Algorithms and Data Structures

Trees (Ch. 4)

Morgan Ericsson

Today

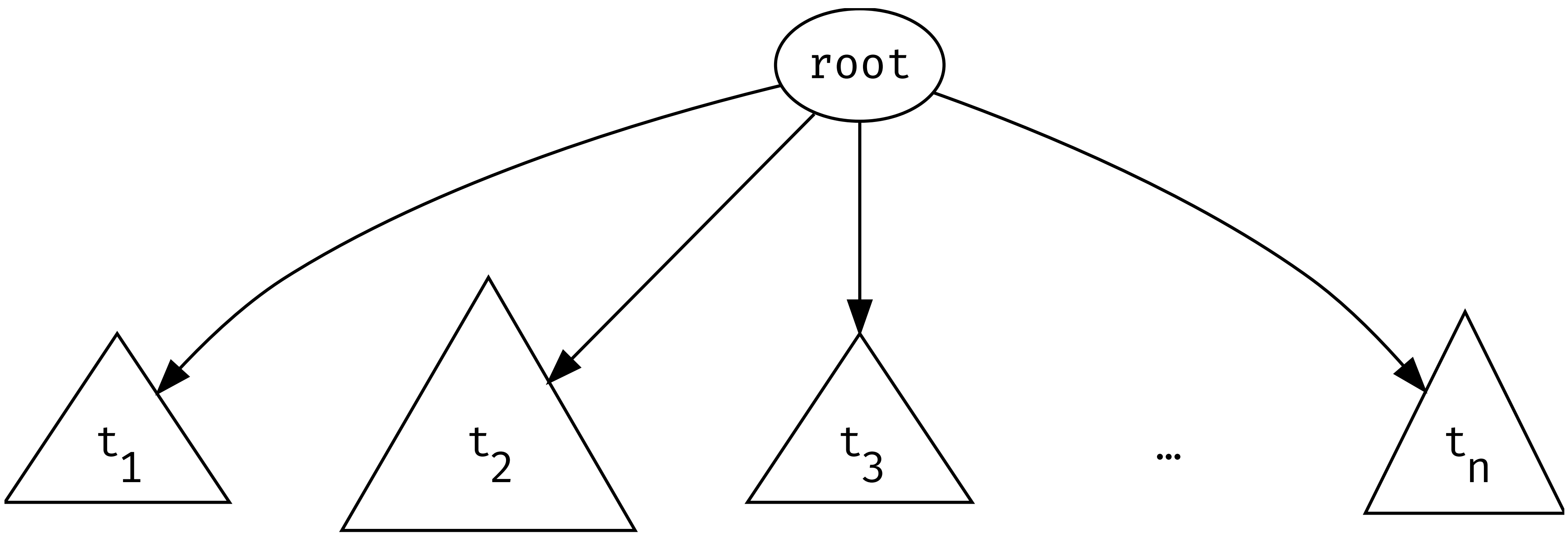
- Trees
  - Binary trees
  - Binary Search Trees
  - AVL-trees
  - Splay trees

Trees

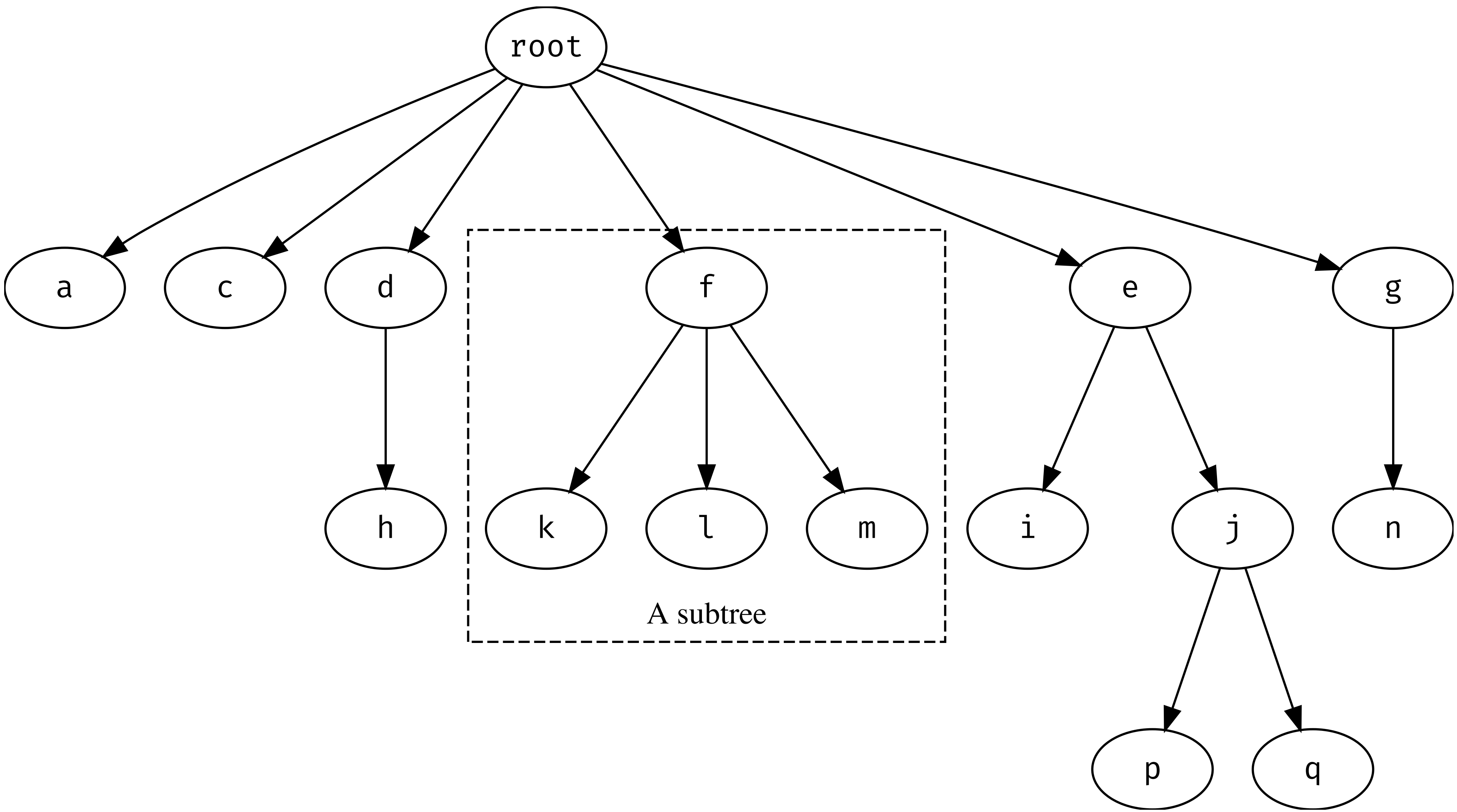
The Tree ADT

- 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:
  - $n_1, n_2, \ldots, n_k$
  - 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 lenght of $n_1, \ldots, n_k$ is $k - 1$

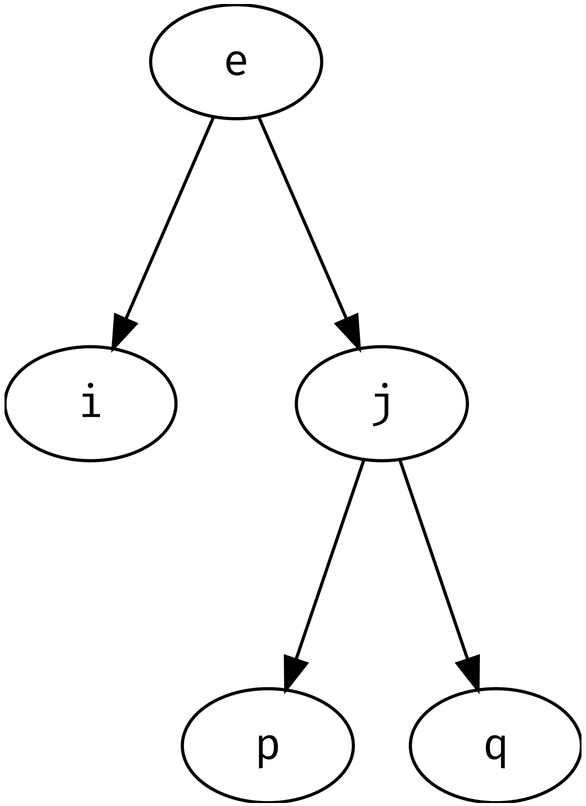
Paths

- 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

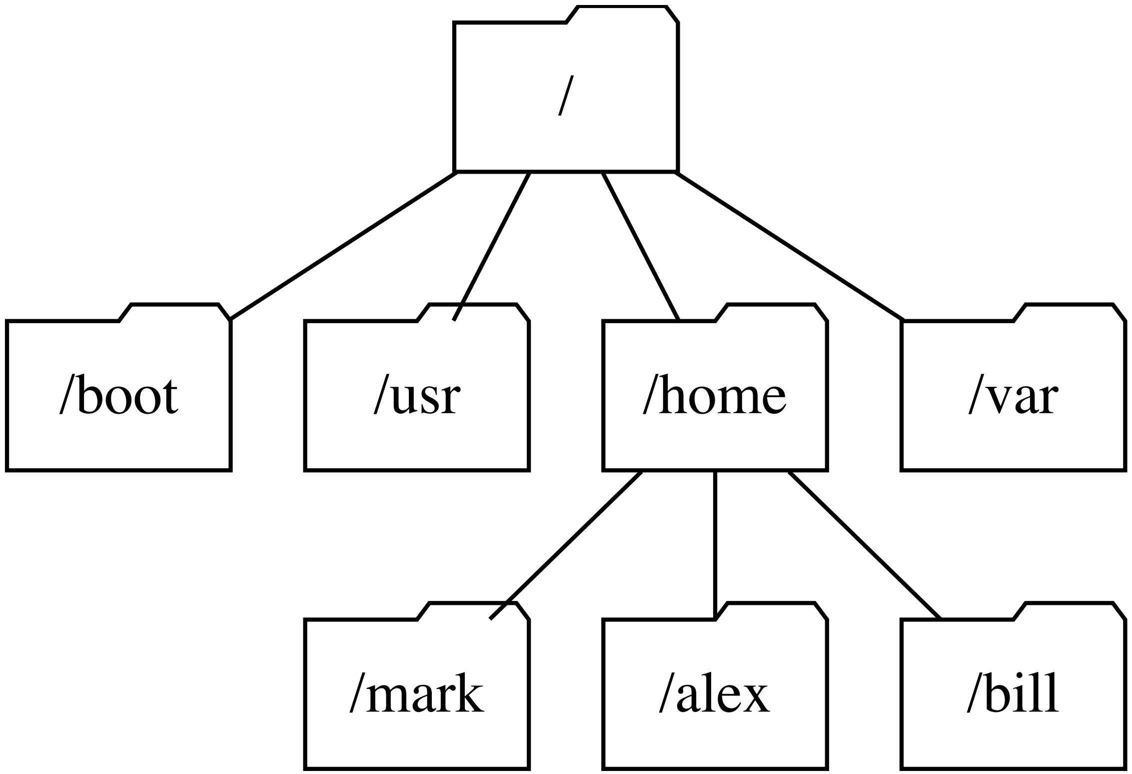
- If there is a path from n_i to n_j then
  - n_i is an *ancestor* of n_j
  - n_j is a *descendant* of n_i
- If $n_i \neq n_j$ then they are *proper*, e.g., *proper ancestor*

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

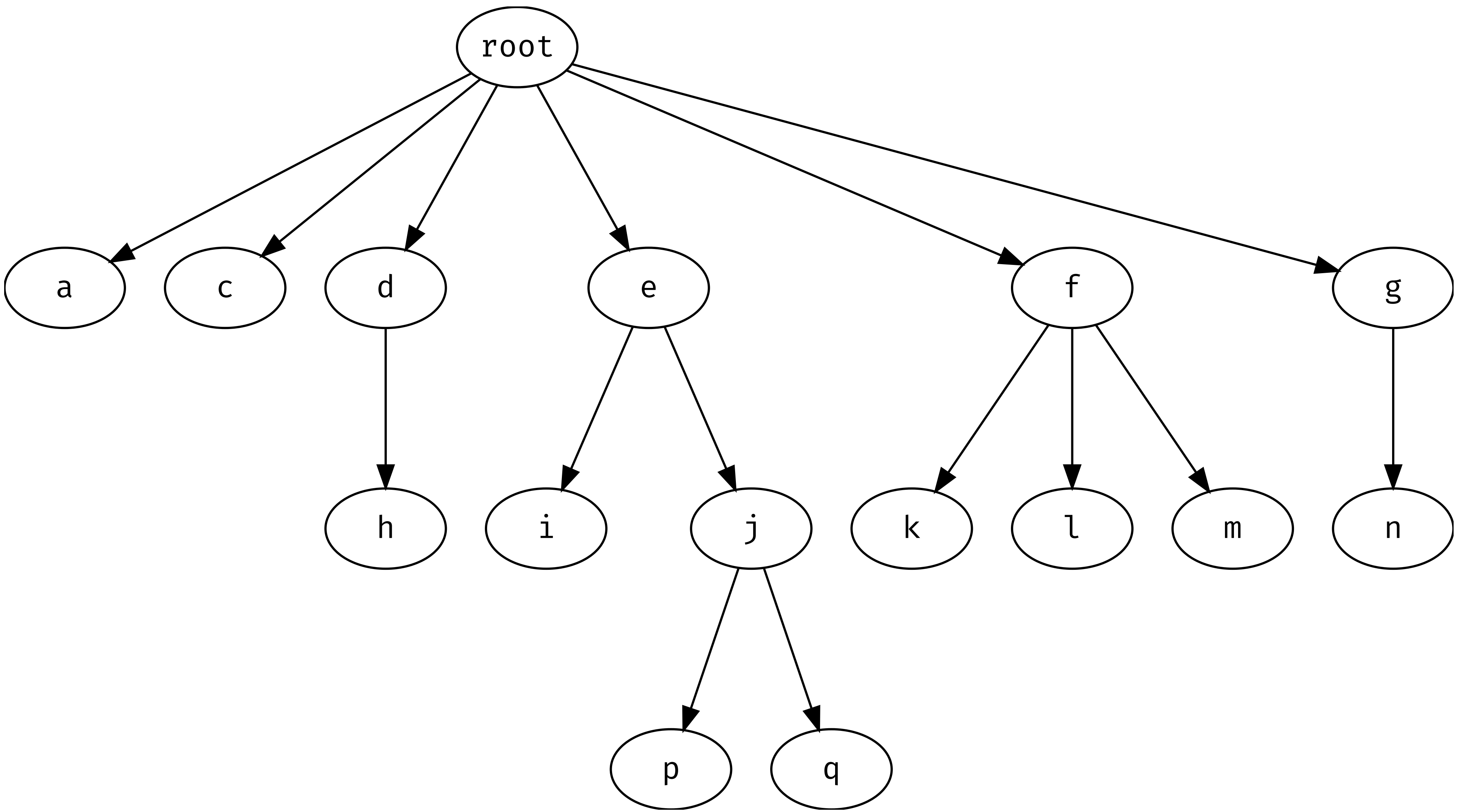
Example: File systems



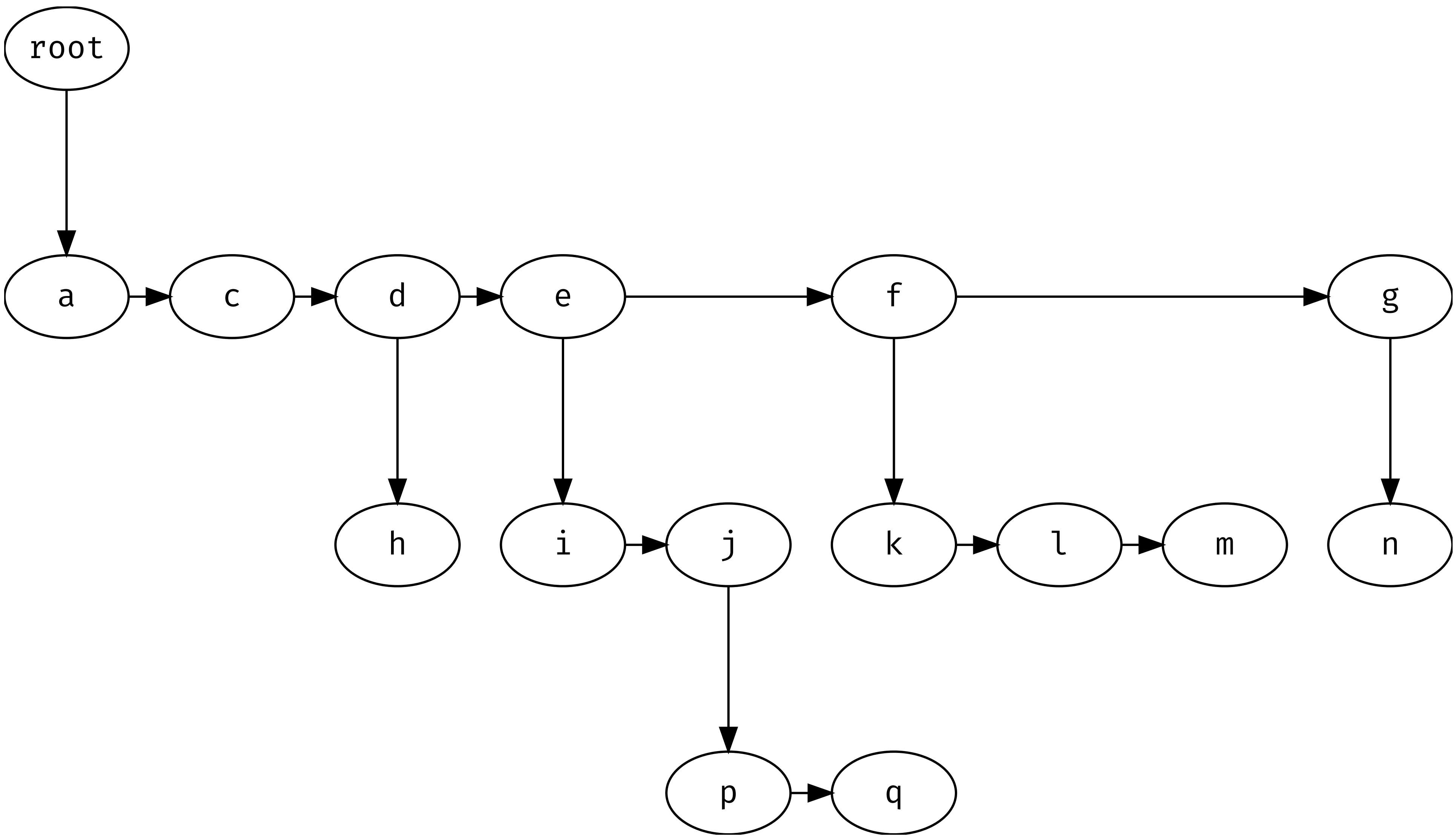
Implementing a tree

- 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



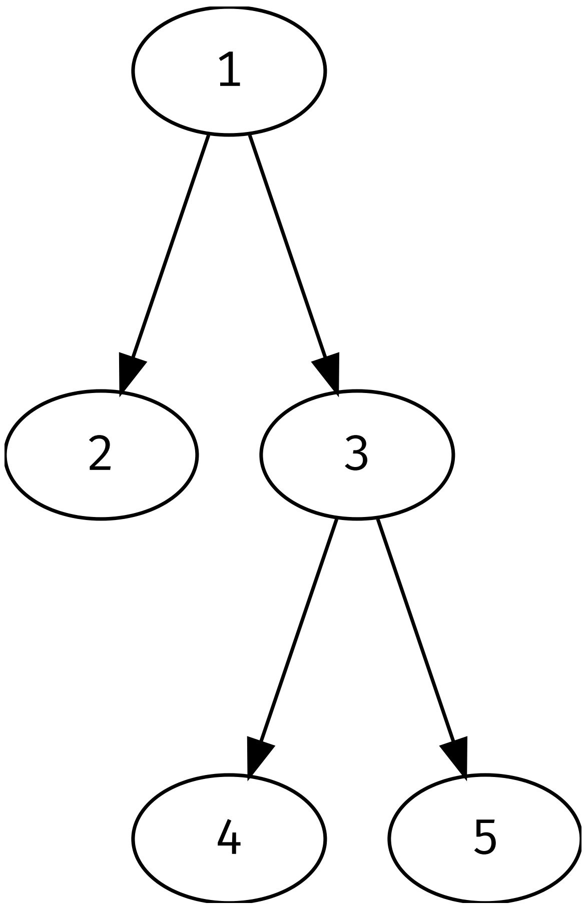
Left-most child, right sibling (LCRS)



LCRSNode

```python
from dataclasses import dataclass

@dataclass
class LCRSNode:
  key: int
  left: 'LCRSNode|None' = None
  right: 'LCRSNode|None' = None
```

Creating a tree




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

Walking the tree

```python
def walk(root:LCRSNode) -> None:
  if root is not None:
    print(root.key)
    walk(root.left)
    walk(root.right)
```

Does it work?

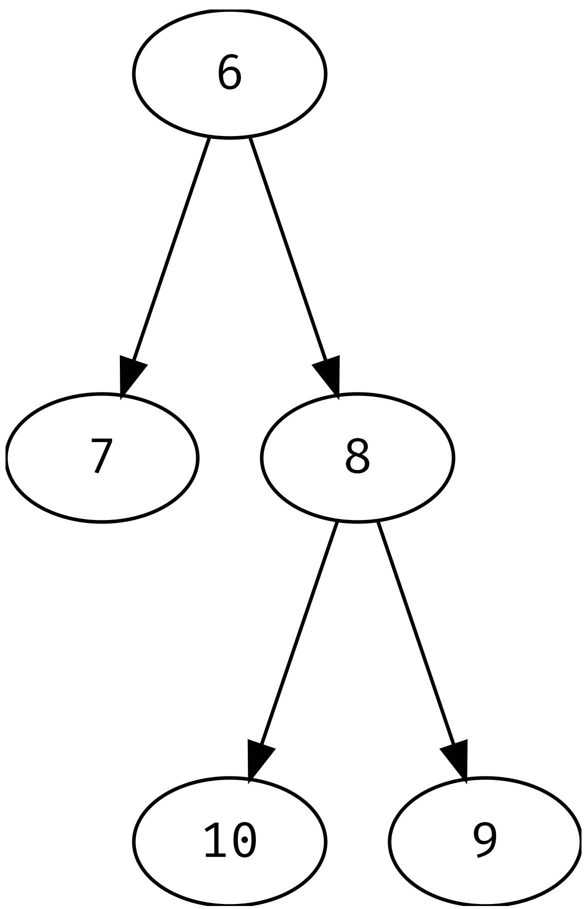
```
walk(r)
```

1
2
3
4
5

Adding children

```python
from fastcore.basics import patch

@patch
def add_child(self:LCRSNode, key:int) -> LCRSNode:
  if self.left is None:
    self.left = LCRSNode(key)
    return self.left
  else:
    p = self.left
    while p.right is not None:
      p = p.right
    p.right = LCRSNode(key)
    return p.right
```

Rewriting our example




```
r = LCRSNode(6)
_ = r.add_child(7)
t = r.add_child(8)
_ = t.add_child(10)
_ = t.add_child(9)
```

Does it work?

```
walk(r)
```

```
6
7
8
10
9
```

Patching in walk

```python
@patch
def walk(self:LCRSNode) -> None:
  print(self.key)
  if self.left is not None:
    self.left.walk()
  if self.right is not None:
    self.right.walk()
```

Does it work?

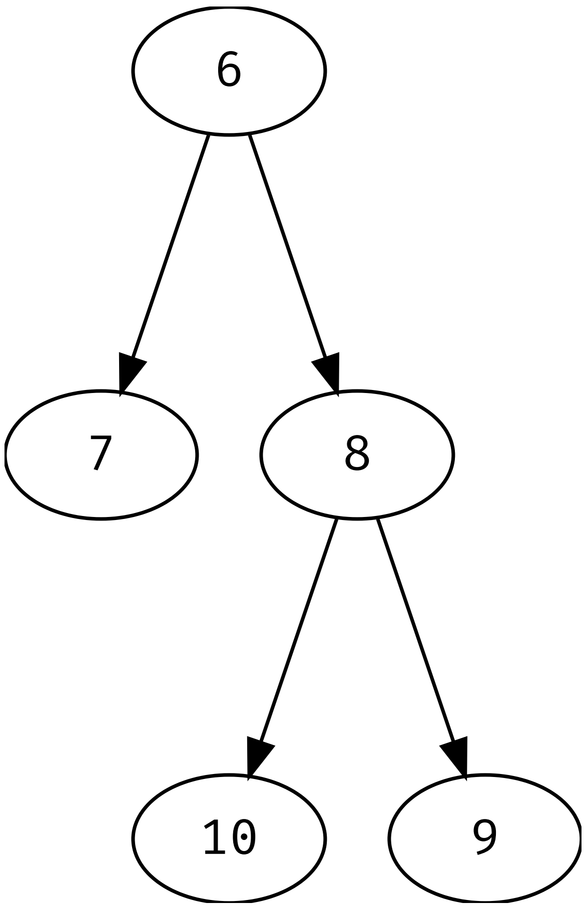
```python
r.walk()
print('---')
r.left.right.walk()
```

```
6
7
8
10
9
---
8
10
9
```

Node degree (number of children)

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

Checking

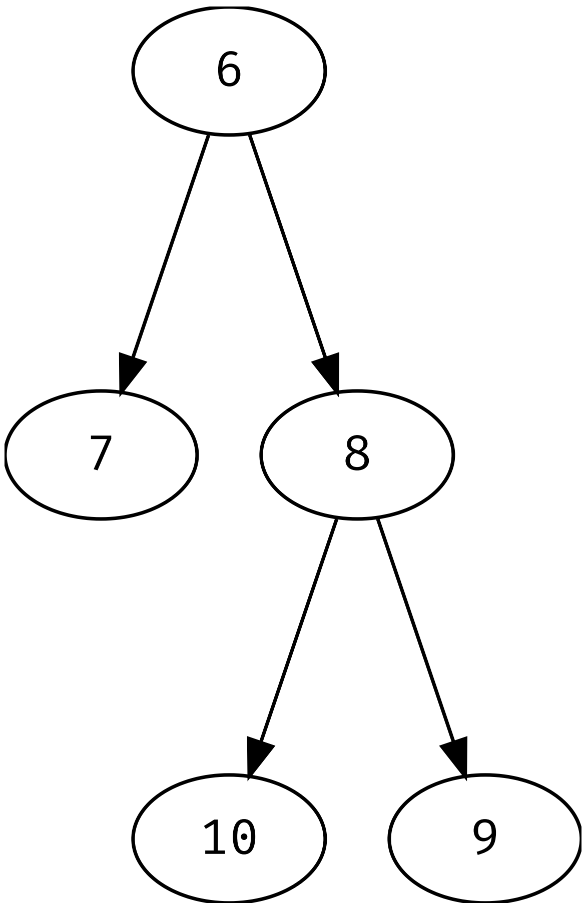



```
assert r.degree == 2
assert r.left.degree == 0
assert r.left.right.degree == 2
```

Is a node a leaf?

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

Checking

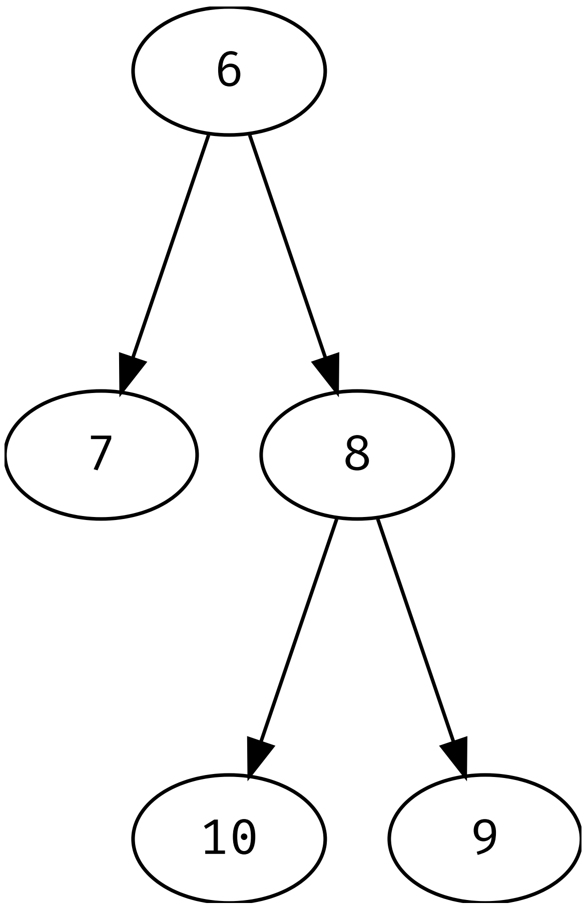



```
assert not r.is_leaf
assert r.left.is_leaf
assert not r.left.right.is_leaf
```

Getting the nth child

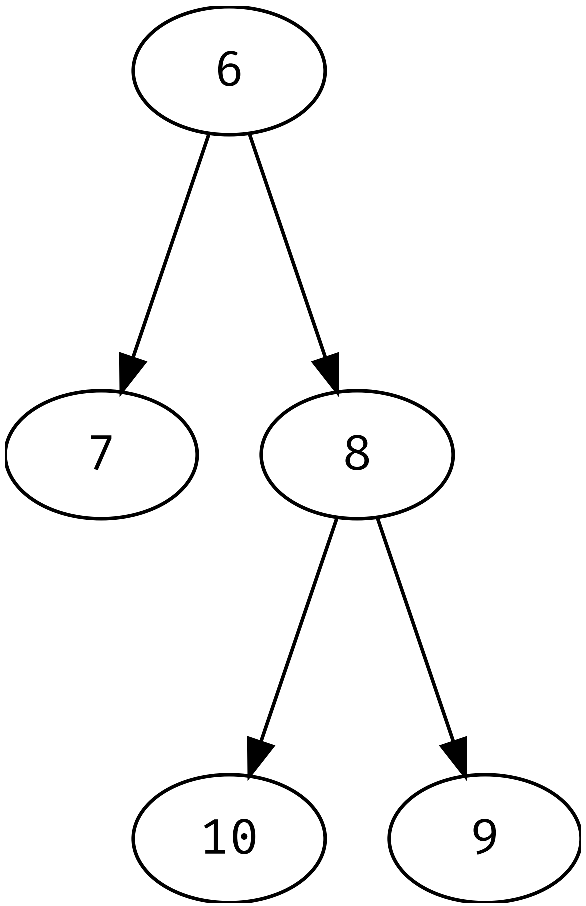
```python
@patch
def __getitem__(self:LCRSNode, key:int) -> LCRSNode:
  c, p = 0, self.left
  while p is not None:
    if c == key:
      return p
    c += 1
    p = p.right
  raise IndexError
```

Checking




```
assert not r.is_leaf
assert r[0].is_leaf
assert not r[1].is_leaf
assert r[1][0].is_leaf
```

Boom 💣




```python
try:
      r[2]
except Exception as e:
  assert \
    isinstance(e, IndexError)
```

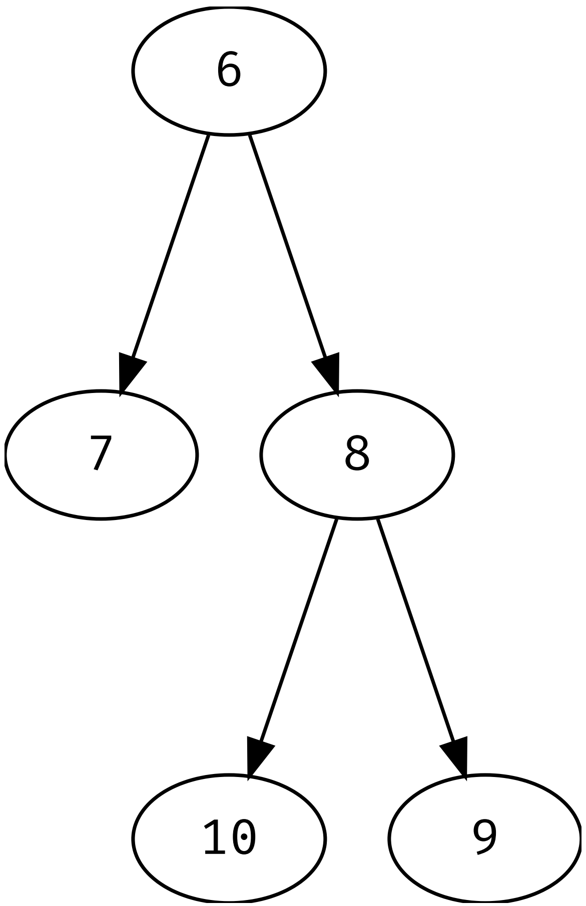
Size and height

```
@patch(as_prop=True)
def size(self:LCRSNode) -> int:
  l, r = 0, 0
  if self.left is not None:
    l = self.left.size
  if self.right is not None:
    r = self.right.size

  return 1 + l + r
```

Size and height

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

Checking




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

The big tree

- Code

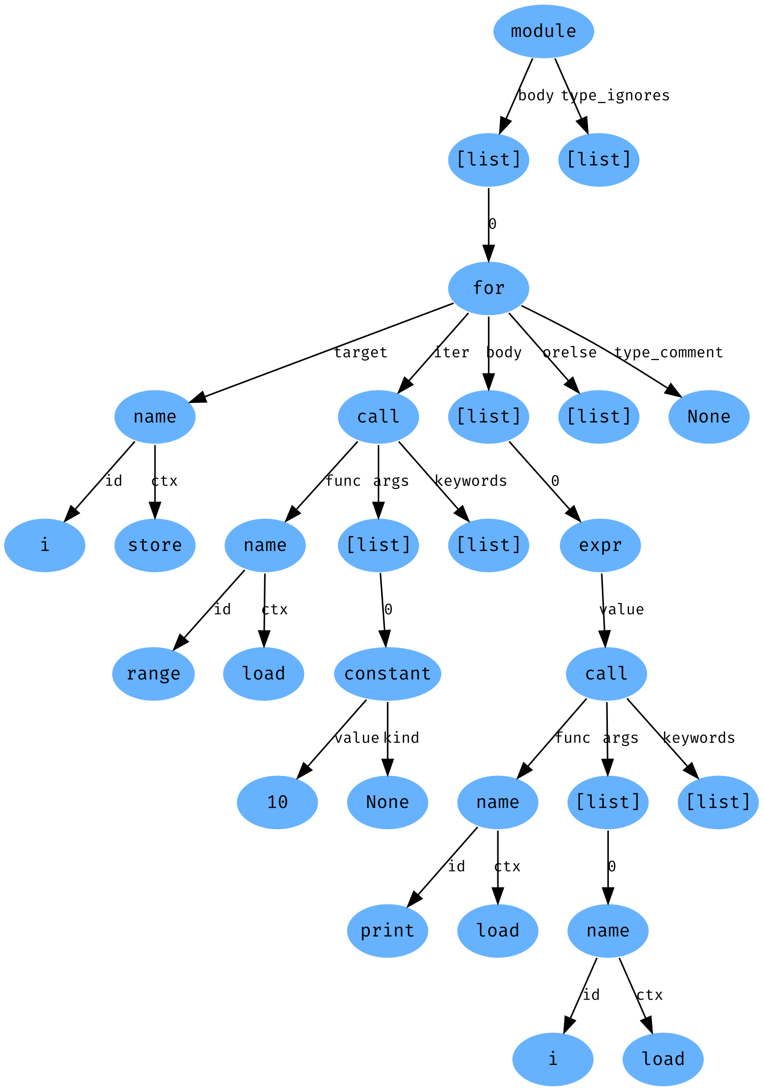
```
r.height=3, should be 3
r.degree=6, should be 6
r.size=16, should be 16
r[3].height=2 (e), should be 2
r[4].degree=3 (f), should be 3
```

Trees

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

Example

```python
for i in range(10):
    print(i)
```

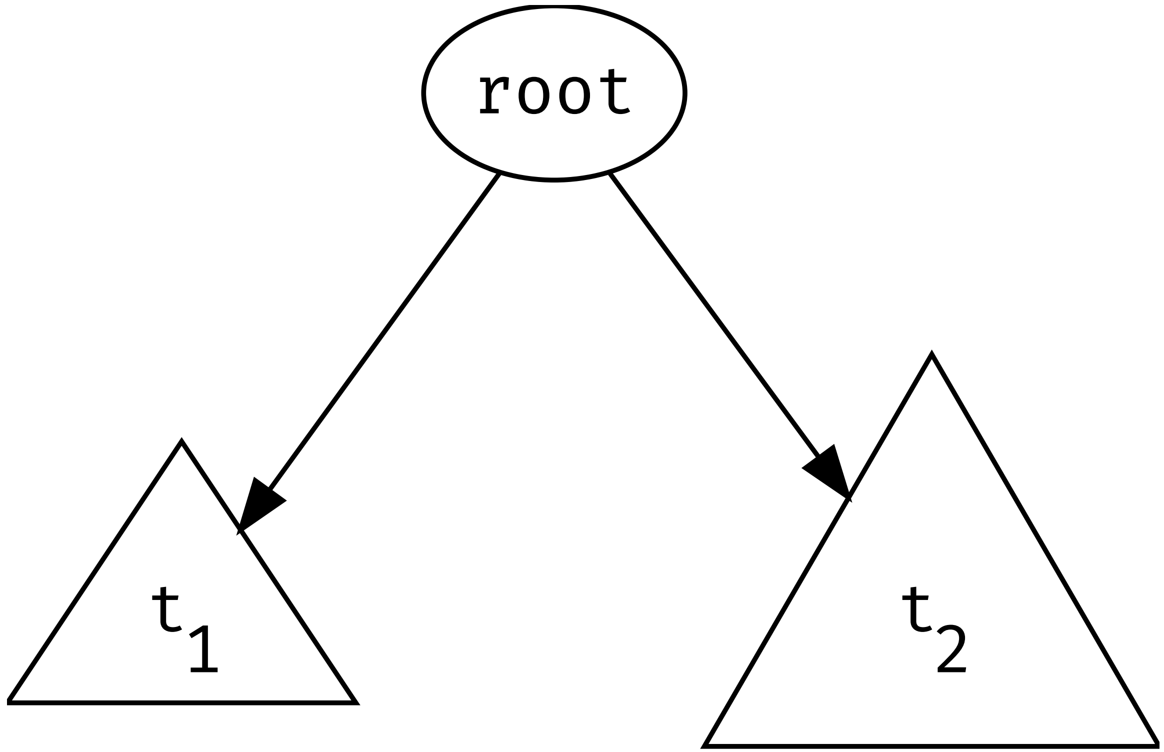


Binary Search Trees

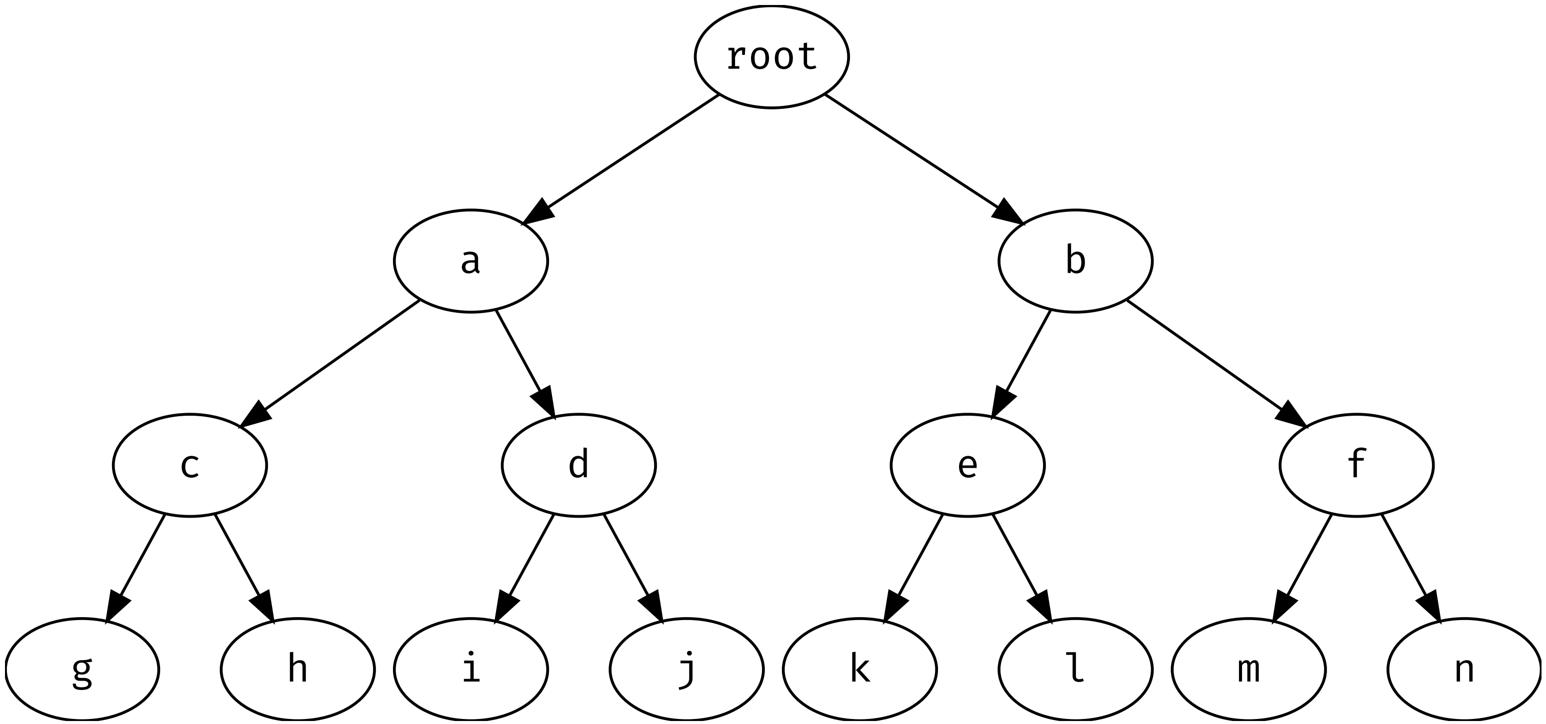
Binary trees

- 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 $\Theta(\sqrt{n})(saysthebook)$
  - a “full” tree has height $\lceil \log_2(n) \rceil - 1$
  - a “degenerate” tree has height $n - 1$

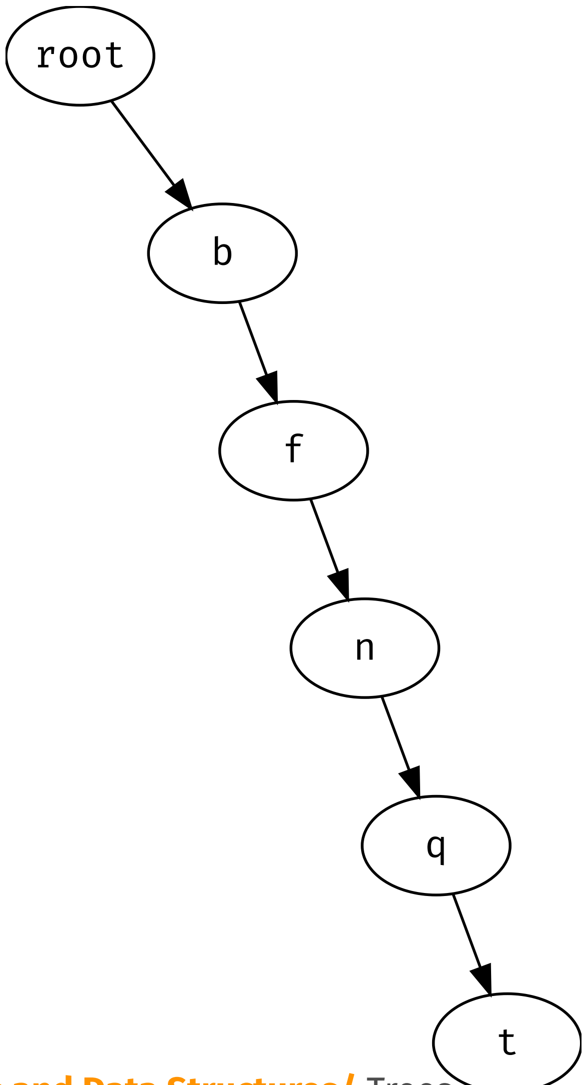
A binary tree



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

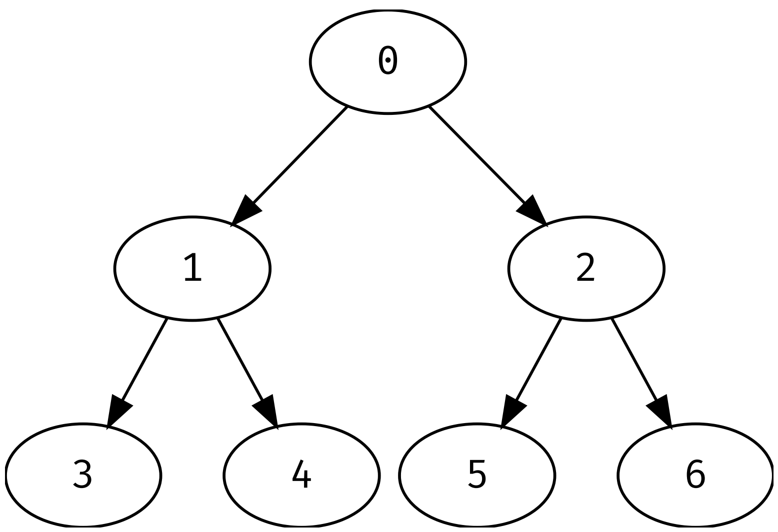
Implementing a binary tree

```python
@dataclass
class BTNode:
  key: int
  left: 'BTNode|None' = None
  right: 'BTNode|None' = None
```

Note

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

Building a small tree




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

Walking the tree

```python
def inorder(r:BTNode):
    if r is None:
        return ''
    else:
        s = inorder(r.left)
        s += f' {r.key} '
        s += inorder(r.right)
        return s
```

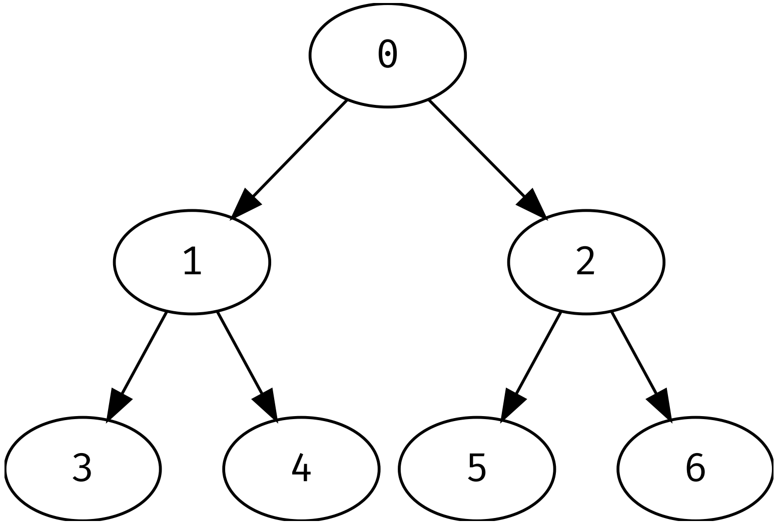
Walking the tree

```python
def preorder(r:BTNode):
    if r is None:
        return ''
    else:
        s = f' {r.key} '
        s += preorder(r.left)
        s += preorder(r.right)
        return s
```

Walking the tree

```python
def postorder(r:BTNode):
    if r is None:
        return ''
    else:
        s = postorder(r.left)
        s += postorder(r.right)
        s += f' {r.key} '
        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
```

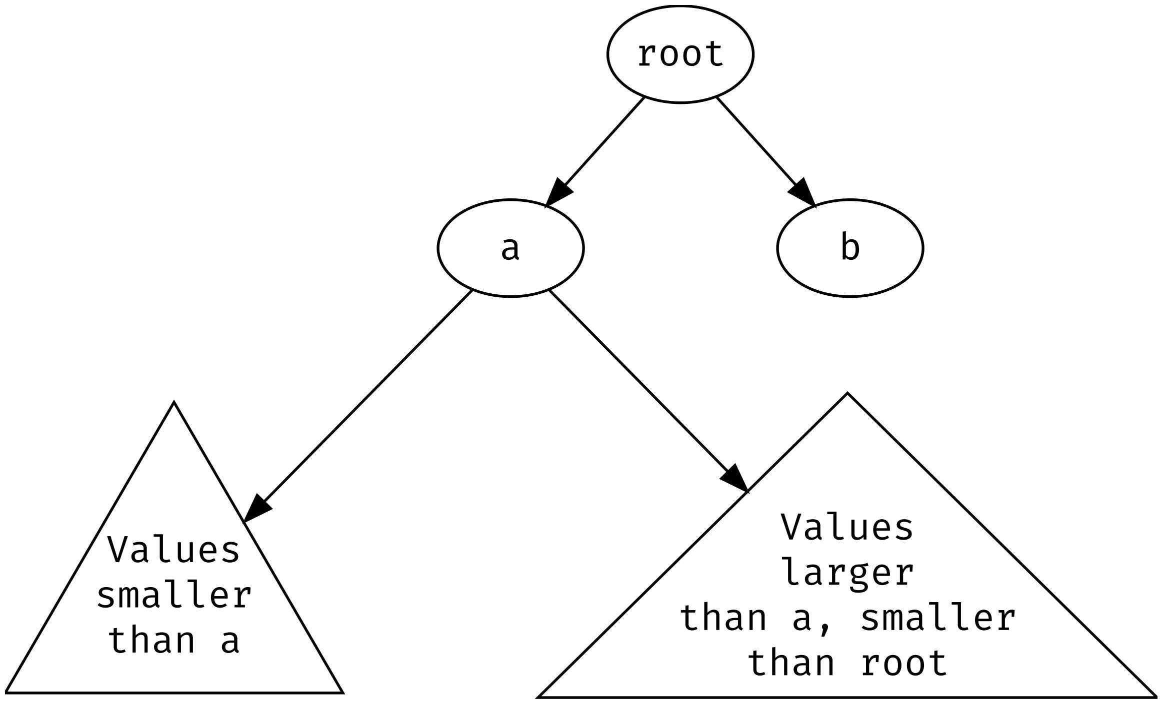
Creating a tree class

```python
class BST:
  def __init__(self) -> None:
    self.root = None
```

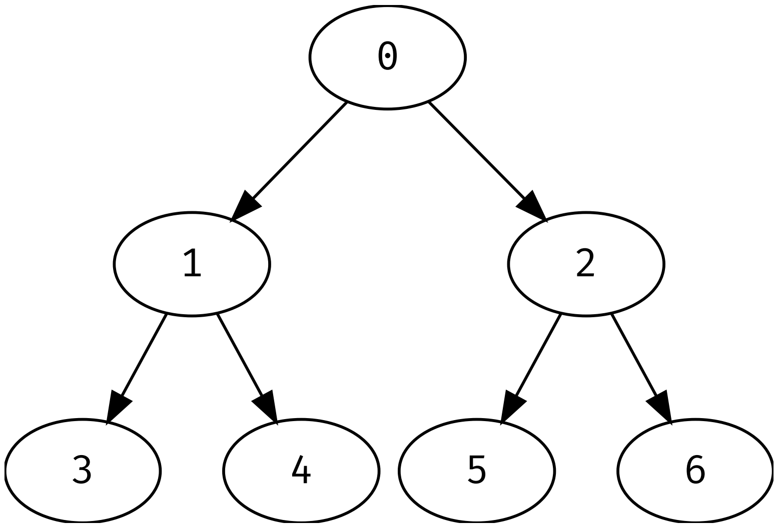
How do we insert?

- 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

```python
@dataclass
class LLNode:
  key: int
  nxt: 'LLNode|None' = None
```

Recursive insert

```python
def lladd(l:LLNode|None, key:int) -> LLNode:
    if l is None:
        return LLNode(key)
    l.nxt = lladd(l.nxt, key)
    return l

lst = None
lst = lladd(lst, 5)
lst = lladd(lst, 7)
print(lst)
```

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

Inserting a value

```python
@patch
def _add(self:BST, n:BTNode|None, key:int) -> BTNode|None:
  if n is None:
    return BTNode(key)

  if n.key > key:
    n.left = self._add(n.left, key)
  elif n.key < key:
    n.right = self._add(n.right, key)

  return n
```

Inserting a value

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

Building a tree

```
t = BST()
t.add(5)
t.add(2)
t.add(7)

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!

```python
@patch
def _inorder(self:BST, n:BTNode|None) -> None:
  if n is not None:
    self._inorder(n.left)
    print(n.key)
    self._inorder(n.right)
```

We need methods to walk the tree!

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

What about an iterator?

- Slightly more complicated
- We rely on recursive calls to keep track of where we are in the tree
- and do not have this implicit information in the iterator
- So, we use a stack to keep track of ancestors
  - Remember, recursive calls uses a stack

Inorder iterator

```python
class InorderIter:
  def __init__(self, n:BTNode|None) -> None:
    self.stack = []
    self._pushLCs(n)

  def _pushLCs(self, n:BTNode|None) -> None:
    while n is not None:
      self.stack.append(n)
      n = n.left
```

Inorder iterator

```
@patch
def __next__(self:InorderIter) -> BTNode:
  if self.stack:
    tmp = self.stack.pop()

    if tmp.right is not None:
      self._pushLCs(tmp.right)

    return tmp
  else:
    raise StopIteration
```

Inorder iterator

```
@patch
def __iter__(self:InorderIter) -> InorderIter:
  return self
```

Inorder iterator

```python
@patch
def __iter__(self:BST) -> InorderIter:
  return InorderIter(self.root)
```

Building a tree

```
t = BST()
t.add(5)
t.add(2)
t.add(7)


for n in t:
  print(n.key)
```

2
5
7

And to check if a value exists

```python
@patch
def _contains(self:BST, n:BTNode|None, key:int) -> bool:
  if n is None:
    return False

  if n.key > key:
    return self._contains(n.left, key)
  elif n.key < key:
    return self._contains(n.right, key)
  else:
    return True
```

And to check if a value exists

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

Testing

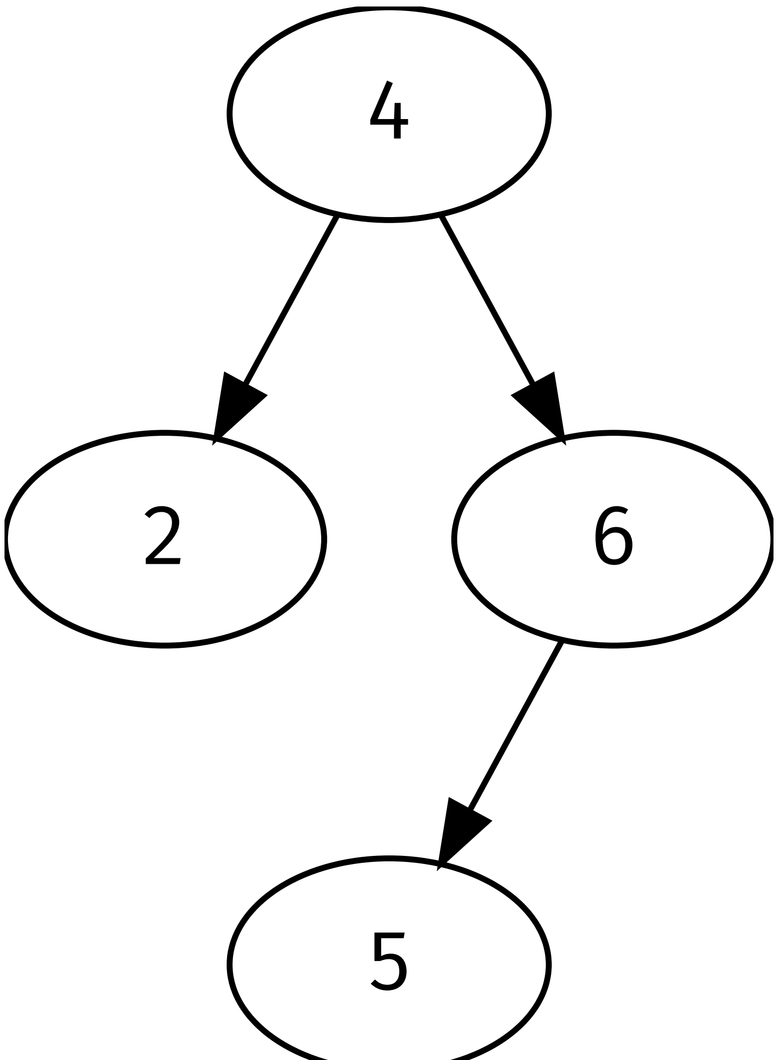
```python
t = BST()
t.add(5)
t.add(2)
t.add(7)

assert 2 in t
assert 7 in t
assert 8 not in t
```

Recursive delete

```python
def lldel(l:LLNode|None, key:int) -> LLNode:
    if l is None:
        return None
    if l.key == key:
        return l.nxt
    else:
        l.nxt = lldel(l.nxt, key)
        return l

# lst = [5, 7]
lst = lldel(lst, 5)
print(lst)
```

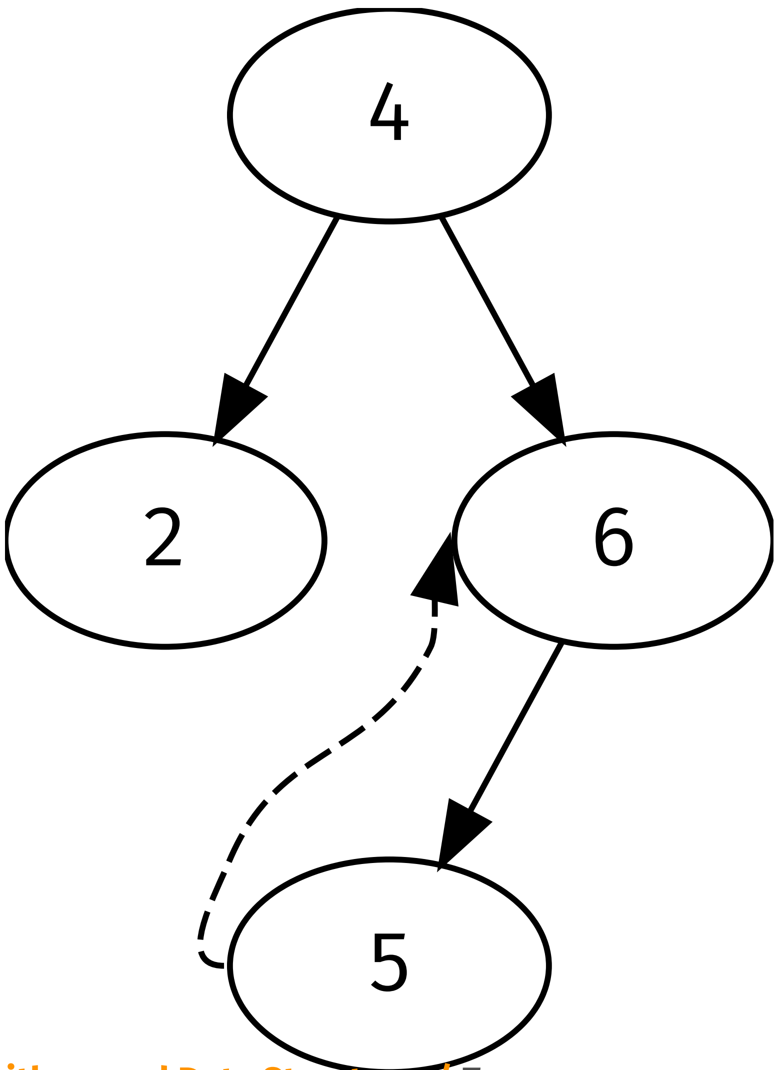
```
LLNode(key=7, nxt=None)
```

Deleting



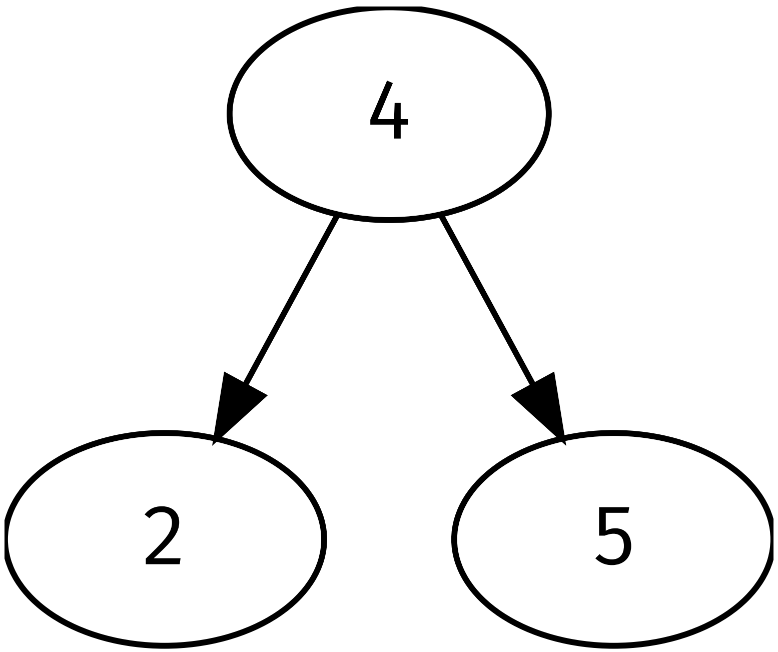
» Assume we want to delete *6*
» If the node has one child, we "lift" it

Deleting



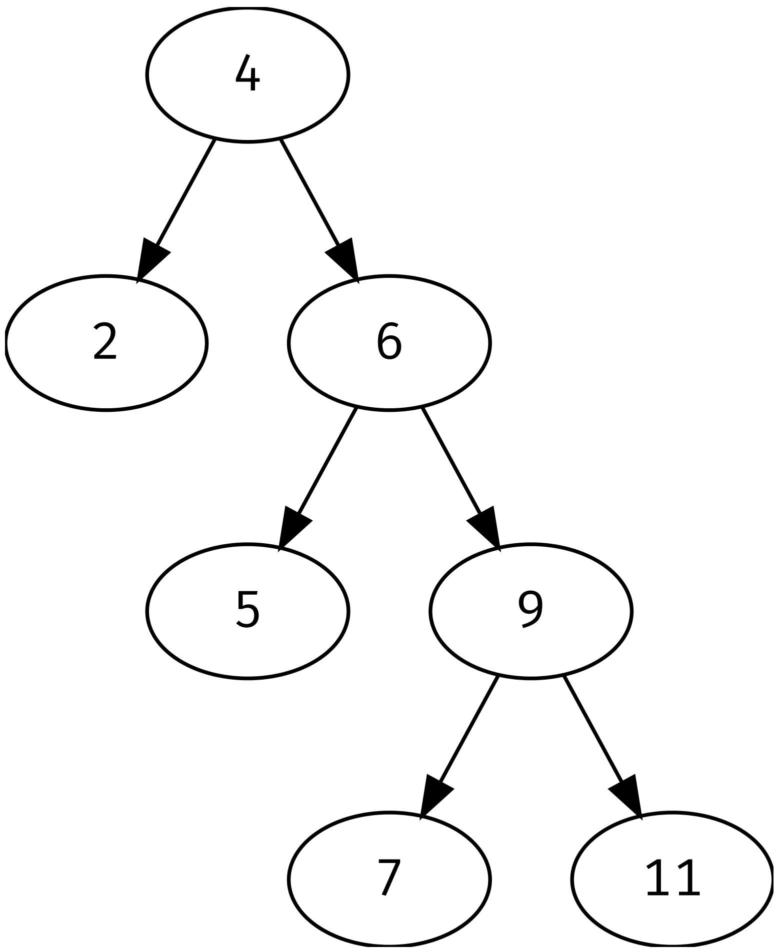
- Assume we want to delete *6*
- If the node has one child, we “lift” it

Deleting



- Assume we want to delete *6*
- If the node has one child, we "lift" it

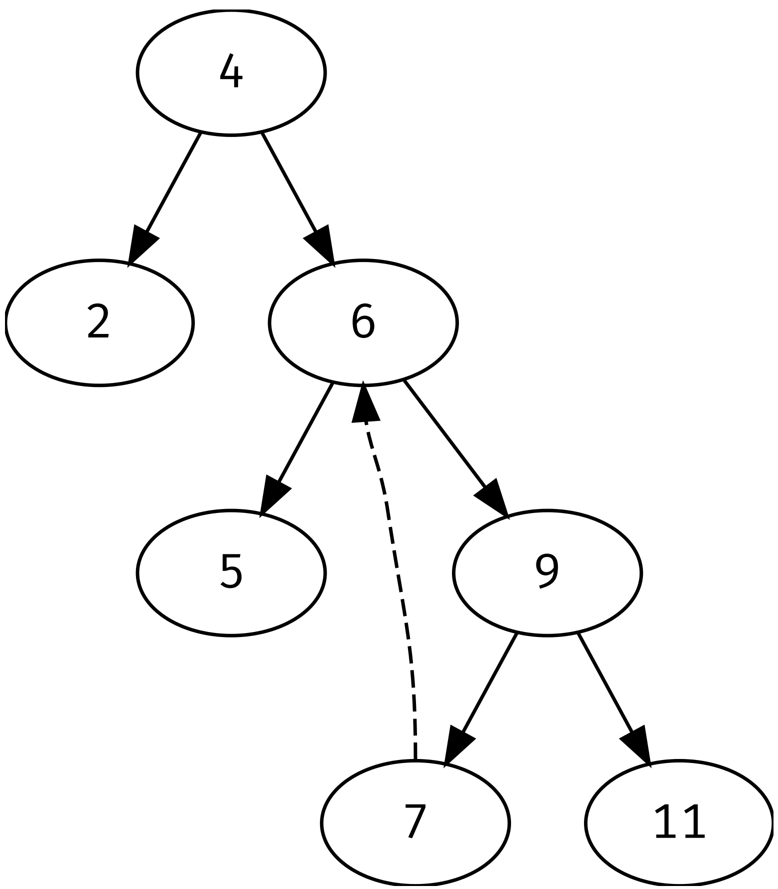
Deleting



» Assume we want to delete *6*

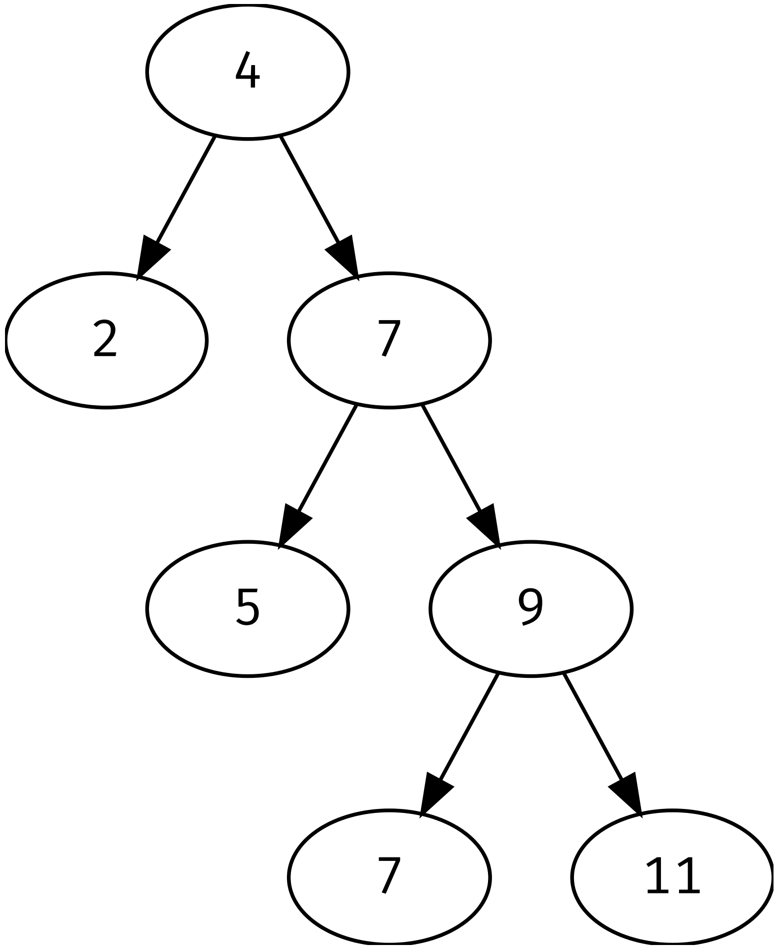
» Trickier when it has two children!

Deleting



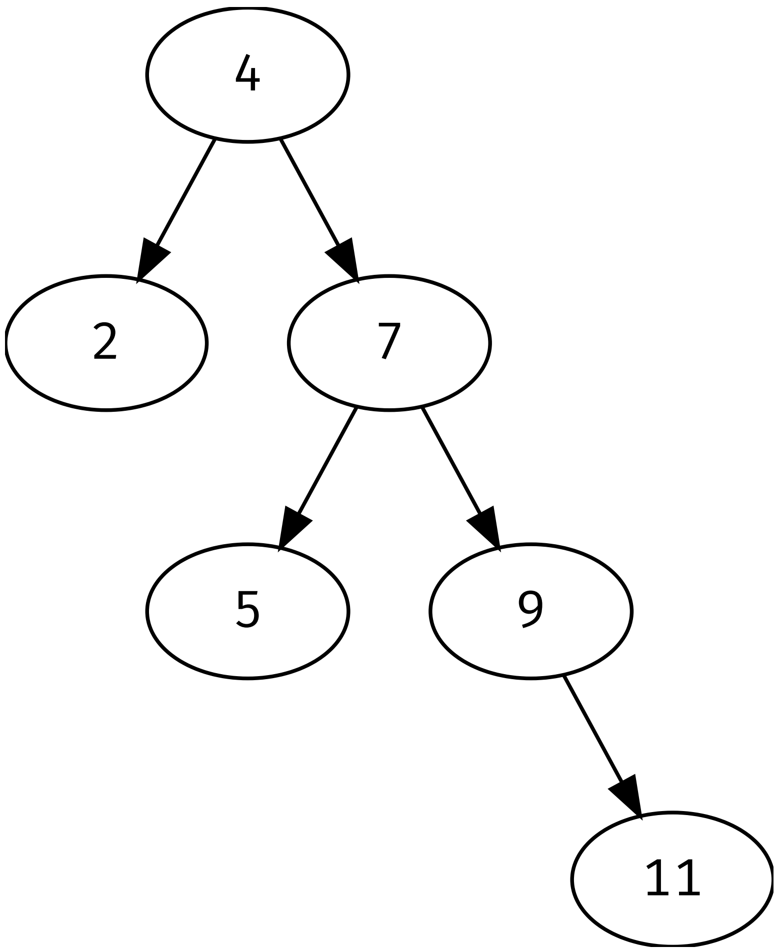
- Assume we want to delete *6*
- Trickier when it has two children!
- We replace the node with the smallest value in the right subtree
    - Cannot have two childen

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 childen

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 childen

Deleting

```python
@patch
def _delete(self:BST, n:BTNode|None, key:int) -> BTNode|None:
  if n is None:
    return None
  if n.key > key:
    n.left = self._delete(n.left, key)
  elif n.key < key:
    n.right = self._delete(n.right, key)
  else:
    if n.right is None:
      return n.left
    if n.left is None:
      return n.right
    n.key = self._min(n.right)
    n.right = self._delete(n.right, n.key)
  return n
```

Finding the smallest node in a subtree

```python
@patch
def _min(self:BST, n:BTNode) -> int:
  if n.left is None:
    return n.key
  else:
    return self._min(n.left)
```

Deleting

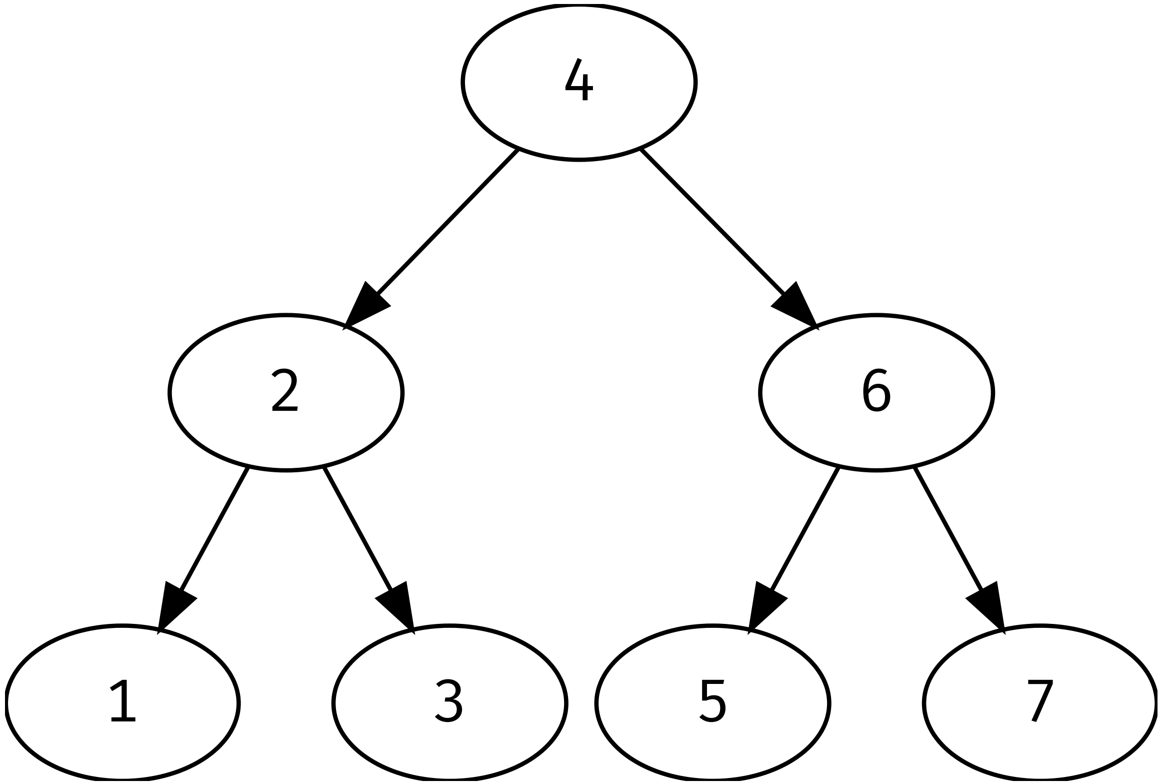
```python
@patch
def delete(self:BST, key:int) -> None:
  self.root = self._delete(self.root, key)
```

Building a tree

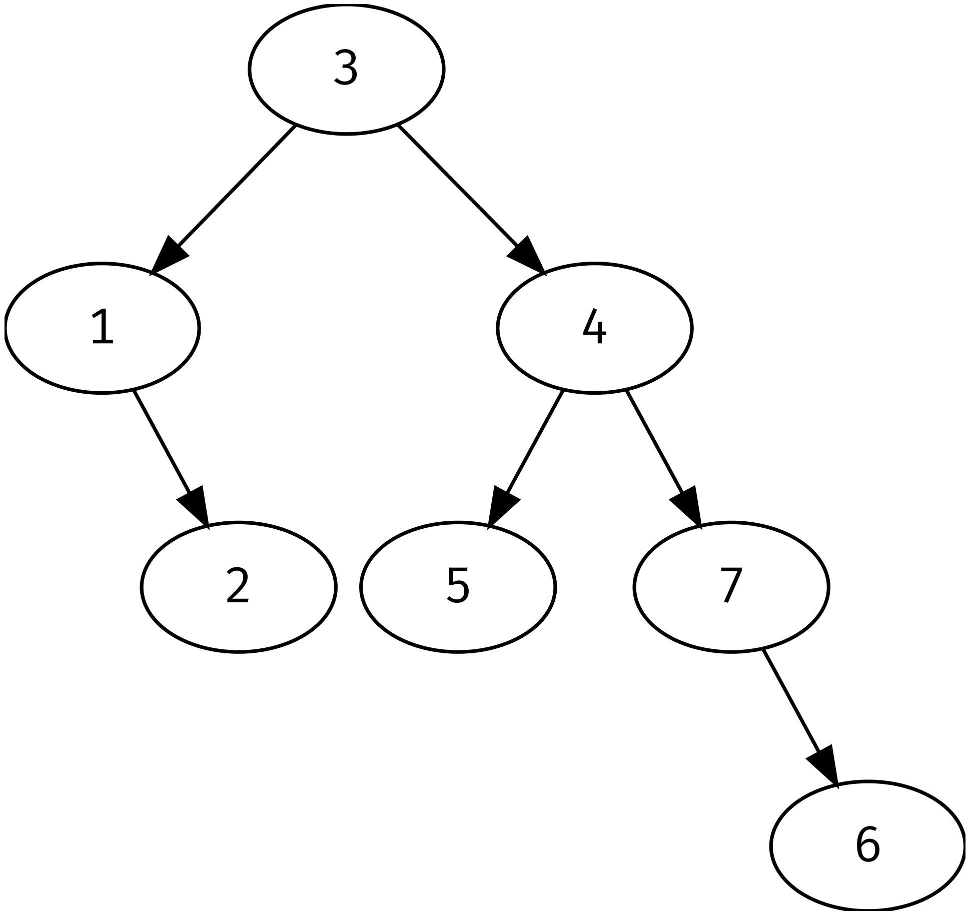
```python
t = BST()
t.add(5)
t.add(2)
t.add(7)

t.print_inorder()
print('---')
t.delete(2)
t.print_inorder()
```

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

Height



Height



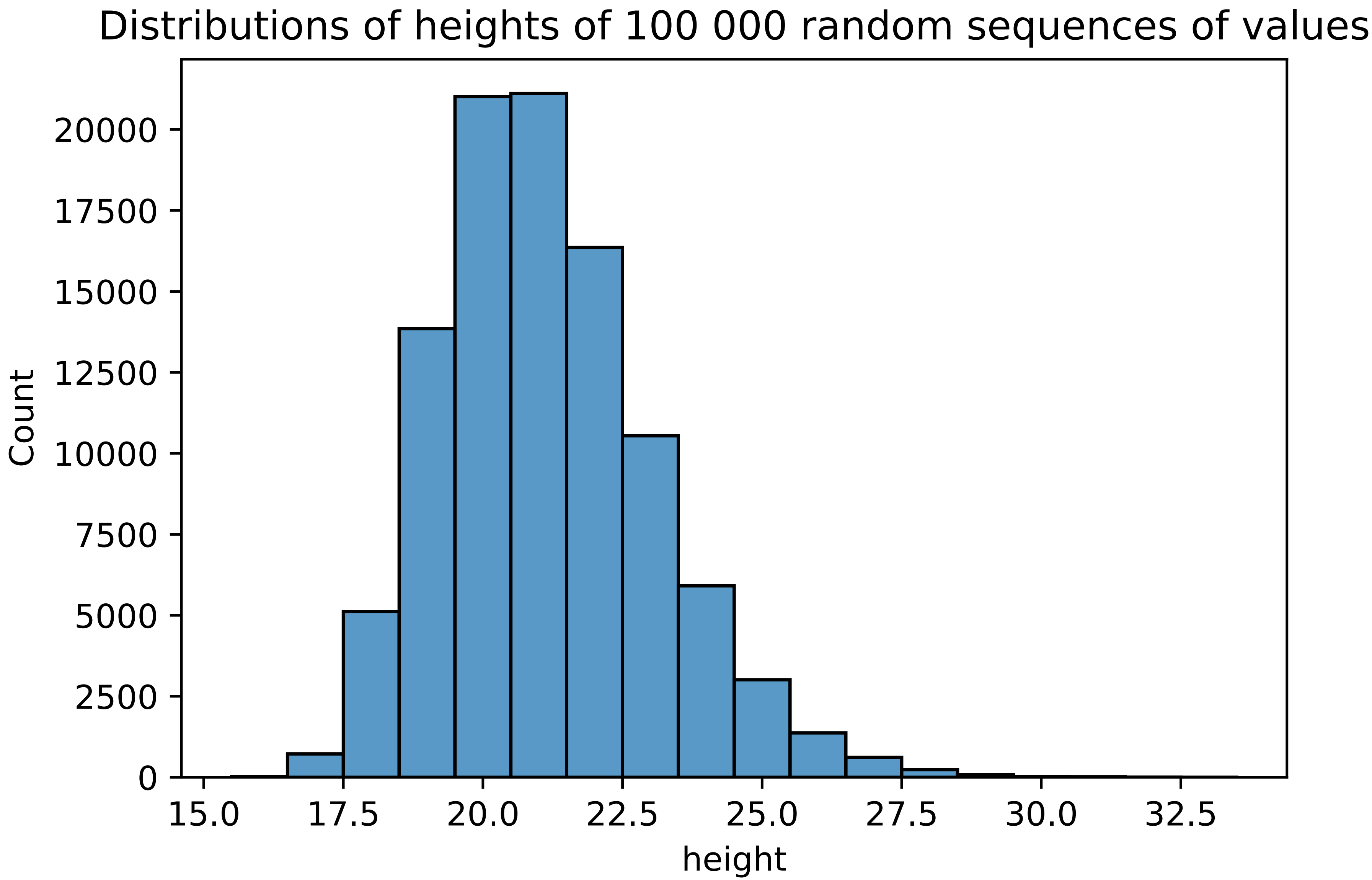
Computing height

```python
@patch
def _height(self:BST, n:BTNode|None) -> int:
  if n is None:
    return -1
  else:
    return 1 + max(self._height(n.left), \
                   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?

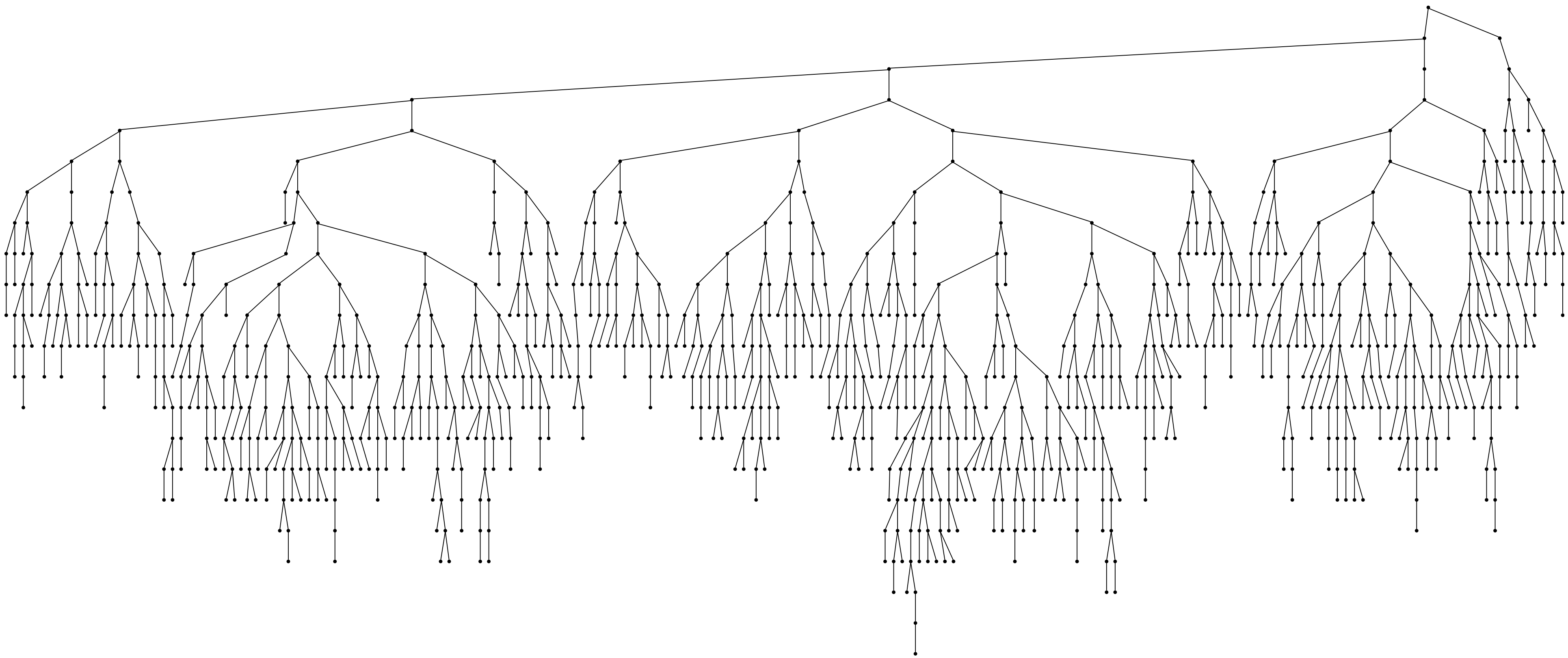
Example (n = 1023)



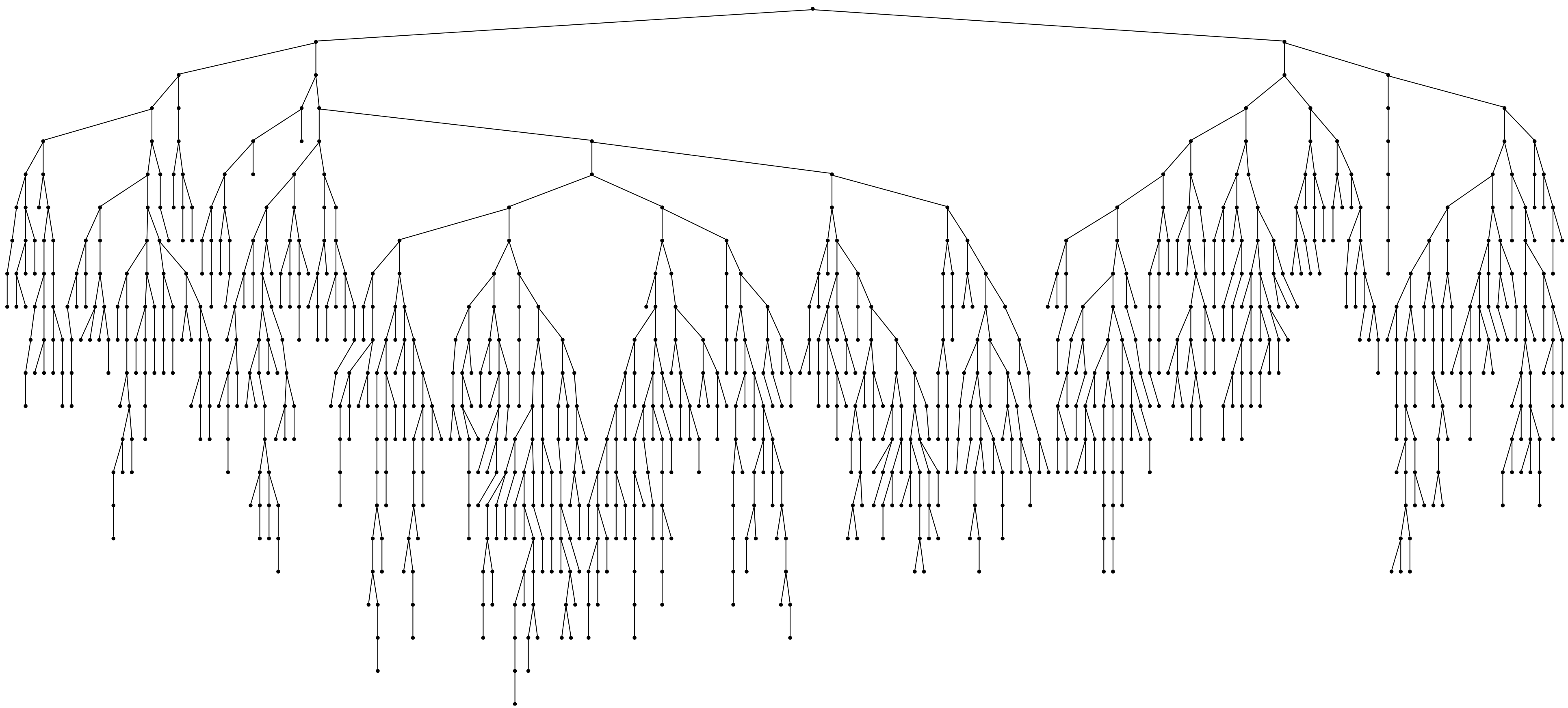
A balanced tree?

- 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

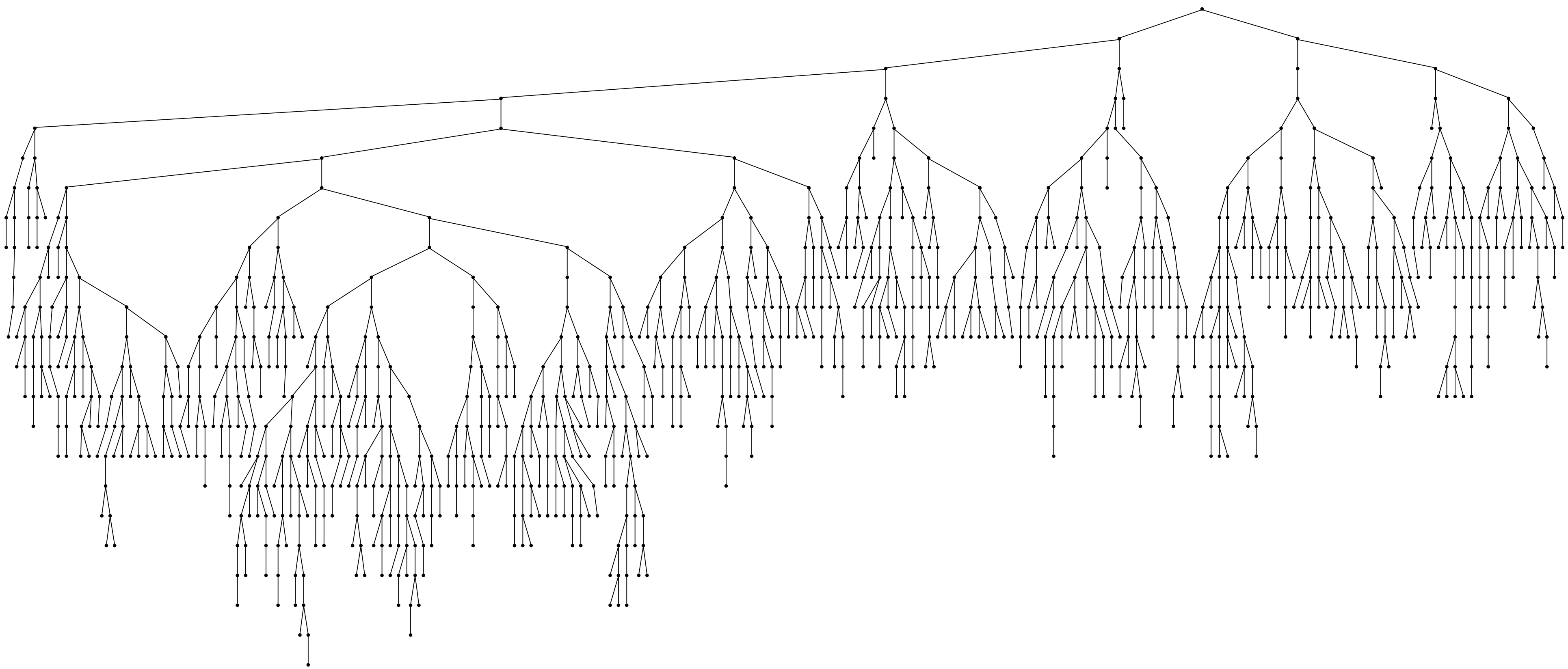
A balanced tree?



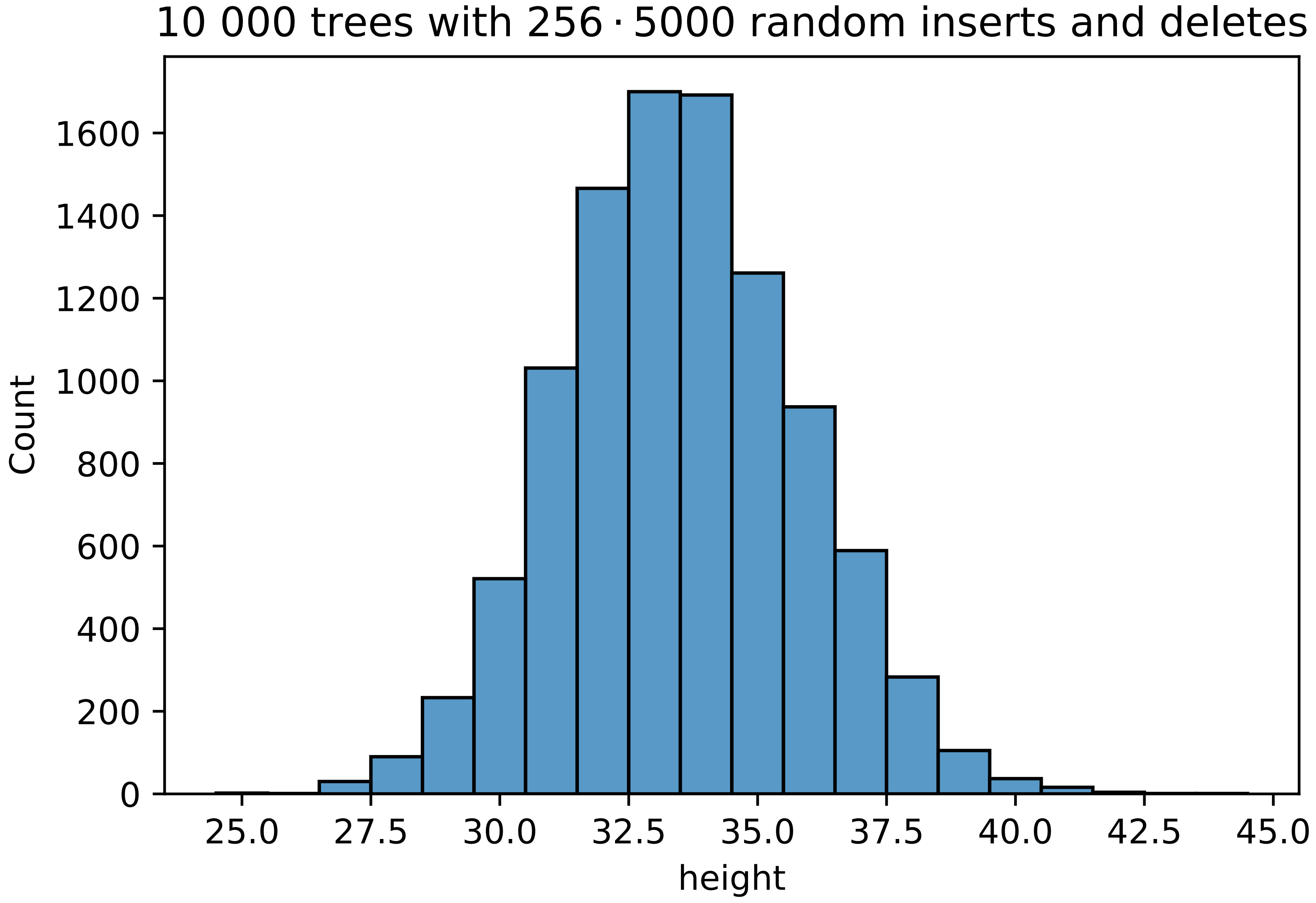
A balanced tree?



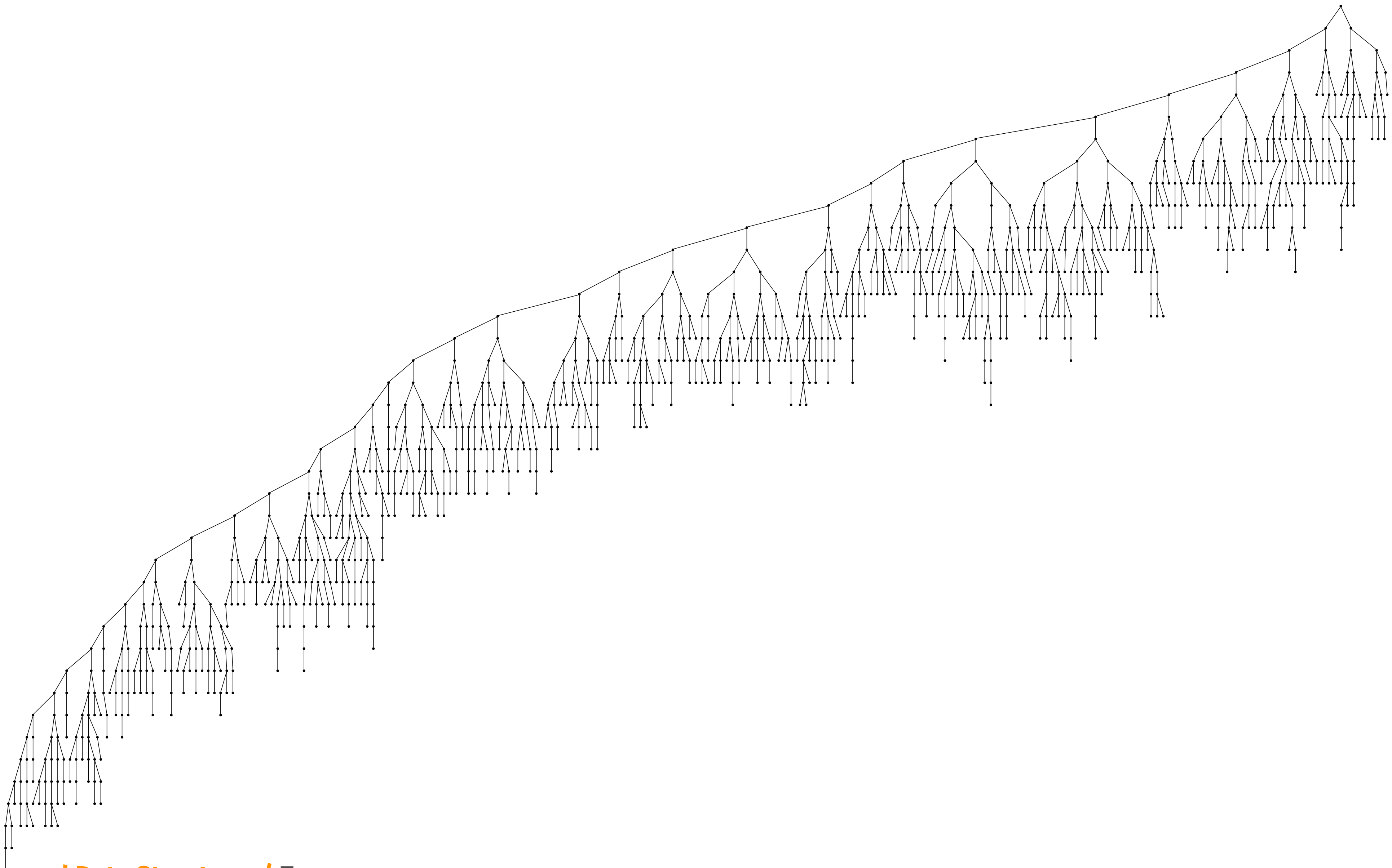
A balanced tree?



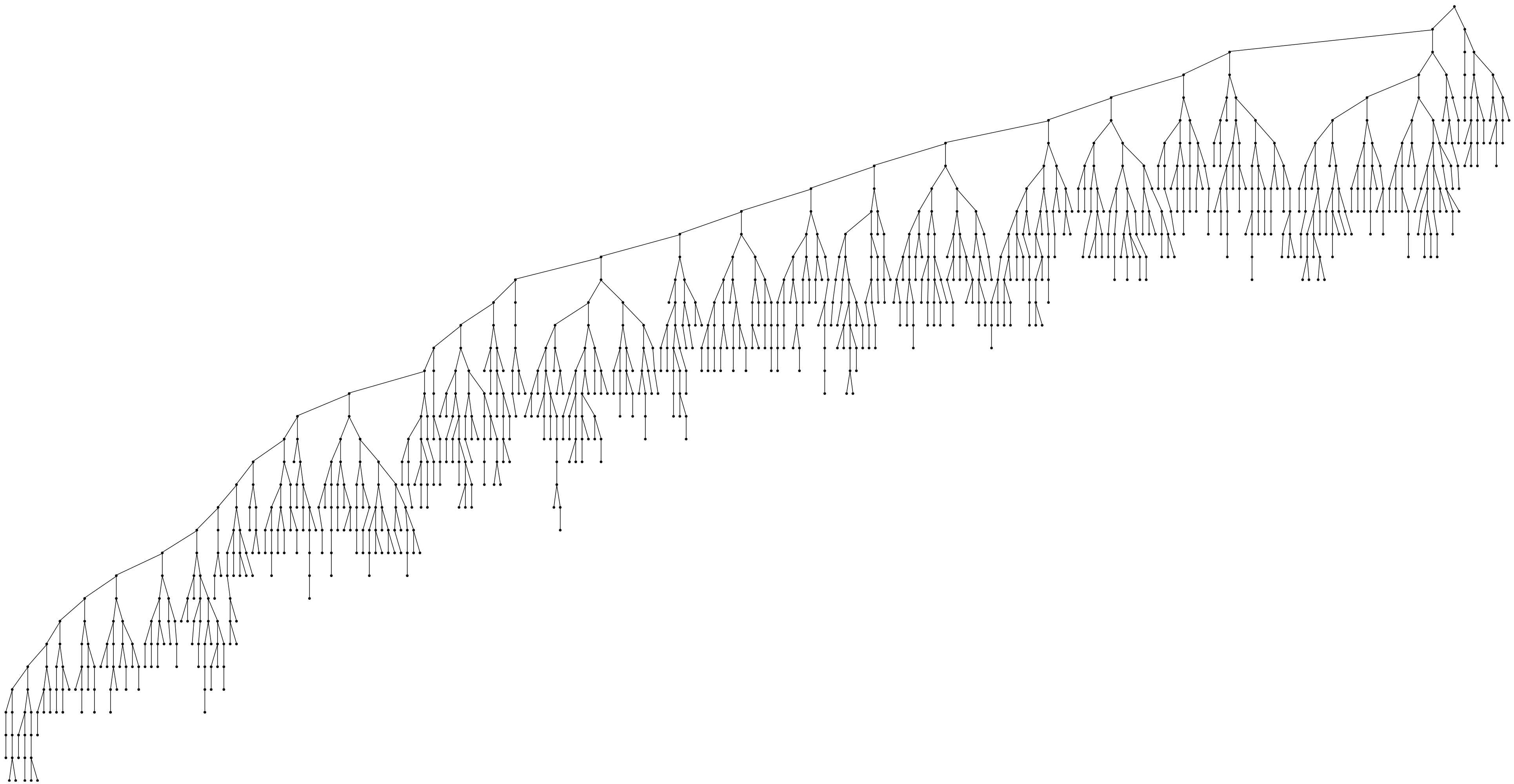
Adding deletes



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

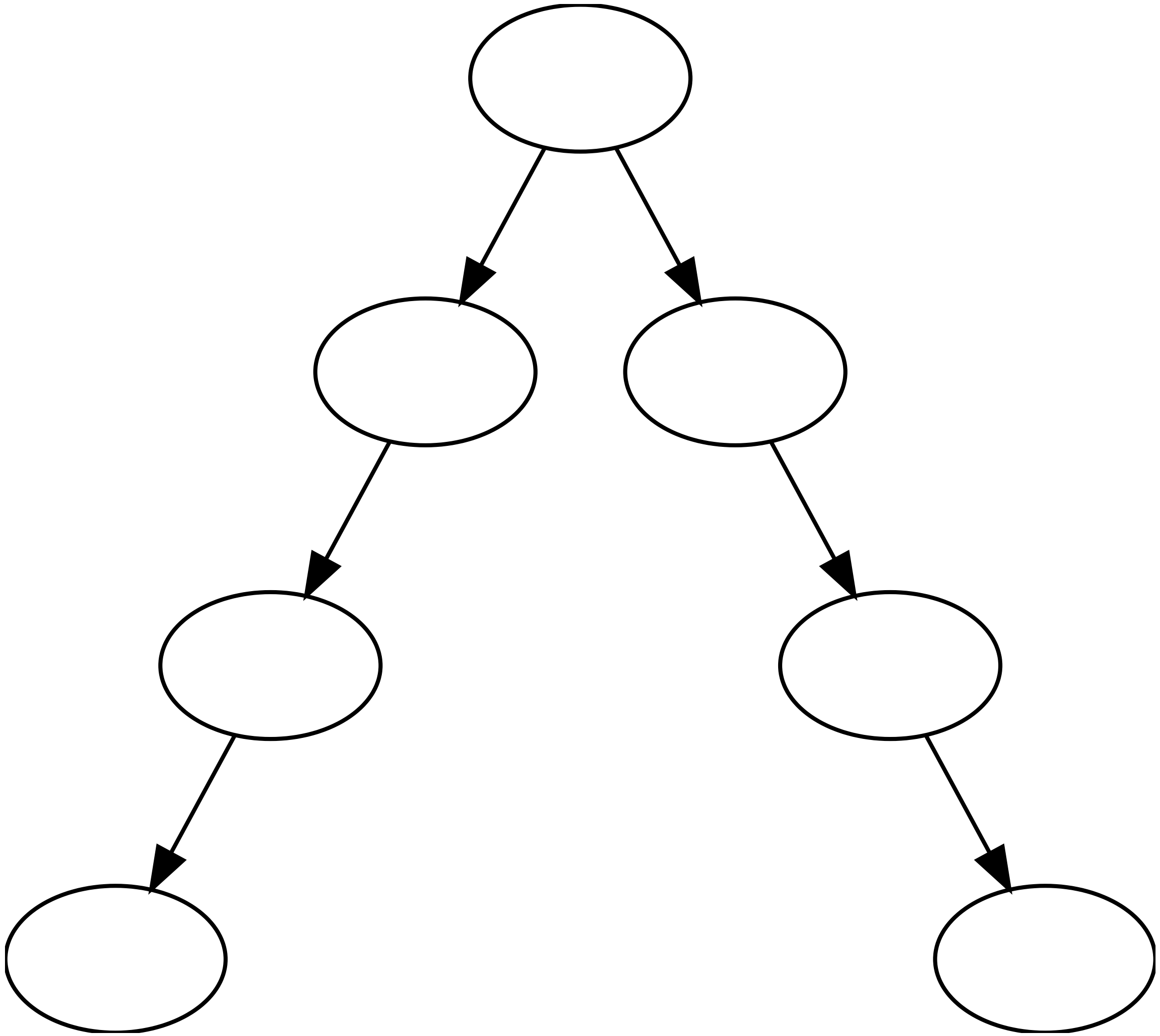
AVL-tree

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

Balance condition

- Should ensure that the depth of the tree is $O(logN)$
- 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

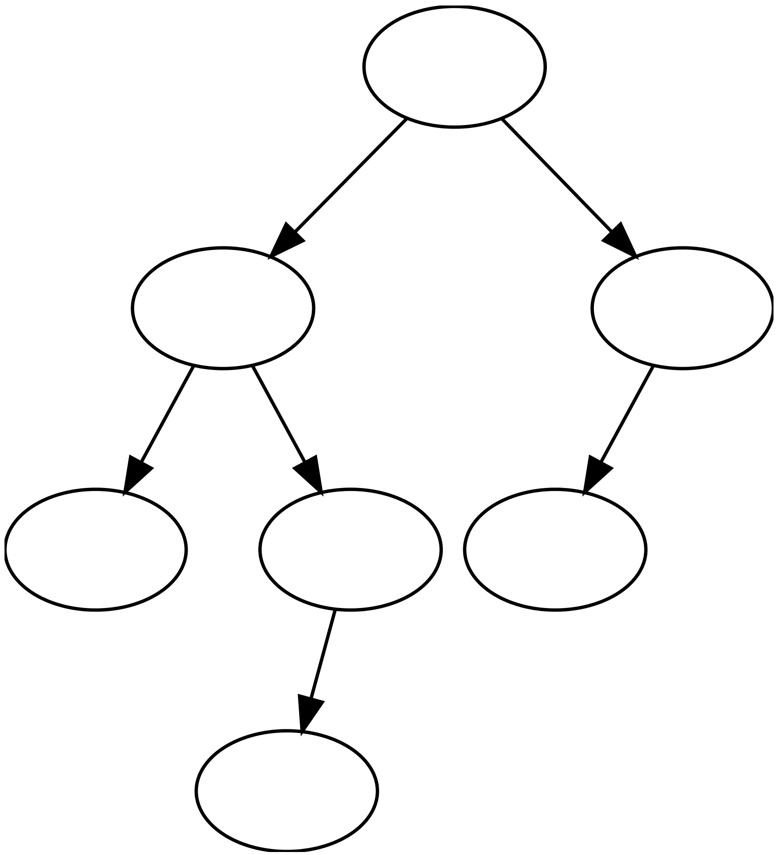
- Balance at root is not enough
- So, each node should have left and right subtrees of the same height
  - Would force perfectly balanced trees
  - But too difficult to maintain

AVL-trees

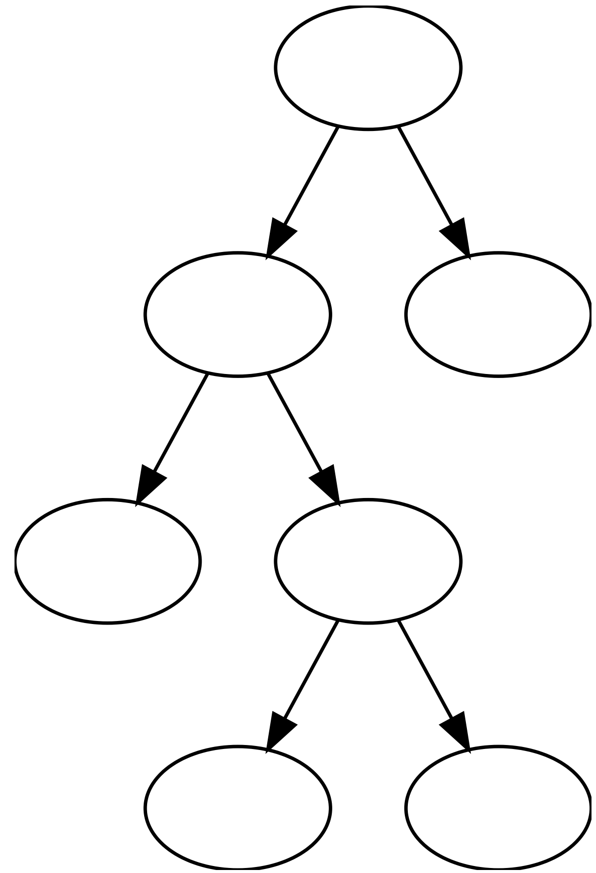
- The heights of the left and right subtrees can differ by at most 1
- Gives a height of about $1.44 \cdot \log_2(N + 2) - 1.328$
  - More than $\log_2$, but not that much
- Minimum nodes at at a height
  - $S(h) = S(h - 1) + s(h - 2) + 1$
- So, a tree with height 9 has at least 143 nodes

AVL-trees

AVL



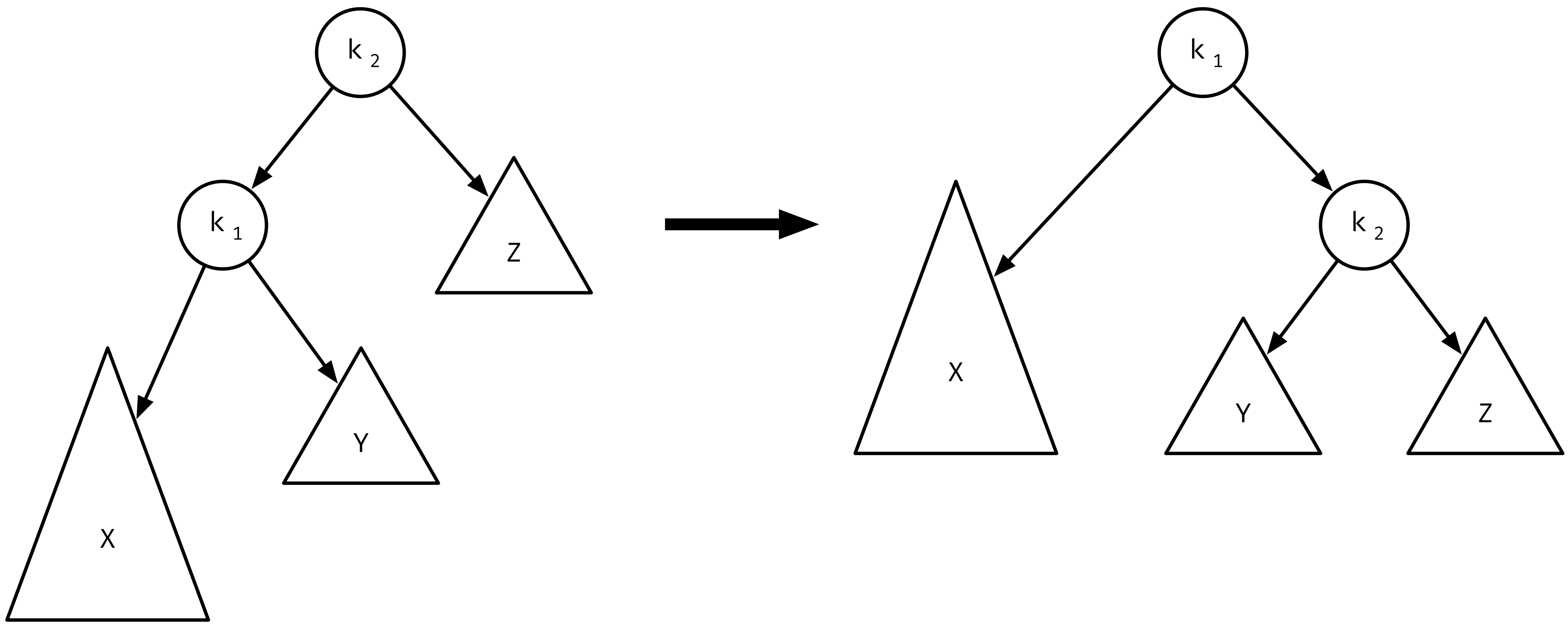
Not AVL



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

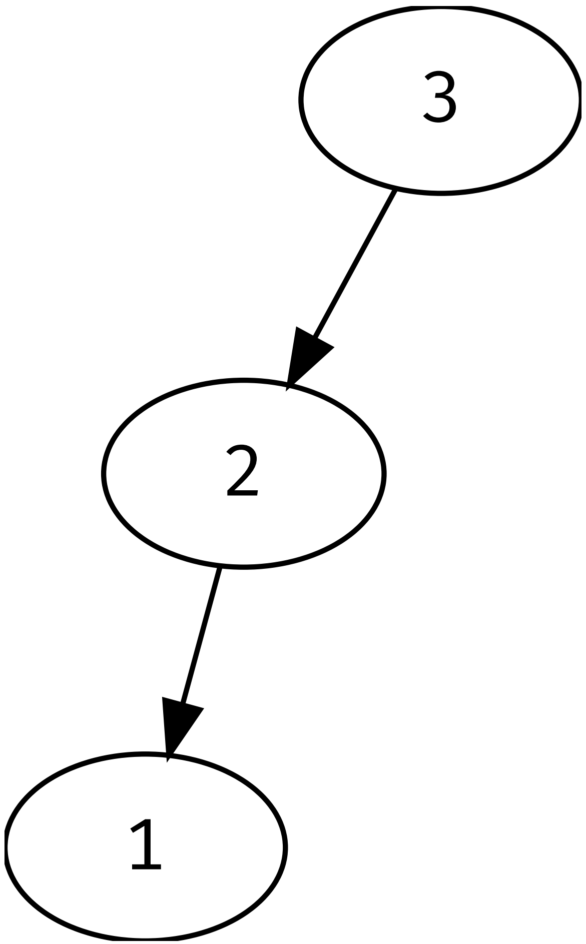
Single rotation (LL and RR)



What is going on?

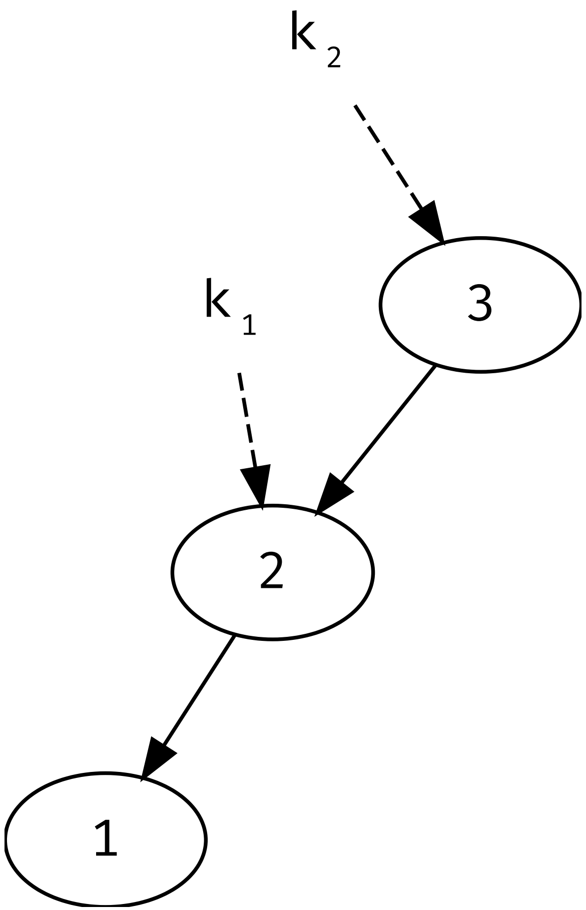
- Node k_2 (the root) is violating the balance condition
  - since X is two levels deeper than Z
  - A change to X caused the violation
- We can fix this by moving X higher and Y and Z lower
  - This means k_1 becomes root
  - and k_2 its right child, since $k_1 < k_2$
  - Y becomes the left child of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

Single rotation with numbers



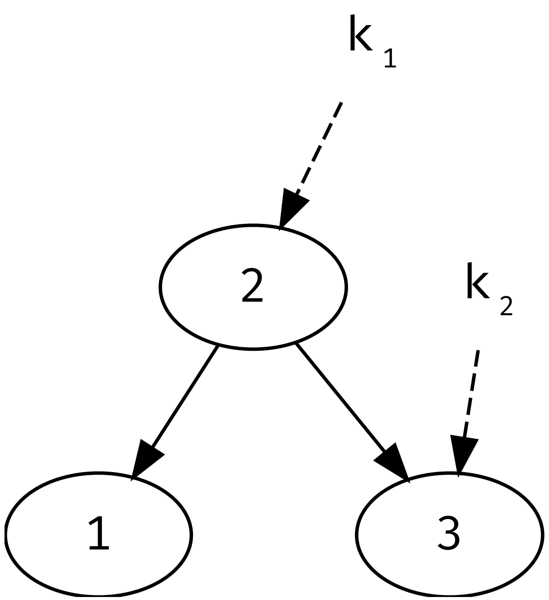
- Problematic: the subtrees of the root differ by more than 1
- We need to rotate at the root
- 2 should become the root and 3 its right child
- Called a left rotate

Single rotation with numbers



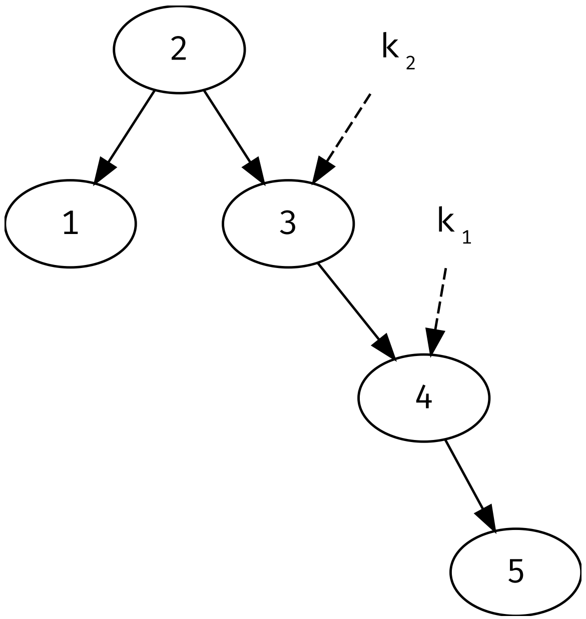
- Left rotate
  - k_1 is $k_2.left$
  - We change $k_2.left$ to $k_1.right$
  - and set $k_1.right$ to k_2

Single rotation with numbers



- Left rotate
  - k_1 is $k_2.\mathit{left}$
  - We change $k_2.\mathit{left}$ to $k_1.\mathit{right}$
  - and set $k_1.\mathit{right}$ to k_2

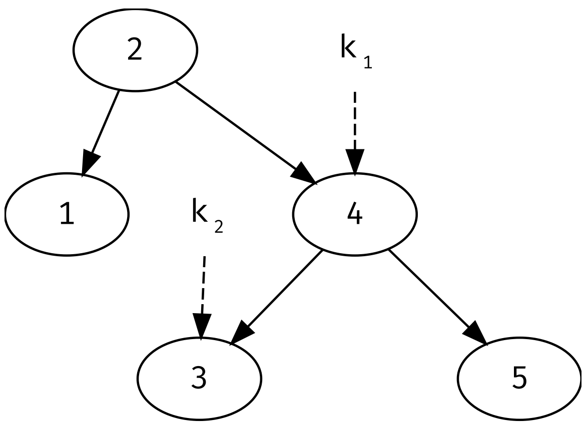
Single rotation with numbers



» 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

```python
@dataclass
class AVLNode:
  key: int
  left: 'AVLNode|None' = None
  right: 'AVLNode|None' = None
  height: int = 0
```

Implementation: AVLTree

```python
class AVLTree:
  def __init__(self) -> None:
    self.root = None
```

Implementation: Add

```python
@patch
def _add(self:AVLTree, n:AVLNode|None, key:int) -> AVLNode:
  if n is None:
    return AVLNode(key)

  if n.key > key:
    n.left = self._add(n.left, key)
  elif n.key < key:
    n.right = self._add(n.right, key)

  return self._balance(n)
```

Implementation: Add

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

Implementation: Balance

```python
@patch
def _balance(self:AVLTree, n:AVLNode|None) -> AVLNode|None:
  if n is None:
    return n

  if self._height(n.left) - self._height(n.right) > 1:
    if self._height(n.left.left) >= self._height(n.left.right):
      n = self._rotate_left(n)
    else:
      n = self._double_left(n)
  elif self._height(n.right) - self._height(n.left) > 1:
    if self._height(n.right.right) >= self._height(n.right.left):
      n = self._rotate_right(n)
    else:
      n = self._double_right(n)

  n.height = max(self._height(n.left), self._height(n.right)) + 1
  return n
```

Implementation: Height

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

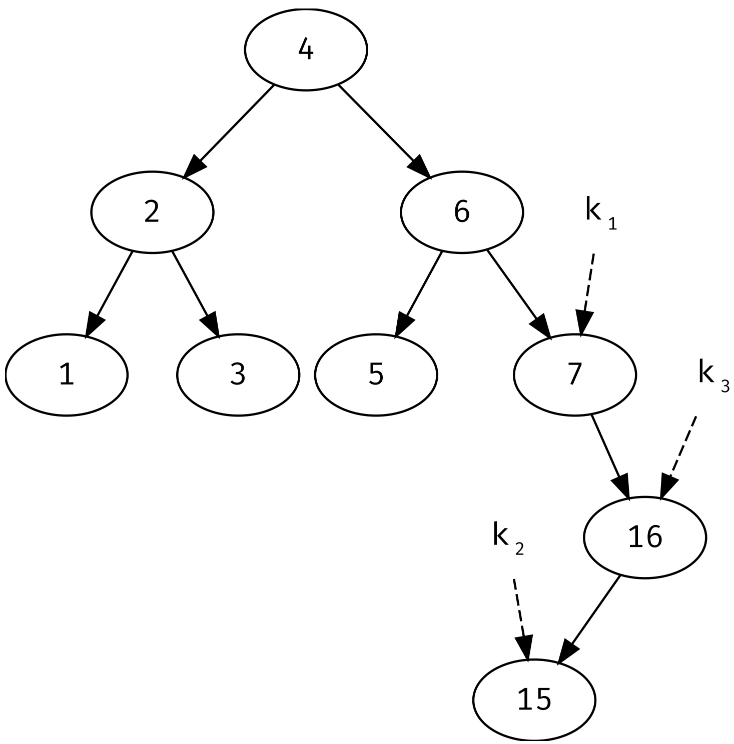
Implementation: Single rotate

```python
@patch
def _rotate_left(self:AVLTree, r2:AVLNode) -> AVLNode:
  r1 = r2.left
  r2.left = r1.right
  r1.right = r2
  r2.height = max(self._height(r2.left), self._height(r2.right)) + 1
  r1.height = max(self._height(r1.left), r2.height) + 1

  return r1
```

Implementation: Single rotate

```python
@patch
def _rotate_right(self:AVLTree, r2:AVLNode) -> AVLNode:
  r1 = r2.right
  r2.right = r1.left
  r1.left = r2
  r2.height = max(self._height(r2.left), self._height(r2.right)) + 1
  r1.height = max(self._height(r1.left), r2.height) + 1

  return r1
```

Double rotation

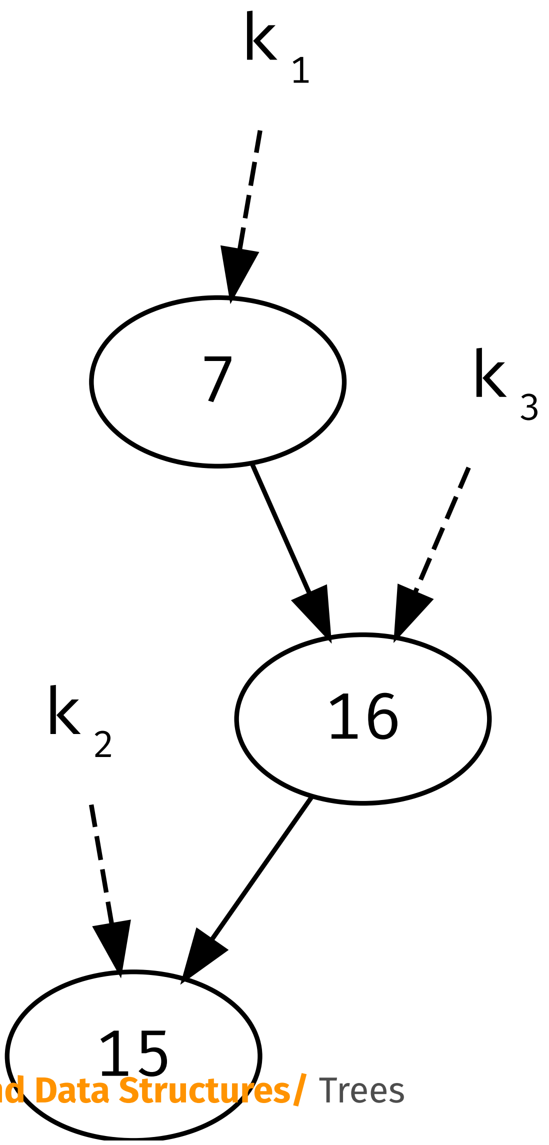


» Previously, we have seen LL and RR

» For RL (and LR), we need to rotate twice

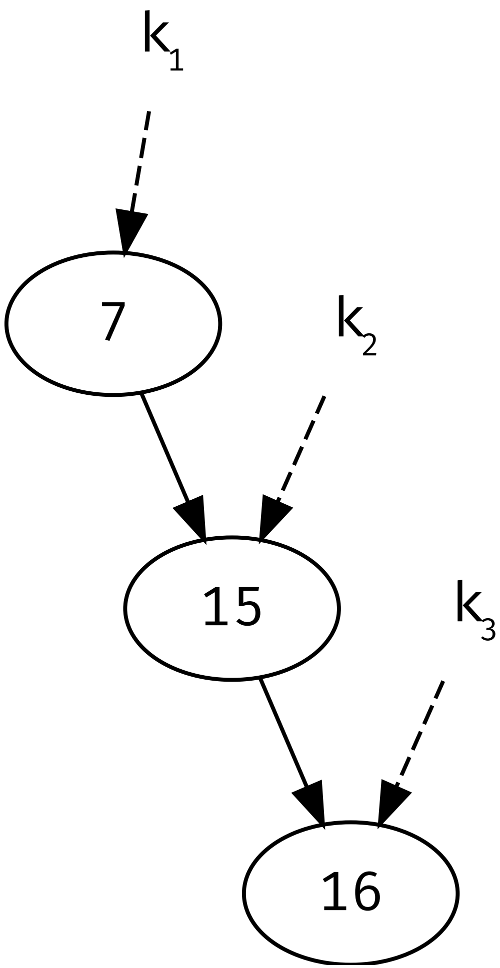
  » For RL, a “double right”

Double right



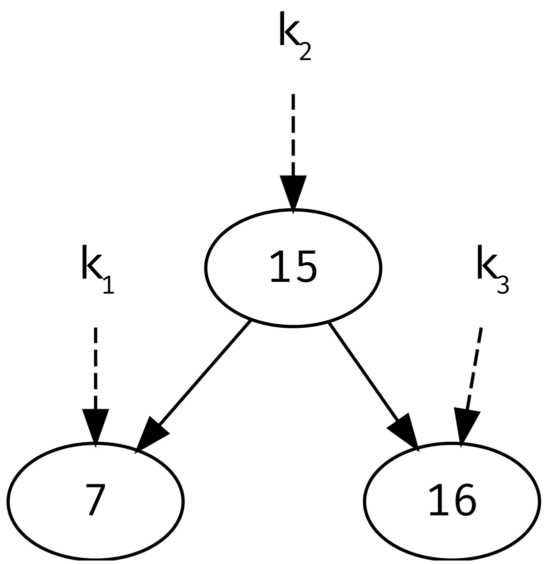
- A double right means
  - rotate right child left
  - rotate self right

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

```python
@patch
def _double_right(self:AVLTree, n:AVLNode) -> AVLNode:
  n.right = self._rotate_left(n.right)
  return self._rotate_right(n)

@patch
def _double_left(self, n:AVLNode) -> AVLNode:
  n.left = self._rotate_right(n.left)
  return self._rotate_left(n)
```

Adding preorder walk

```python
@patch
def print_preorder(self:AVLTree) -> None:
  self._preorder(self.root)

@patch
def _preorder(self:AVLTree, n:AVLNode|None) -> None:
  if n is not None:
    print(n.key)
    self._preorder(n.left)
    self._preorder(n.right)
```

Example

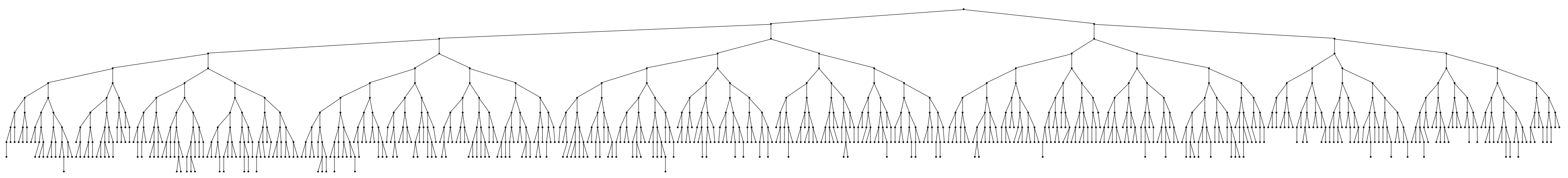
```python
t = AVLTree()
t.add(3)
t.add(2)
t.print_preorder()
print('----')
t.add(1)
t.print_preorder()
```

```
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...

```python
@patch
def _delete(self:AVLTree, n:AVLNode|None, key:int) -> AVLNode|None:
  if n is None:
    return None

  if n.key > key:
    n.left = self._delete(n.left, key)
  elif n.key < key:
    n.right = self._delete(n.right, key)
  else:
    if n.right is None:
      return n.left
    if n.left is None:
      return n.right
    n.key = self._min(n.right)
    n.right = self._delete(n.right, n.key)

  return self._balance(n)
```

Splay trees

Splay trees

- » Many applications have “data locality”
  - » A node is accessed multiple times within a reasonble 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 consective 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

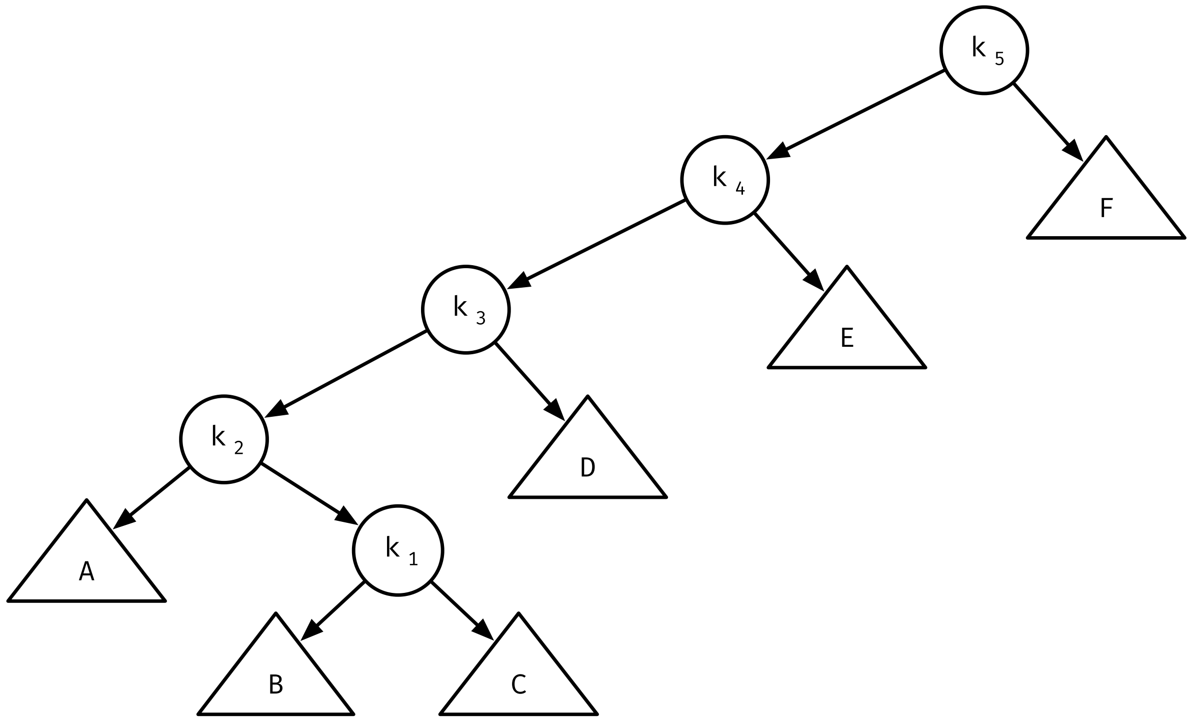
- If an operation is $O(n)$ and we want $O(\log_2 n)$ it is clear that we must do something to fix it
  - In splay trees, we fix by moving
  - So, if first $O(n)$, then consecutive close to $O(1)$

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

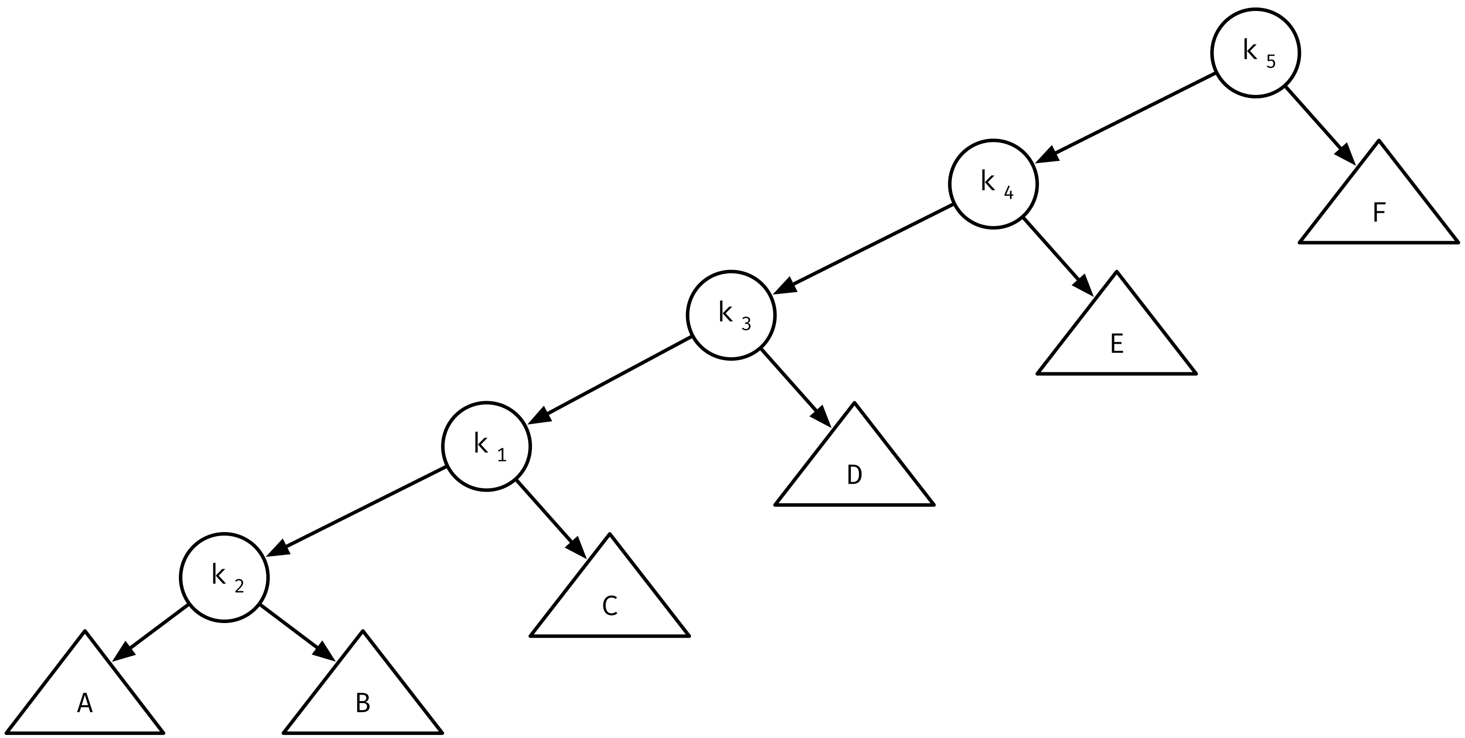
First idea

We search for k_1



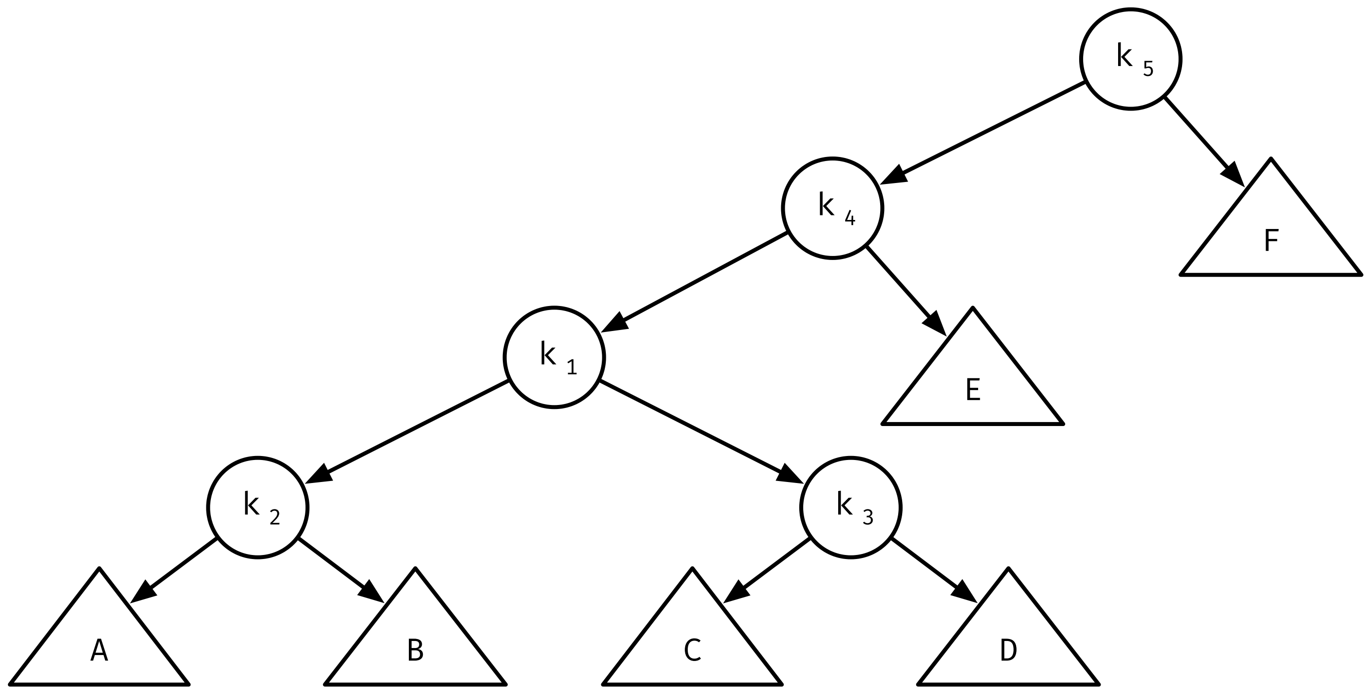
First idea

And rotate it upwards



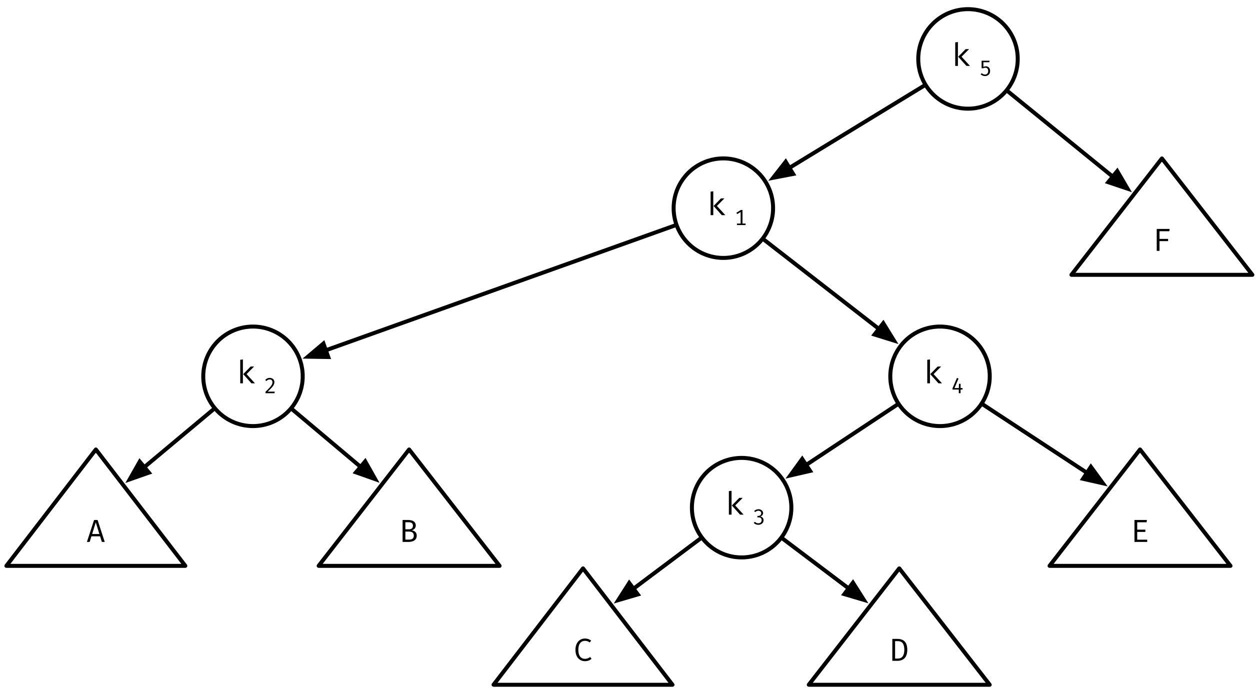
First idea

And rotate it upwards



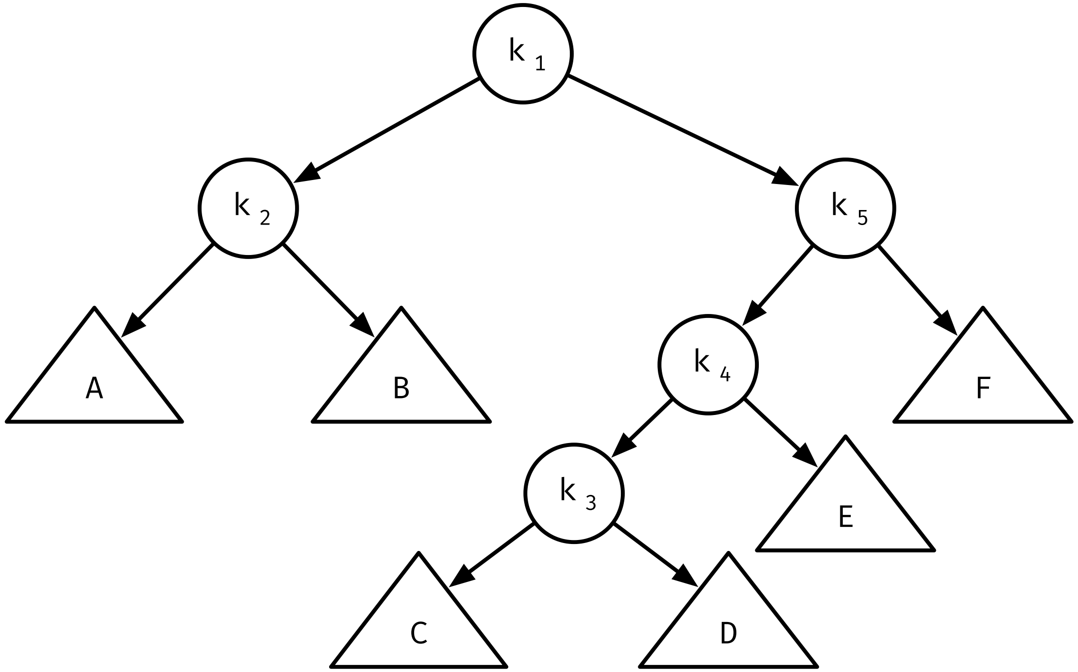
First idea

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

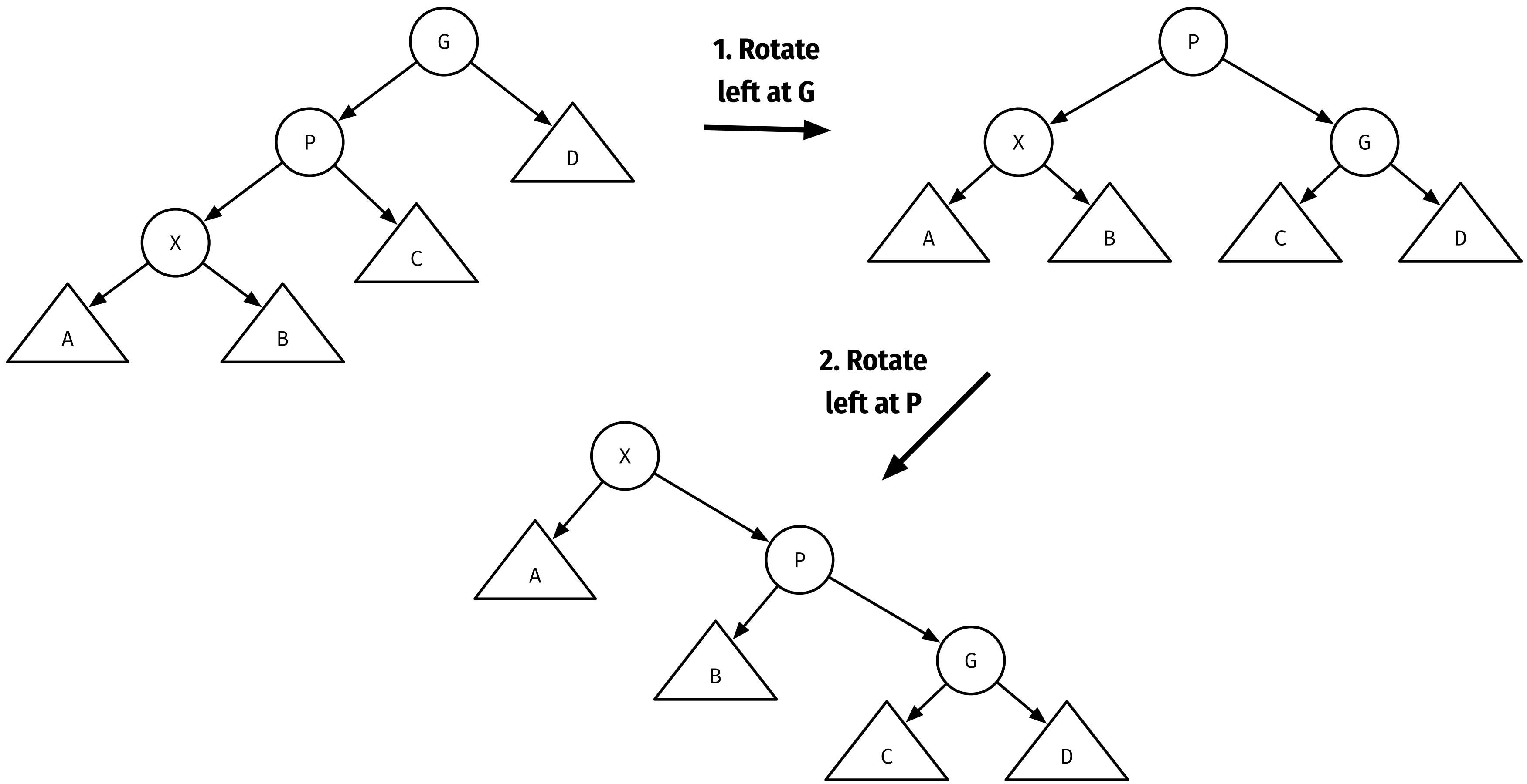
Better idea

- 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

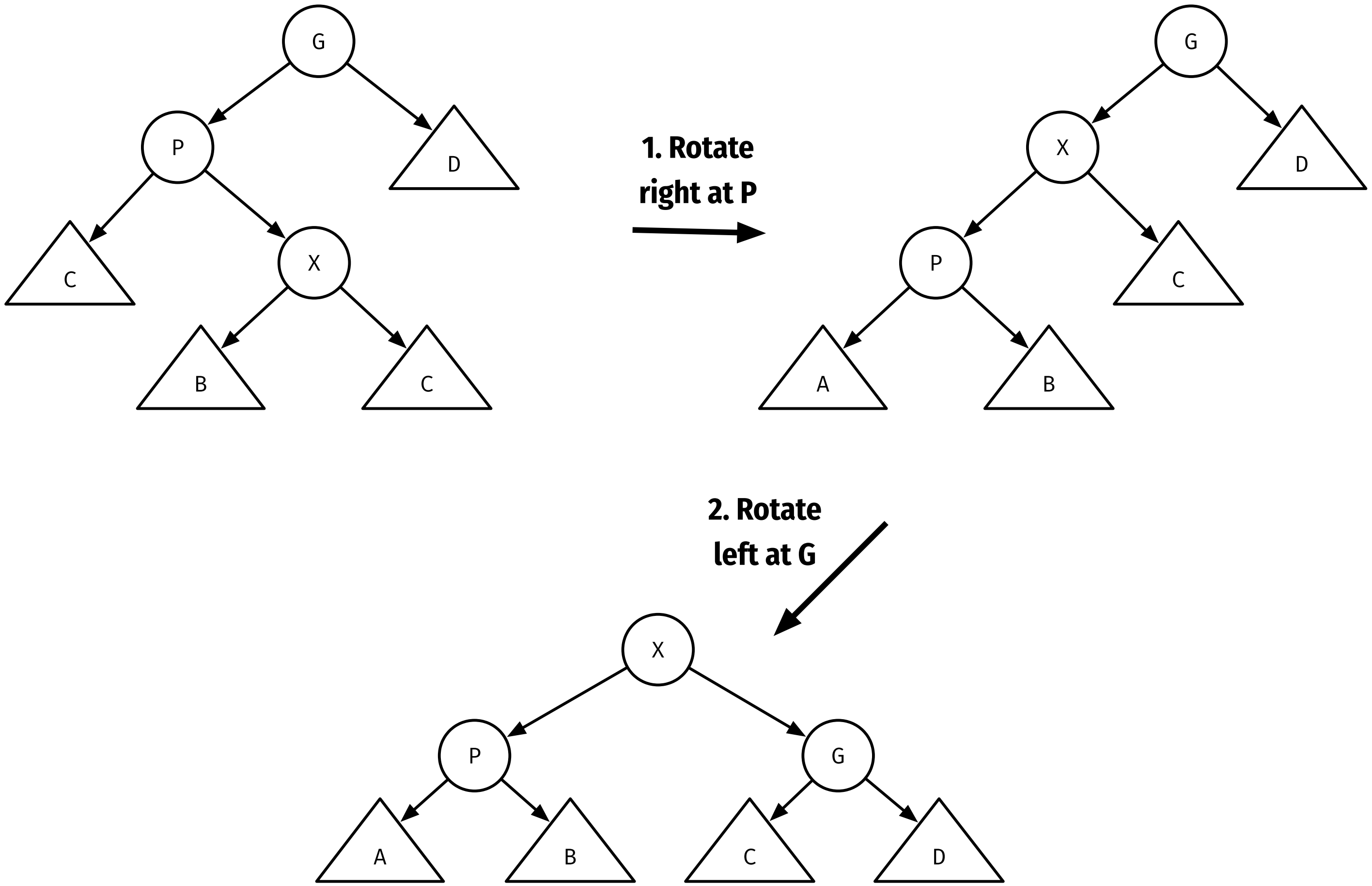
I do not like these names...

- 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



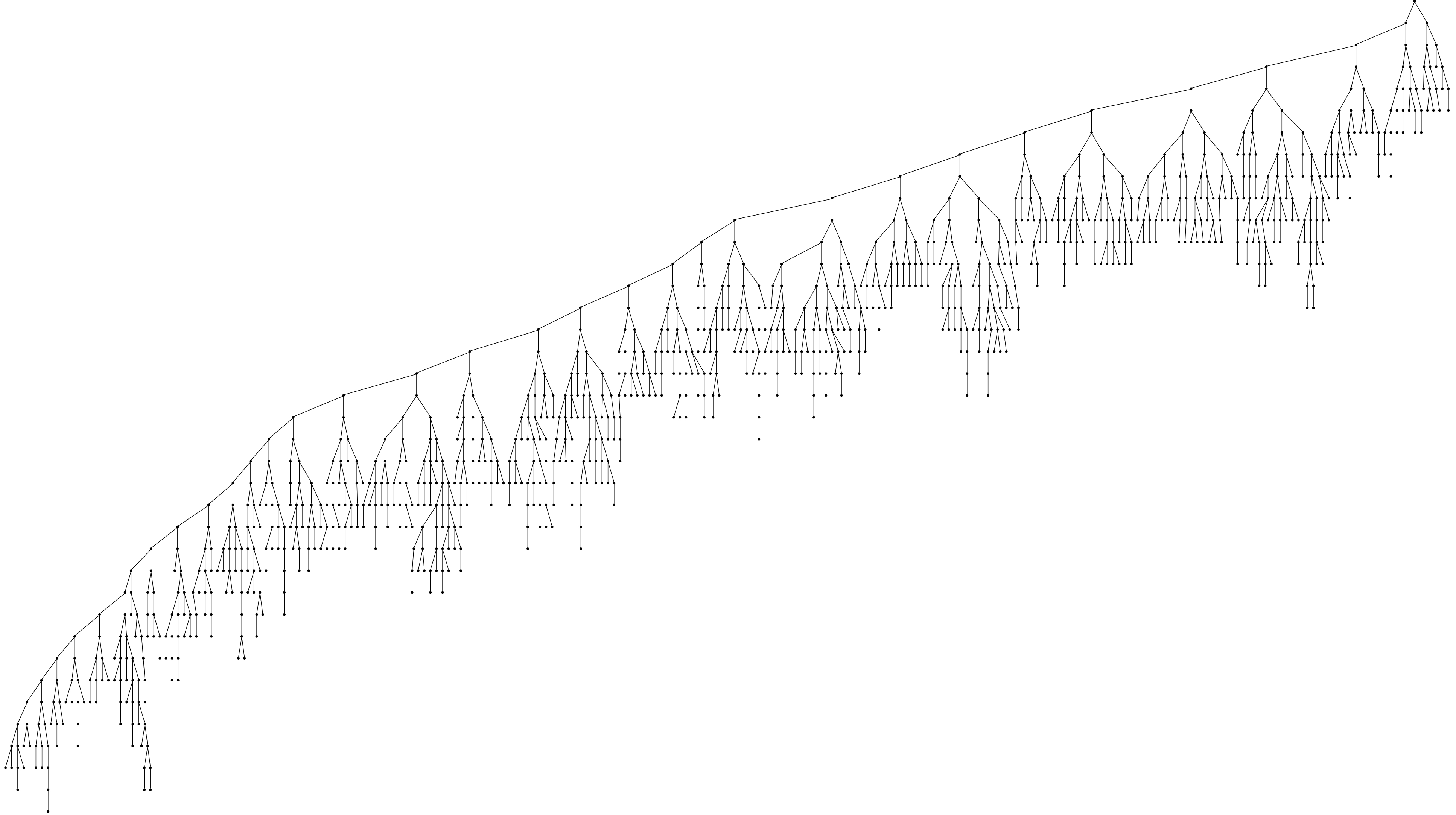
G
P
D
C
X
B
C
1. Rotate
right at P
G
X
D
P
C
A
B
2. Rotate
left at G
X
P
G
A
B
C
D

Splay trees

- We can move any node to the root by combing 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

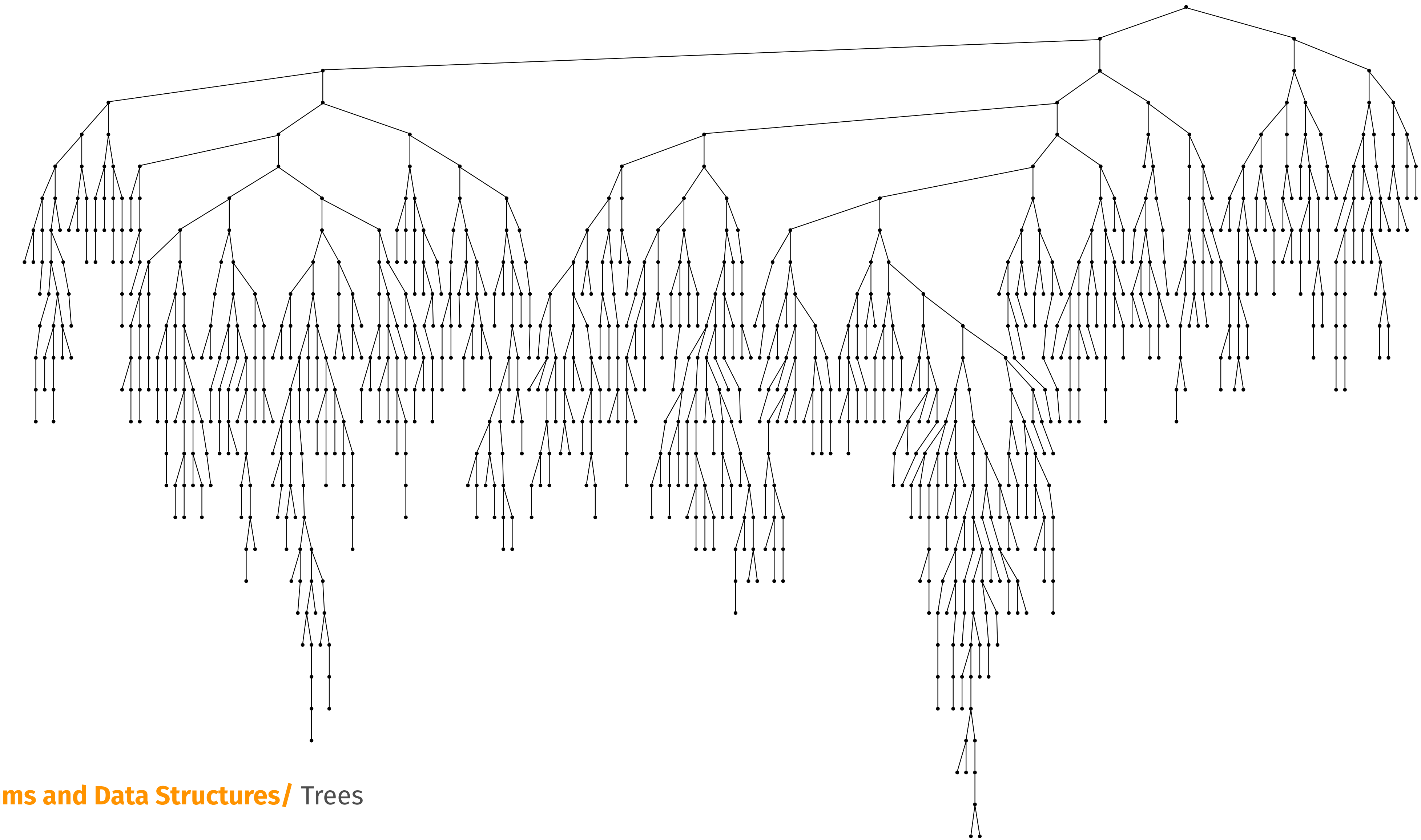
Experiment

A tree after multiple inserts and deletes



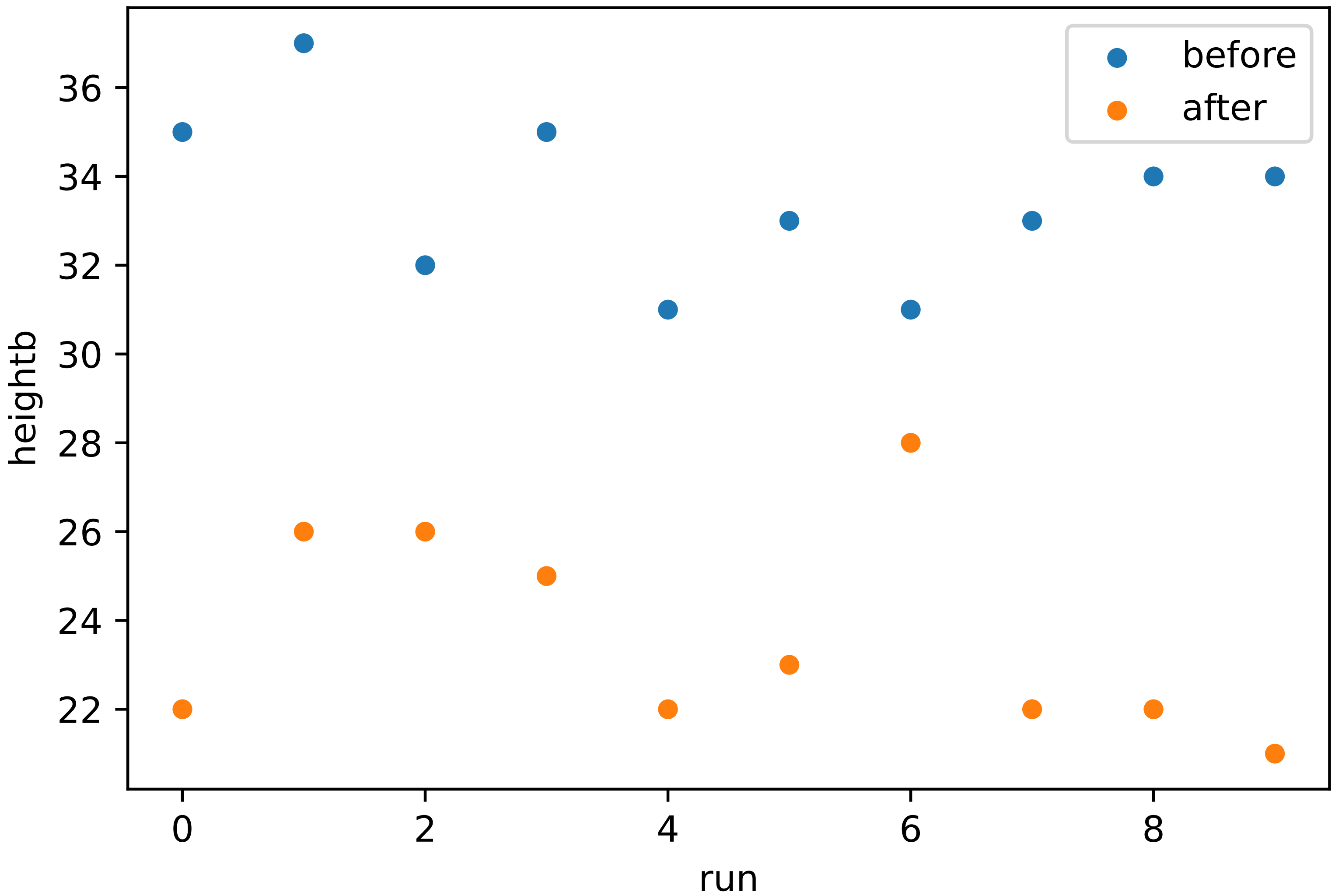
Experiment

Same tree after 5000 random finds



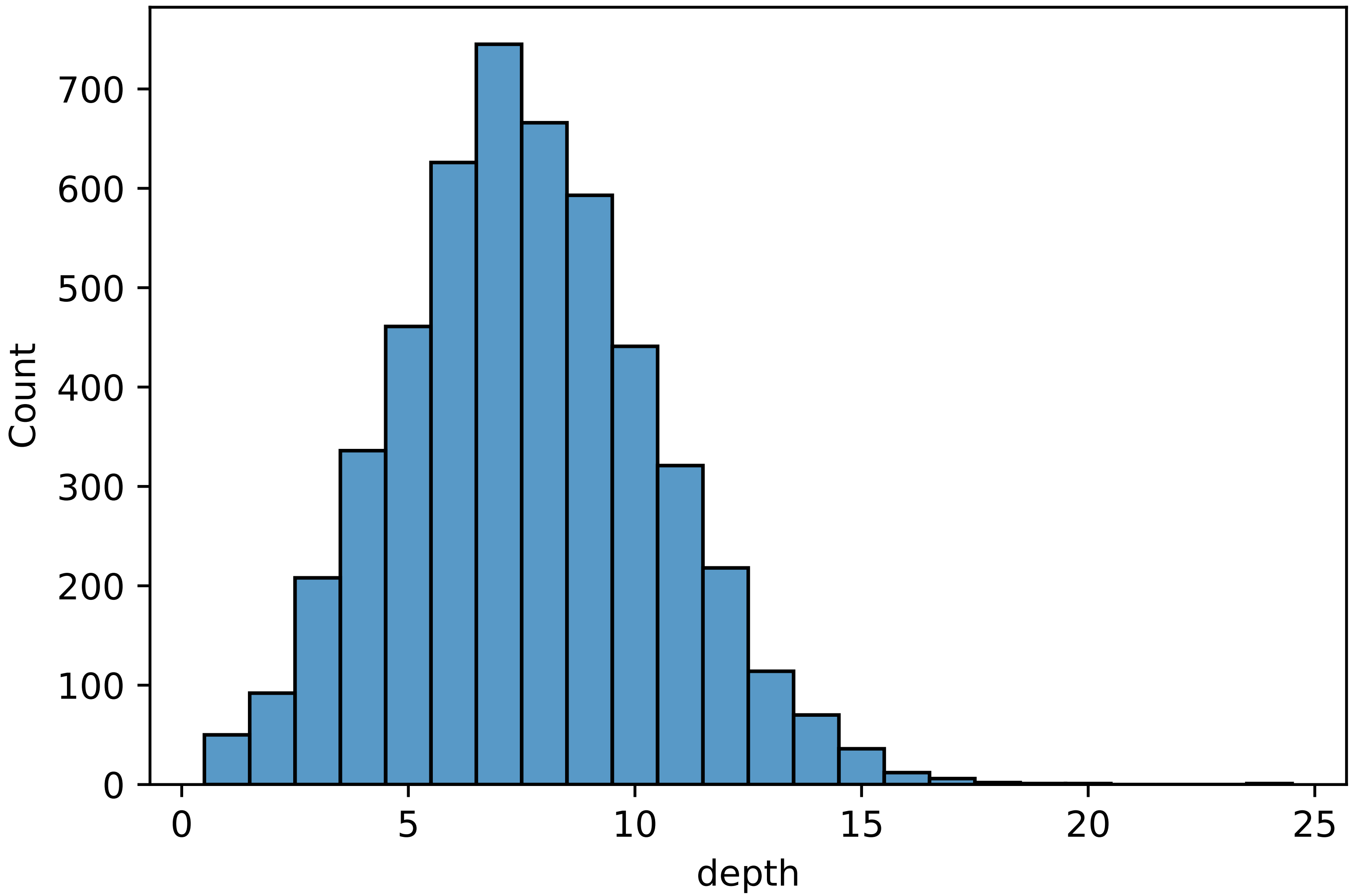
Experiment

Heights of 10 trees before and after splaying



Experiment

Depth of the value we are search for



Experiment

Adding warmup