

# Algorithms and Data Structures

## Trees (Ch. 4)

Morgan Ericsson

# Today

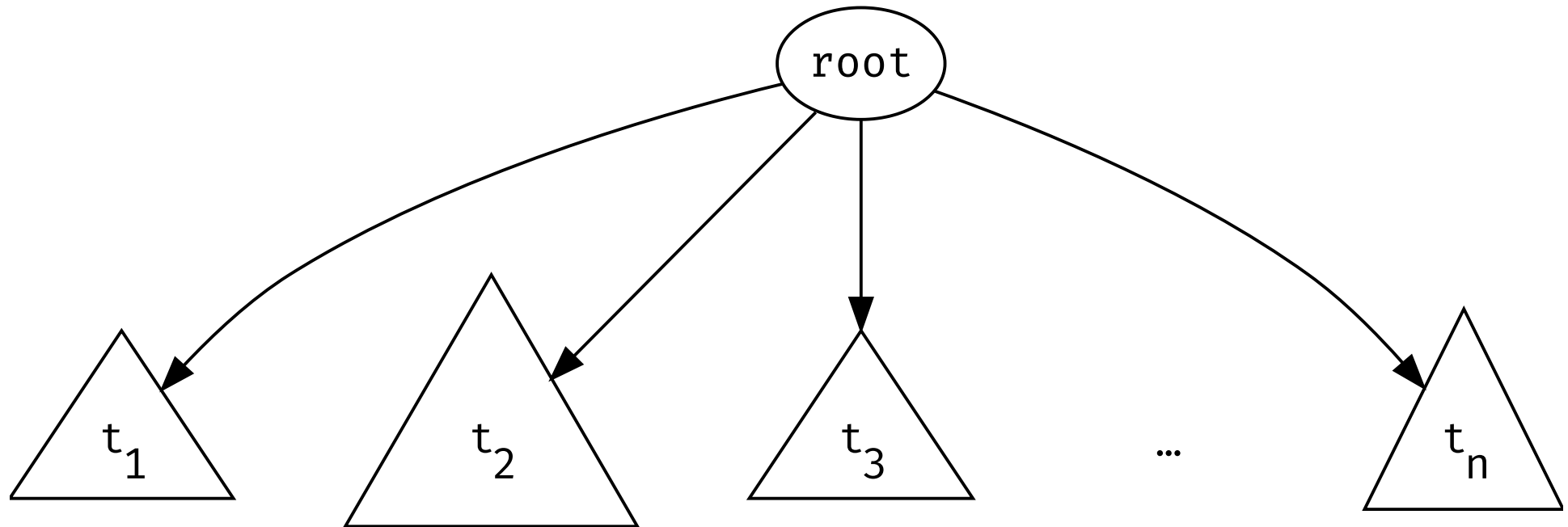
- » Trees
  - » Binary trees
  - » Binary Search Trees
  - » AVL-trees
  - » Splay trees

# Trees

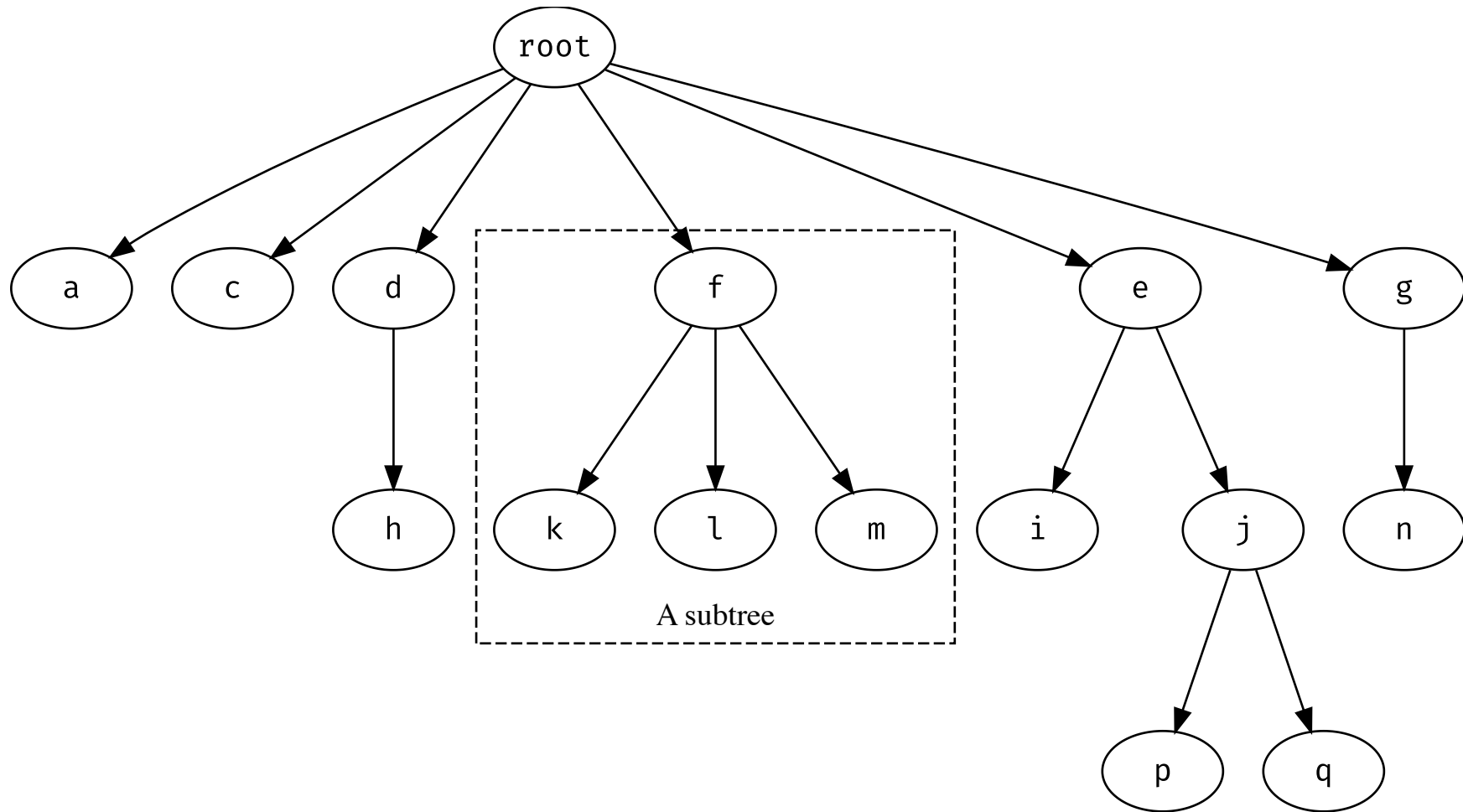
# The Tree ADT

- » A tree is a collection of nodes
- » If it is not empty,
  - » then it has a distinguished node  $r$  that is the root,
  - » and zero or more subtrees that are connected from the root by a directed edge
- » The root of each subtree is a *child* of  $r$ , and  $r$  is the parent of each subtree
- » Each subtree is a tree

# A tree



# A tree



# Trees

- » A node can have an arbitrary number of children
- » Nodes with no children are *leaves*
- » Nodes with the same parent are *siblings*

# Paths

- » A path from node  $n_1$  to node  $n_k$  is defined as a sequence of nodes:
  - »  $n_1, n_2, \dots, n_k$
  - »  $n_i$  is the parent of  $n_{i+1}$  for  $i \leq i < k$
- » The *length* of a path is the number of edges it contains
  - » So, the length of  $n_1, \dots, n_k$  is  $k - 1$



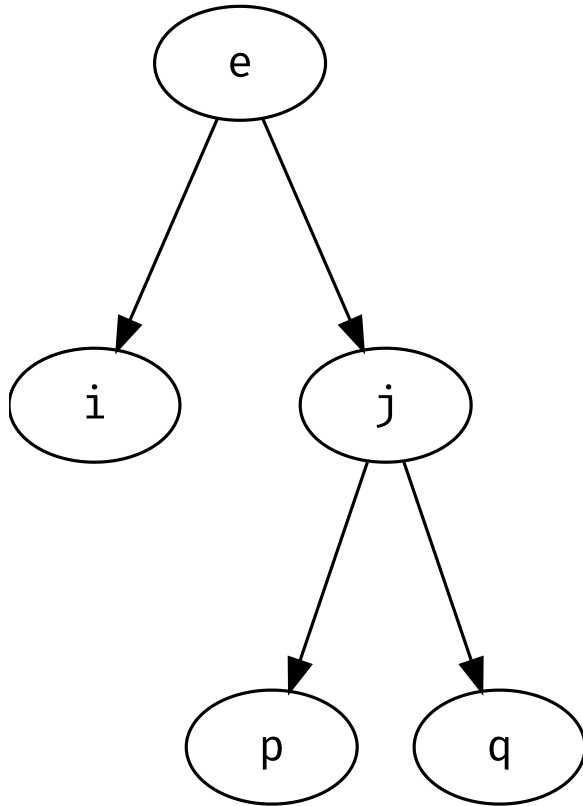
# Paths

- » The *depth* of a node,  $n_i$  is the length of the path from the root to  $n_i$
- » The *height* of a node,  $n_i$  is the longest path from  $n_i$  to a leaf
  - » All leaves have height 0
  - » The height of the tree is the height of the root

# Paths

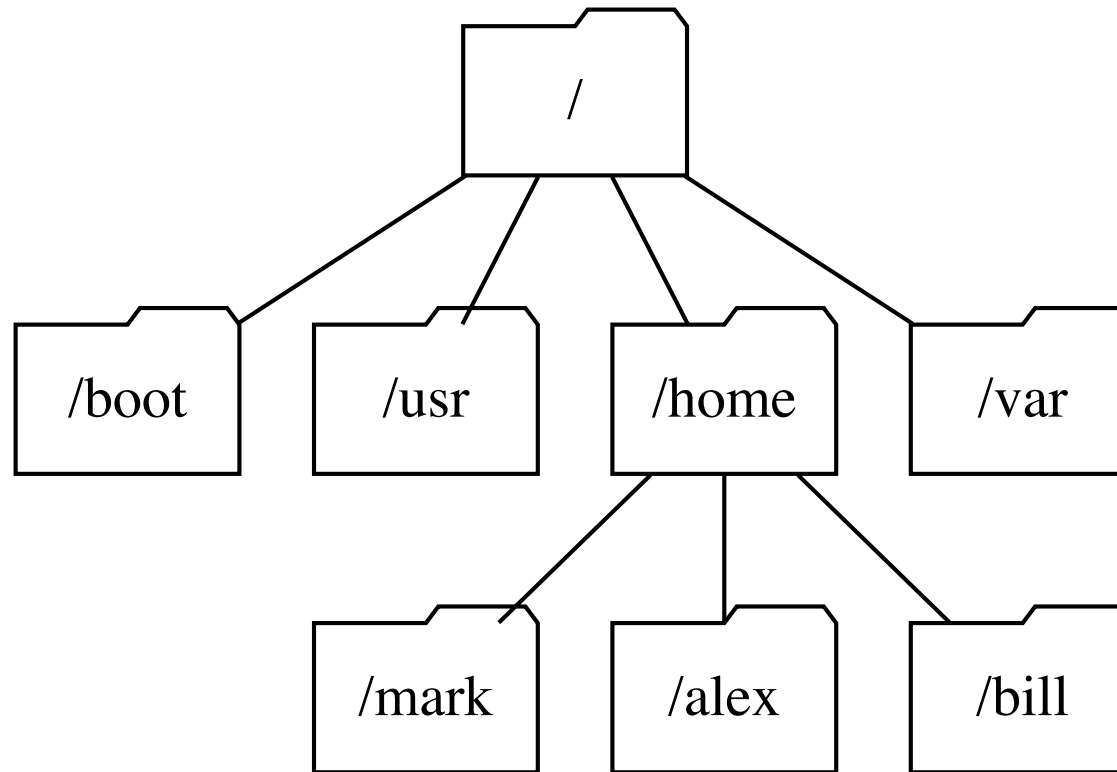
- » If there is a path from  $n_i$  to  $n_j$  then
  - »  $n_i$  is an *ancestor* of  $n_j$
  - »  $n_j$  is a *descendant* of  $n_i$
- » If  $n_i \neq n_j$  then they are *proper*, e.g., *proper ancestor*

# Example



- » *e* is the *root*
- » There is a *path*, *e, j, q* from *e* to *q* of *length 2*
- » The *depth* of *i* is 1 and the *height* is 0
- » *j* is a *proper ancestor* of *q*

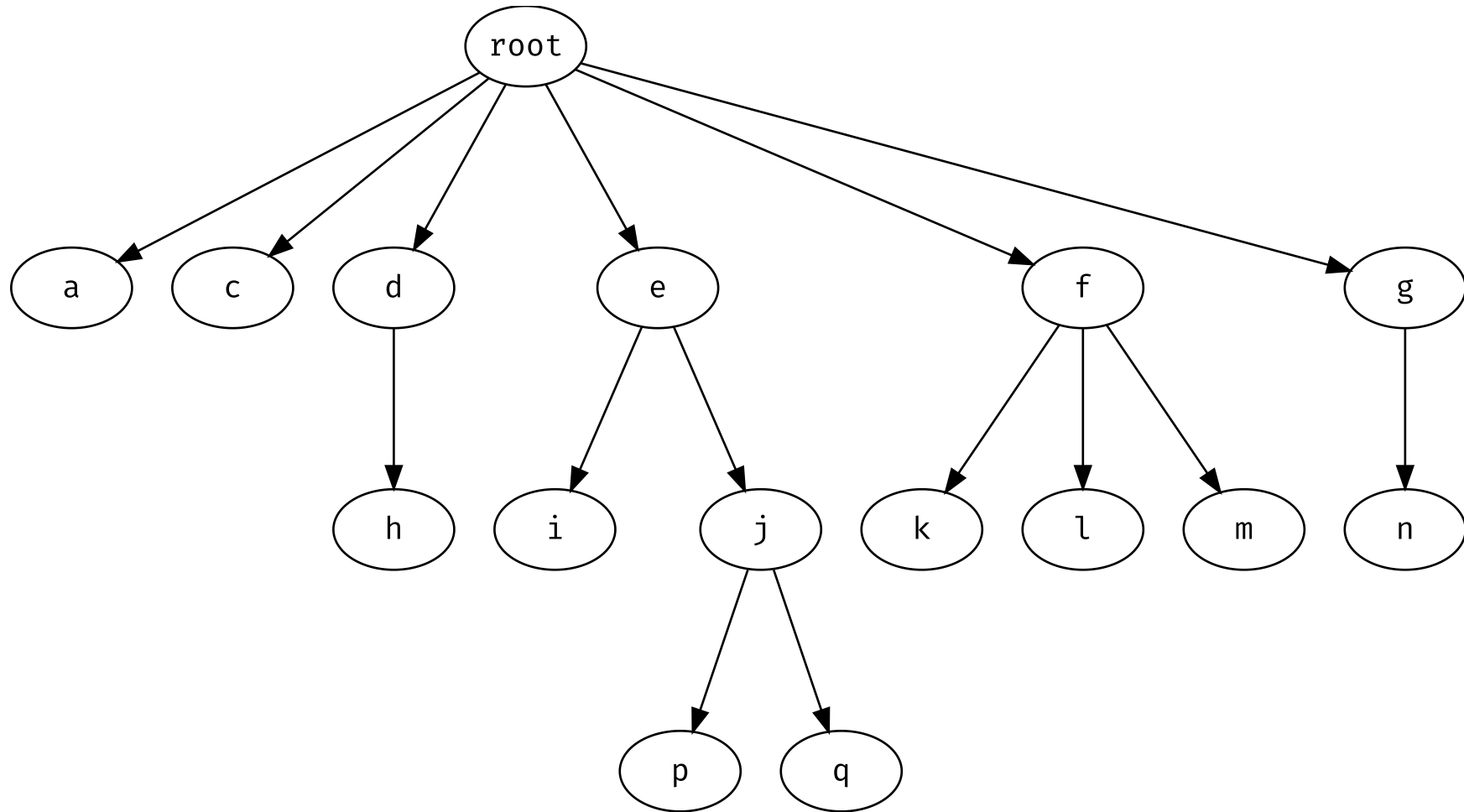
# Example: File systems



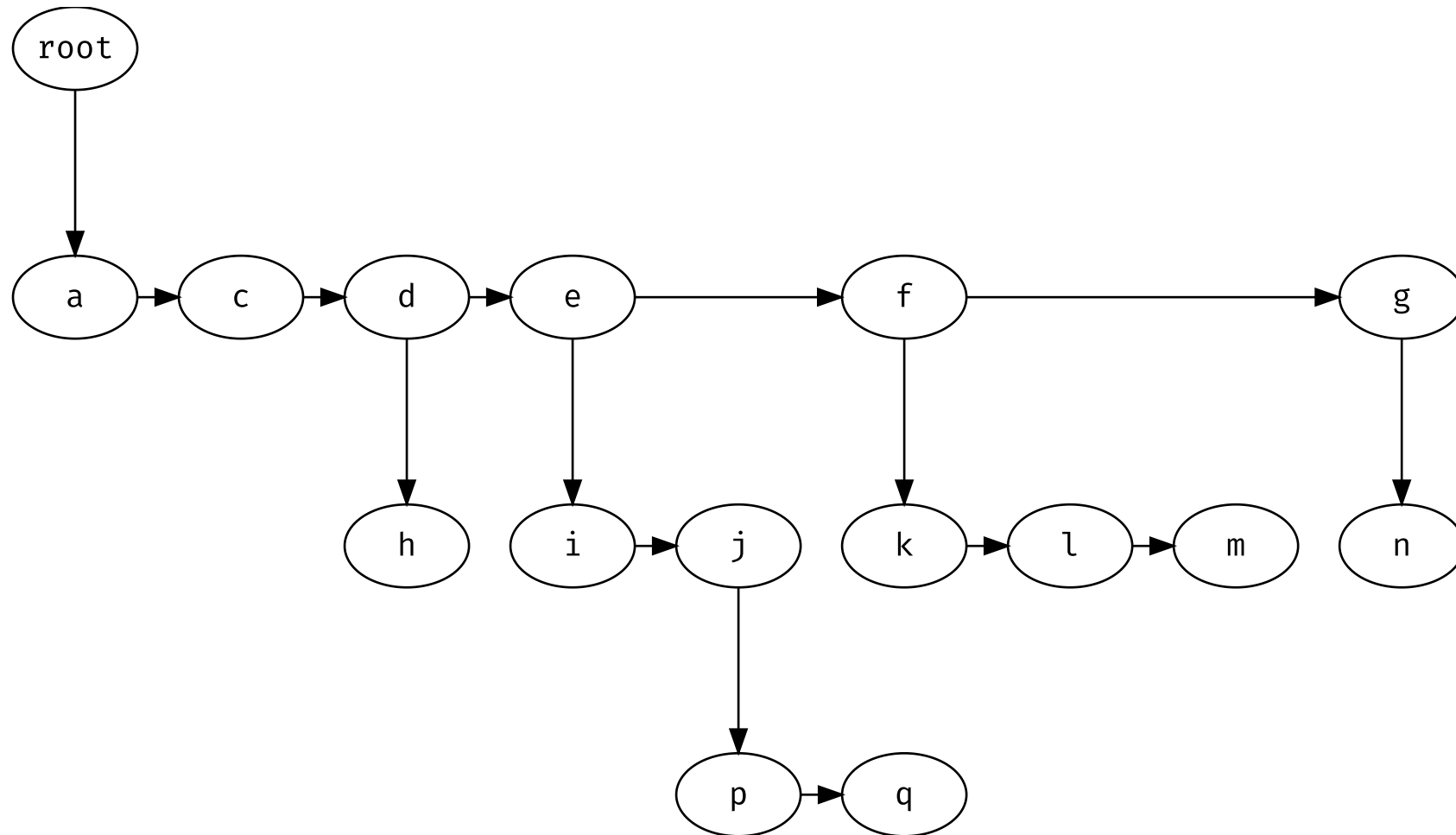
# Implementing a tree

- » A tree as an arbitrary number of nodes
- » A node has an arbitrary number of children
  - » Can vary greatly, so not a great idea to keep references to all children in the node
- » Left-most child, right sibling (also known as First child, next sibling)
- » Keep two pointers in each node
  - » Left child
  - » Right sibling

# Remember the tree



# Left-child-right-sibling

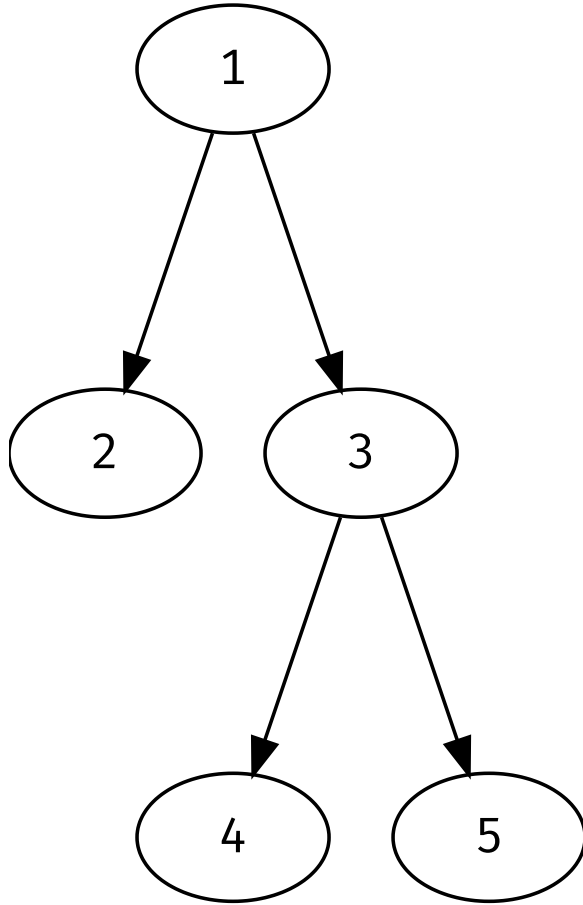


# LCRSNode

```
1 from dataclasses import dataclass
2
3 @dataclass
4 class LCRSNode:
5     key: int
6     left: 'LCRSNode | None' = None
7     right: 'LCRSNode | None' = None
```



# Creating a tree



```
1 r = LCRSNode(1)
2 r.left = LCRSNode(2)
3 r.left.right = LCRSNode(3)
4 r.left.right.left = LCRSNode(4)
5 r.left.right.left.right = \
6     LCRSNode(5)
```

# Walking the tree

```
1 def walk(root:LCRSNode) -> None:
2     if root is not None:
3         print(root.key)
4         walk(root.left)
5         walk(root.right)
```

# Does it work?

```
1 walk(r)
```

1

2

3

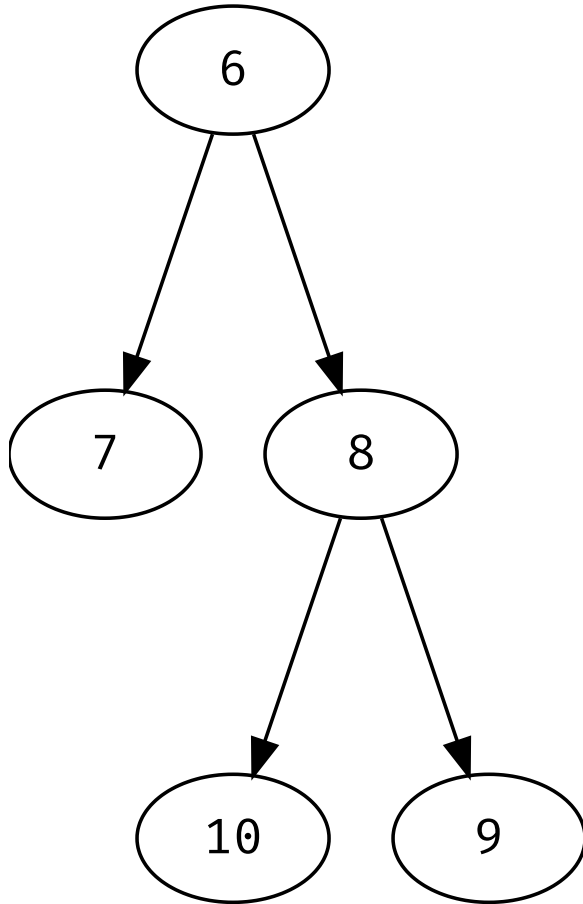
4

5

# Adding children

```
1 from fastcore.basics import patch
2
3 @patch
4 def add_child(self:LCRSNode, key:int) -> LCRSNode:
5     if self.left is None:
6         self.left = LCRSNode(key)
7         return self.left
8     else:
9         p = self.left
10        while p.right is not None:
11            p = p.right
12        p.right = LCRSNode(key)
13        return p.right
```

# Rewriting our example



```
1 r = LCRSNode(6)
2 _ = r.add_child(7)
3 t = r.add_child(8)
4 _ = t.add_child(10)
5 _ = t.add_child(9)
```

# Does it work?

```
1 walk(r)
```

6

7

8

10

9

# Patching in walk

```
1 @patch
2 def walk(self:LCRSNode) -> None:
3     print(self.key)
4     if self.left is not None:
5         self.left.walk()
6     if self.right is not None:
7         self.right.walk()
```

# Does it work?

```
1 r.walk()  
2 print( '---' )  
3 r.left.right.walk()
```

6

7

8

10

9

---

8

10

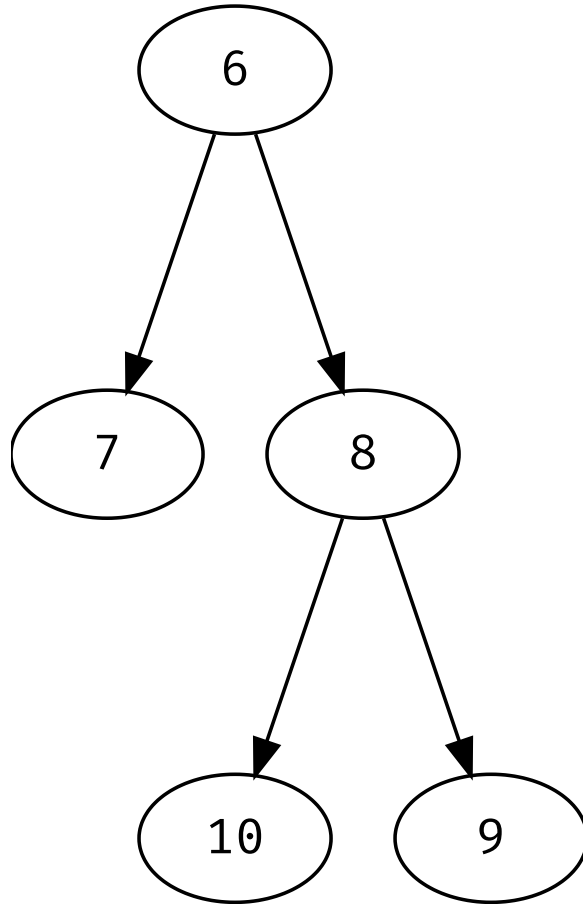
9



# Node degree (number of children)

```
1 @patch(as_prop=True)
2 def degree(self:LCRSNode) -> int:
3     s, p = 0, self.left
4     while p is not None:
5         s += 1
6         p = p.right
7     return s
```

# Checking

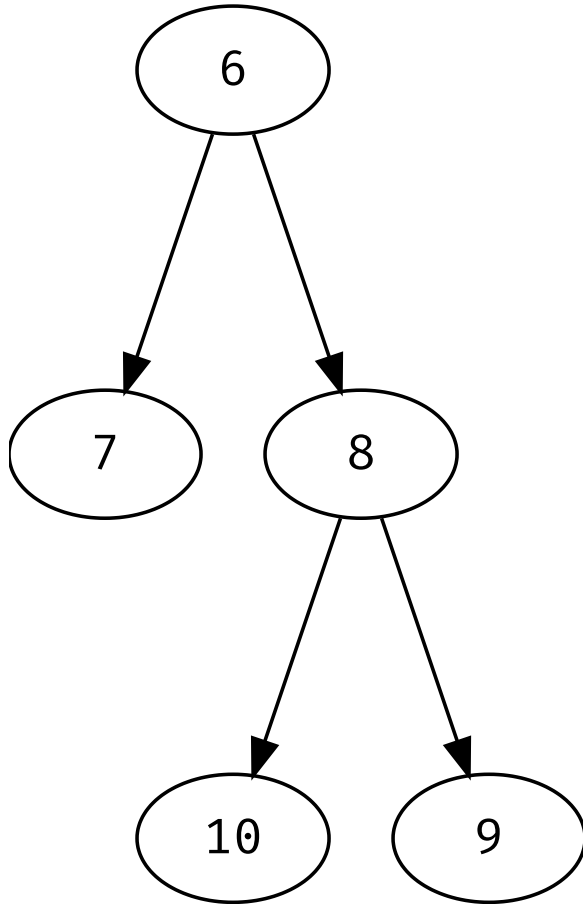


```
1 assert r.degree == 2
2 assert r.left.degree == 0
3 assert r.left.right.degree == 2
```

# Is a node a leaf?

```
1 @patch(as_prop=True)
2 def is_leaf(self:LCRSNode) -> bool:
3     return self.left is None
```

# Checking

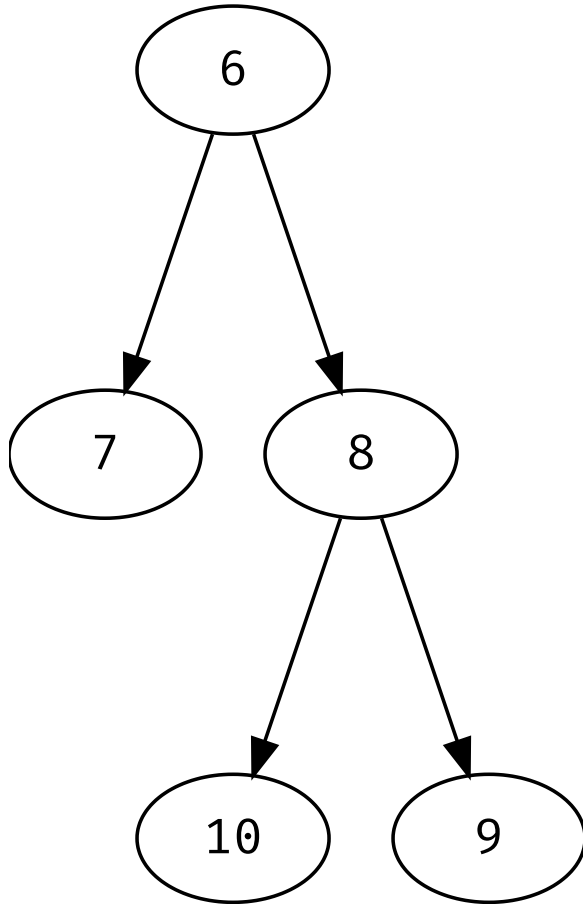


```
1  assert not r.is_leaf
2  assert r.left.is_leaf
3  assert not r.left.right.is_leaf
```

# Getting the $n^{\text{th}}$ child

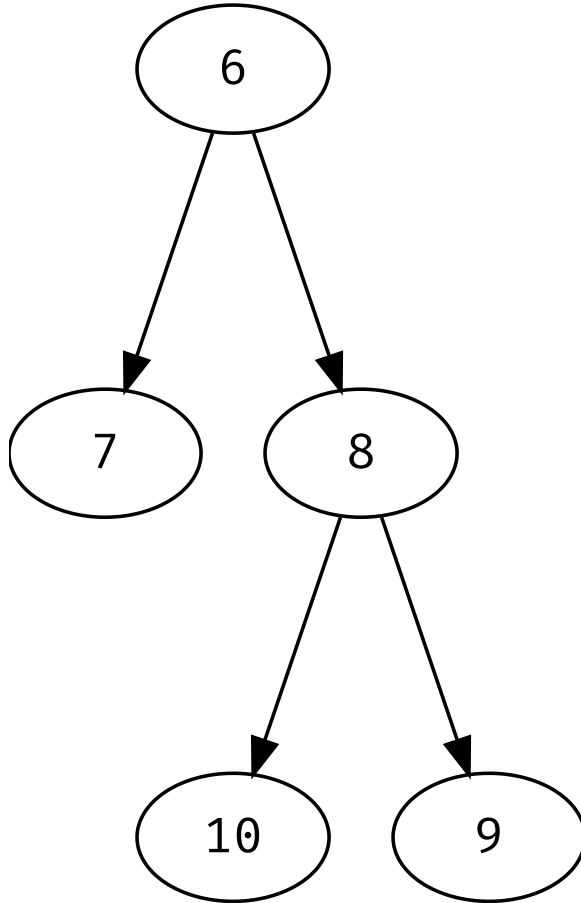
```
1 @patch
2 def __getitem__(self:LCRSNode, key:int) -> LCRSNode:
3     c, p = 0, self.left
4     while p is not None:
5         if c == key:
6             return p
7         c += 1
8         p = p.right
9     raise IndexError
```

# Checking



```
1  assert not r.is_leaf
2  assert r[0].is_leaf
3  assert not r[1].is_leaf
4  assert r[1][0].is_leaf
```

# Boom



```
1 try:
2     r[2]
3 except Exception as e:
4     assert \
5         isinstance(e, IndexError)
```

# Size and height

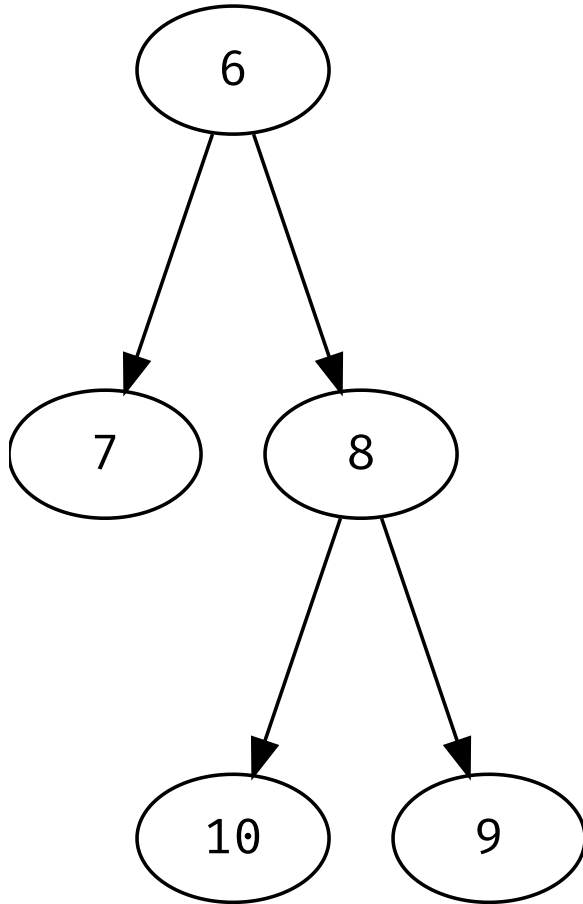
```
1 @patch(as_prop=True)
2 def size(self:LCRSNode) -> int:
3     l, r = 0, 0
4     if self.left is not None:
5         l = self.left.size
6     if self.right is not None:
7         r = self.right.size
8
9     return 1 + l + r
```



# Size and height

```
1 @patch(as_prop=True)
2 def height(self:LCRSNode) -> int:
3     h = 0
4     p = self.left
5     while p is not None:
6         h = max(h, 1 + p.height)
7         p = p.right
8     return h
```

# Checking



```
1  assert r.size == 5
2  assert r.height == 2
3  assert r[0].height == 0
4  assert r[1].height == 1
```

# The big tree

## ► Code

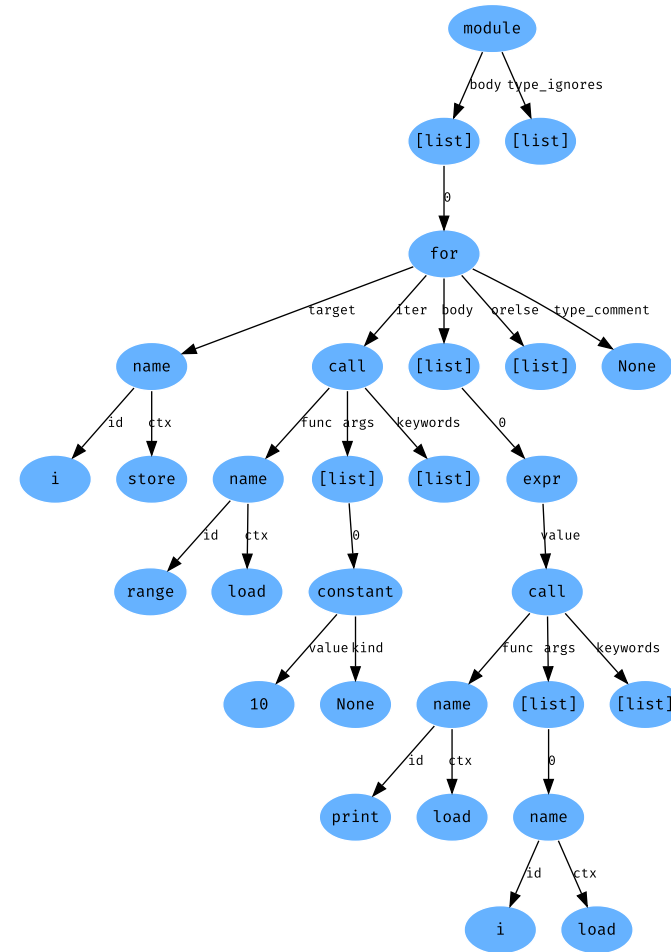
```
r.height=3, should be 3  
r.degree=6, should be 6  
r.size=16, should be 16  
r[3].height=2 (e), should be 2  
r[4].degree=3 (f), should be 3
```

# Trees

- » Many uses in computer science
  - » File/directory structure
  - » HTML/DOM
  - » Parse tree
  - » ...

# Example

```
1 for i in range(10):
2     print(i)
```

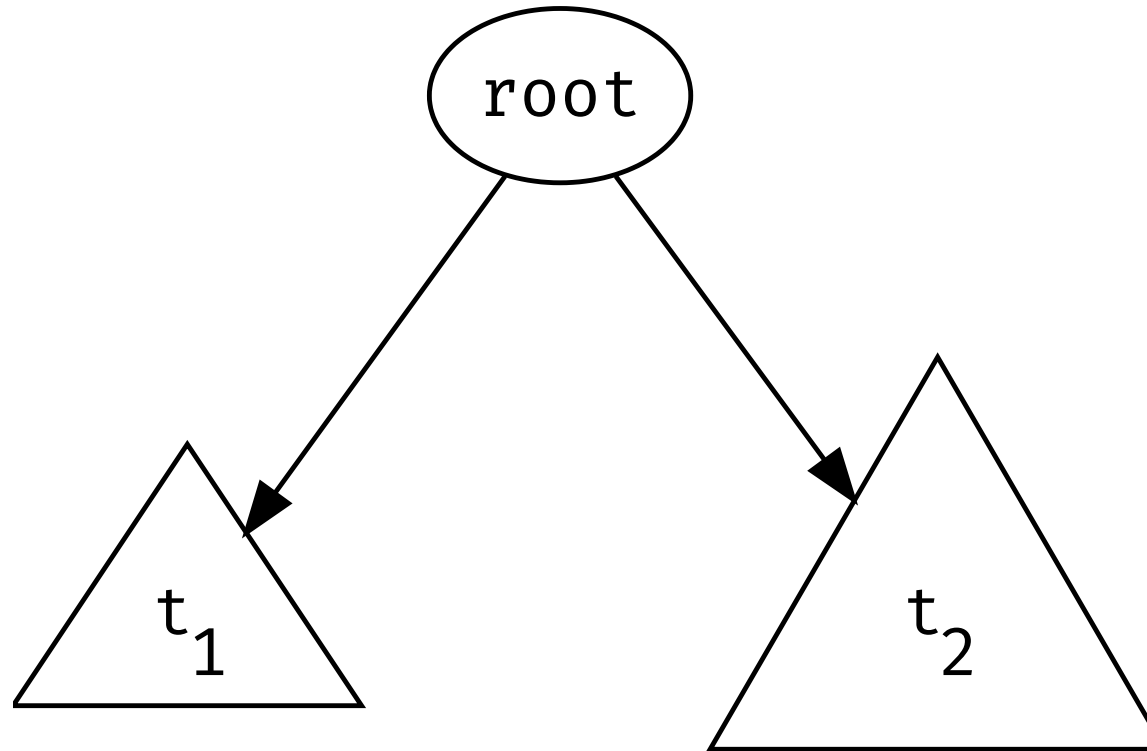


# Binary Search Trees

# Binary trees

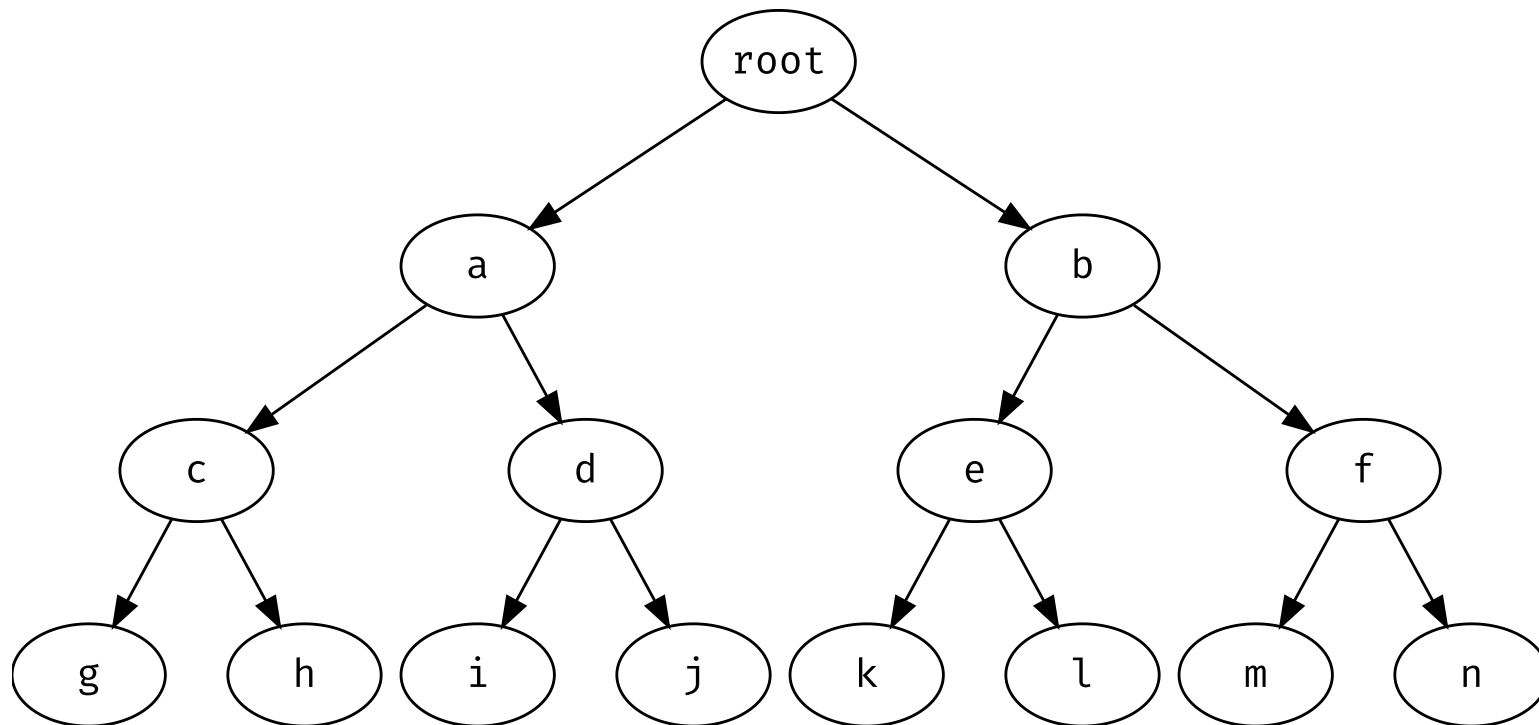
- » A binary tree is a tree where each node has at most two children
- » Since only two, each node can hold points to all its children
- » We can reason about height:
  - » an average binary tree has height  $\Theta(\sqrt{n})$  (says the book)
  - » a “full” tree has height  $\lceil \log_2(n) \rceil - 1$
  - » a “degenerate” tree has height  $n - 1$

# A binary tree

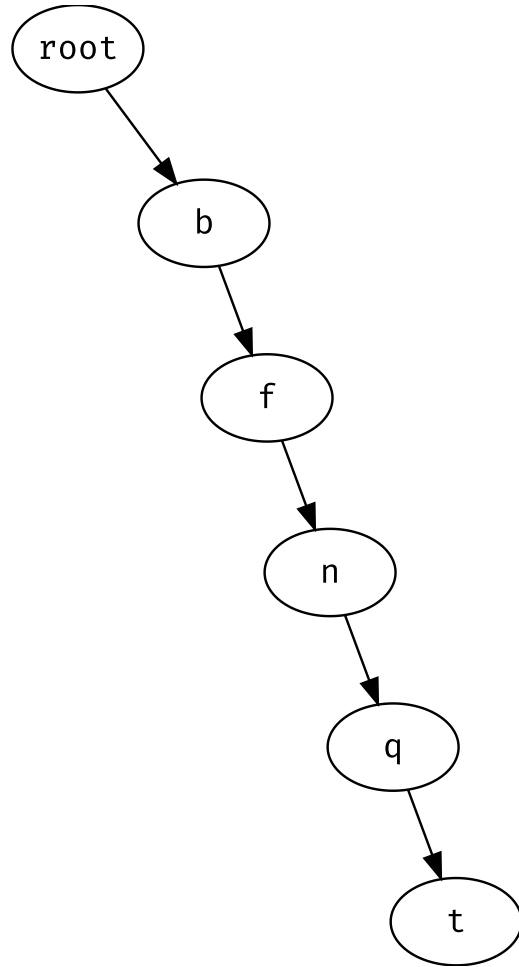




# A full tree



# A “degenerate” tree



- » A degenerate tree becomes a linked list
- » The height is  $n - 1$  compared to  $\lceil \log_2(n + 1) \rceil$  for a full tree
  - » Height of the example is 5
  - » If “full”, it would be 3
- » Will be important in the future

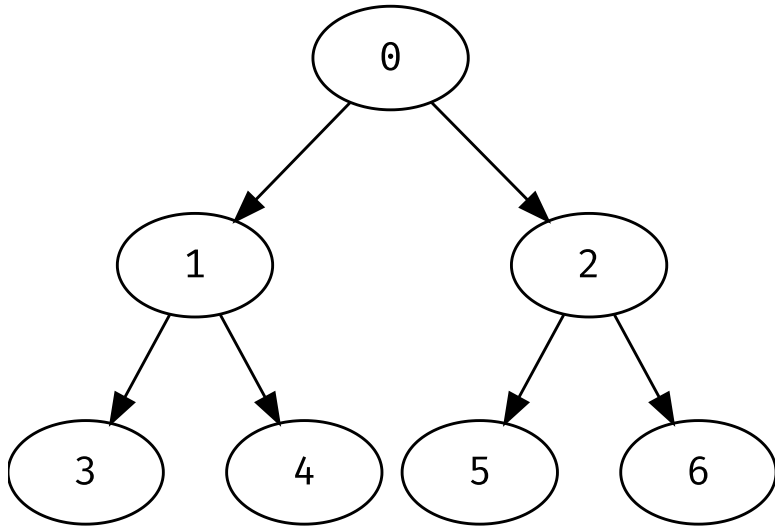
# Implementing a binary tree

```
1 @dataclass
2 class BTreeNode:
3     key: int
4     left: 'BTreeNode|None' = None
5     right: 'BTreeNode|None' = None
```

## Note

This is identical to `LCRSTreeNode`, but we define a new type to avoid confusion

# Building a small tree



```
1 r = BTreeNode(0)
2 r.left = BTreeNode(1)
3 r.right = BTreeNode(2)
4 r.left.left = BTreeNode(3)
5 r.left.right = BTreeNode(4)
6 r.right.left = BTreeNode(5)
7 r.right.right = BTreeNode(6)
```

# Walking the tree

```
1 def inorder(r:BTNode):  
2     if r is None:  
3         return ''  
4     else:  
5         s = inorder(r.left)  
6         s += f' {r.key} '  
7         s += inorder(r.right)  
8         return s
```

# Walking the tree

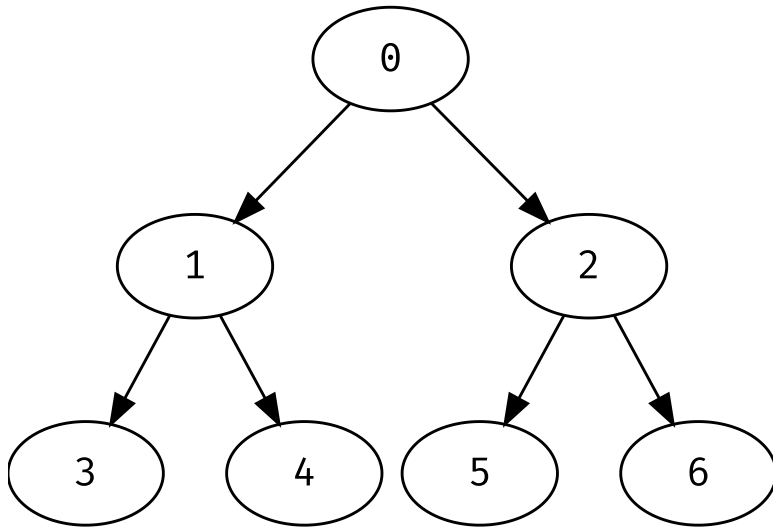
```
1 def preorder(r:BTNode):
2     if r is None:
3         return ''
4     else:
5         s = f' {r.key} '
6         s += preorder(r.left)
7         s += preorder(r.right)
8     return s
```

# Walking the tree

```
1 def postorder(r:BTNode):
2     if r is None:
3         return ''
4     else:
5         s = postorder(r.left)
6         s += postorder(r.right)
7         s += f' {r.key} '
8     return s
```

# Testing on our small tree

in:	3	1	4	0	5	2	6
pre:	0	1	3	4	2	5	6
post:	3	4	1	5	6	2	0





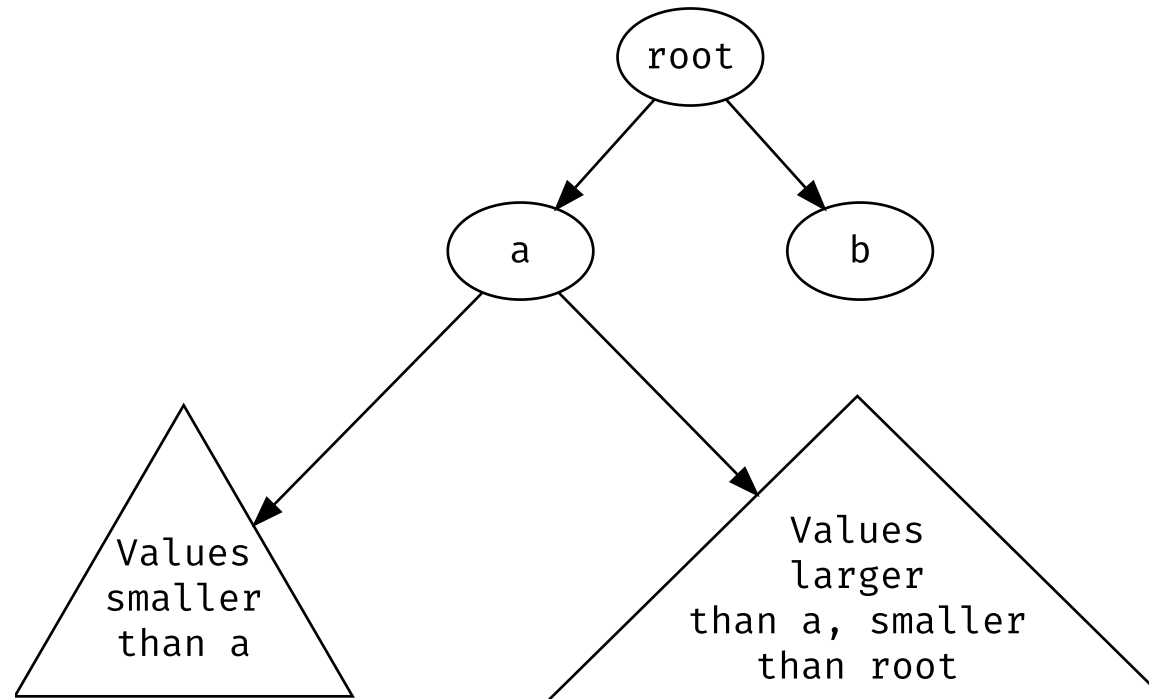
# Creating a tree class

```
1 class BST:
2     def __init__(self) -> None:
3         self.root = None
```

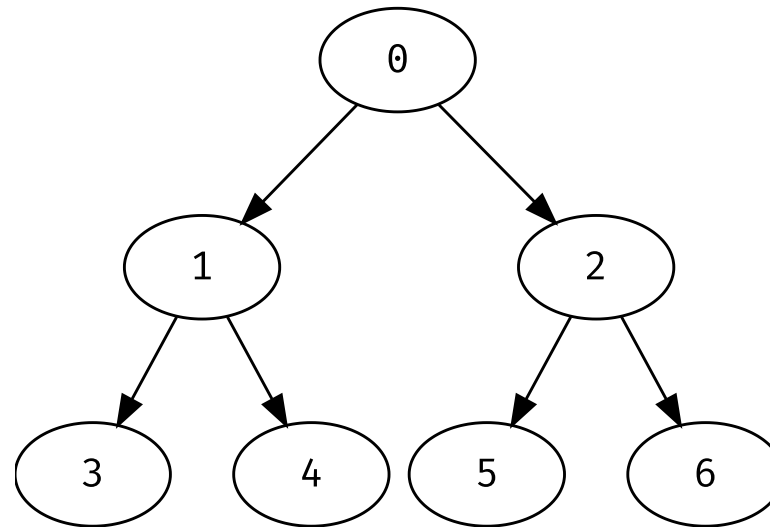
# How do we insert?

- » If we insert a value, where do we place it?
  - » Easy in list
  - » In tree, left or right?
- » Simple idea, put smaller to the left and larger to the right
  - » Binary search tree (BST)

# Binary search tree



# Not a BST!



# Recursive insert

```
1 @dataclass
2 class LLNode:
3     key: int
4     nxt: 'LLNode|None' = None
```

# Recursive insert

```
1 def lladd(l:LLNode|None, key:int) -> LLNode:
2     if l is None:
3         return LLNode(key)
4     l.nxt = lladd(l.nxt, key)
5     return l
6
7 lst = None
8 lst = lladd(lst, 5)
9 lst = lladd(lst, 7)
10 print(lst)
```

LLNode(key=5, nxt=LLNode(key=7, nxt=None))

# Inserting a value

```
1 @patch
2 def _add(self:BST, n:BTNode|None, key:int) -> BTNode|None:
3     if n is None:
4         return BTNode(key)
5
6     if n.key > key:
7         n.left = self._add(n.left, key)
8     elif n.key < key:
9         n.right = self._add(n.right, key)
10
11     return n
```

# Inserting a value

```
1 @patch
2 def add(self:BST, key:int) -> None:
3     self.root = self._add(self.root, key)
```



# Building a tree

```
1 t = BST()  
2 t.add(5)  
3 t.add(2)  
4 t.add(7)  
5  
6 print(t.root)
```

```
BTNode(key=5, left=BTNode(key=2, left=None, right=None),  
right=BTNode(key=7, left=None, right=None))
```

# We need methods to walk the tree!

```
1 @patch
2 def _inorder(self:BST, n:BTNode|None) -> None:
3     if n is not None:
4         self._inorder(n.left)
5         print(n.key)
6         self._inorder(n.right)
```

# We need methods to walk the tree!

```
1 @patch
2 def print_inorder(self:BST) -> None:
3     self._inorder(self.root)
```

# Building a tree

```
1 t = BST()  
2 t.add(5)  
3 t.add(2)  
4 t.add(7)  
5  
6 t.print_inorder()
```

2  
5  
7

# And to check if a value exists

```
1 @patch
2 def _contains(self:BST, n:BTNode|None, key:int) -> bool:
3     if n is None:
4         return False
5
6     if n.key > key:
7         return self._contains(n.left, key)
8     elif n.key < key:
9         return self._contains(n.right, key)
10    else:
11        return True
```

# And to check if a value exists

```
1 @patch
2 def __contains__(self:BST, key:int) -> bool:
3     return self._contains(self.root, key)
```

# Testing

```
1 t = BST()  
2 t.add(5)  
3 t.add(2)  
4 t.add(7)  
5  
6 assert 2 in t  
7 assert 7 in t  
8 assert 8 not in t
```

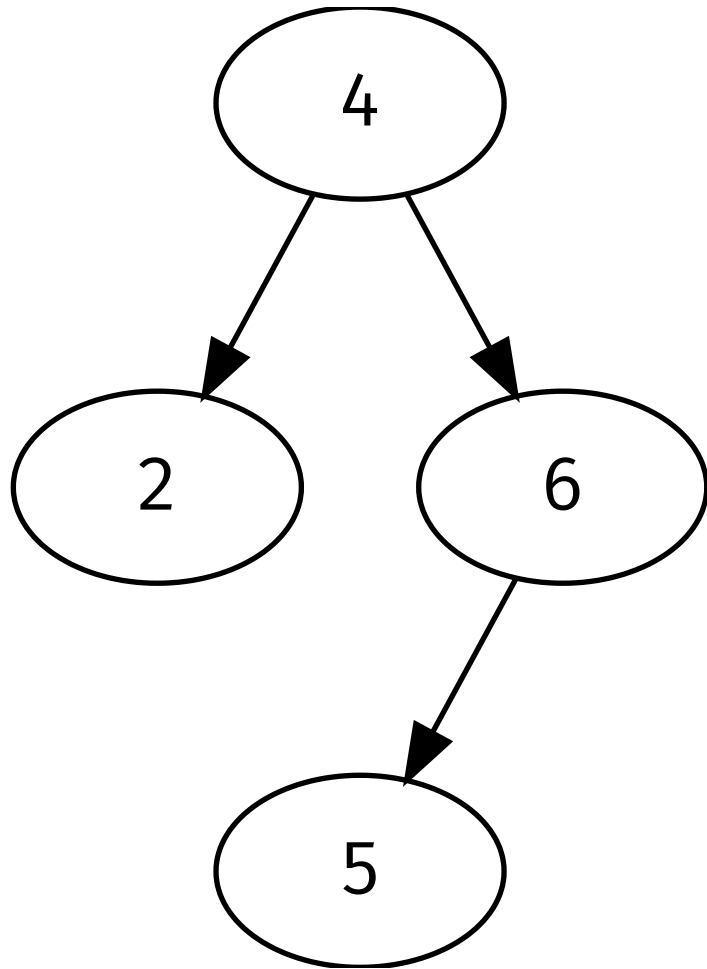
# Recursive delete

```
1 def lldel(l:LLNode|None, key:int) -> LLNode:
2     if l is None:
3         return None
4     if l.key == key:
5         return l.nxt
6     else:
7         l.nxt = lldel(l.nxt, key)
8         return l
9
10 # lst = [5, 7]
11 lst = lldel(lst, 5)
12 print(lst)
```

LLNode(key=7, nxt=None)

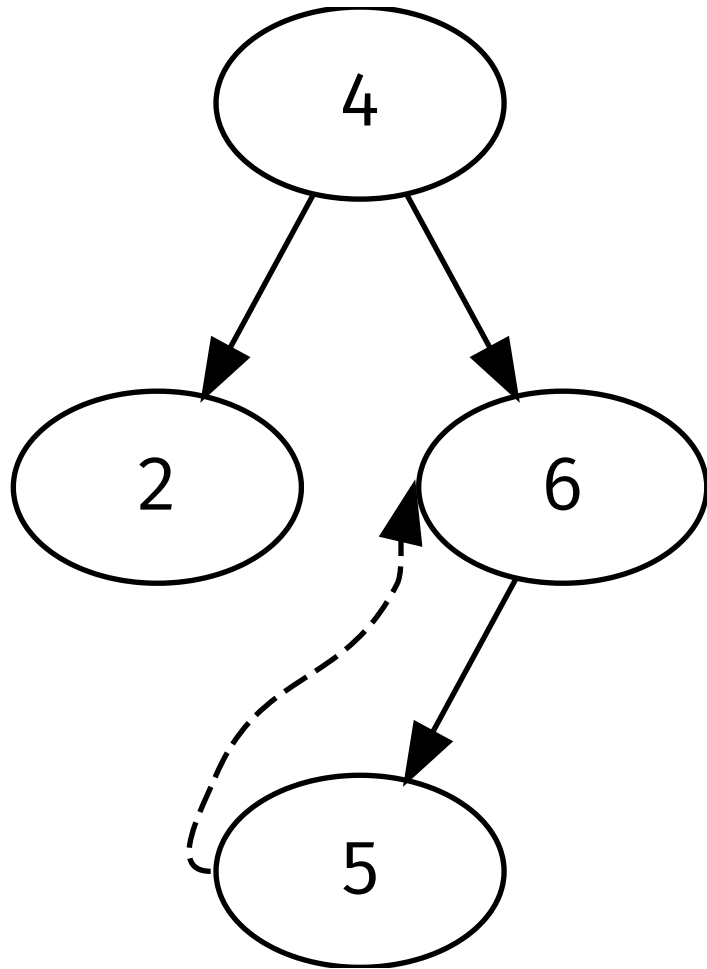


# Deleting



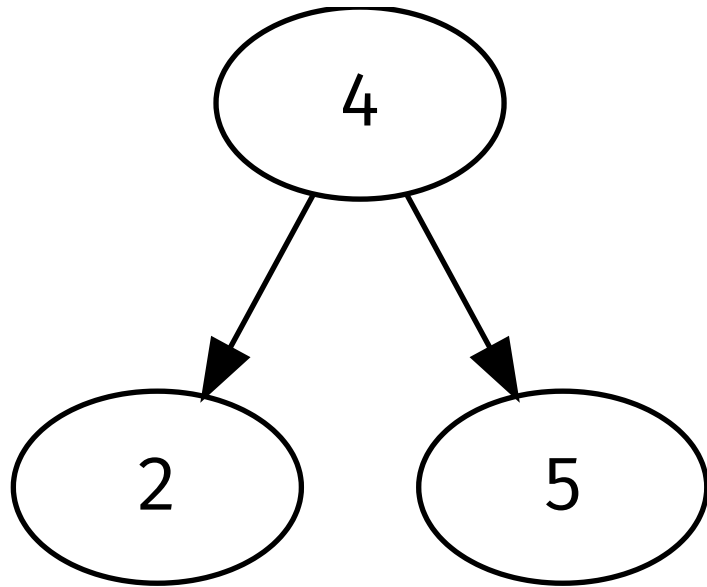
- » Assume we want to delete 6
- » If the node has one child, we “lift” it

# Deleting



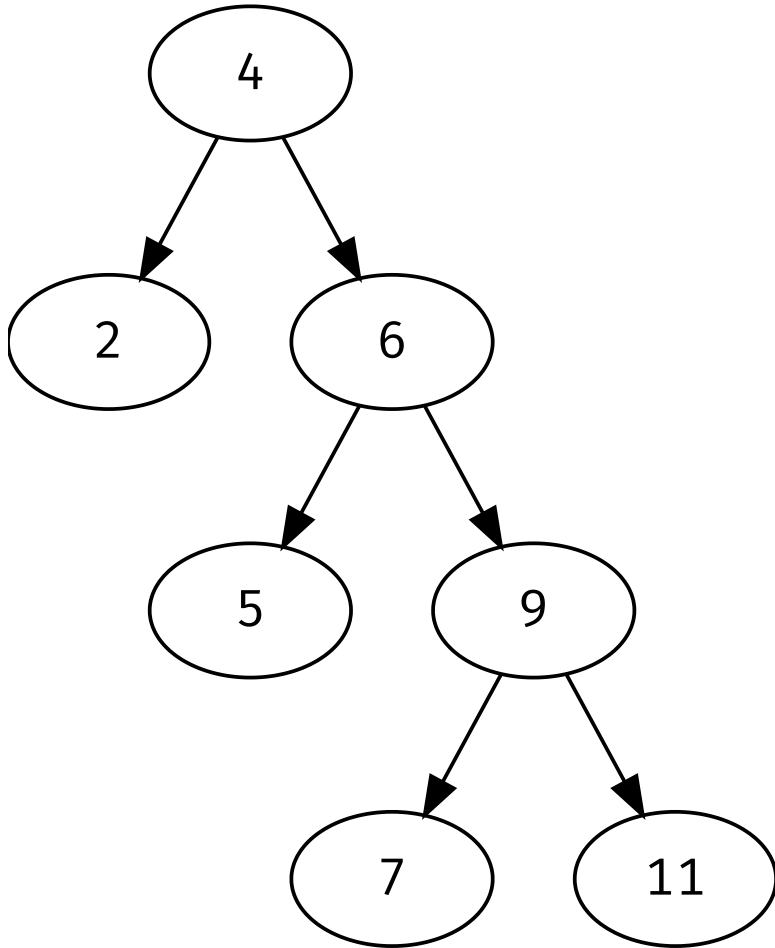
- » Assume we want to delete 6
- » If the node has one child, we “lift” it

# Deleting



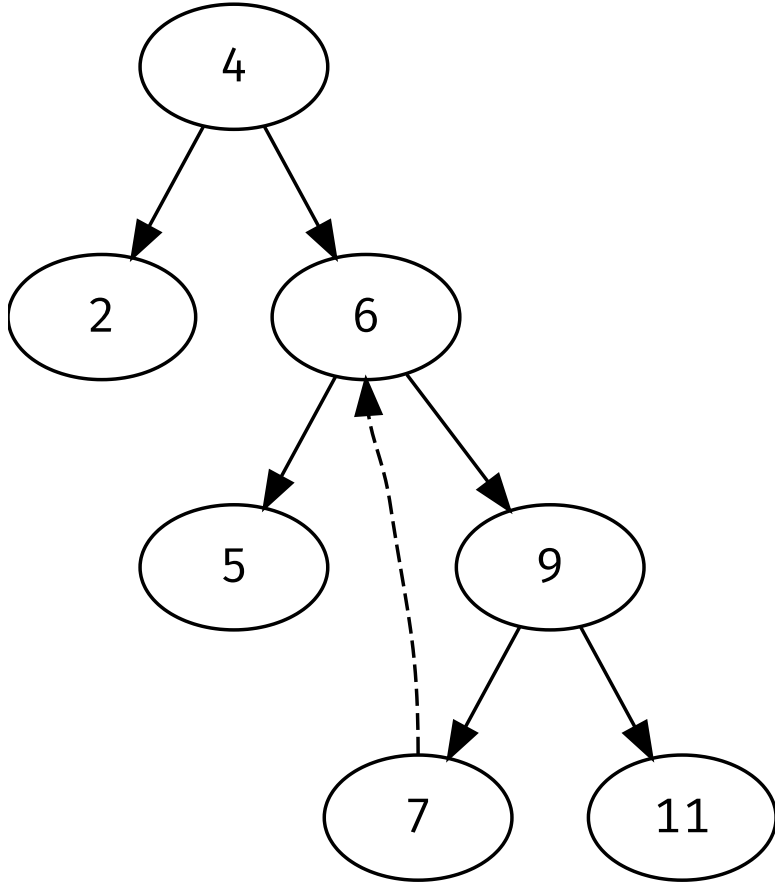
- » Assume we want to delete 6
- » If the node has one child, we “lift” it

# Deleting



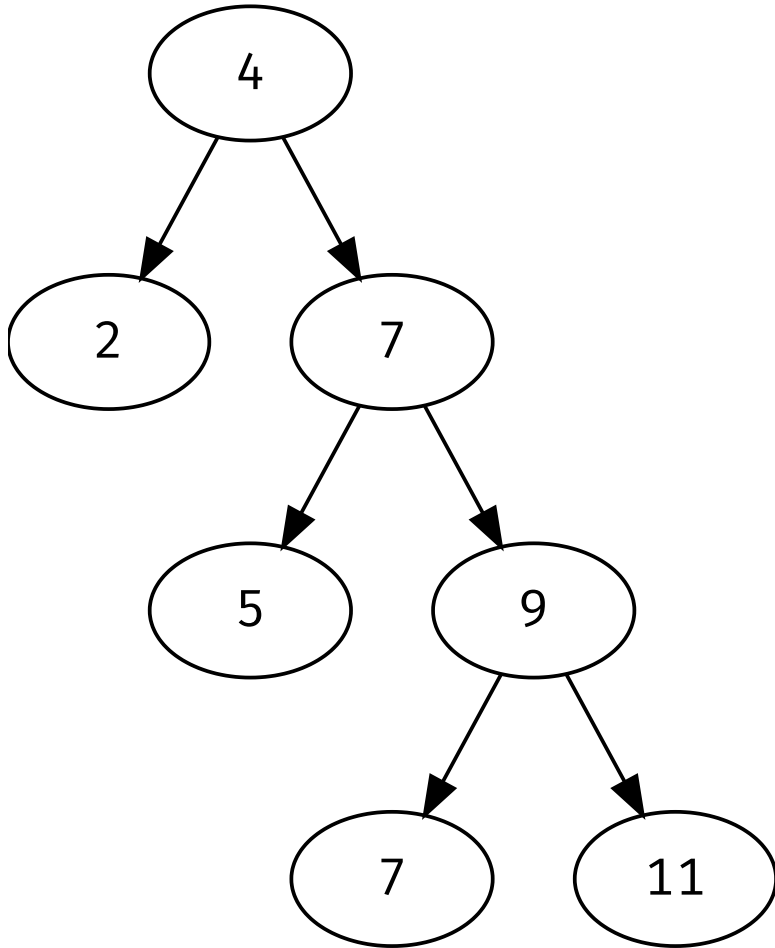
- » Assume we want to delete 6
- » Trickier when it has two children!

# Deleting



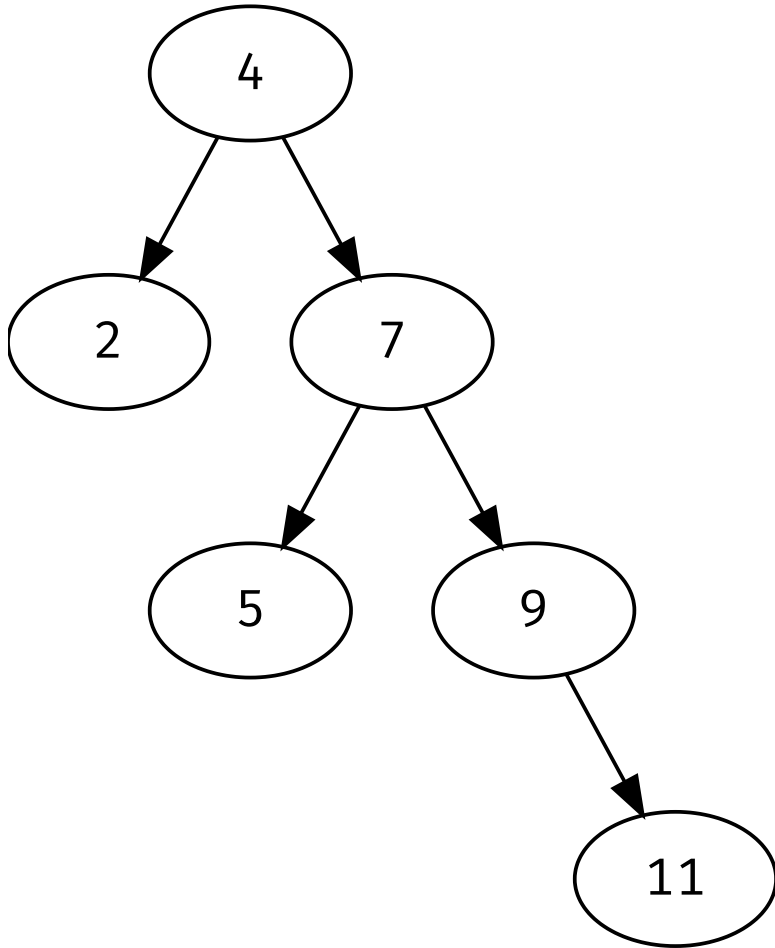
- » Assume we want to delete 6
- » Trickier when it has two children!
- » We replace the node with the smallest value in the right subtree
  - » Cannot have two children

# Deleting



- » Assume we want to delete 6
- » Trickier when it has two children!
- » We replace the node with the smallest value in the right subtree
  - » Cannot have two children

# Deleting



- » Assume we want to delete 6
- » Trickier when it has two children!
- » We replace the node with the smallest value in the right subtree
  - » Cannot have two children

# Deleting

```
1 @patch
2 def _delete(self:BST, n:BTNode|None, key:int) -> BTNode|None:
3     if n is None:
4         return None
5     if n.key > key:
6         n.left = self._delete(n.left, key)
7     elif n.key < key:
8         n.right = self._delete(n.right, key)
9     else:
10        if n.right is None:
11            return n.left
12        if n.left is None:
13            return n.right
14        n.key = self._min(n.right)
15        n.right = self._delete(n.right, n.key)
16    return n
```



# Finding the smallest node in a subtree

```
1 @patch
2 def _min(self:BST, n:BTNode) -> int:
3     if n.left is None:
4         return n.key
5     else:
6         return self._min(n.left)
```

# Deleting

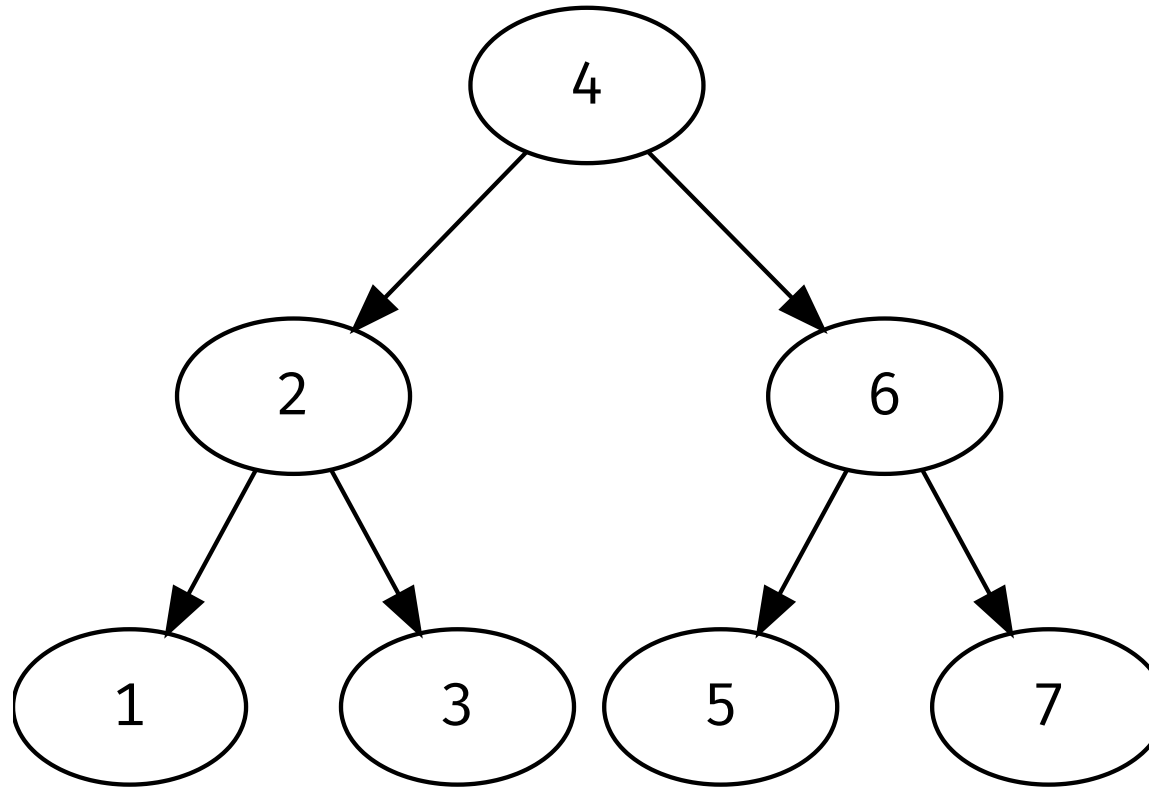
```
1 @patch
2 def delete(self:BST, key:int) -> None:
3     self.root = self._delete(self.root, key)
```

# Building a tree

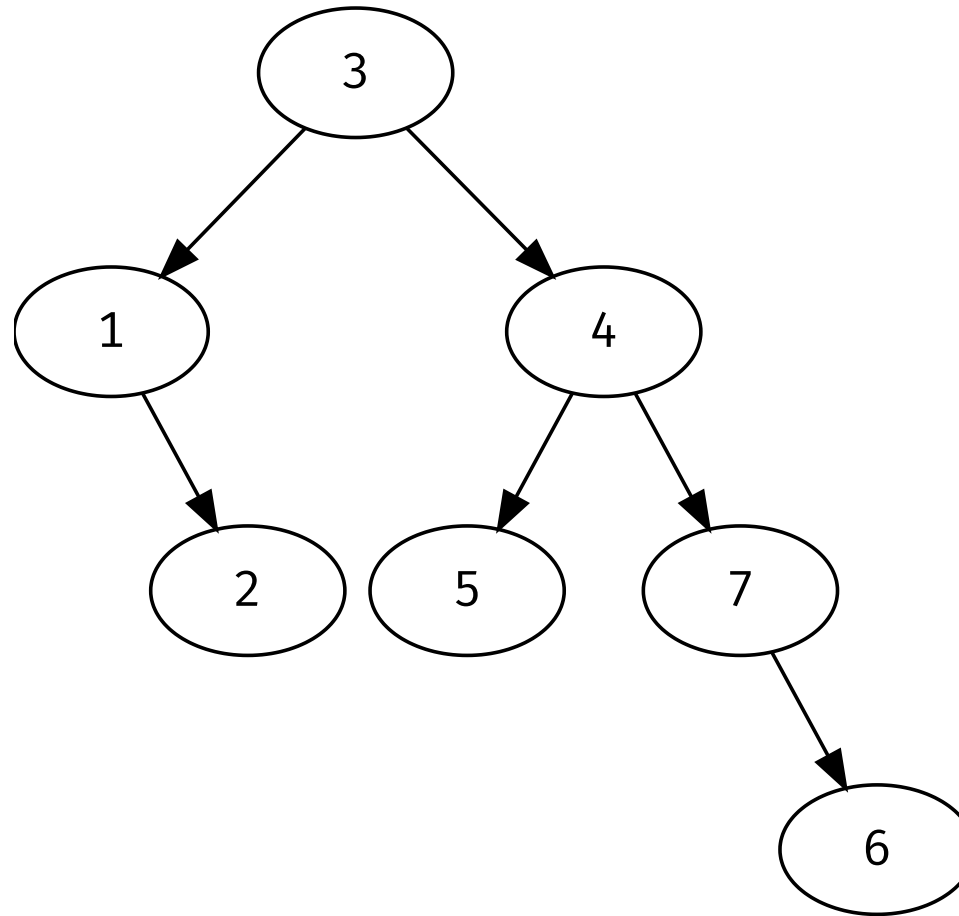
```
1 t = BST()  
2 t.add(5)  
3 t.add(2)  
4 t.add(7)  
5  
6 t.print_inorder()  
7 print('---')  
8 t.delete(2)  
9 t.print_inorder()
```

```
2  
5  
7  
---  
5  
7
```

# Height



# Height



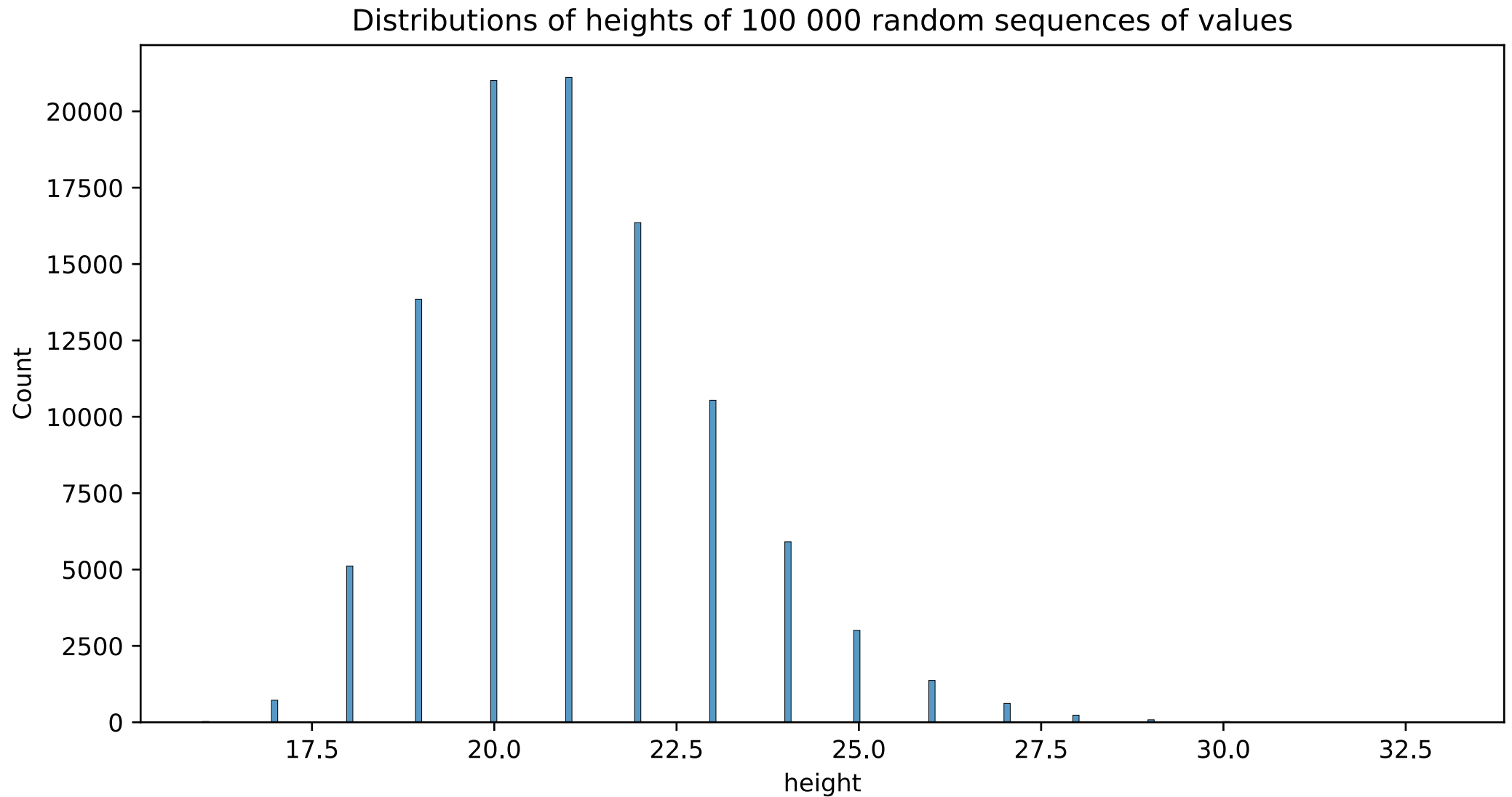
# Computing height

```
1 @patch
2 def _height(self:BST, n:BTNode|None) -> int:
3     if n is None:
4         return -1
5     else:
6         return 1 + max(self._height(n.left), \
7                        self._height(n.right))
```

# What is the height of an average tree

- » We know that best and worst case
- » What is the height of an average tree?

# Example (n = 1023)

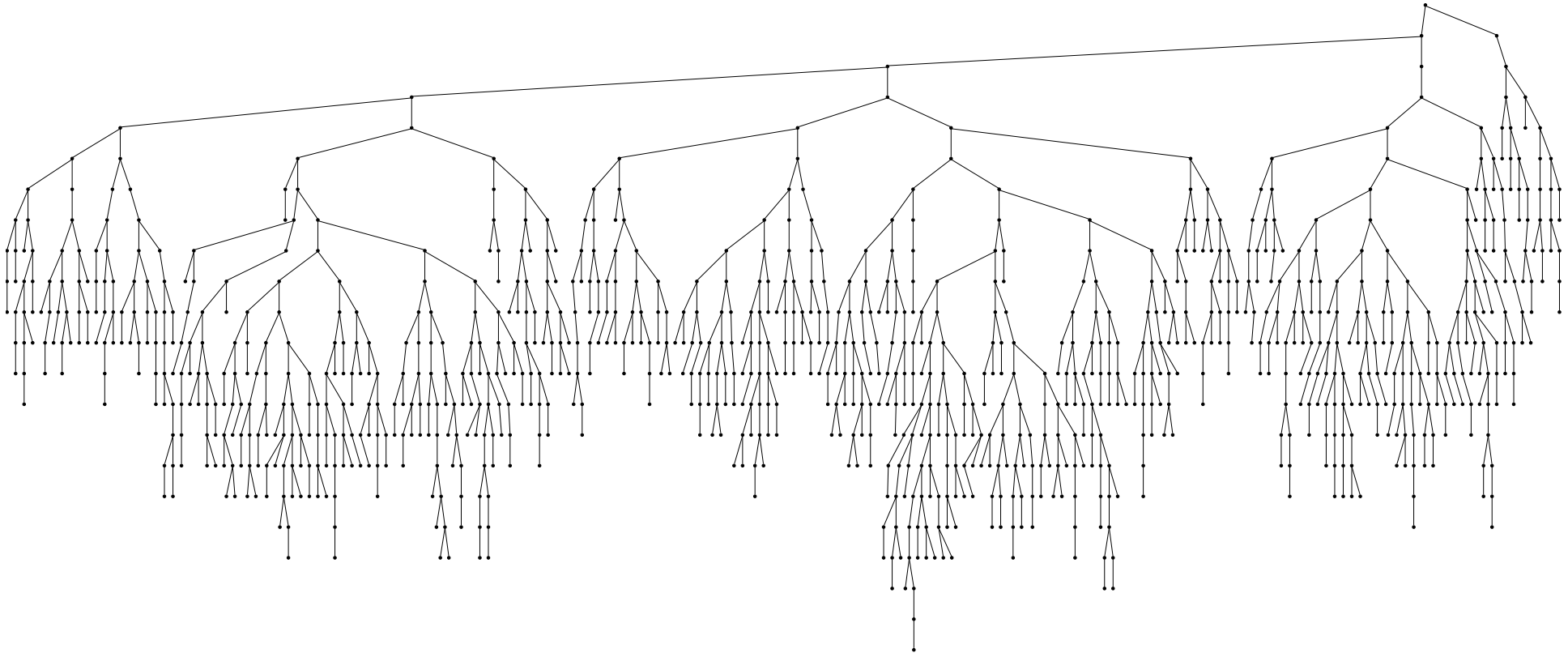




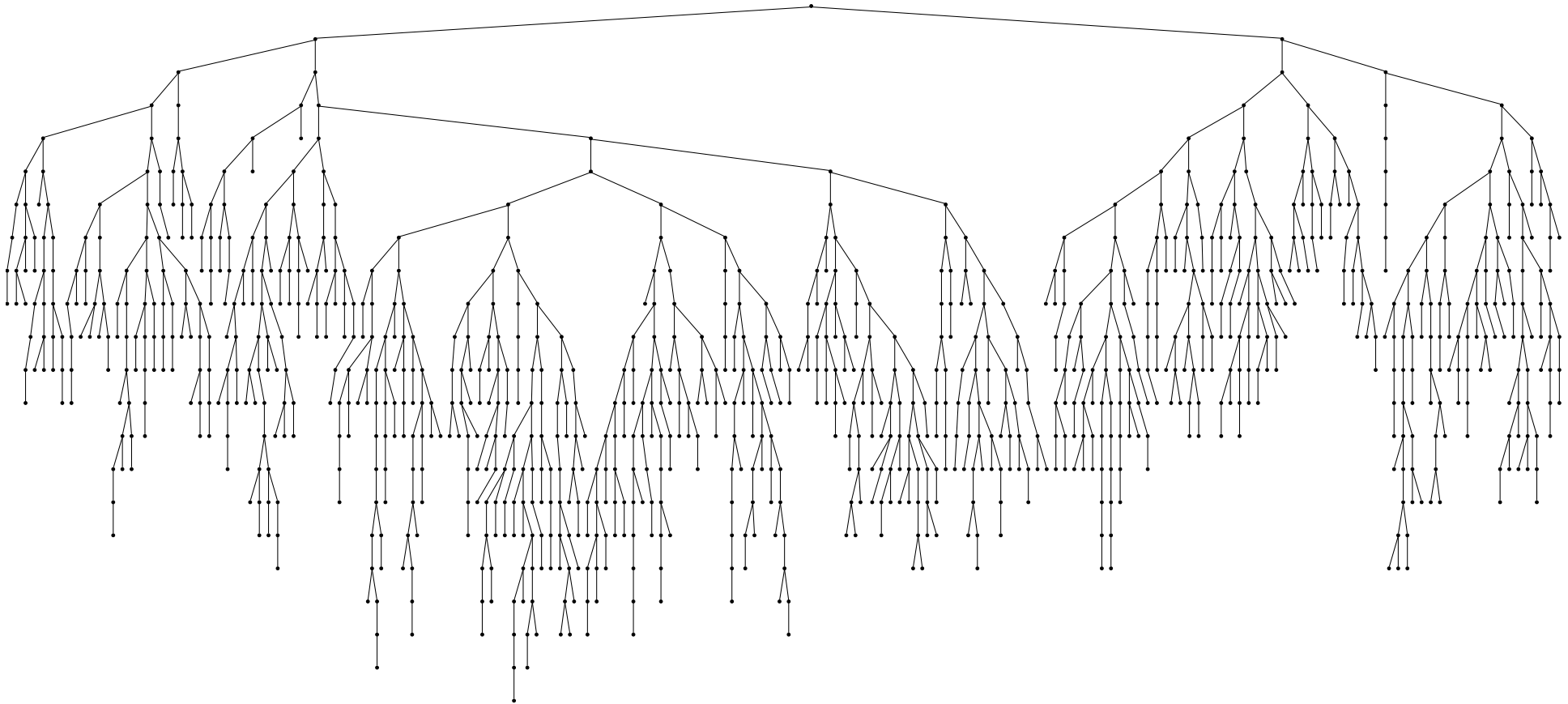
# A balanced tree?

- » The actual heights range from 16 to 33.
- » The best and worst cases are 9 and 1022
- » So, much closer to best than worst
  - » About 2x worse on average

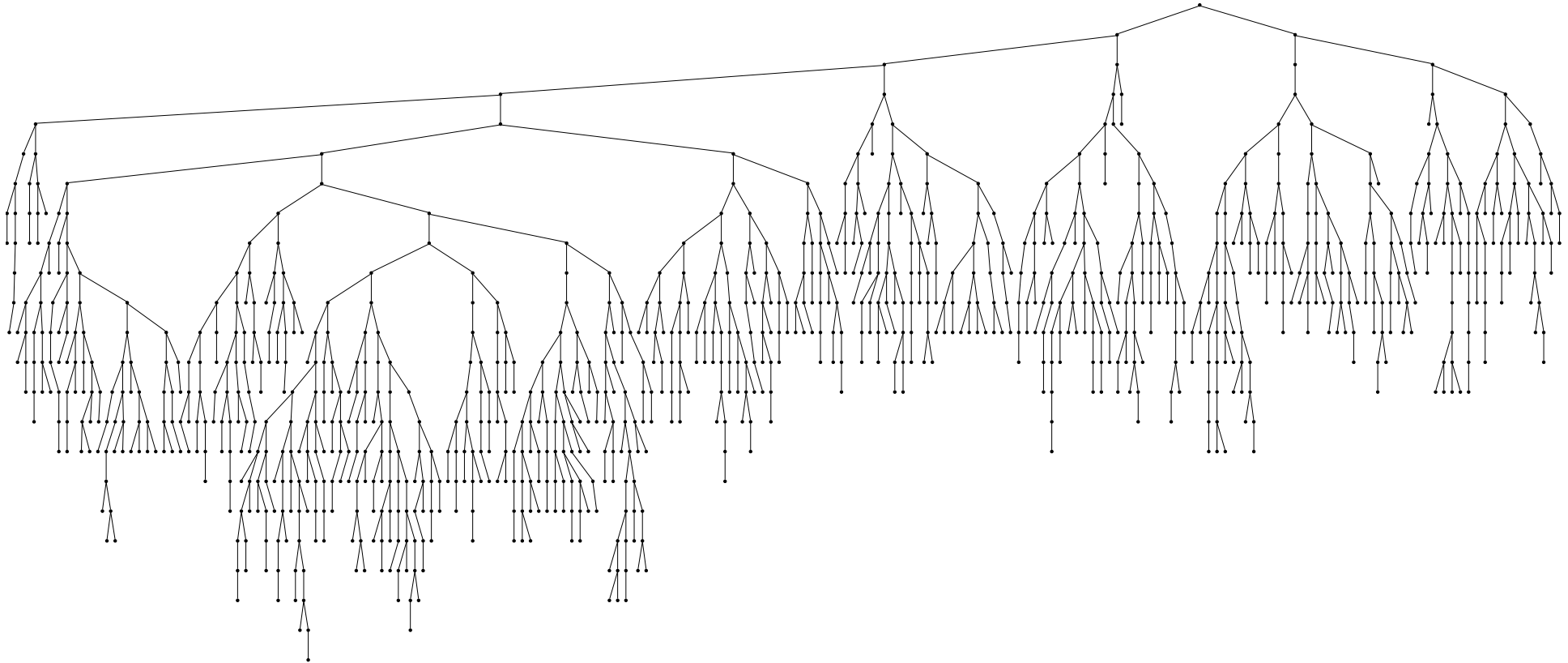
# A balanced tree?



# A balanced tree?

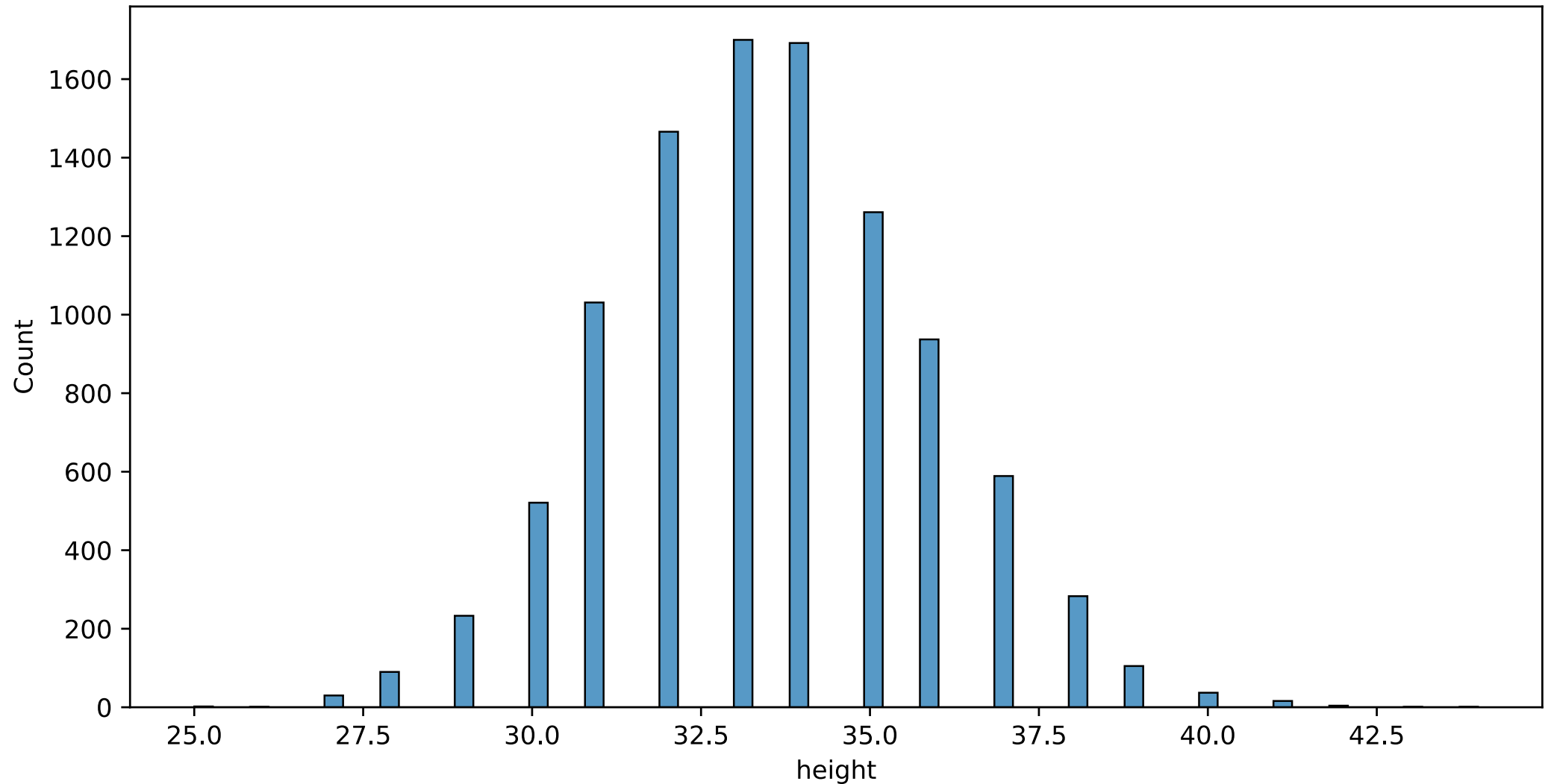


# A balanced tree?

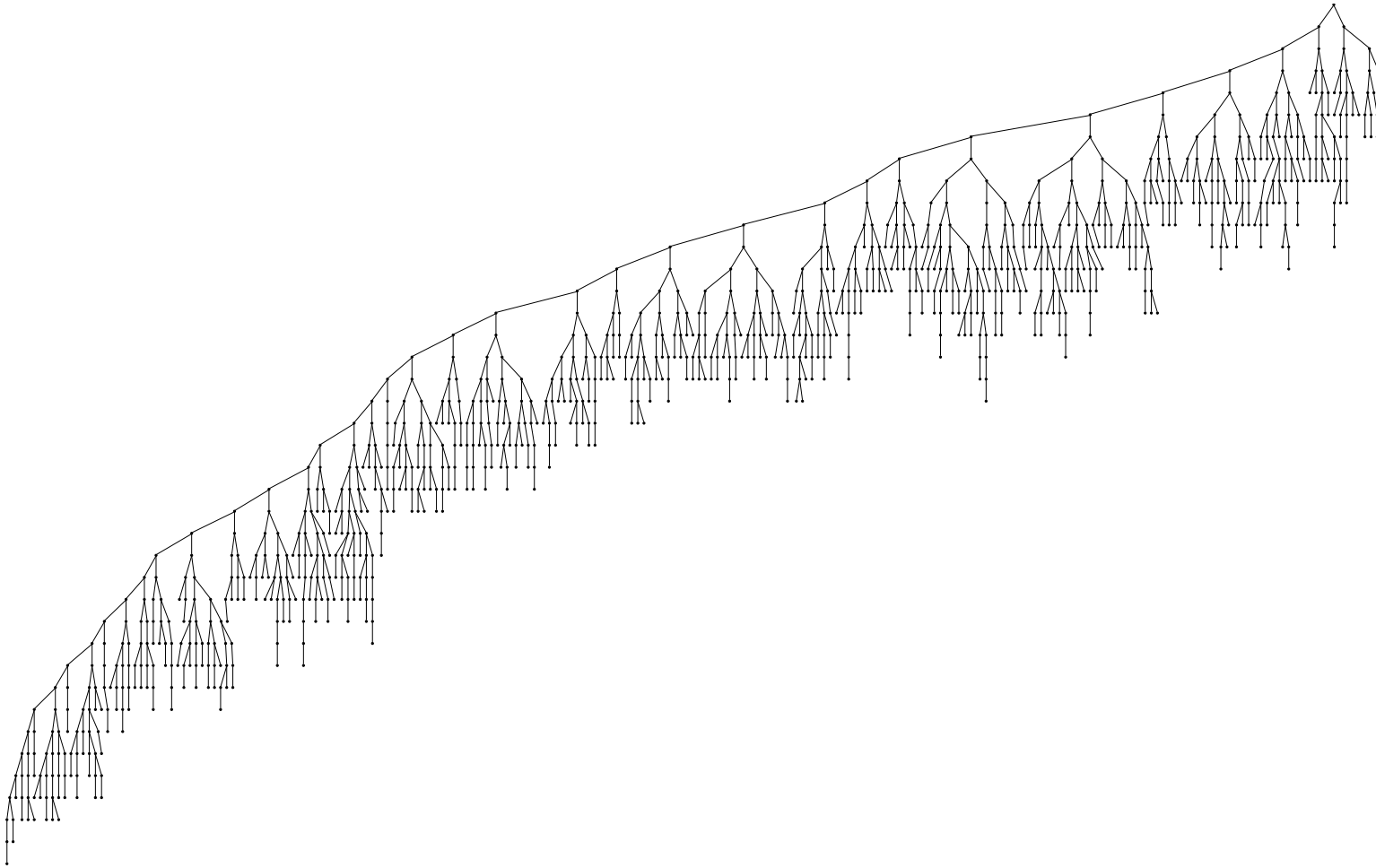


# Adding deletes

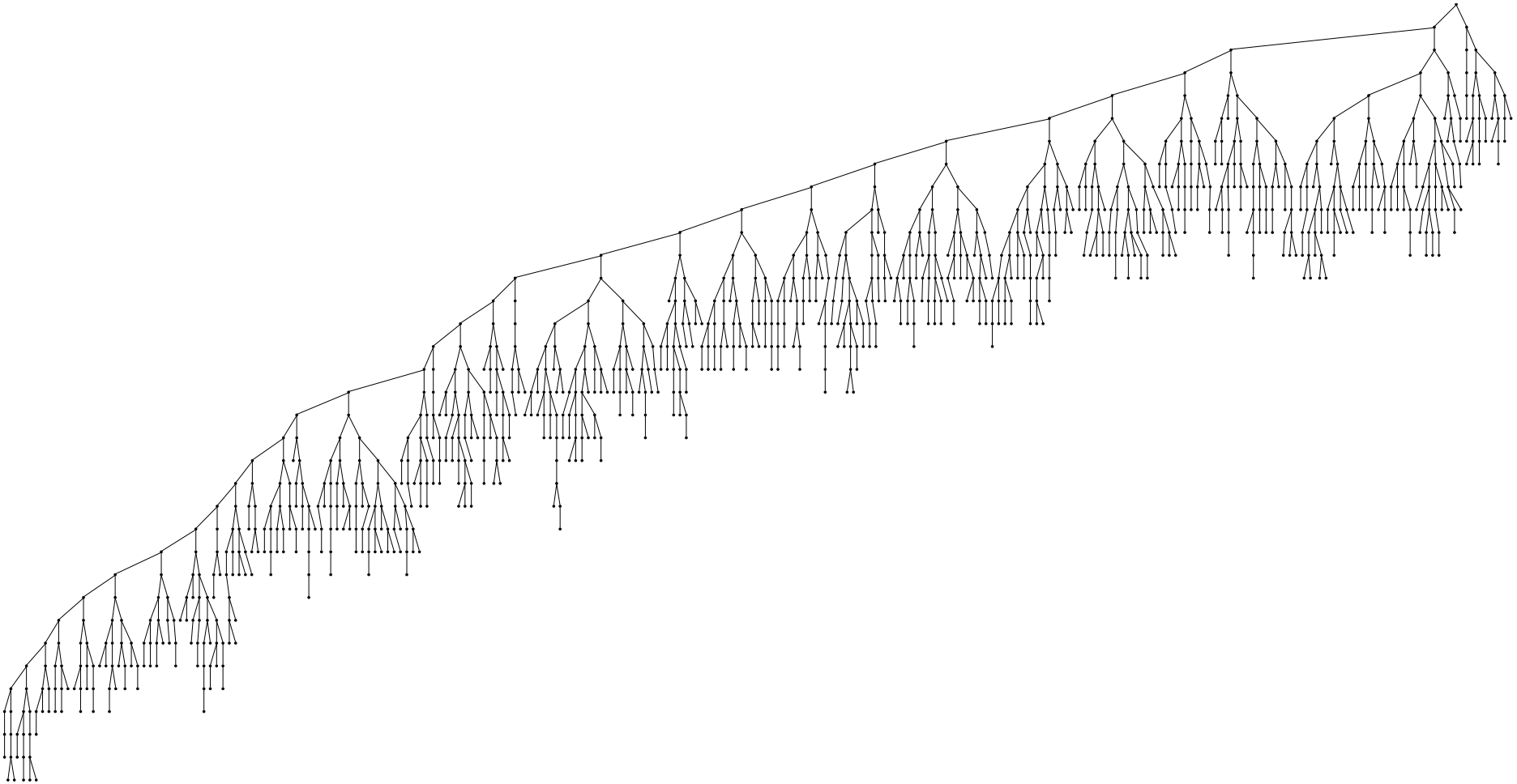
10 000 trees with 256 · 5000 random inserts and deletes



# A balanced tree?



# A balanced tree?



# Operations

- » The cost of all operations depends on the height of the tree
- » For balanced trees, all operations are  $O(\log n)$
- » For degenerate trees, all operations are  $O(n)$
- » We know that average trees are rarely balanced or degenerate
- » If we allow deletes, an average tree has height  $O(\sqrt{n})$



# AVL-trees

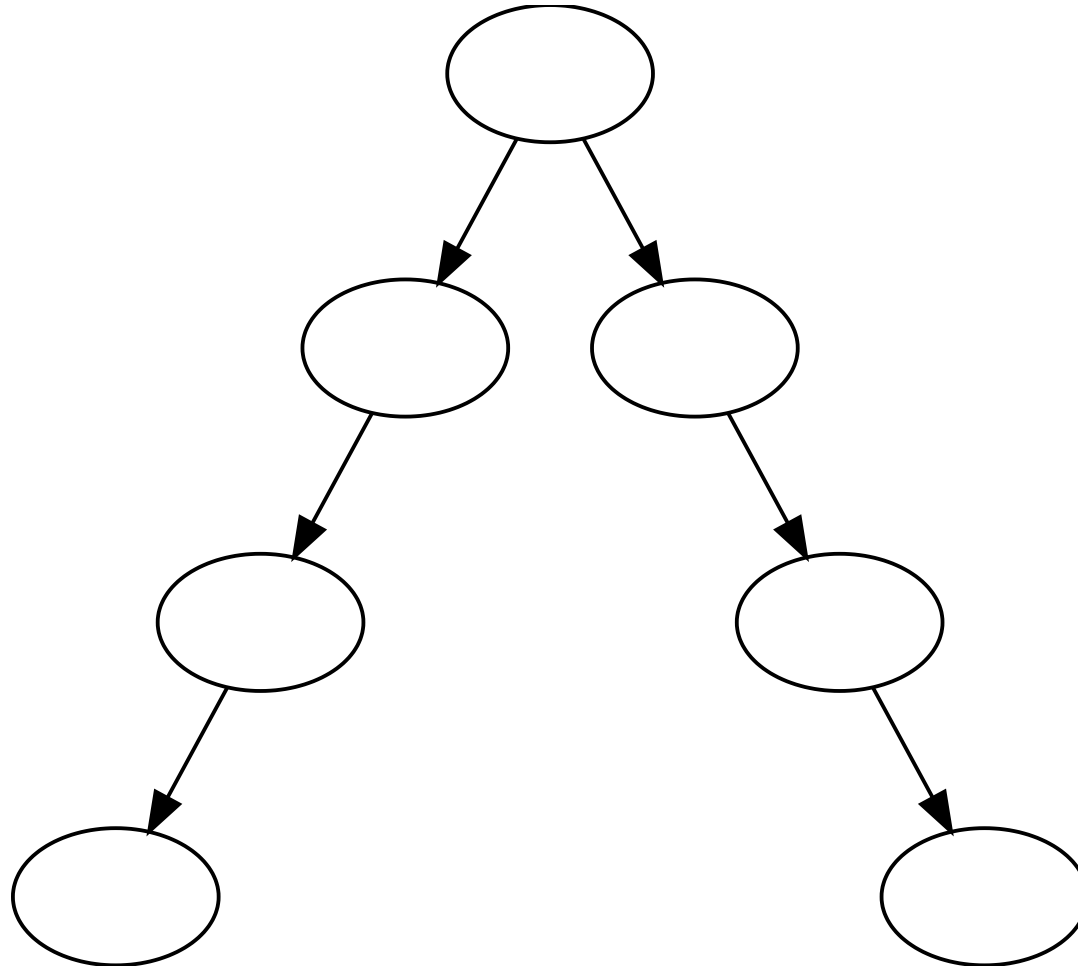
# AVL-tree

- » Adelson-Velskii and Landis
- » A binary search tree with a *balance condition*

# Balance condition

- » Should ensure that the depth of the tree is  $O(\log N)$
- » Must be easy to maintain
- » First idea, the left and right subtrees should be the same height
  - » Can result in poorly balanced trees

# “Balanced” tree



# Balance conditions

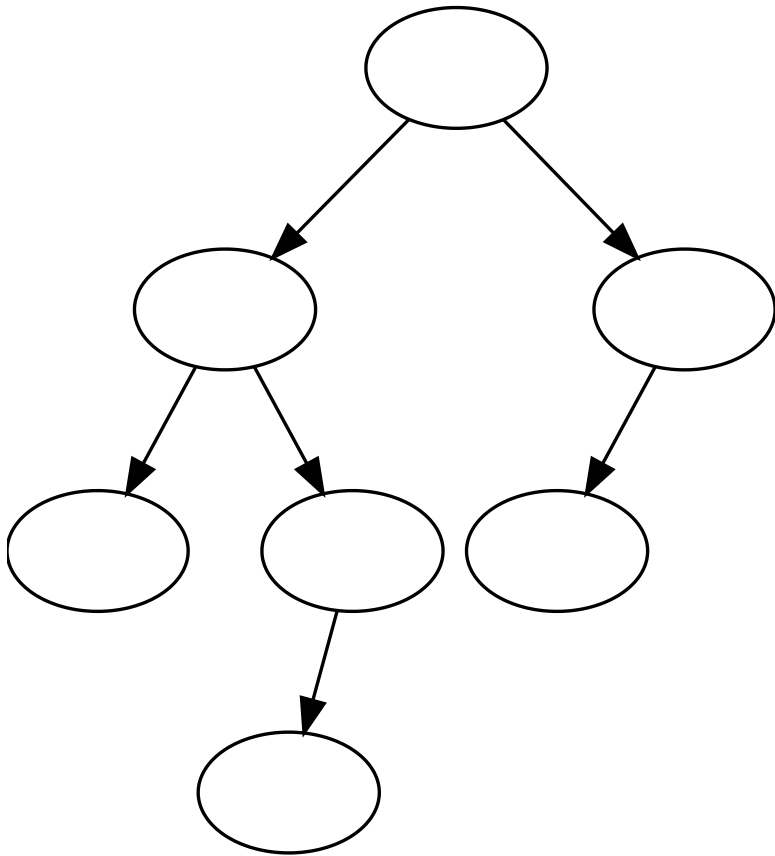
- » Balance at root is not enough
- » So, each node should have left and right subtrees of the same height
  - » Would force perfectly balanced trees
  - » But too difficult to maintain

# AVL-trees

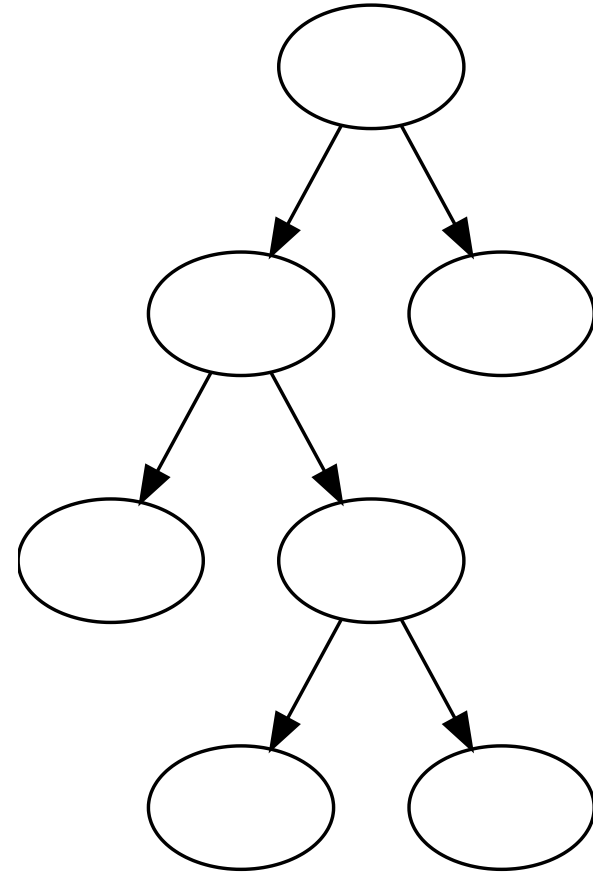
- » The heights of the left and right subtrees can differ by at most 1
- » Gives a height of about  $1.44 \cdot \log_2(N + 2) - 1.328$ 
  - » More than  $\log_2$ , but not that much
- » Minimum nodes at a height
  - »  $S(h) = S(h - 1) + S(h - 2) + 1$
- » So, a tree with height 9 has at least 143 nodes

# AVL-trees

**AVL**



**Not AVL**

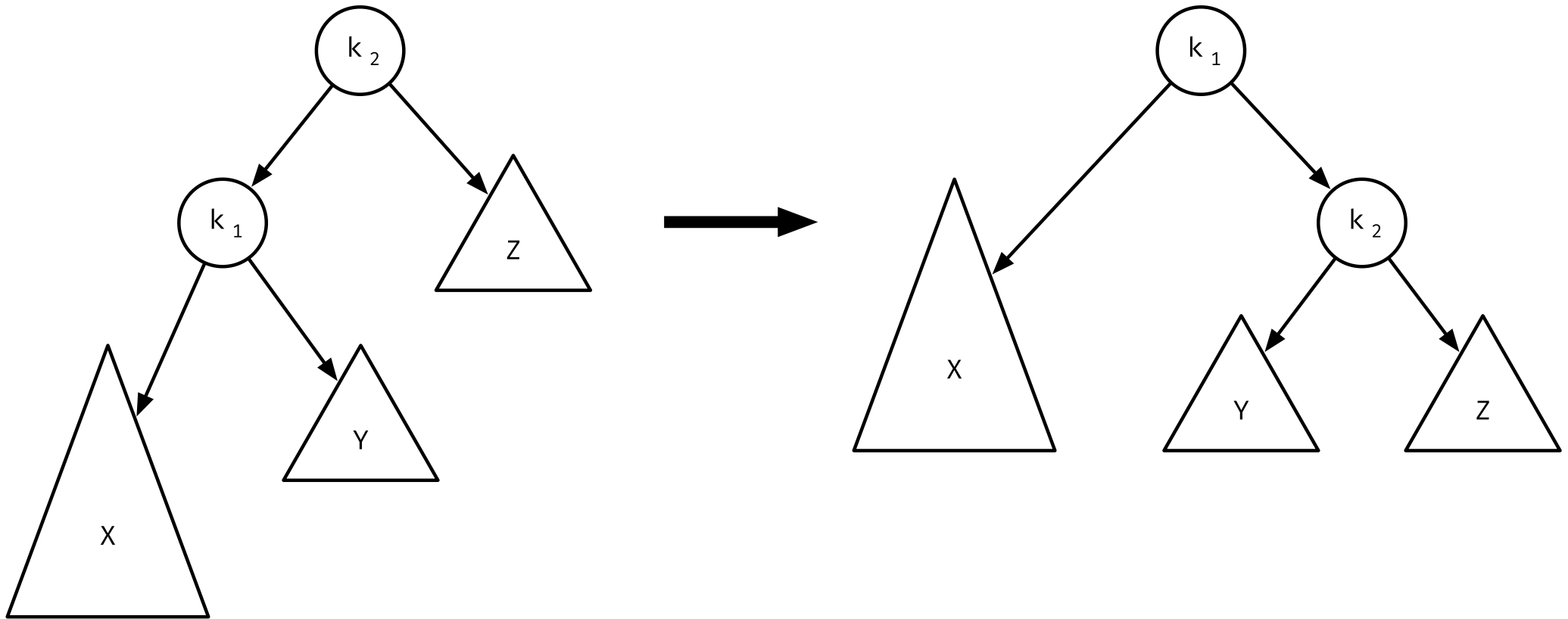


# Implementation

- » Problem: inserting values can destroy the balance
- » So, insert must make sure the tree is balanced after
- » Four possible cases: insert into left (L) subtree of left (L) child, LR, RL, and RR
  - » Two are symmetric: LL and RR, and LR and RL
  - » And one pair is easier, LL and RR



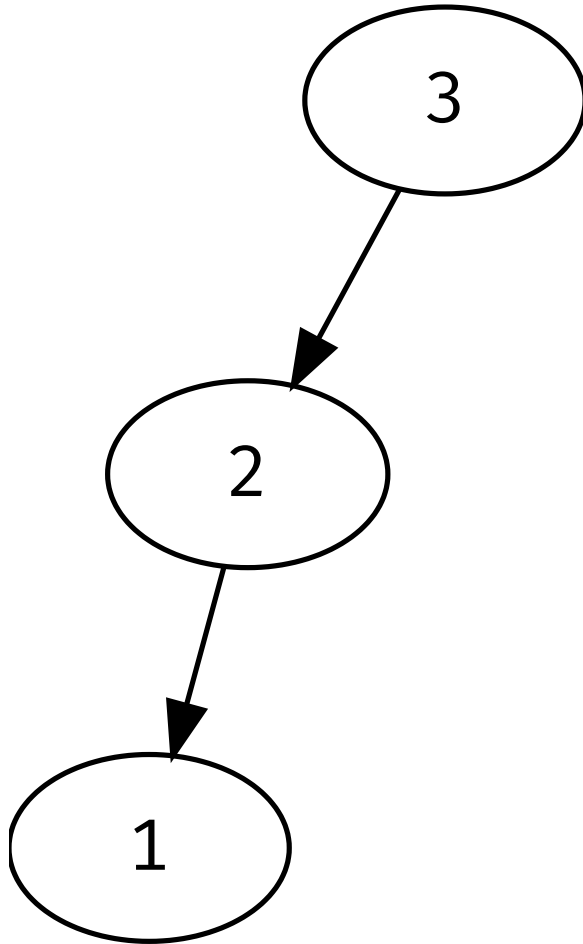
# Single rotation (LL and RR)



# What is going on?

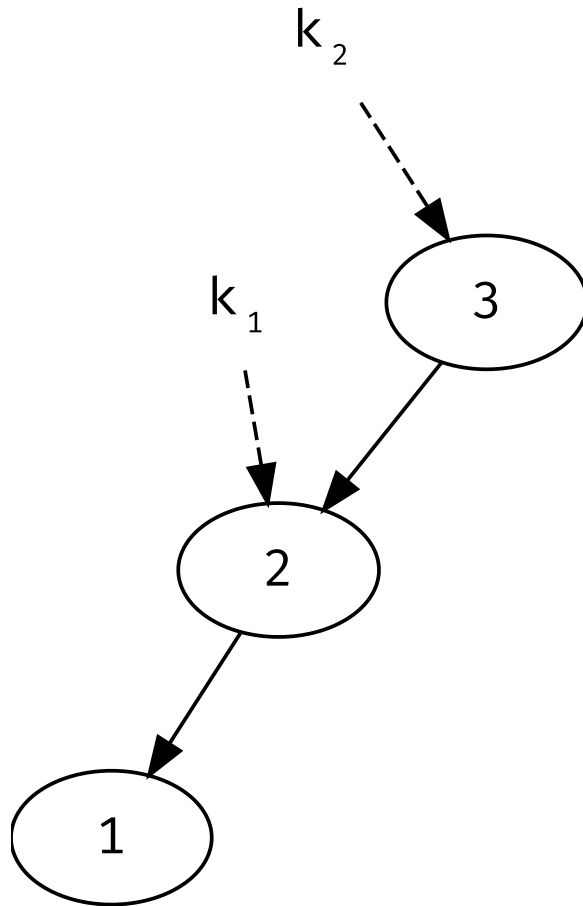
- » Node  $k_2$  (the root) is violating the balance condition
  - » since  $X$  is two levels deeper than  $Z$
  - » A change to  $X$  caused the violation
- » We can fix this by moving  $X$  higher and  $Y$  and  $Z$  lower
  - » This means  $k_1$  becomes root
  - » and  $k_2$  its right child, since  $k_1 < k_2$
  - »  $Y$  becomes the left child of  $k_2$  since  $k_1 < Y < k_2$

# Single rotation with numbers



- » Problematic: the subtrees of the root differ by more than 1
- » We need to rotate at the root
- » 2 should become the root and 3 its right child
- » Called a left rotate

# Single rotation with numbers



» Left rotate

»  $k_1$  is  $k_2$ .left

» We change  $k_2$ .left to  $k_1$ .right

» and set  $k_1$ .right to  $k_2$

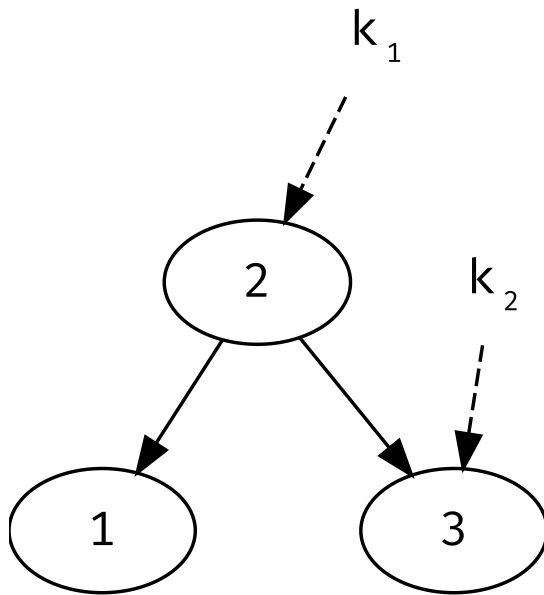
# Single rotation with numbers

» Left rotate

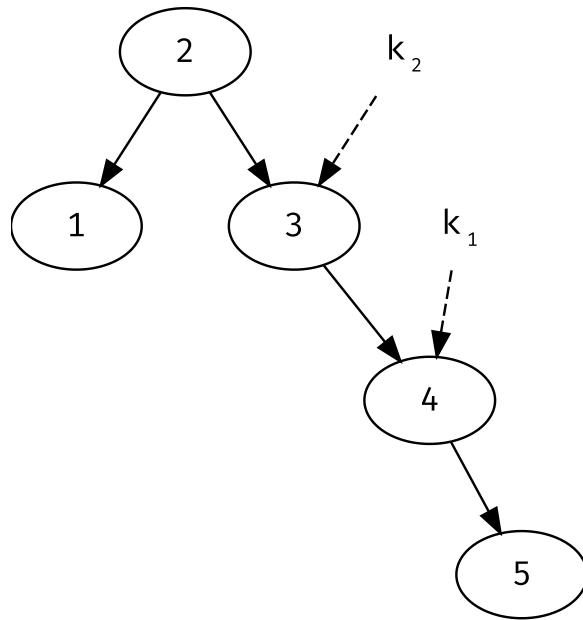
»  $k_1$  is  $k_2$ .left

» We change  $k_2$ .left to  $k_1$ .right

» and set  $k_1$ .right to  $k_2$

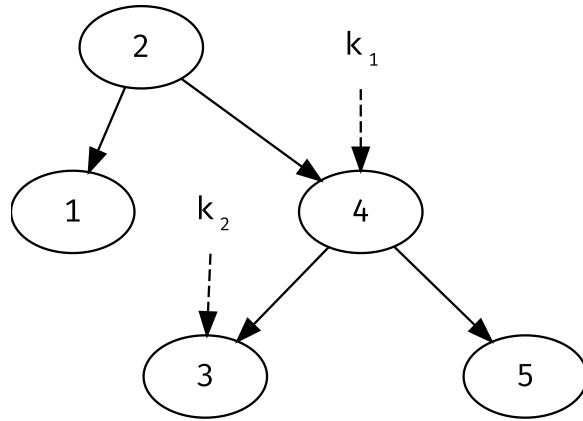


# Single rotation with numbers



- » Adding 4 and 5 causes another balance issue
- » This time, the opposite, so rotate

# Single rotation with numbers



- » Adding 4 and 5 causes another balance issue
- » This time, the opposite, so rotate

# Implementation: Node

```
1 @dataclass
2 class AVLNode:
3     key: int
4     left: 'AVLNode|None' = None
5     right: 'AVLNode|None' = None
6     height: int = 0
```



# Implementation: AVLTree

```
1 class AVLTree:
2     def __init__(self) -> None:
3         self.root = None
```

# Implementation: Add

```
1 @patch
2 def _add(self:AVLTree, n:AVLNode|None, key:int) -> AVLNode:
3     if n is None:
4         return AVLNode(key)
5
6     if n.key > key:
7         n.left = self._add(n.left, key)
8     elif n.key < key:
9         n.right = self._add(n.right, key)
10
11     return self._balance(n)
```

# Implementation: Add

```
1 @patch
2 def add(self:AVLTree, key:int) -> None:
3     self.root = self._add(self.root, key)
```

# Implementation: Balance

```
1 @patch
2 def _balance(self:AVLTree, n:AVLNode|None) -> AVLNode|None:
3     if n is None:
4         return n
5
6     if self._height(n.left) - self._height(n.right) > 1:
7         if self._height(n.left.left) >= self._height(n.left.right):
8             n = self._rotate_left(n)
9         else:
10            n = self._double_left(n)
11     elif self._height(n.right) - self._height(n.left) > 1:
12         if self._height(n.right.right) >= self._height(n.right.left):
13             n = self._rotate_right(n)
14         else:
15            n = self._double_right(n)
16
17     n.height = max(self._height(n.left), self._height(n.right)) + 1
18     return n
```

# Implementation: Height

```
1 @patch
2 def _height(self:AVLTree, n:AVLNode|None) -> int:
3     if n is None:
4         return -1
5     return n.height
```

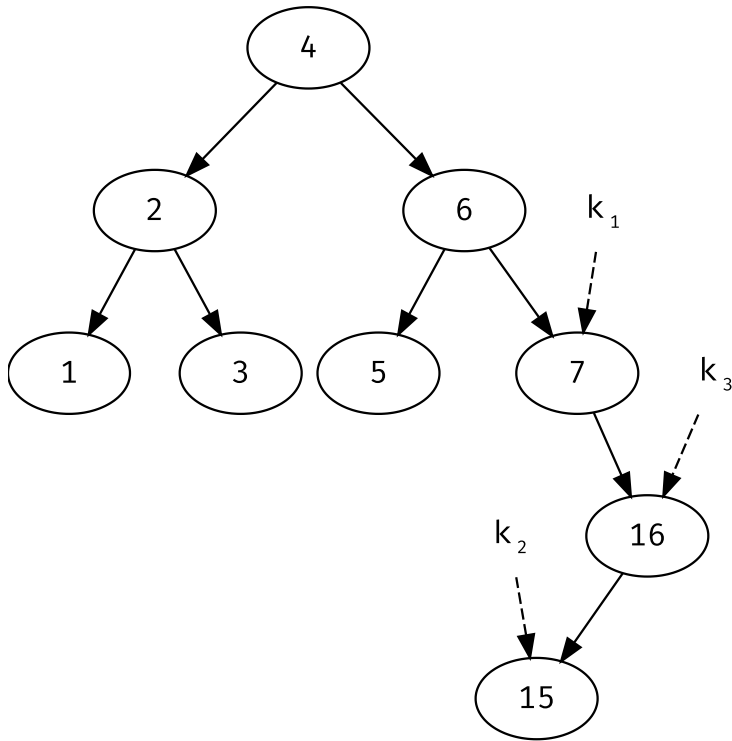
# Implementation: Single rotate

```
1 @patch
2 def _rotate_left(self:AVLTree, r2:AVLNode) -> AVLNode:
3     r1 = r2.left
4     r2.left = r1.right
5     r1.right = r2
6     r2.height = max(self._height(r2.left), self._height(r2.right)) + 1
7     r1.height = max(self._height(r1.left), r2.height) + 1
8
9     return r1
```

# Implementation: Single rotate

```
1 @patch
2 def _rotate_right(self:AVLTree, r2:AVLNode) -> AVLNode:
3     r1 = r2.right
4     r2.right = r1.left
5     r1.left = r2
6     r2.height = max(self._height(r2.left), self._height(r2.right)) + 1
7     r1.height = max(self._height(r1.left), r2.height) + 1
8
9     return r1
```

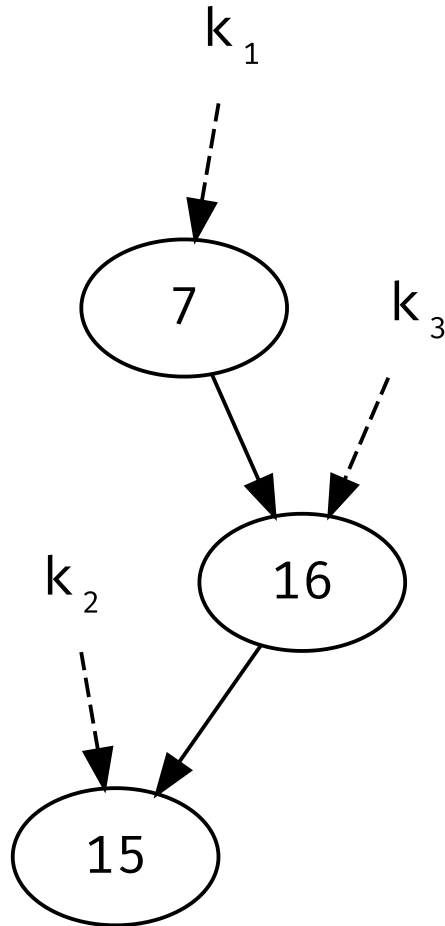
# Double rotation



- » Previously, we have seen LL and RR
- » For RL (and LR), we need to rotate twice
  - » For RL, a “double right”

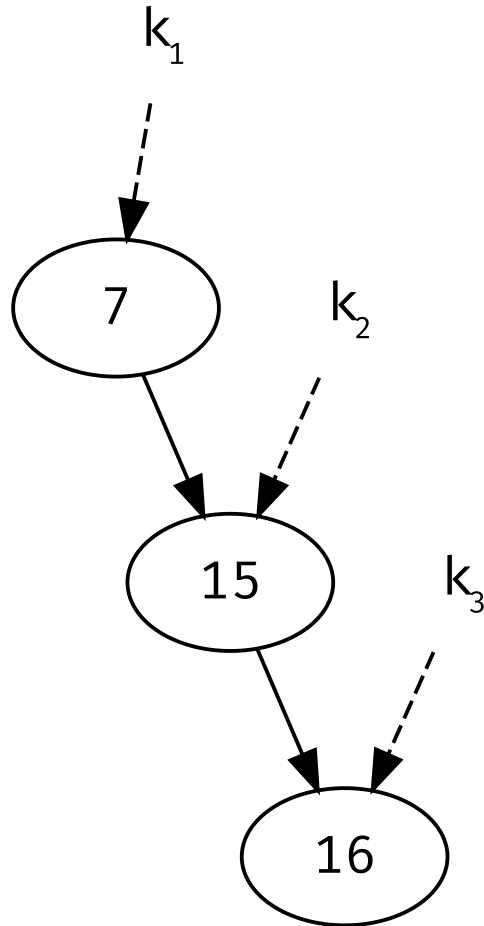


# Double right



- » A double right means
  - » rotate right child left
  - » rotate self right

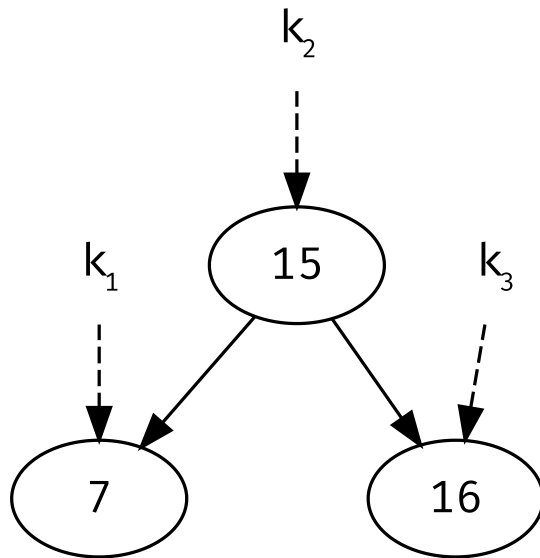
# Rotate left child right



- » A double right means
  - » rotate right child left
  - » rotate self right

# And self right

- » A double right means
  - » rotate right child left
  - » rotate self right



# Implementation: rotate double

```
1 @patch
2 def _double_right(self:AVLTree, n:AVLNode) -> AVLNode:
3     n.right = self._rotate_left(n.right)
4     return self._rotate_right(n)
5
6 @patch
7 def _double_left(self, n:AVLNode) -> AVLNode:
8     n.left = self._rotate_right(n.left)
9     return self._rotate_left(n)
```

# Adding preorder walk

```
1 @patch
2 def print_preorder(self:AVLTree) -> None:
3     self._preorder(self.root)
4
5 @patch
6 def _preorder(self:AVLTree, n:AVLNode | None) -> None:
7     if n is not None:
8         print(n.key)
9         self._preorder(n.left)
10        self._preorder(n.right)
```

# Example

```
1 t = AVLTree()  
2 t.add(3)  
3 t.add(2)  
4 t.print_preorder()  
5 print('----')  
6 t.add(1)  
7 t.print_preorder()
```

3

2

----

2

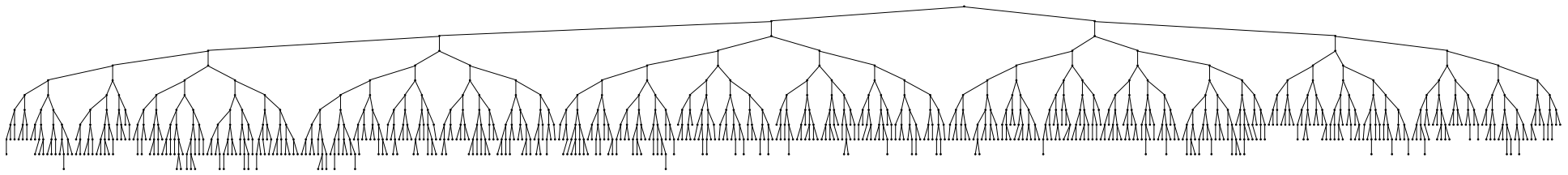
1

3

# Random inserts

- » We expect a tree with 1023 nodes to have a height of about 13
  - »  $1.44 \cdot \log_2 1023 - 1.328$
- » Running a 100 000 inserts, we find that the height is between 10 to 12
  - » Mean is 11.000310, so very close to 11
  - » Compared to a mean of 33.5 with binary search trees and no balancing effort
- » The above is with deletes

# A balanced tree?





# Oh, deletes...

```
1 @patch
2 def _delete(self:AVLTree, n:AVLNode|None, key:int) -> AVLNode|None:
3     if n is None:
4         return None
5
6     if n.key > key:
7         n.left = self._delete(n.left, key)
8     elif n.key < key:
9         n.right = self._delete(n.right, key)
10    else:
11        if n.right is None:
12            return n.left
13        if n.left is None:
14            return n.right
15        n.key = self._min(n.right)
16        n.right = self._delete(n.right, n.key)
17
18    return self._balance(n)
```

# Splay trees

# Splay trees

- » Many applications have “data locality”
  - » A node is accessed multiple times within a reasonable timeframe
- » Splay trees push a node to the root after it is accessed
- » Uses a series of rotations from AVL trees
- » Can also help balance the tree

# Amortized cost

- » Splay trees guarantees that  $m$  consecutive operations is  $O(m \log_2 n)$
- » A single operation can still be  $\Theta(n)$ , so the bound is not  $O(\log_2 n)$
- » This is called *amortized* running time
  - » if  $m$  operations are  $O(m \cdot f(n))$
  - » the amortized cost is  $O(f(n))$

# Splay trees

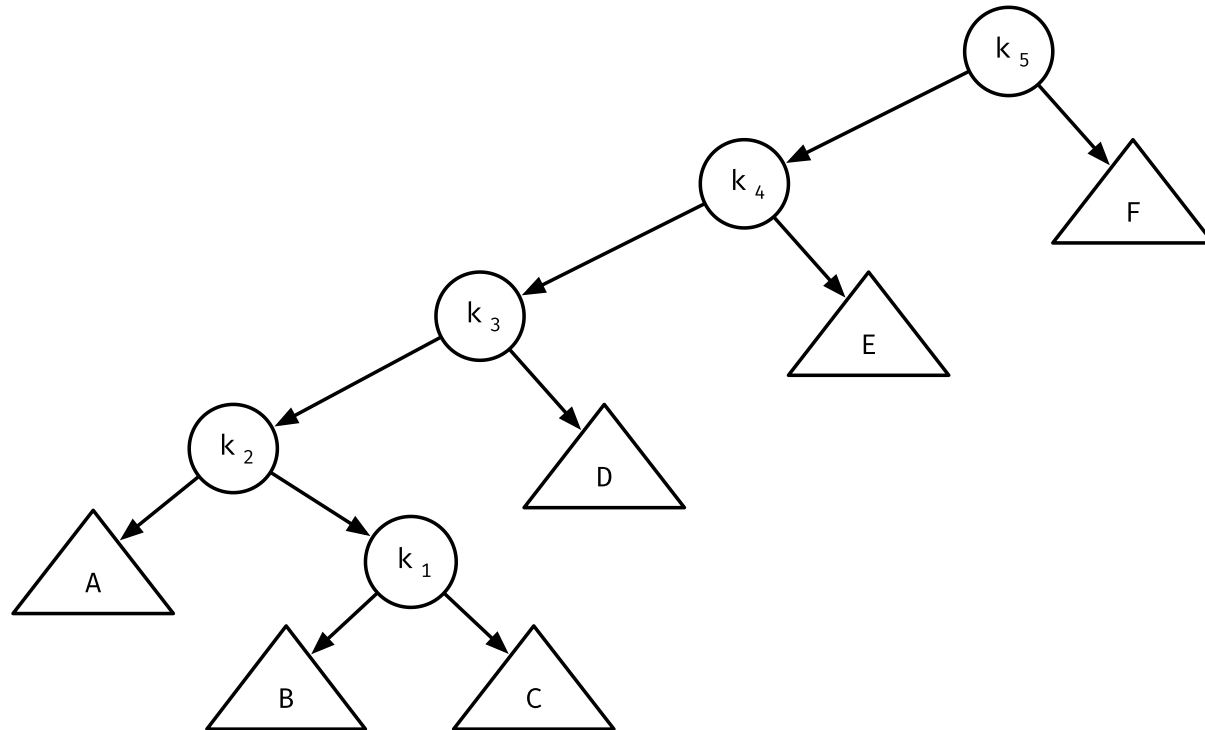
- » If an operation is  $O(n)$  and we want  $O(\log_2 n)$  it is clear that we must do something to fix it
  - » In splay trees, we fix by moving
  - » So, if first  $O(n)$ , then consecutive close to  $O(1)$

# First idea

- » Single rotation from node to root
- » Will get the node to root
- » Nodes on the path will “suffer”
  - » I.e., move further from the root
- »  $\Omega(m \cdot n)$
- » So, not good enough

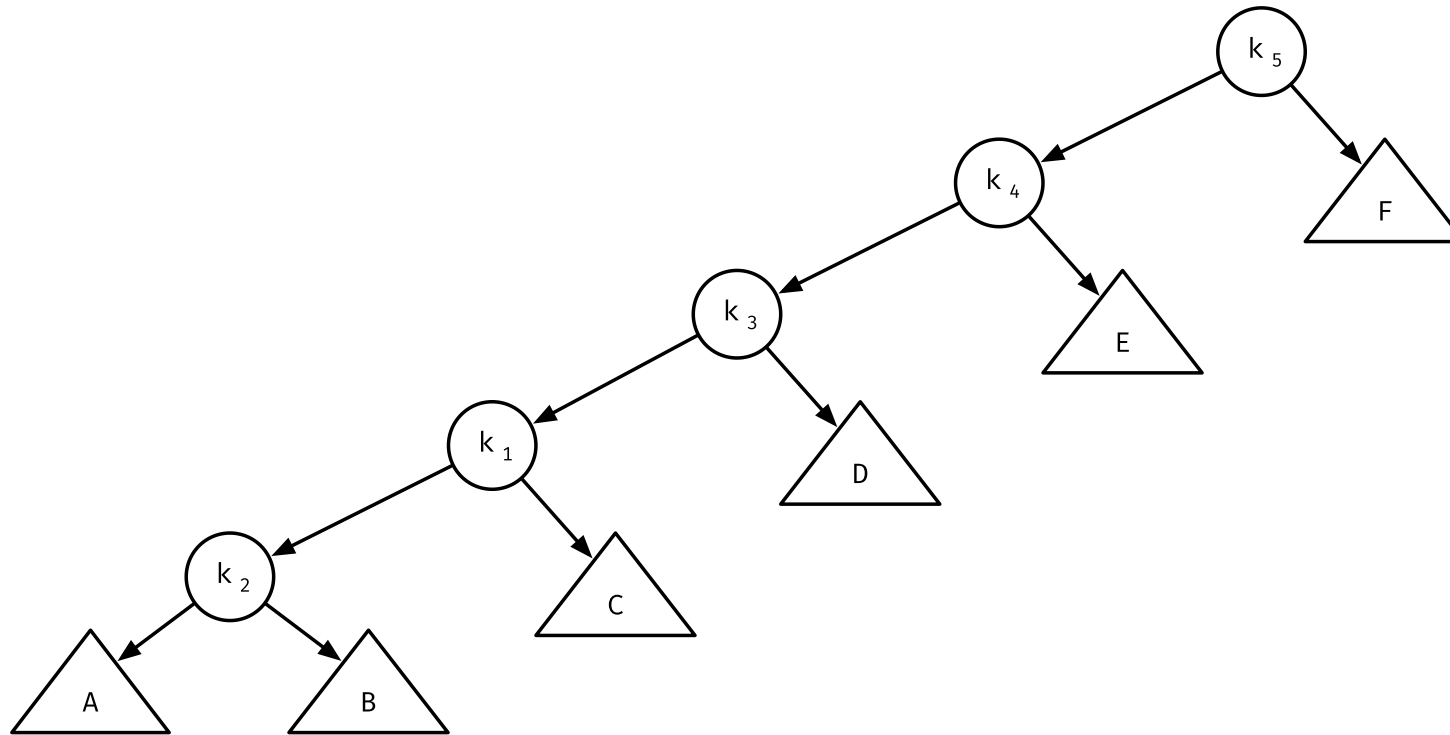
# First idea

We search for  $k_1$



# First idea

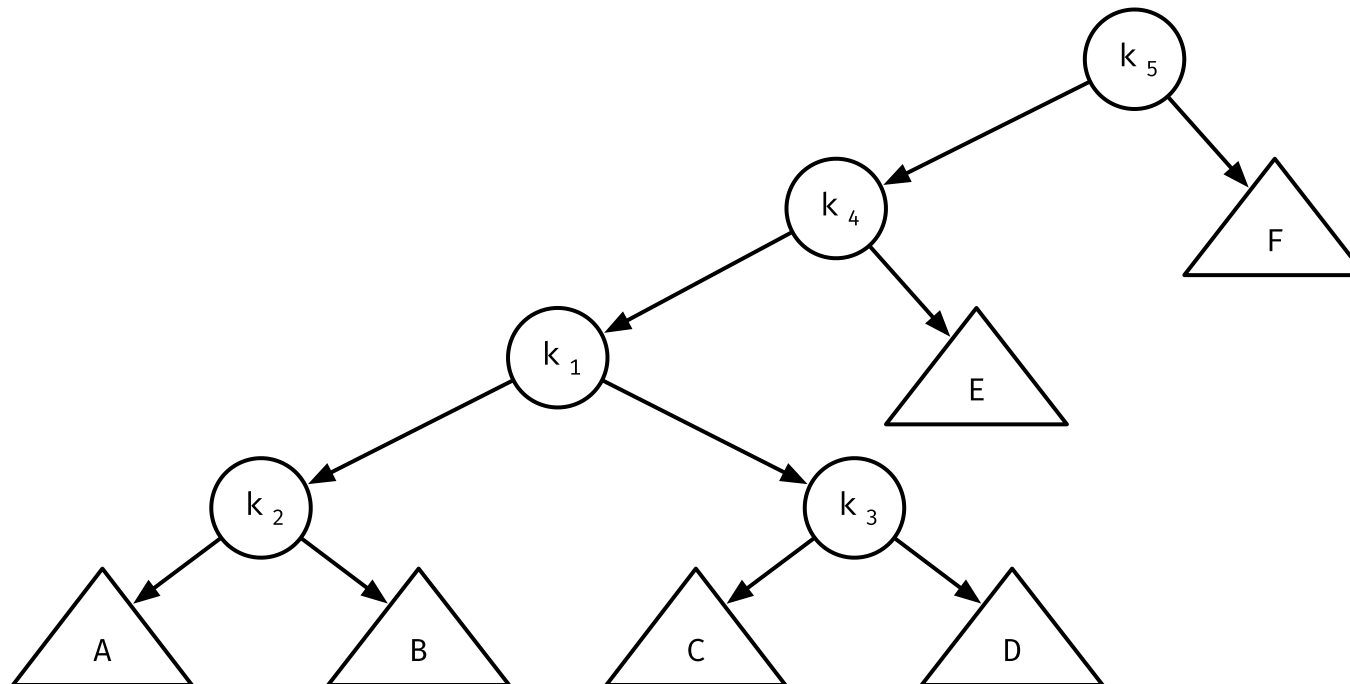
And rotate it upwards





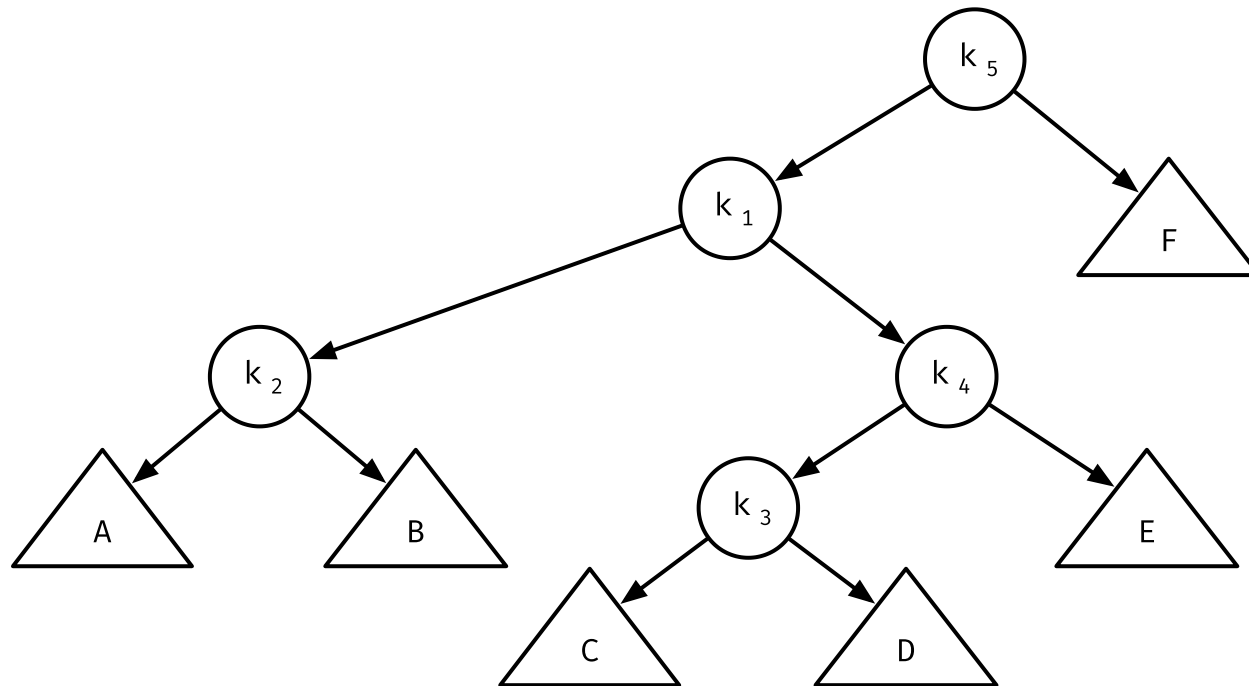
# First idea

And rotate it upwards



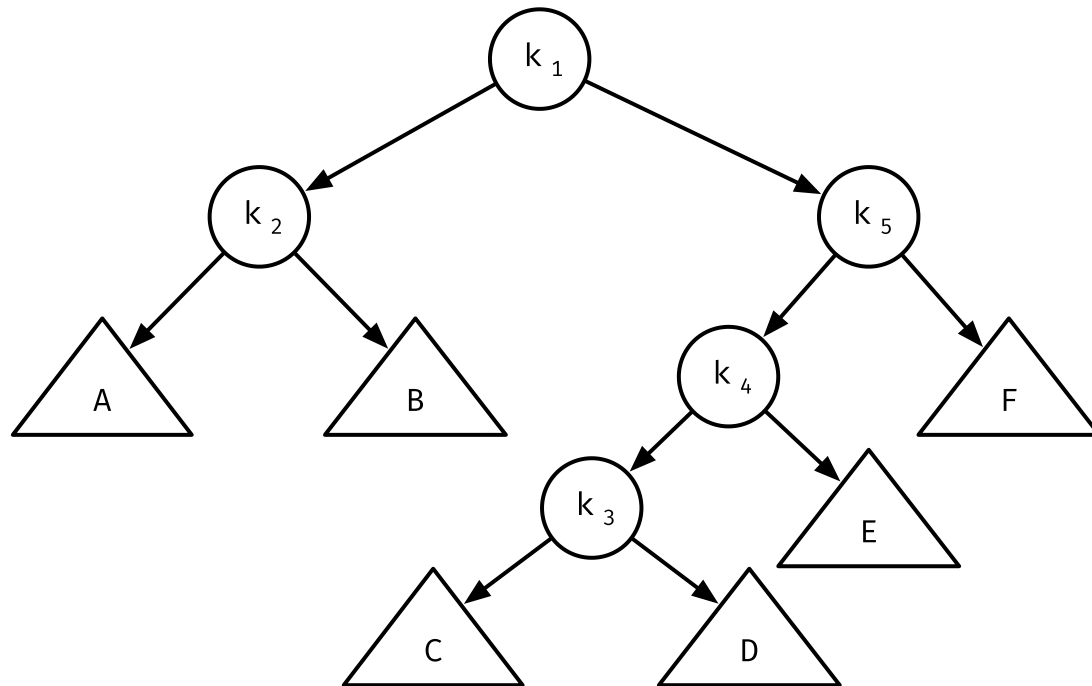
# First idea

And rotate it upwards



# First idea

$k_3$  is now worse than before



# Better idea

- » We need to be smarter about our rotations
- » A few cases:
  - »  $X$  is the node we rotate
  - »  $P$  is its parent
  - »  $G$  is its grandparent

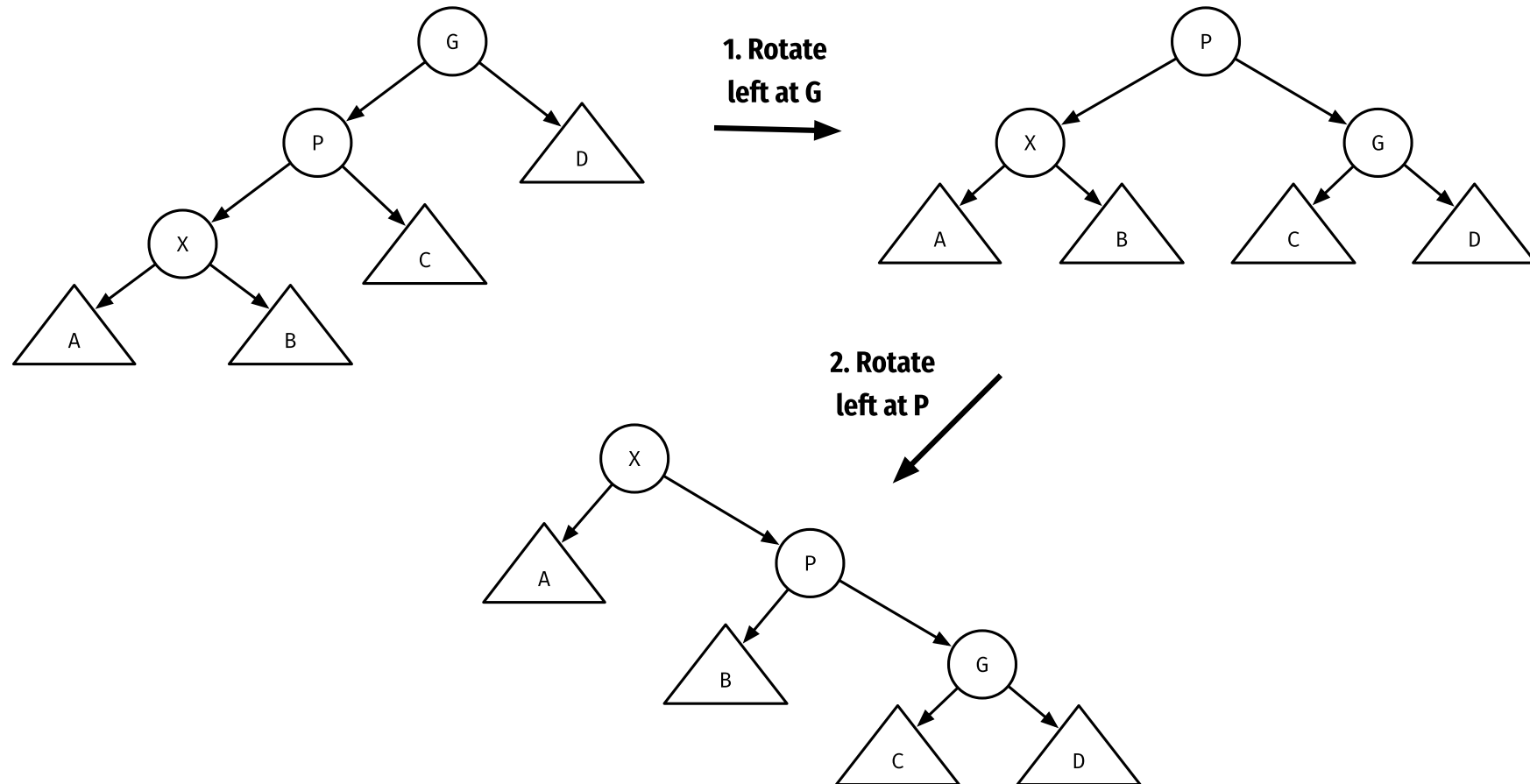
# Better idea

- » If  $P$  is the root, then we rotate  $X$  and the root
- » If  $X$  is a right child and  $P$  is a left child, we *zig-zag*
- » If  $X$  and  $P$  are both left children, we *zig-zig*
- » Note the symmetric cases, just as for AVL trees

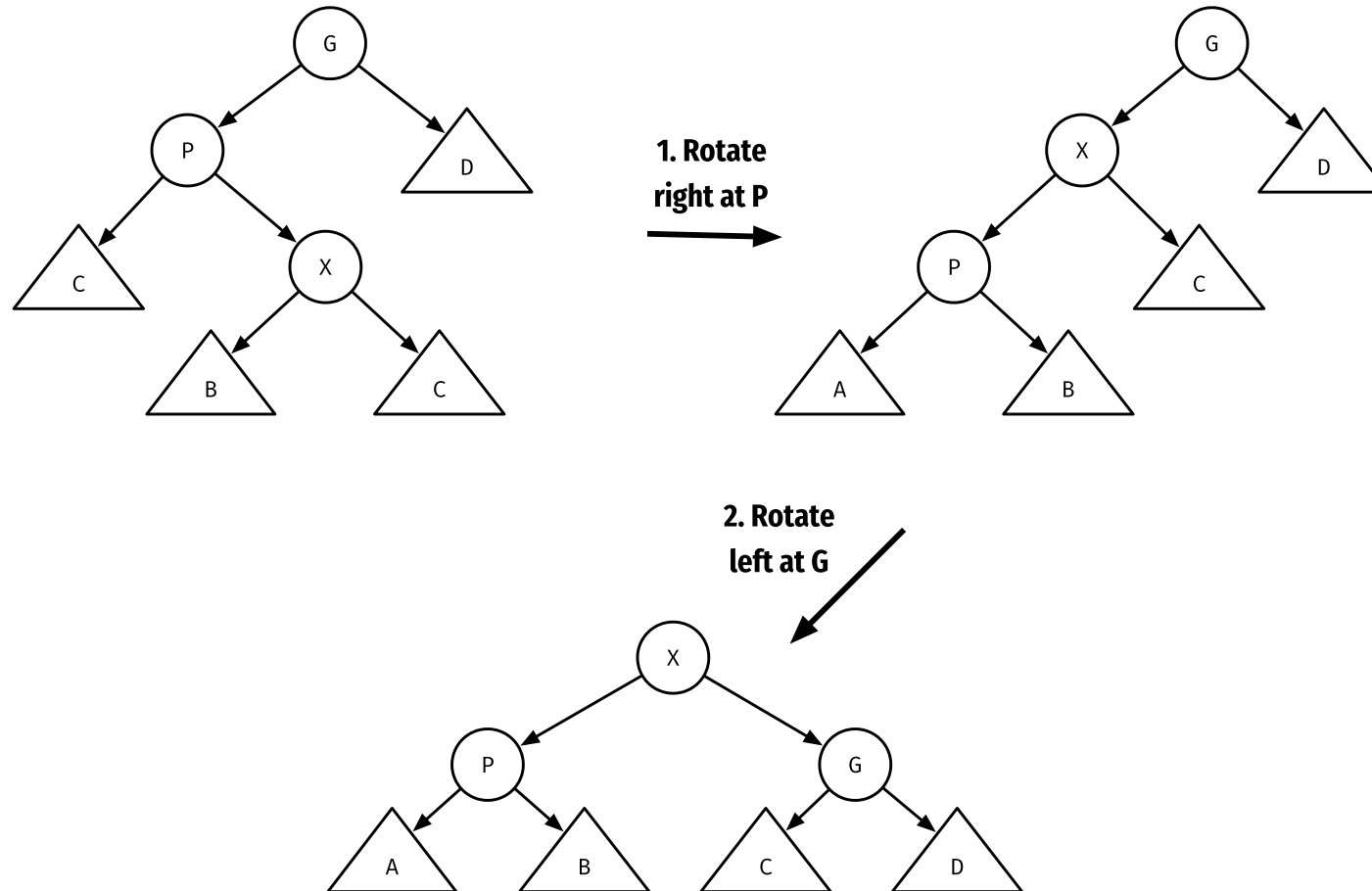
# I do not like these names...

- » A zig-zig means that the same rotation is performed twice
  - » LL or RR
- » A zig-zag means that a rotation followed by the mirror
  - » LR or RL
- » Some use zig for one and zag for the other and have four combinations

# Zig-zig



# Zig-zag



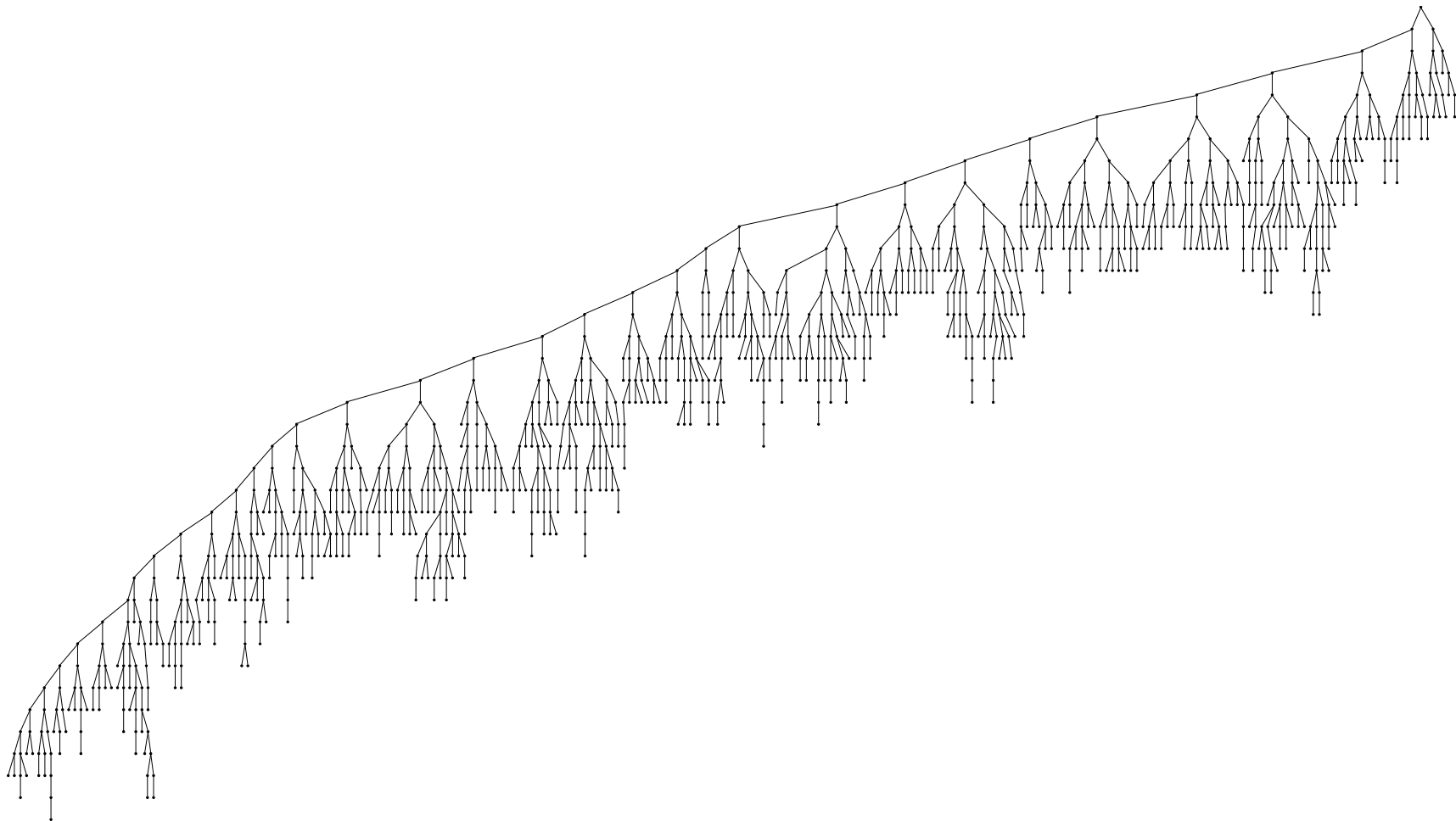


# Splay trees

- » We can move any node to the root by combining zig, zig-zig, and zig-zag
- » We do this each time we search for a node
- » This will ensure that nodes that we have searched for will be closer to the root and be quicker to find again

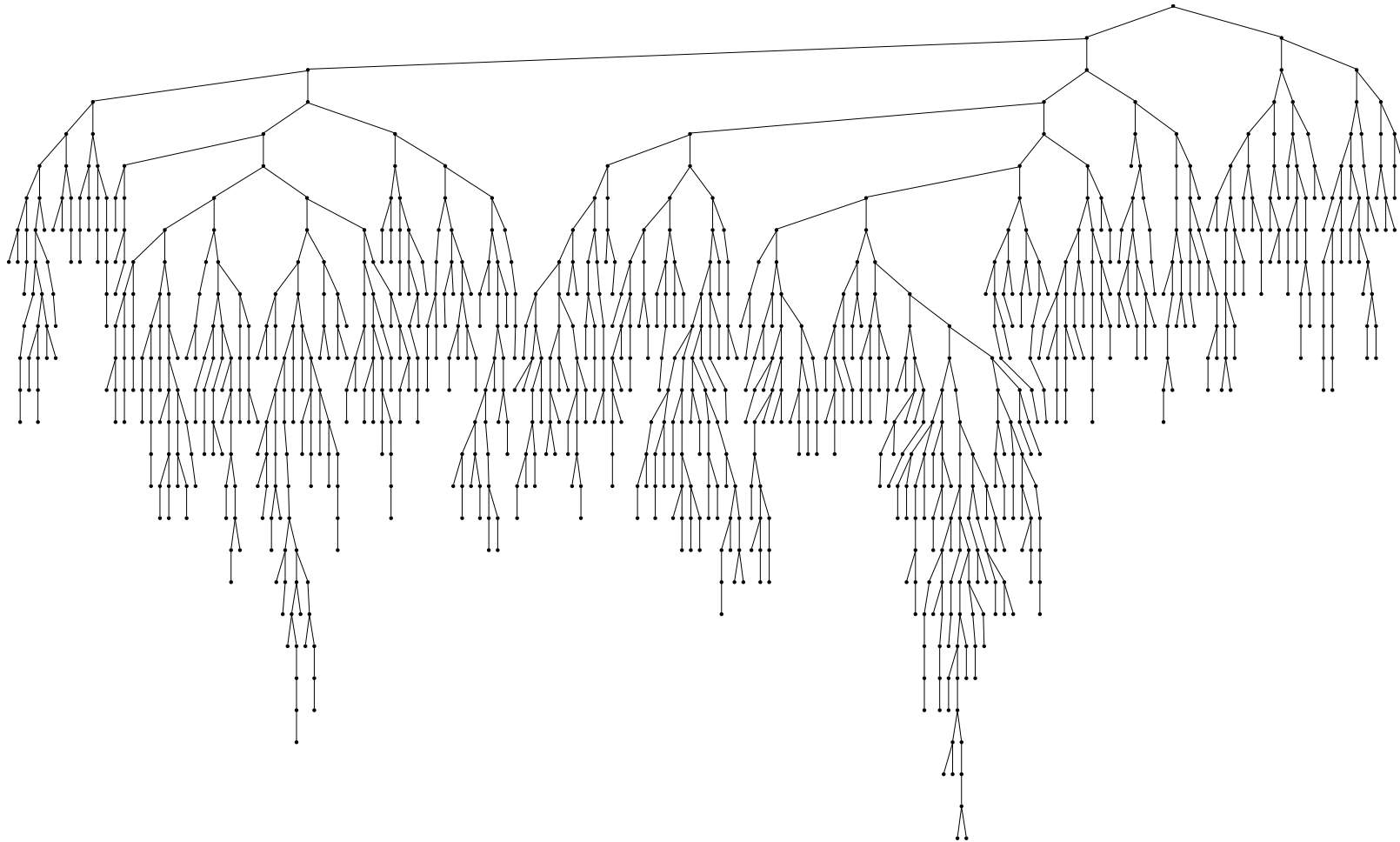
# Experiment

**A tree after multiple inserts and deletes**



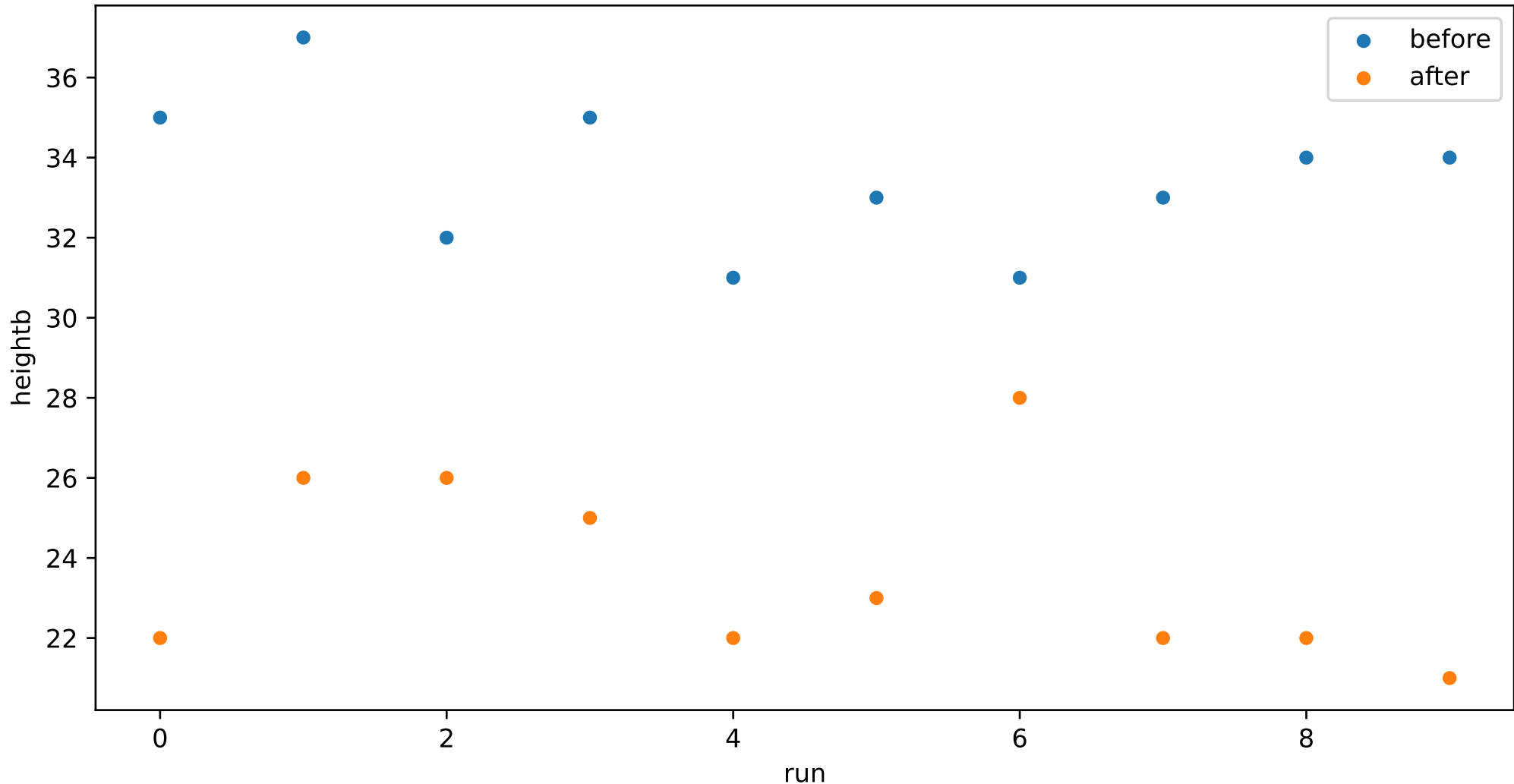
# Experiment

**Same tree after 5000 random finds**



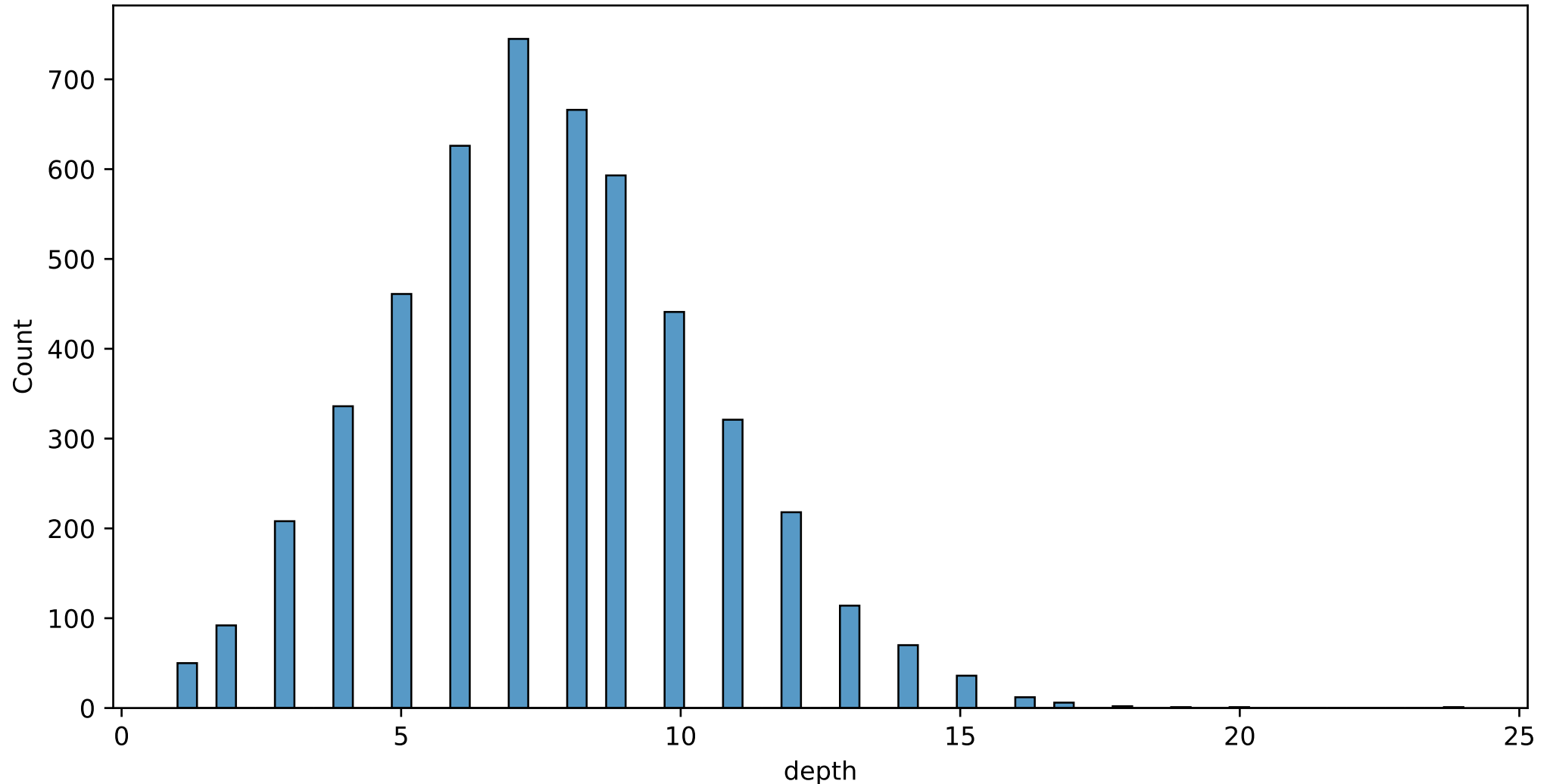
# Experiment

## Heights of 10 trees before and after splaying



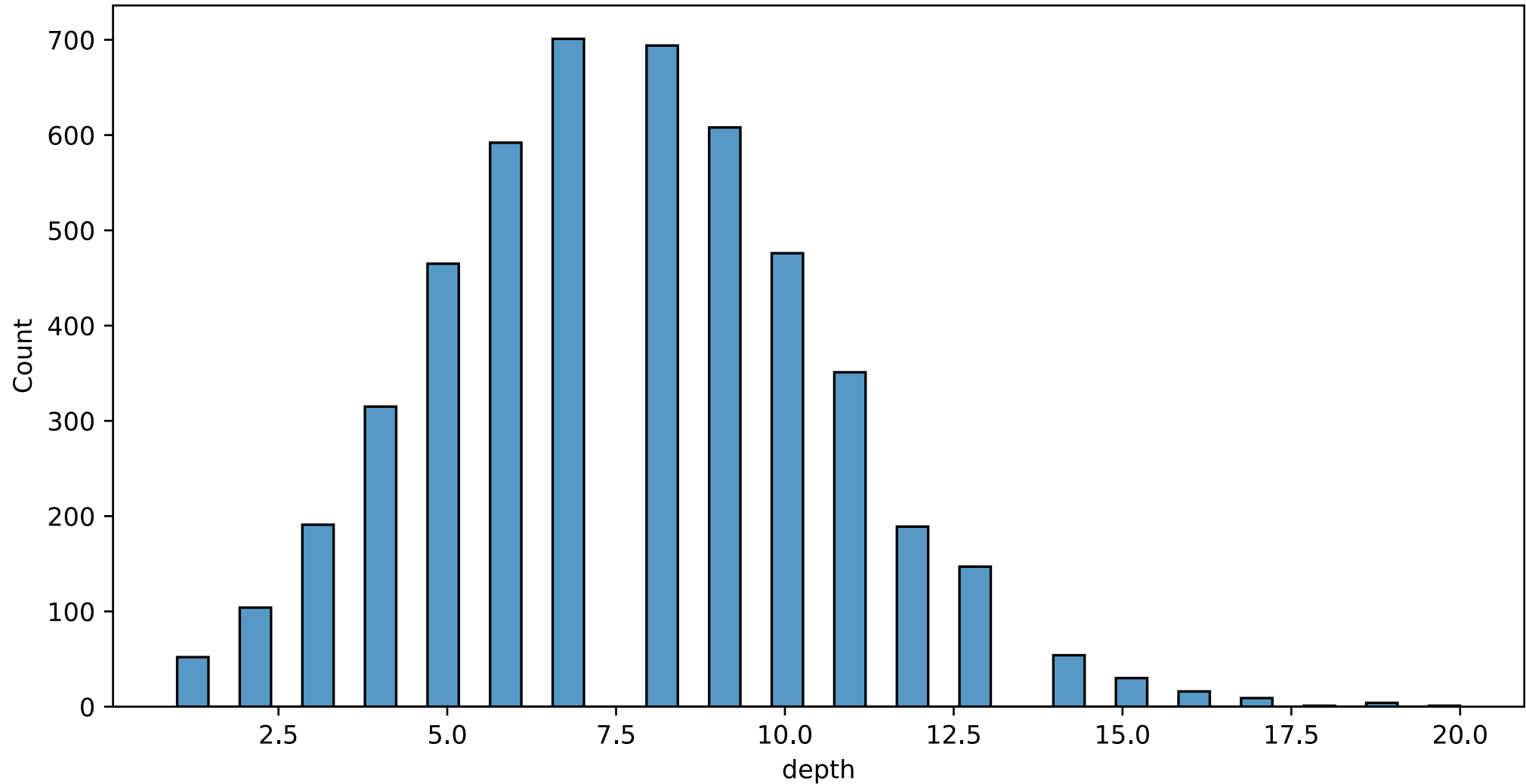
# Experiment

Depth of the value we are search for



# Experiment

## Adding warmup



# Reading instructions

# Reading instructions

» Ch. 4.1 - 4.6



