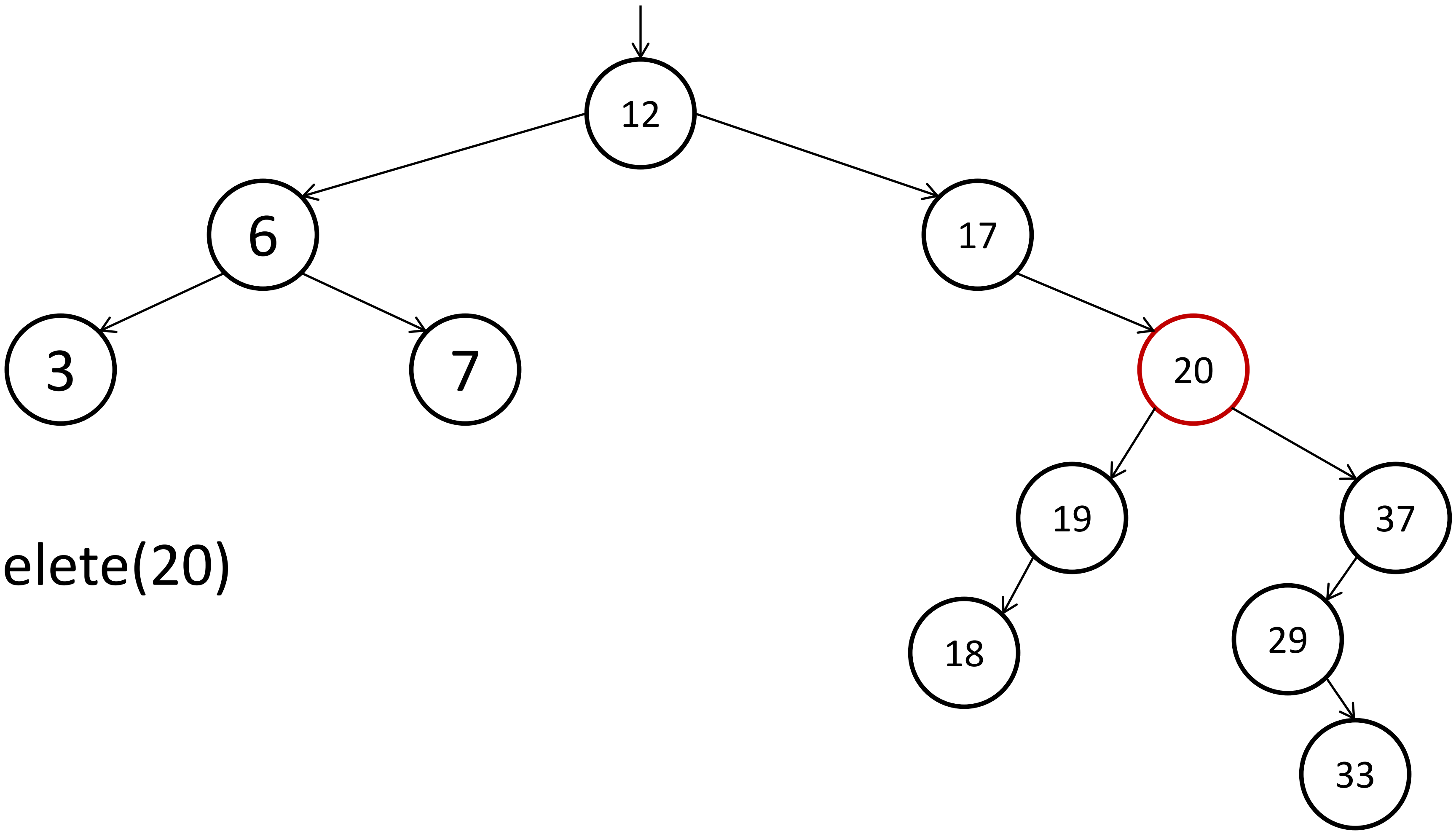
- insert(r,v) = {
  - If $v < r$:
    - if left child L exists, call insert(L,v)
    - otherwise make v left child
  - If $v \geq r$:
    - if right child R exists, call insert(R,v)
    - otherwise make v right child

  }

Deleting from a BST

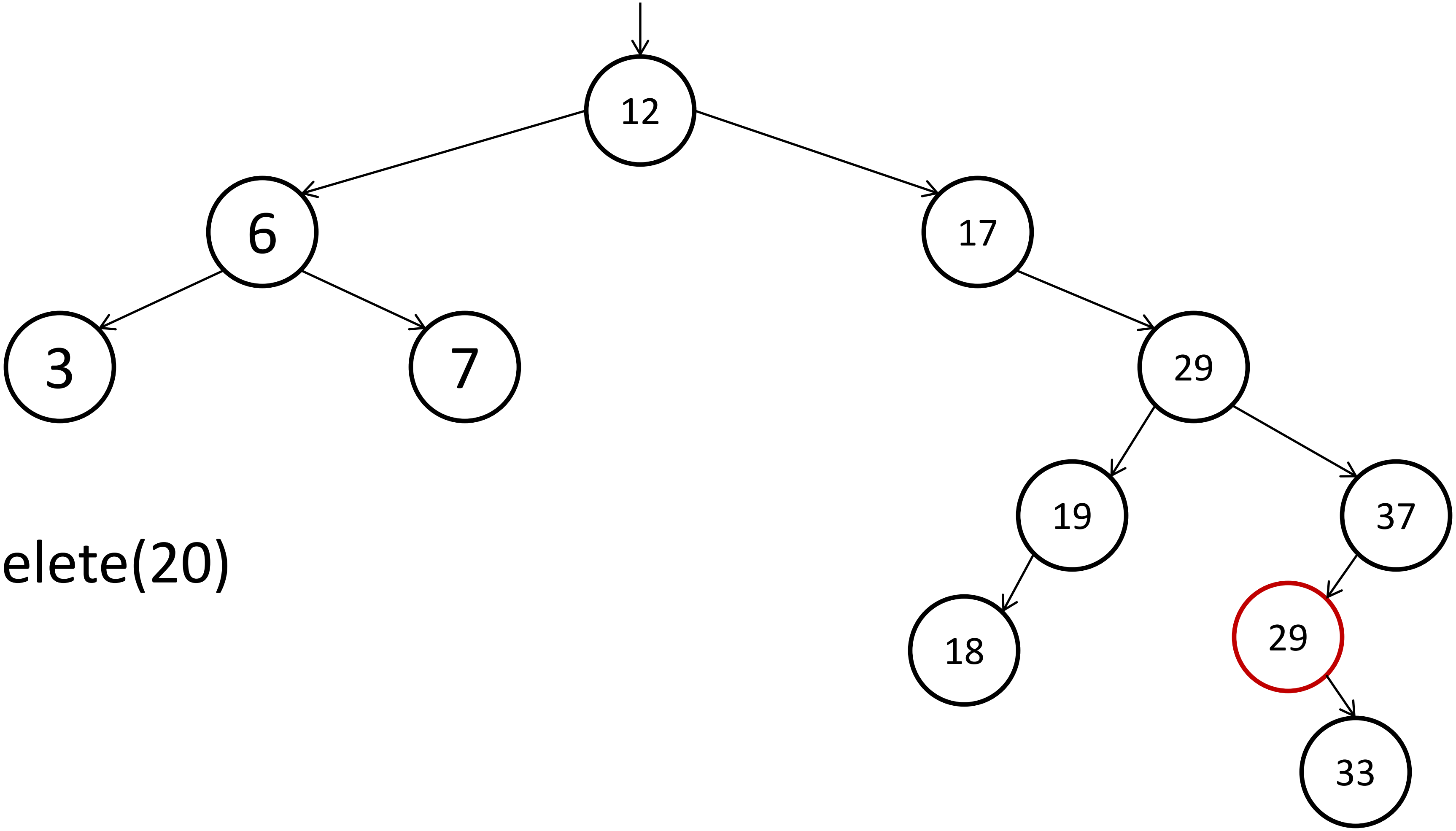
- Find value
  - If leaf, delete
  - If exactly 1 child, delete and replace with child
  - If two children, delete and replace with smallest node in right subtree

Deleting from a BST



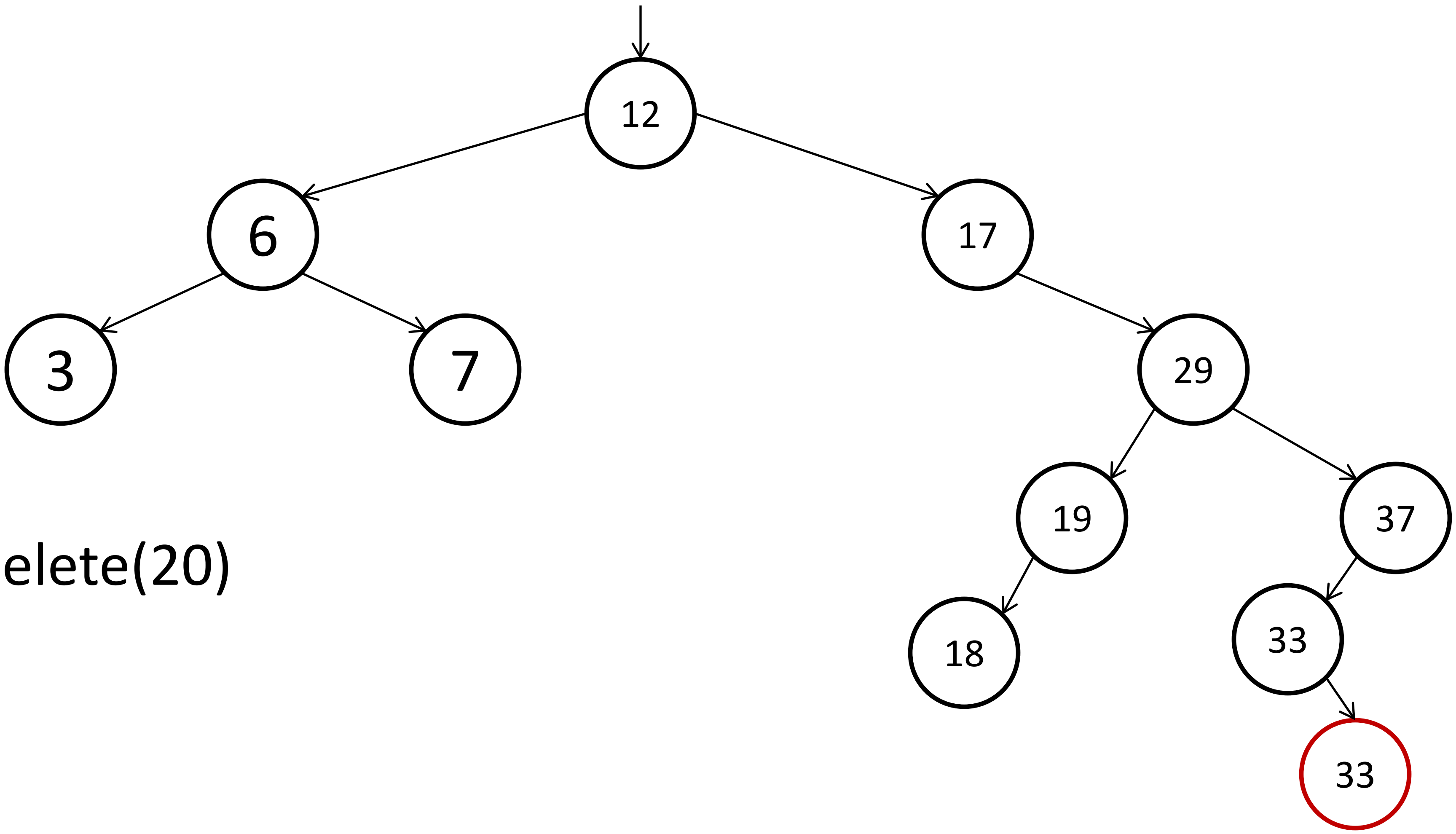
delete(20)

Deleting from a BST



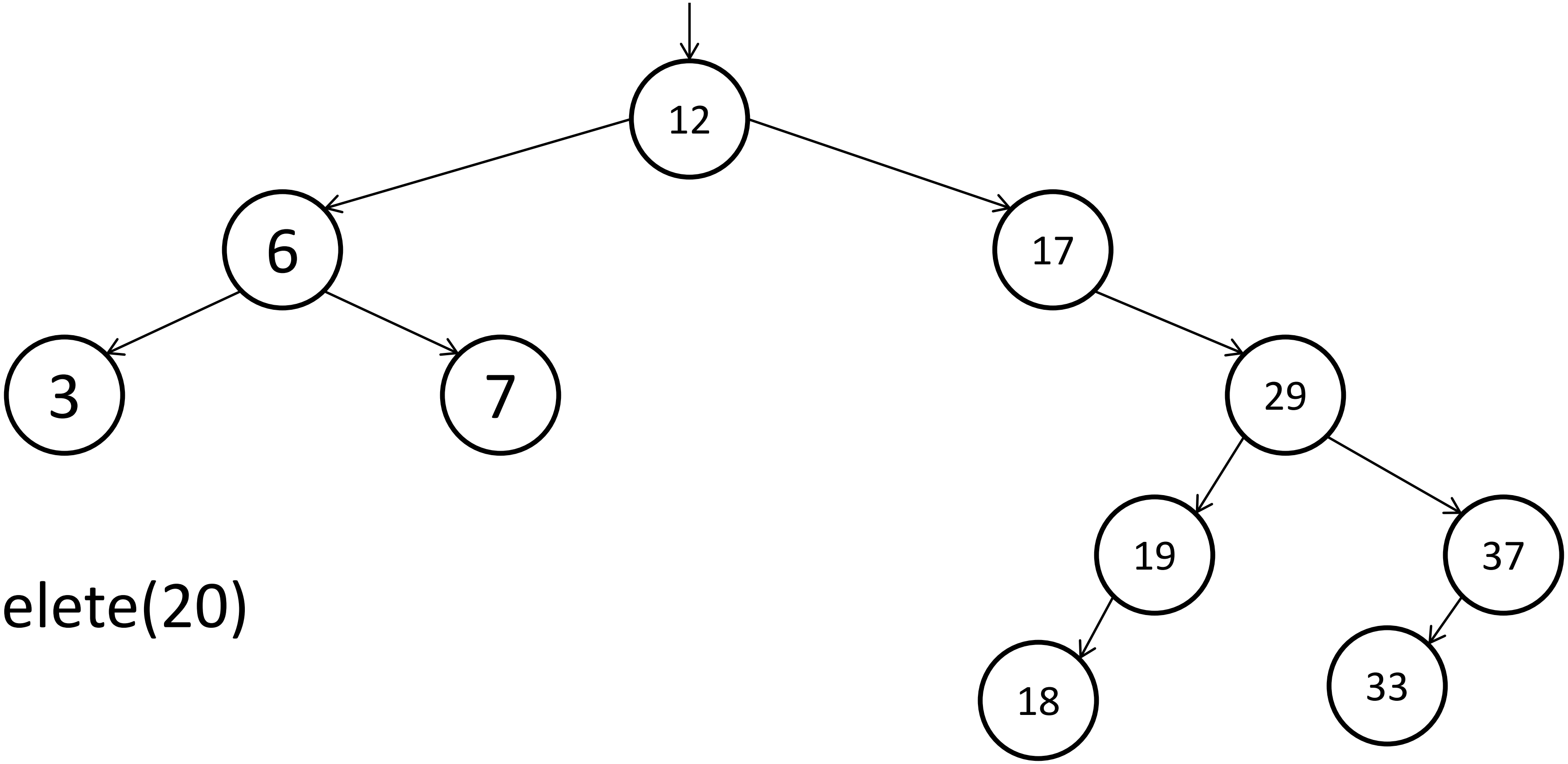
delete(20)

Deleting from a BST



delete(20)

Deleting from a BST



delete(20)

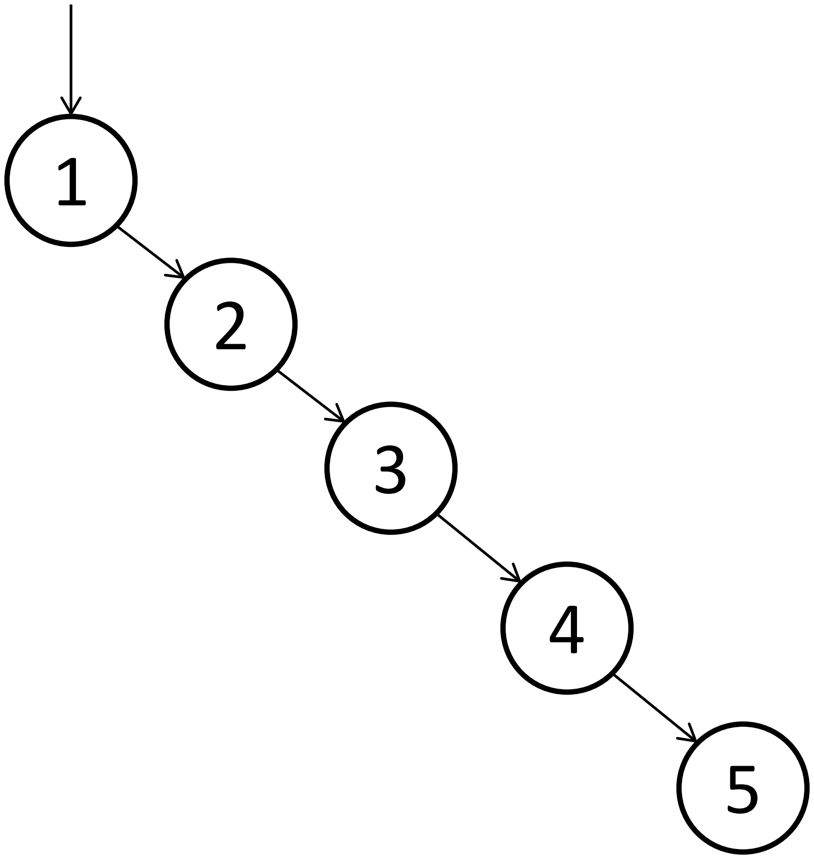
Deleting from a BST

- Let v be the lowest node deleted
- We only visit ancestors of v
- Therefore, running time is O(h)

Keeping Balance

Problem: Inserting in Order

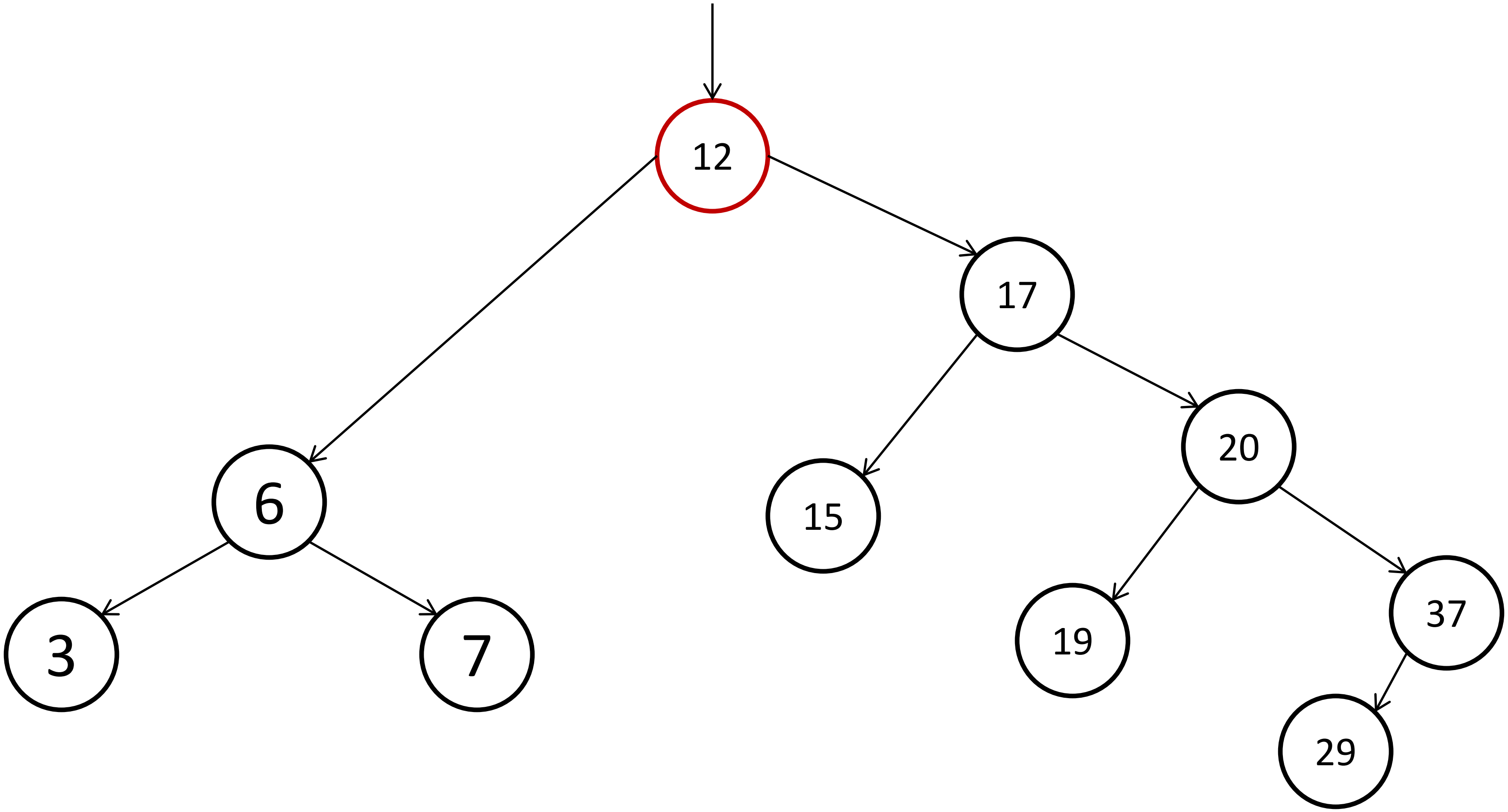
- What if we start with an empty tree, and add 1, then 2, then 3, … up to n?



Degenerate tree!

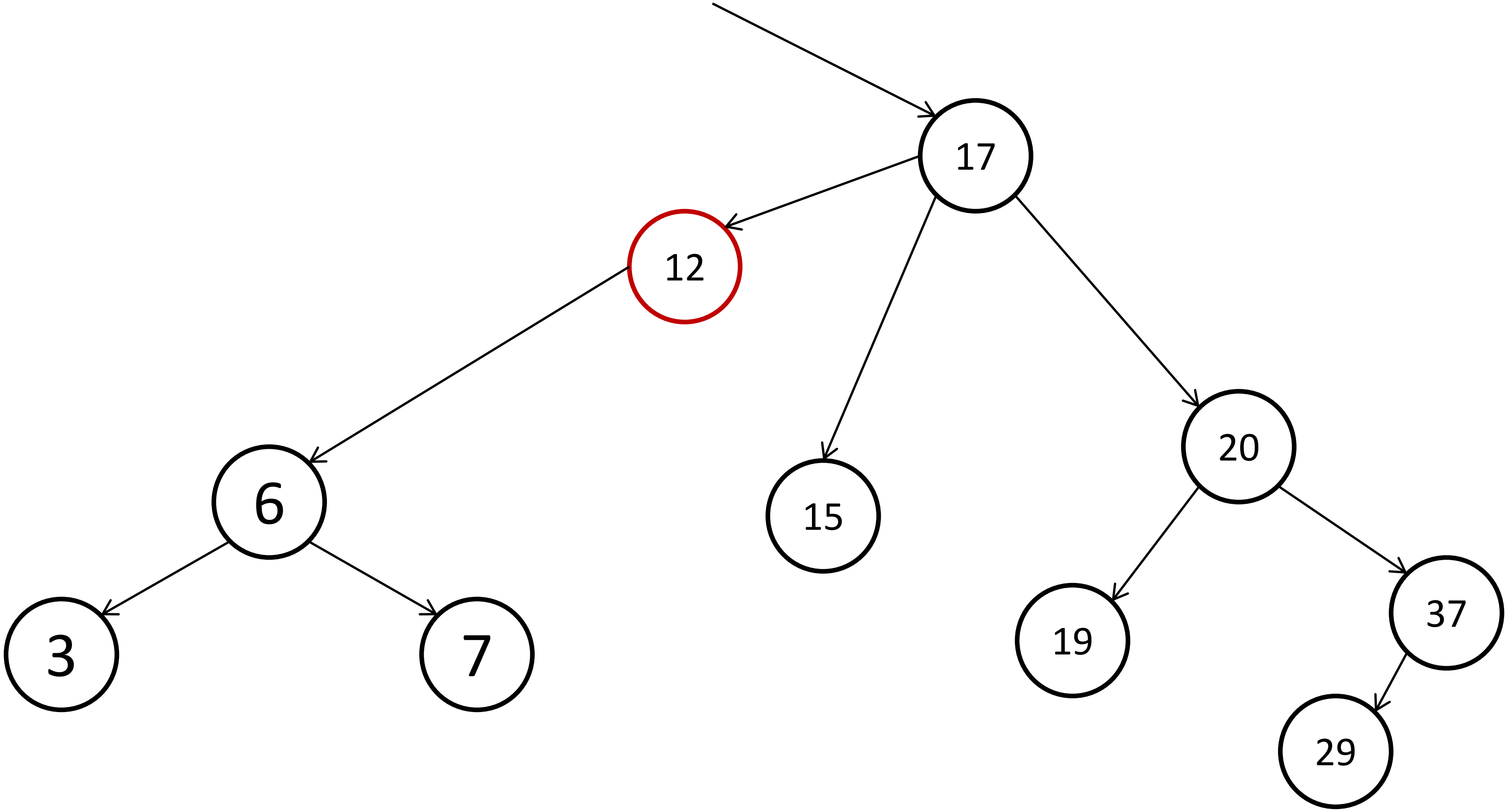
Restoring Balance: Rotations

- Rotate Left



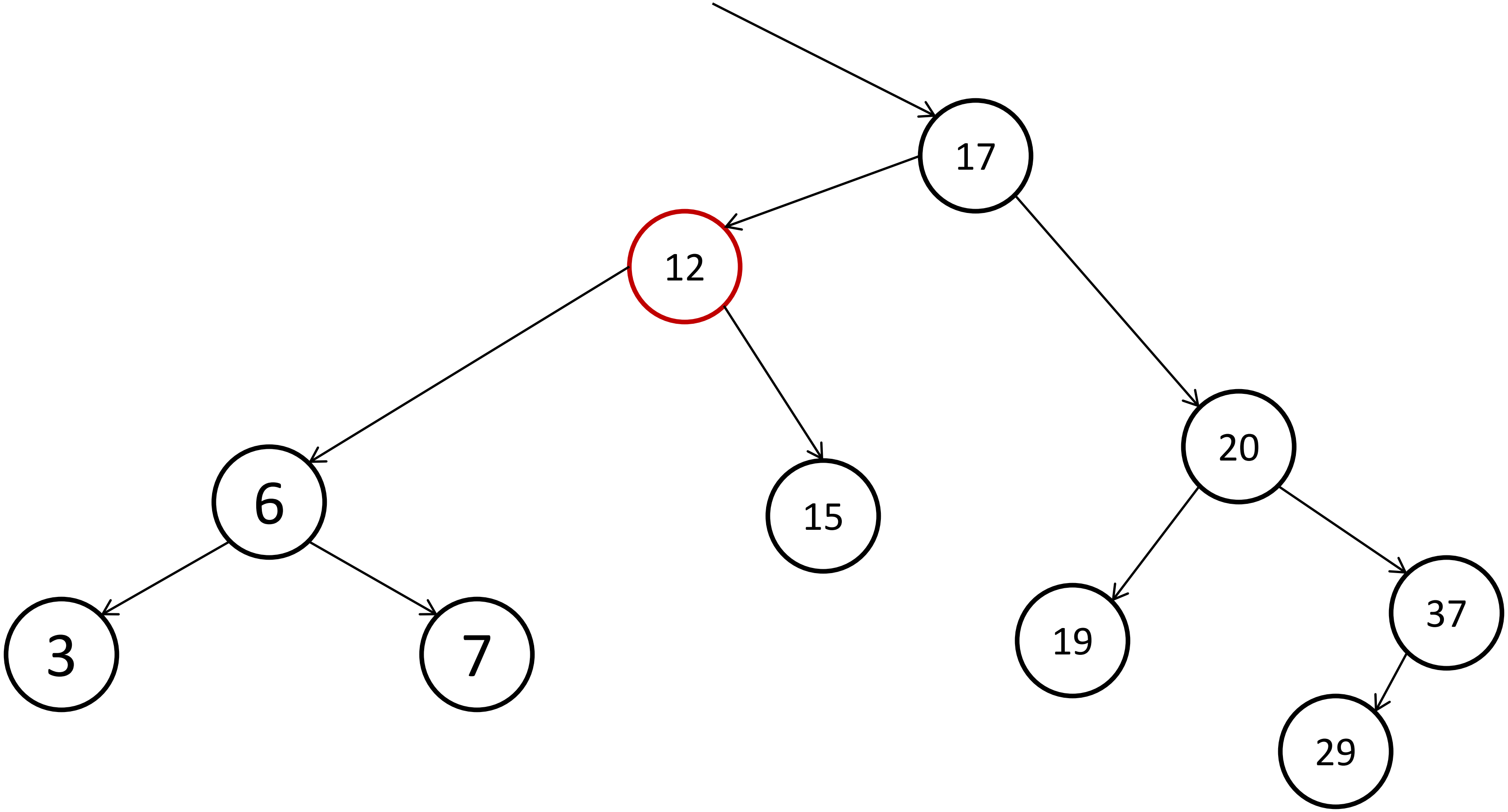
Restoring Balance: Rotations

- Rotate Left



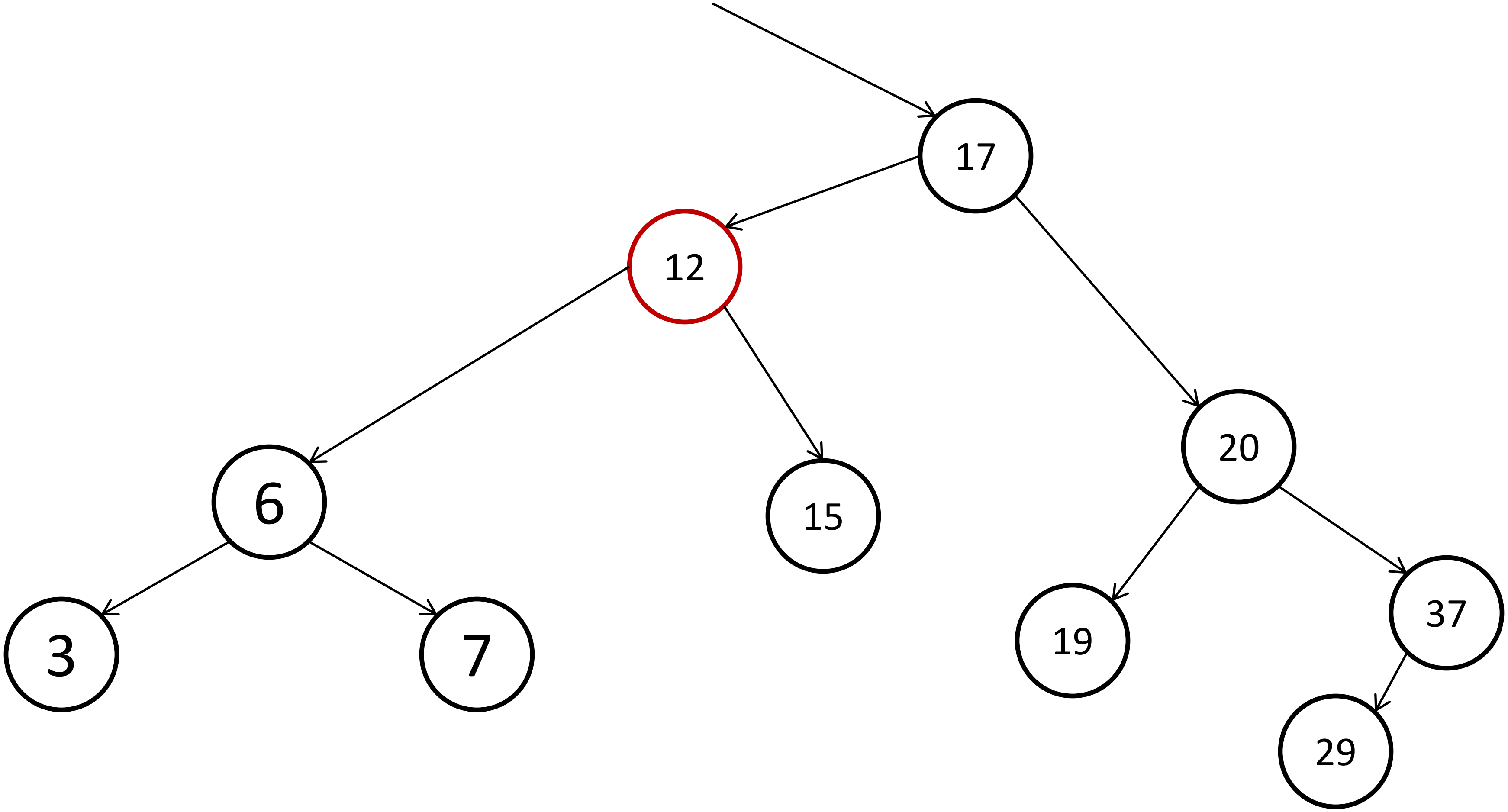
Restoring Balance: Rotations

- Rotate Left



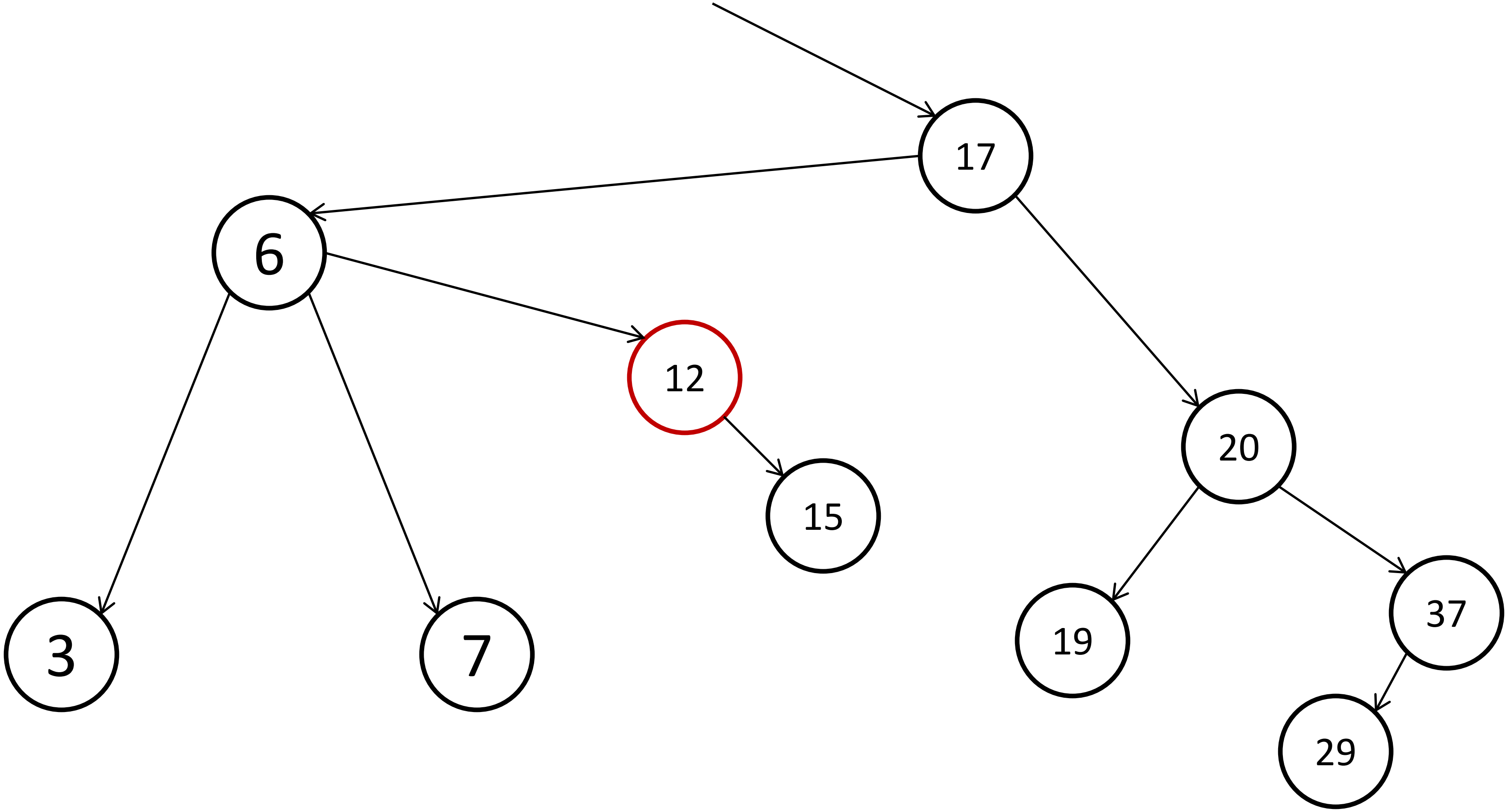
Restoring Balance: Rotations

- Rotate Right



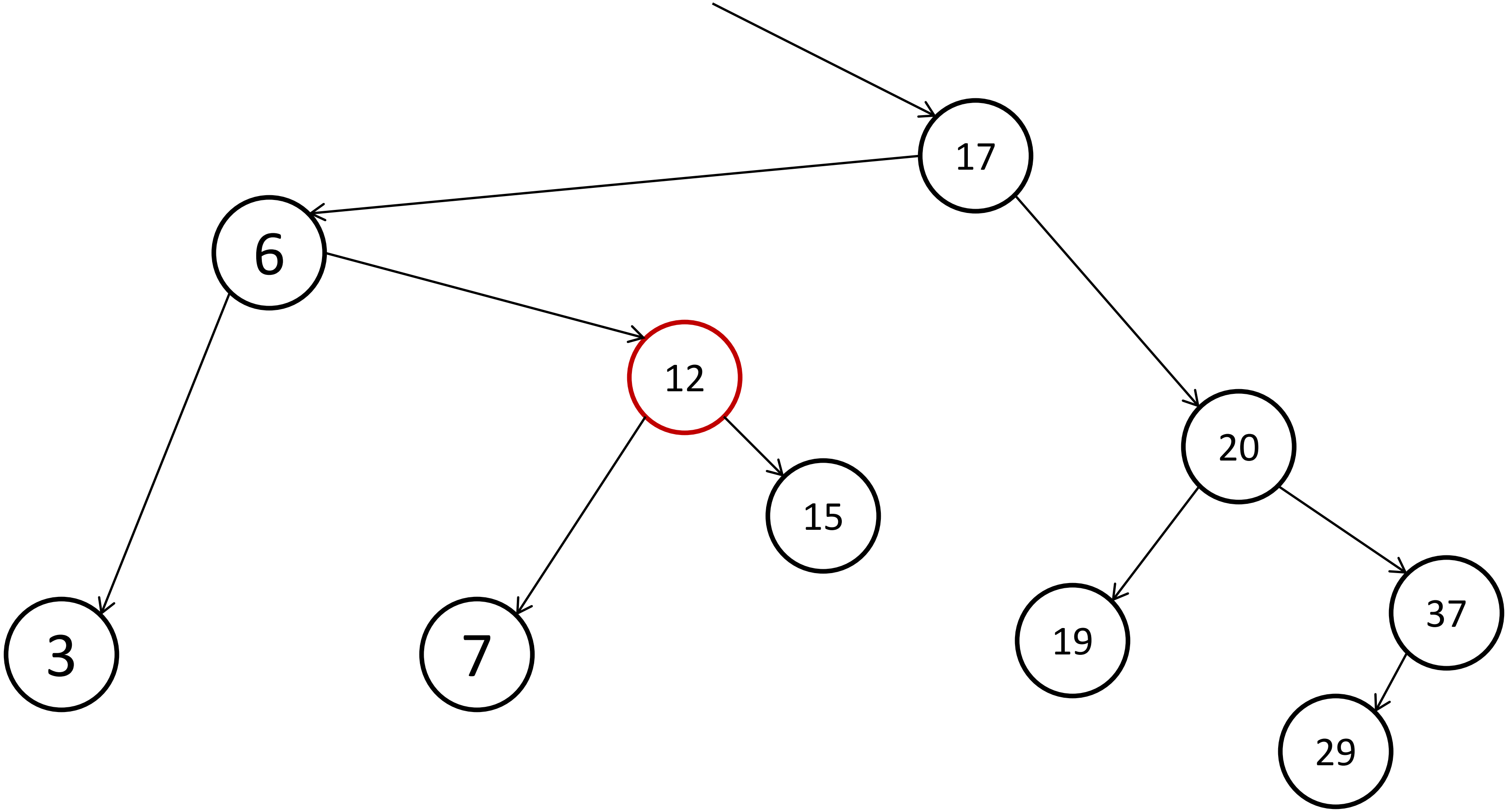
Restoring Balance: Rotations

- Rotate Right



Restoring Balance: Rotations

- Rotate Right



Restoring Balance: Rotations

- Rotations only affect 3 pointers, O(1) time.
- Rotate left decreases depth of right subtree by at least 1
- Rotate right decreases depth of left subtree by at least 1.

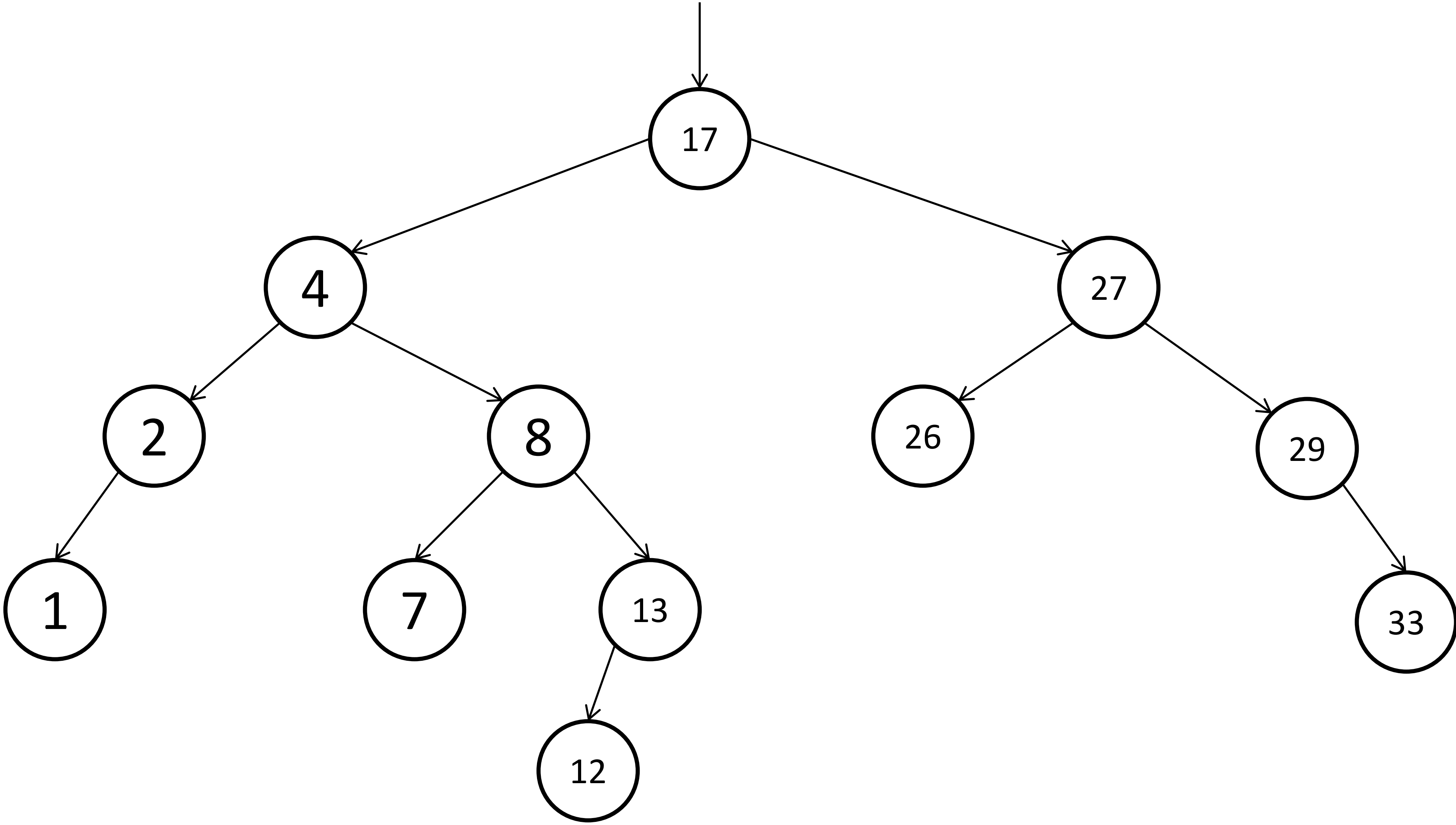
Self-Balancing BSTs

- Using rotations, keeps BST balanced during insertions and removals
- Use O(log |V|) rotations/operations per insertion, removal.

Example: AVL Trees

- Each node also stores the height of the subtree rooted at that node
- Property: the height of the children of any node can only differ by 1 from each other

Example: AVL Trees



Example: AVL Trees

- Let ϕ be the golden ratio ≈1.62 ($\phi^2 = \phi + 1$)
- Claim: $n \geq \varphi^h - 1$
  - Proof: True for h = 0,1
  - If h>1, one of the subtrees must have height h-1. Nodes in this subtree (by induction): $n_1 \geq \varphi^{h-1} - 1$
  - The other must have height at least h-2. Nodes in this subtree: $n_2 \geq \varphi^{h-2} - 1$
  - Total number of nodes:

$$|V| = 1 + n_1 + n_2 \geq \varphi^{h-1} + \varphi^{h-2} - 1 = \varphi^h - 1$$

Example: AVL Trees

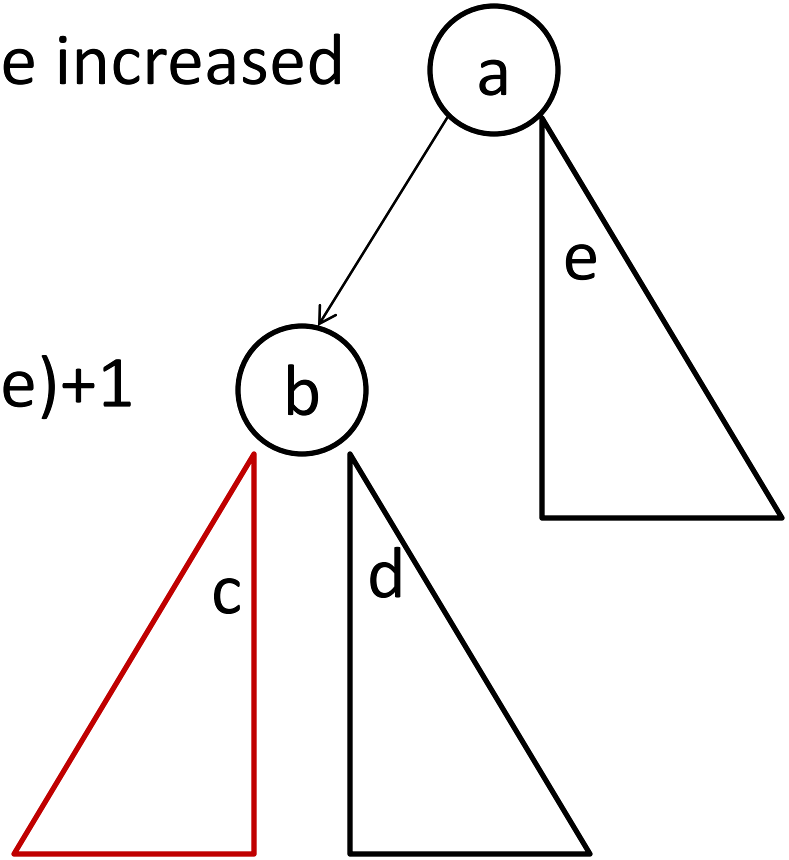
$$|V| \geq \varphi^{h} - 1$$

- This means we can bound h as

$$h \leq \log_{\varphi}(|V|+1) = O(\log|V|)$$

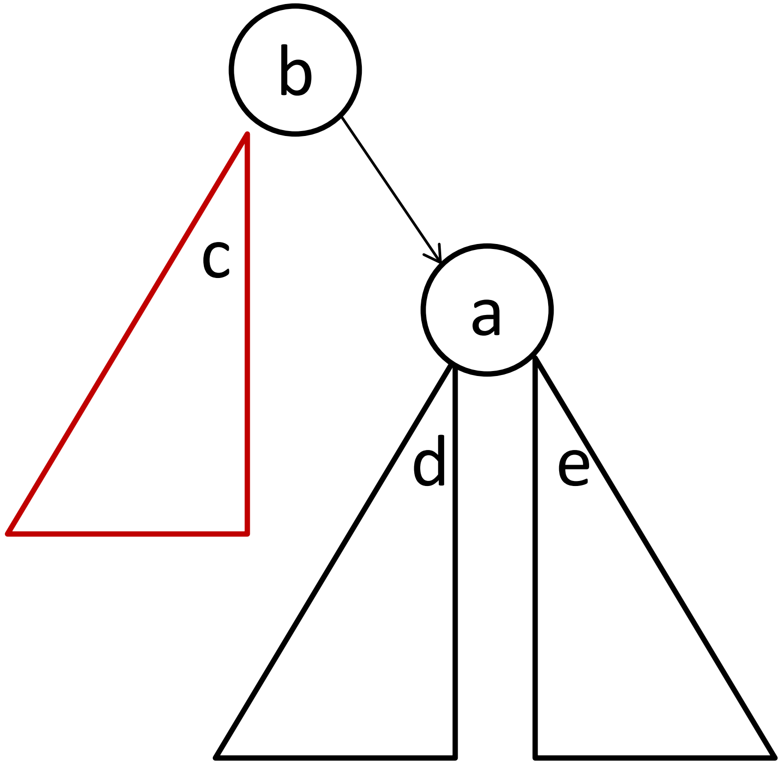
Inserting into AVL Trees

- Say inserting into subtree c causes node a to violate AVL property
  - Then h(b) and h(c) must have increased
  - h(b)=h(e)+2
  - h(c)-1≤h(d)≤h(c)
  - Thus h(b)=h(c)+1, so h(c)=h(e)+1



Inserting into AVL Trees

- Rotate a to the right
  - – $h(a)=1+\max(h(d),h(e))$
  - – Recall $h(c)-1 \le h(d) \le h(c)$ and $h(e)=h(c)-1$
  - – Thus $h(e) \le h(d) \le h(e)+1$
    - a is balanced
  - – $h(a)=1+h(e)=h(c)$
    - b is balanced



Inserting into AVL Trees

- Other cases more complicated, require 2 rotations

Maintaining Height Data During Operations

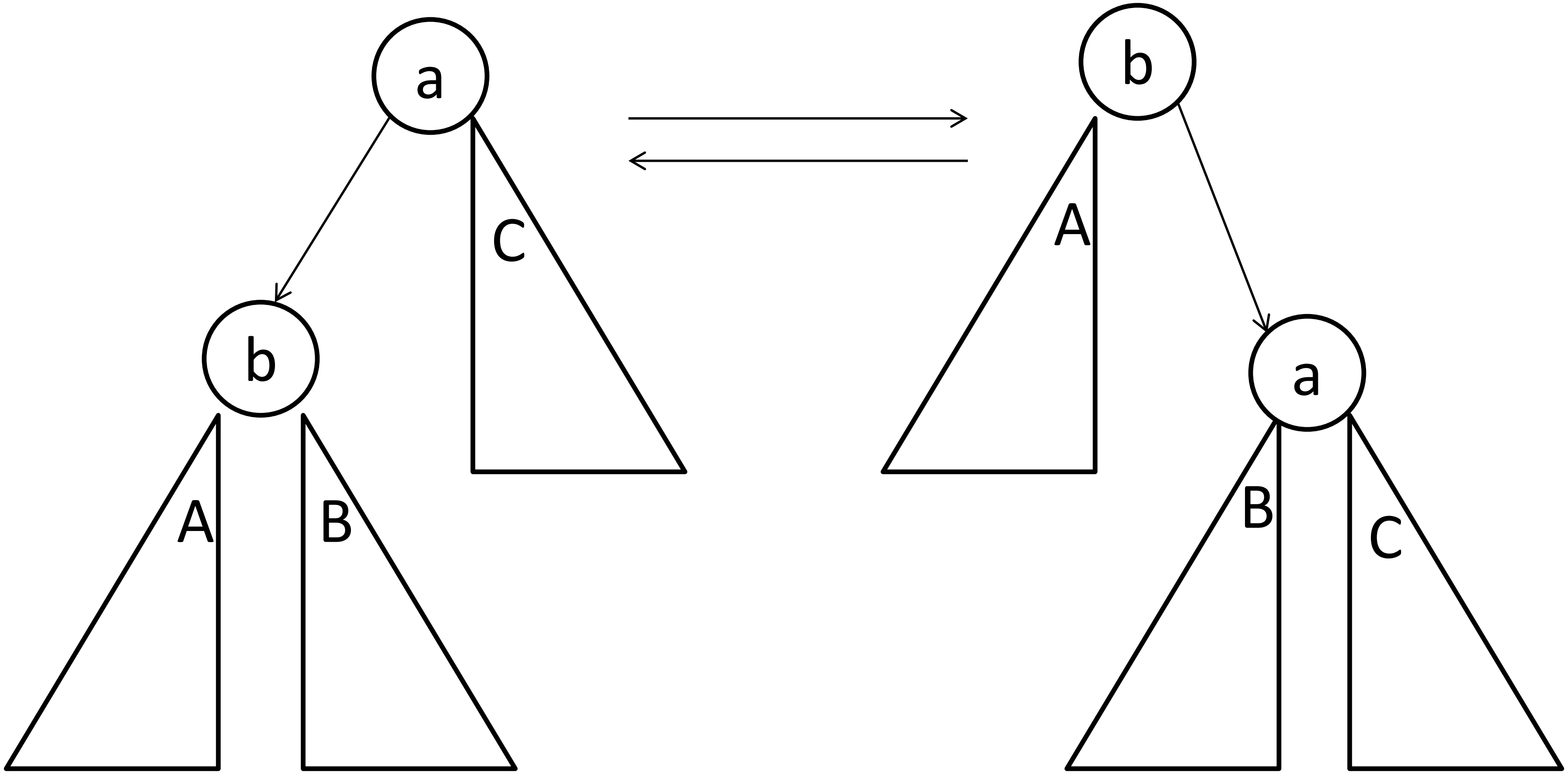
- When we add a node, its height is 1
- The only nodes whose height have changed are the ancestors
- Work back up the tree recalculating heights

Maintaining Height Data During Operations

- When we delete a node, let v be the lowest node that gets deleted
- The only nodes whose height have changed are the ancestors of v
- Work back up the tree recalculating heights

Maintaining Height Data During Operations

- Rotations: Just need to recalculate h(a) and h(b) (and ancestors of b)

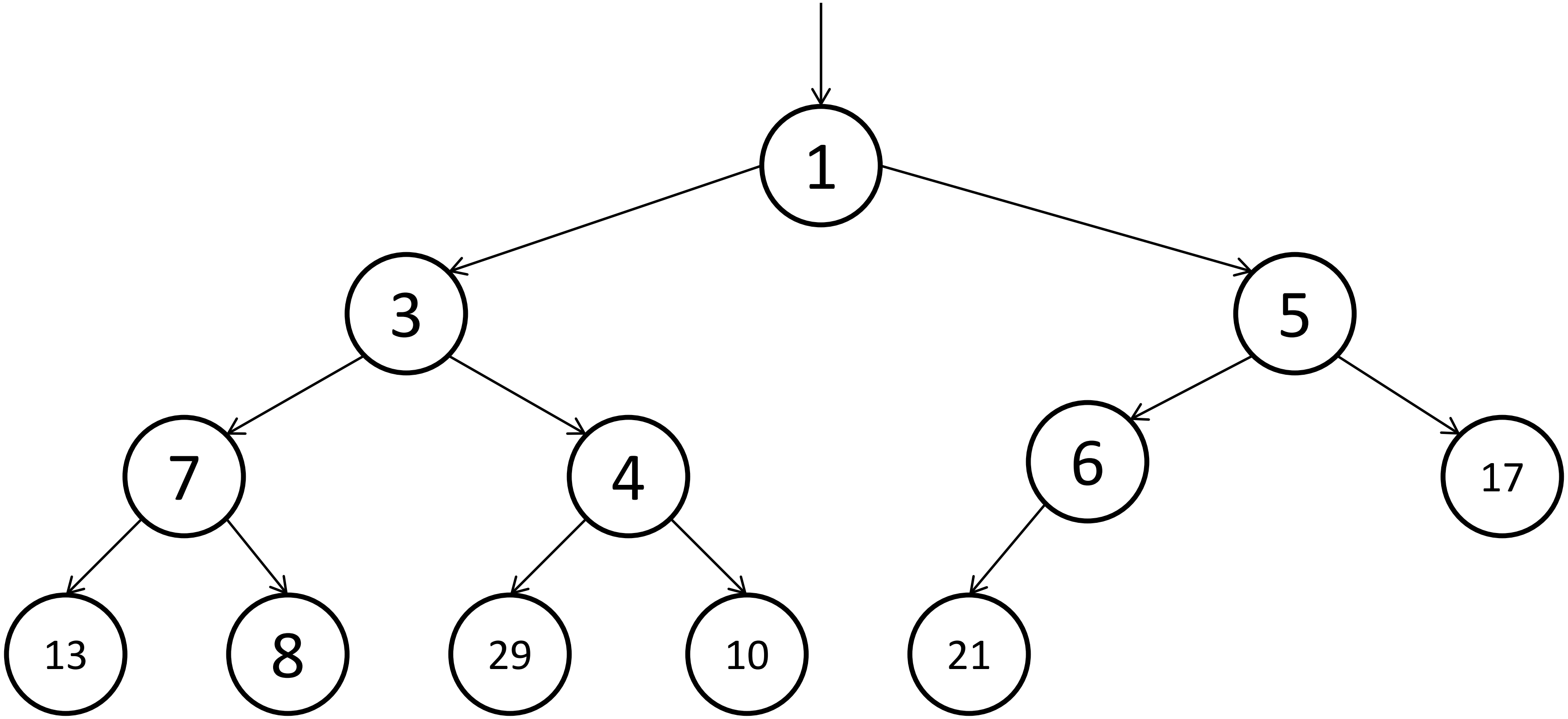


Heaps

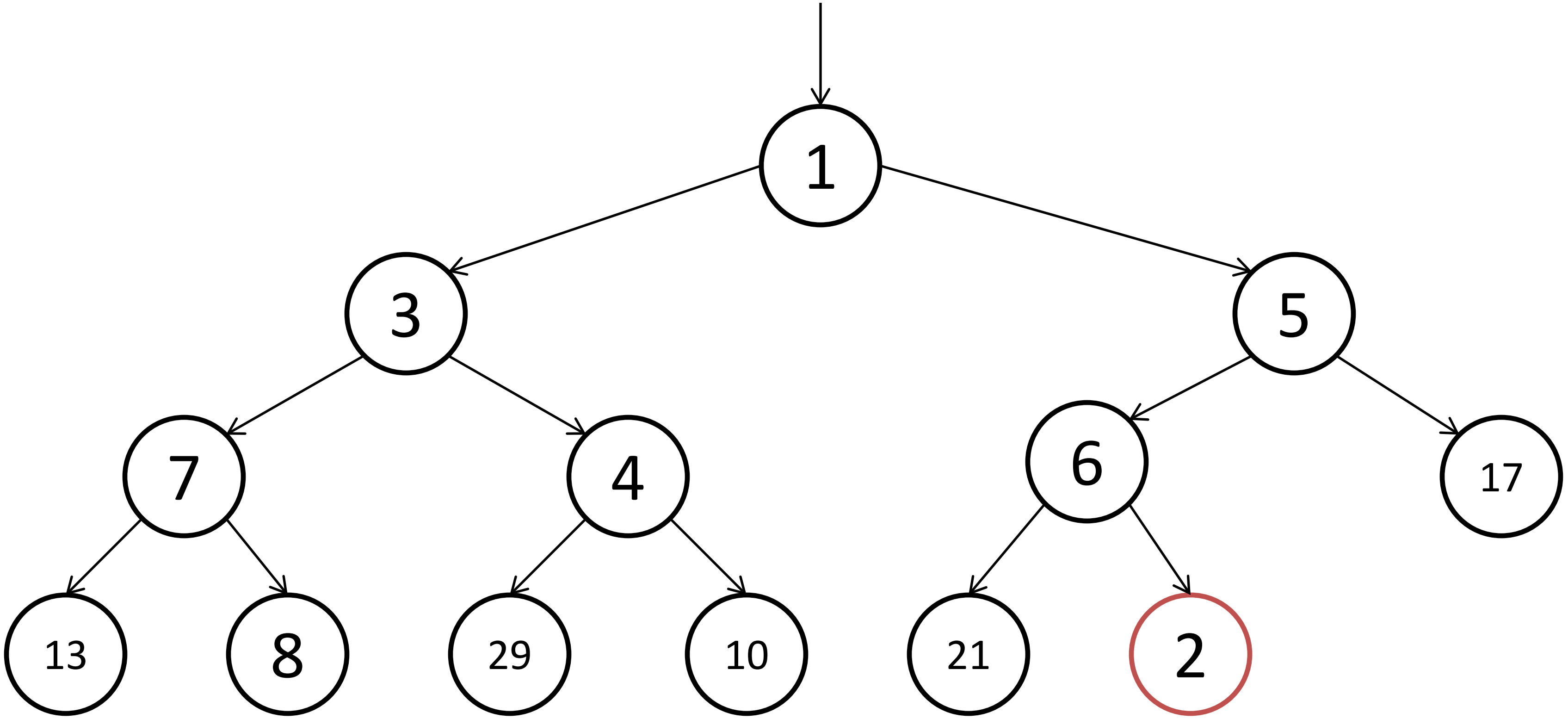
Heaps

- Complete tree
- Property: every node has a smaller value than its children
  - Root contains lowest value
- Supported operations:
  - deletemin(): returns and removes lowest element
  - insert(x): adds element
  - decreasekey(x): Sometimes the order of values may change. Updates heap if the value x decreases

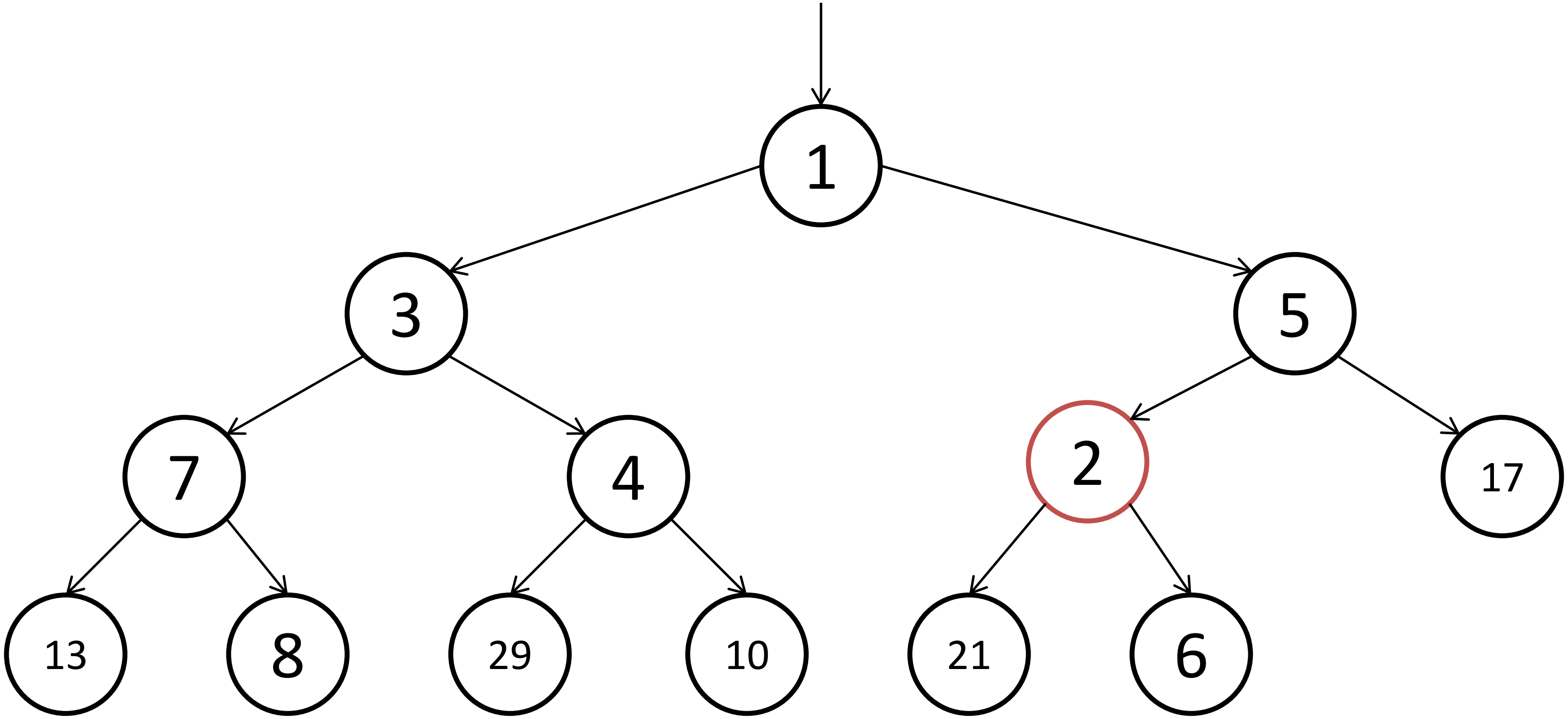
Binary Heaps



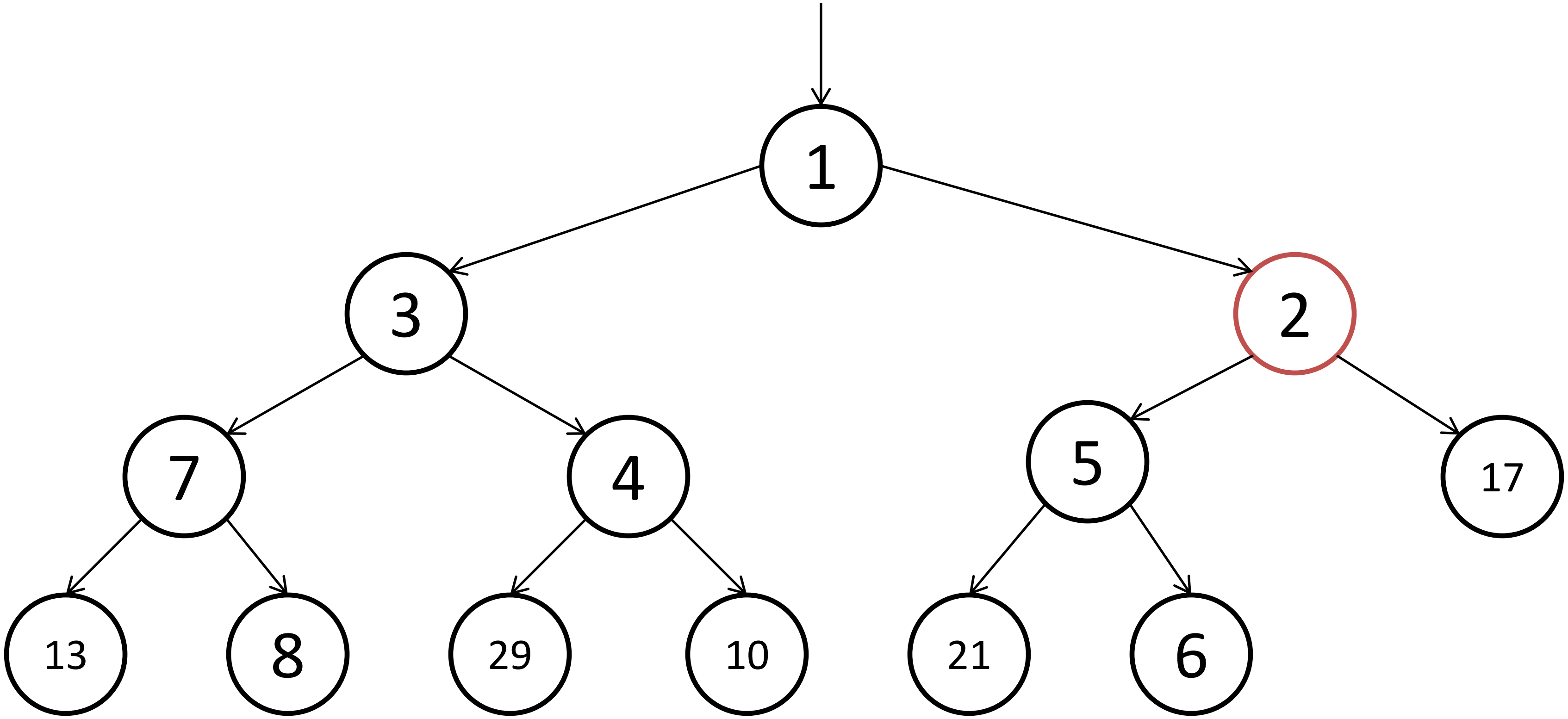
insert(2)



insert(2)



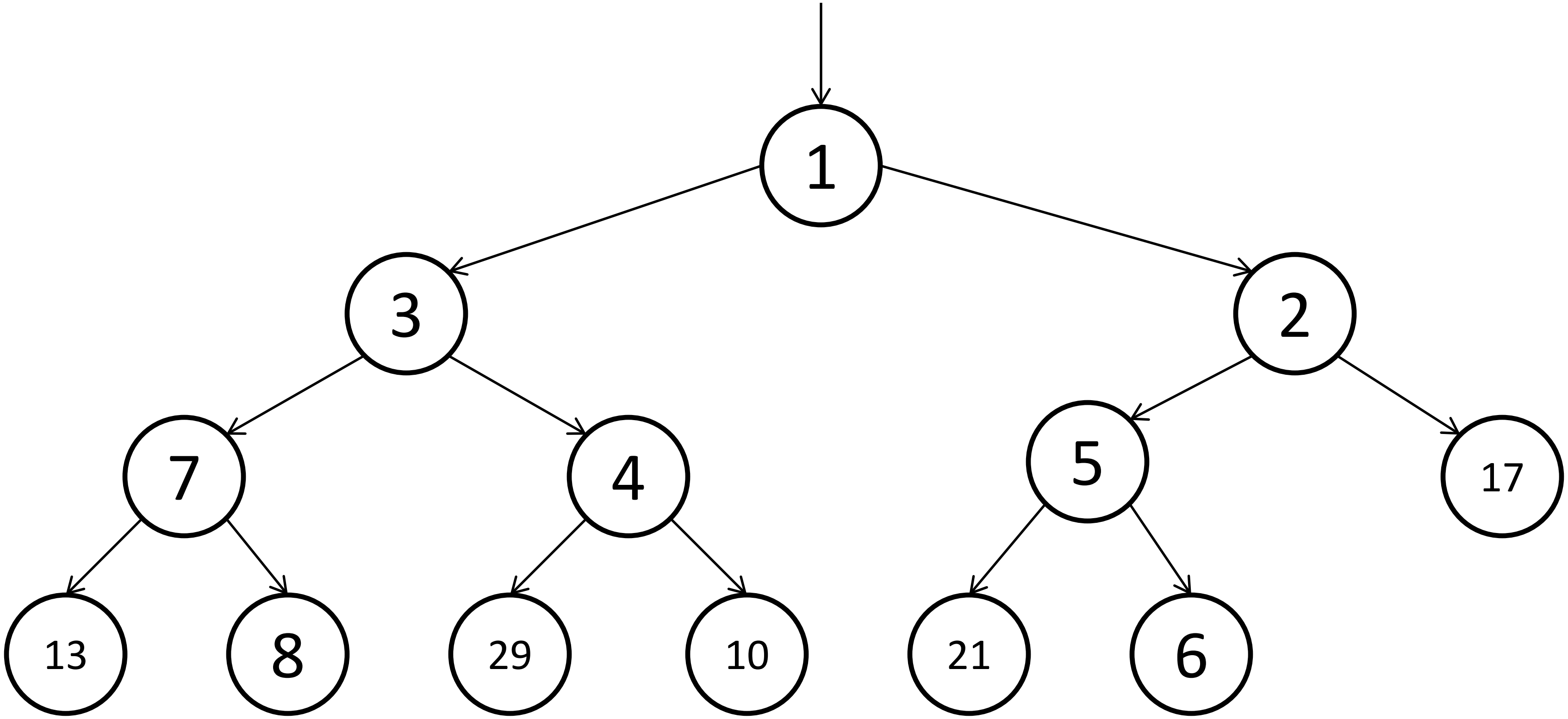
insert(2)



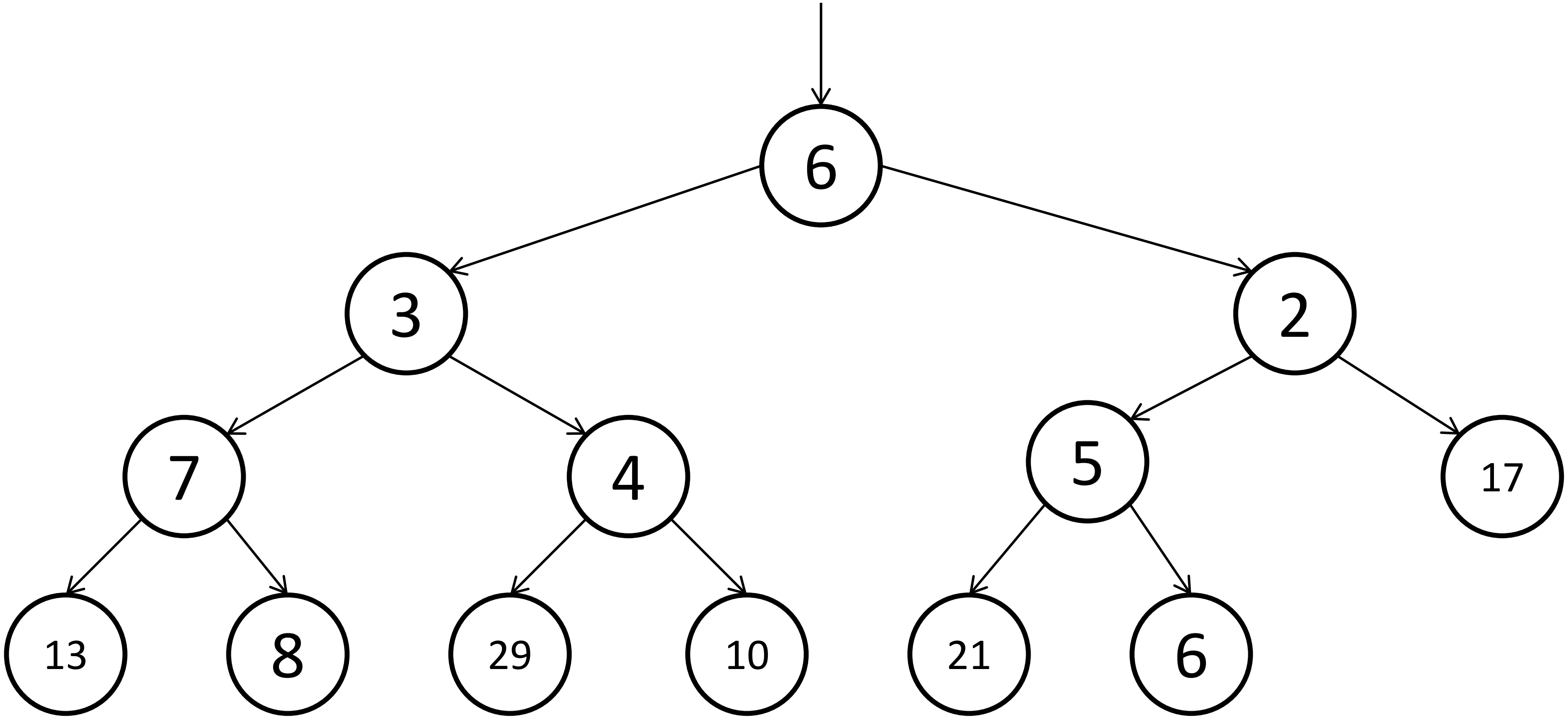
Insert

- Add element to end of last row
- Check if node is greater than parent
  - If so, swap, and repeat from parent
  - Otherwise, done
- O(h)=O(log |V|) time

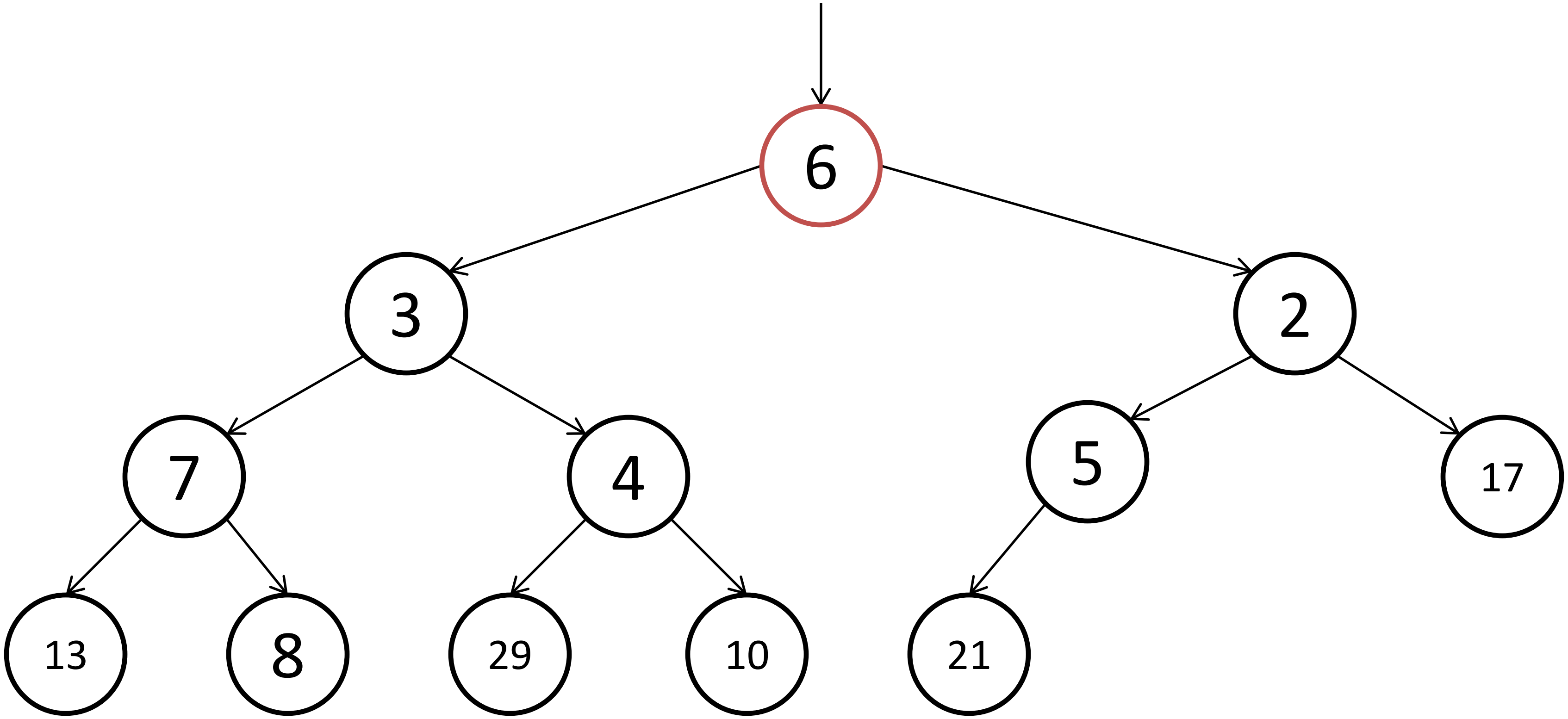
deletemin()



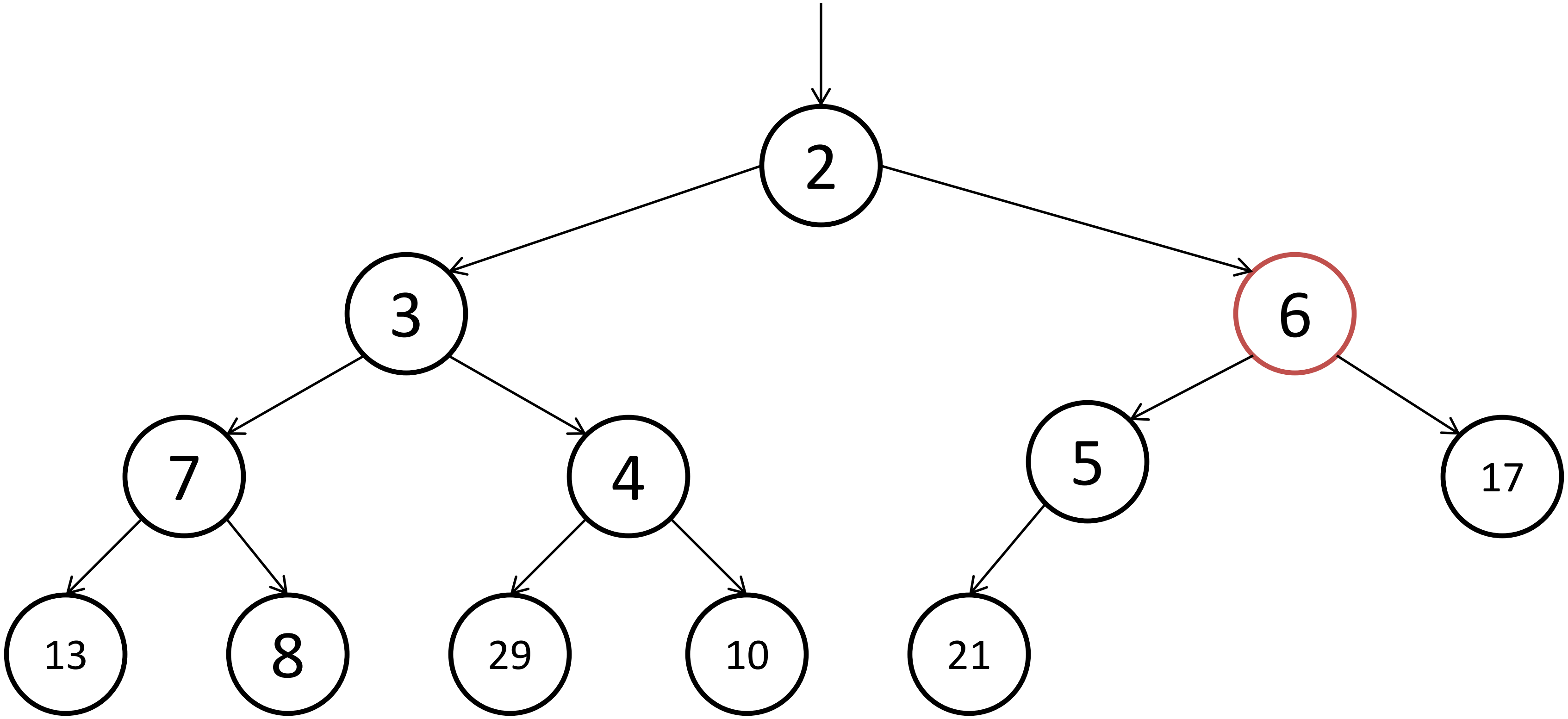
deletemin()



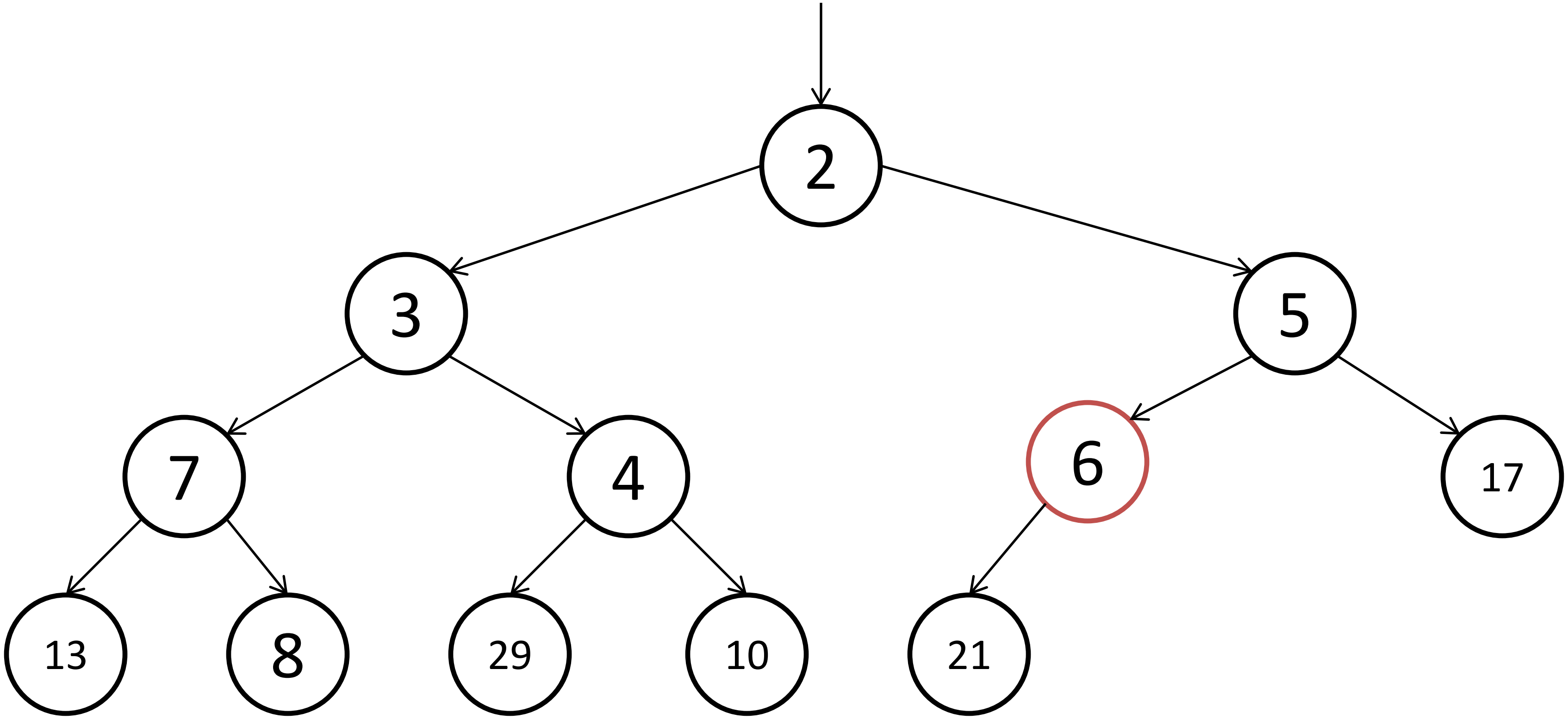
deletemin()



deletemin()



deletemin()



deletemin()

- Delete root, move rightmost node of last row to root
- Check if node is smaller than both children
  - If so, done
  - Otherwise, swap with smaller child, and repeat.
- O(h) = O(log |V|) time

decreasekey

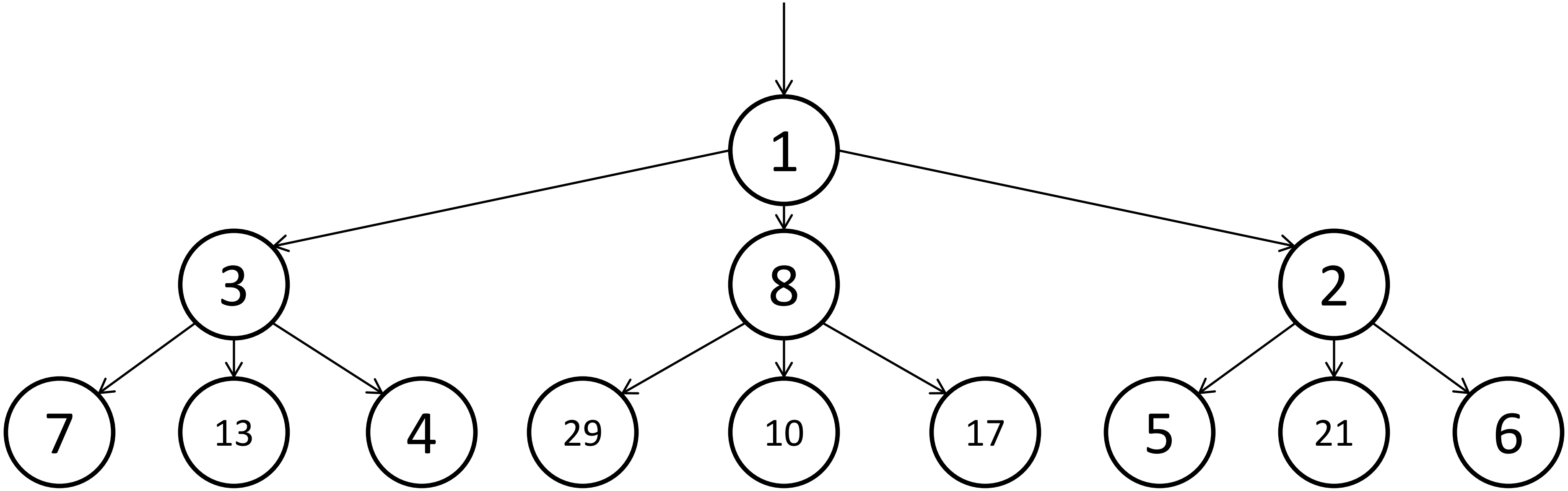
- The ordering of nodes may change
  - Example: heap contains strings, ordered by length
  - We might modify a string, making it shorter
- As long as the node we are modifying gets smaller, decreasekey will restore the heap property
- Just like inserts: check if less than parent. If so swap and repeat

Efficient Representation

- Dynamic array of length |V|
- Root: index 0
- Children of node at index i: 2i, 2i+1
- Parent of index i: floor(i/2)
- End of array = last node in last level
  - Can get to end efficiently

Generalization: d-ary Heaps

- Instead of two children per node, have d



Running Time

- insert/decreasekey: $O(\log |V| / \log d)$
- deletemin: $O(d \log |V| / \log d)$