CSE 250 Lecture 24

Traversing and Balancing Trees

Tree Traversals

Tree Traversals

- Pre-order (top-down)
  - visit **root**, visit **left** subtree, visit **right** subtree
- In-order
  - visit **left** subtree, visit **root**, visit **right** subtree
- Post-order (bottom-up)
  - visit **left** subtree, visit **right** subtree, visit **root**

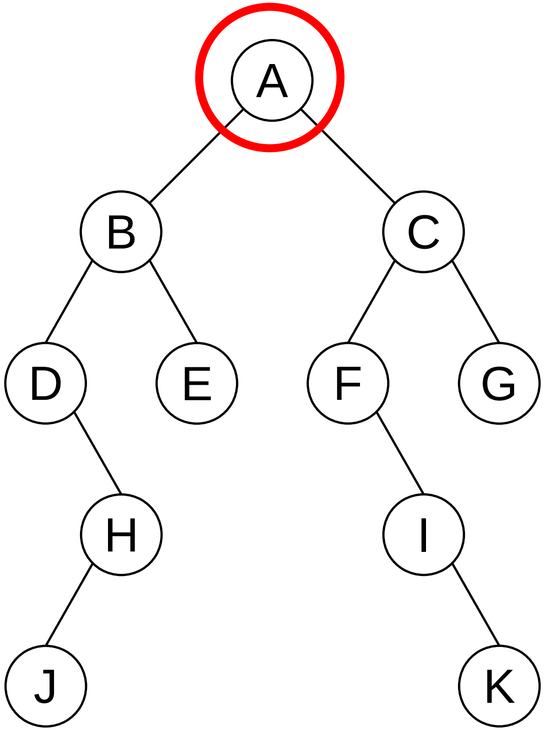
Tree Traversals

- How expensive is it to call...
  - new Iterator()
  - iterator.next
  - for(i <- iterator) { O(1) }

Tree Iteration: In-Order

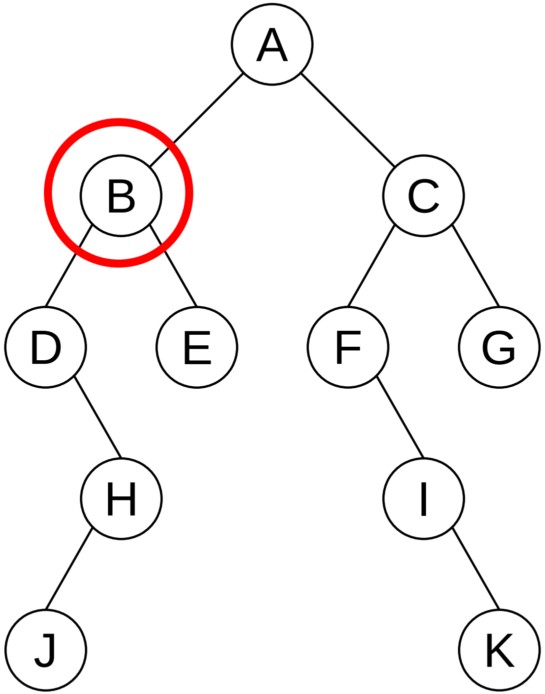
```
def inorderVisit[T](root: ImmutableTree[T]) =
{
  root match {
    case TreeNode(v, left, right) =>
      /* visit left */
      inorderVisit(left)

      /* visit root */
      visit(v)

      /* visit right */
      inorderVisit(right)

    case EmptyTree =>
      /* Do Nothing */
  }
}
```

Tree Iteration: In-Order



inorderVisit(A)

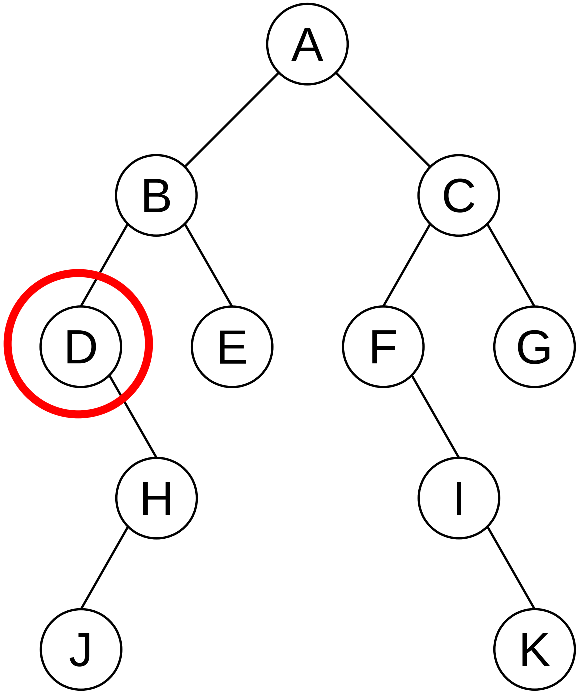
Tree Iteration: In-Order



inorderVisit(A)

inorderVisit(B)

Tree Iteration: In-Order



inorderVisit(A)

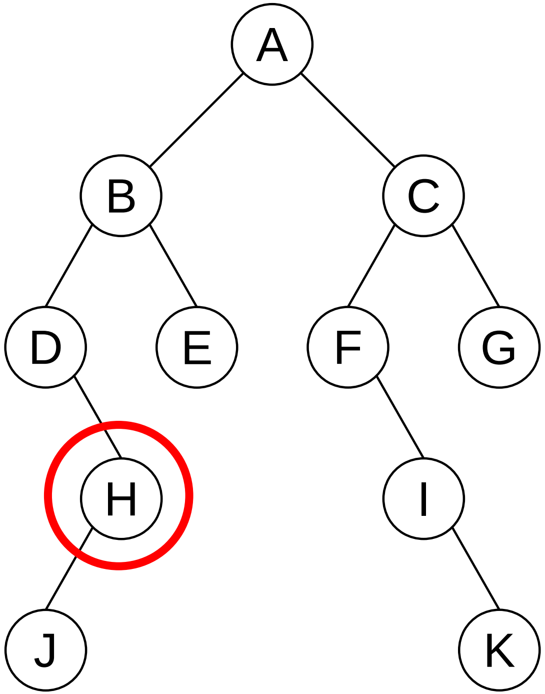
inorderVisit(B)

inorderVisit(D)

visit(D)

D

Tree Iteration: In-Order



inorderVisit(A)

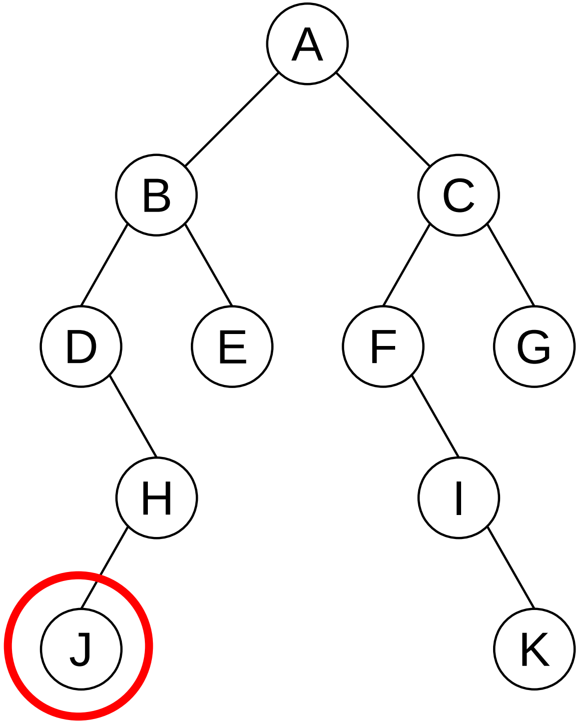
inorderVisit(B)

inorderVisit(D)

inorderVisit(H)

D

Tree Iteration: In-Order



inorderVisit(A)

inorderVisit(B)

inorderVisit(D)

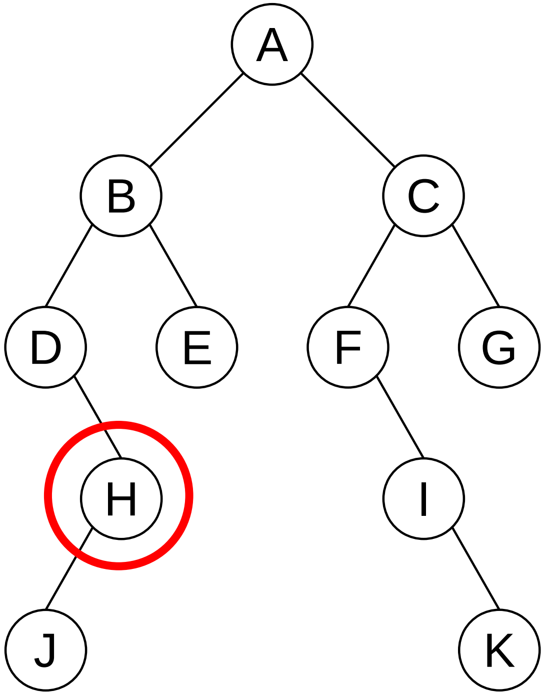
inorderVisit(H)

inorderVisit(J)

visit(J)

D, J

Tree Iteration: In-Order



inorderVisit(A)

inorderVisit(B)

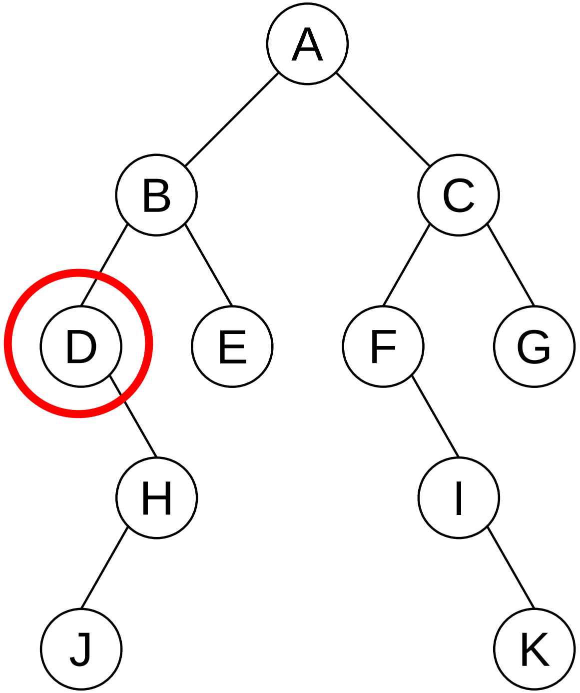
inorderVisit(D)

inorderVisit(H)

visit(H)

D, J, H

Tree Iteration: In-Order



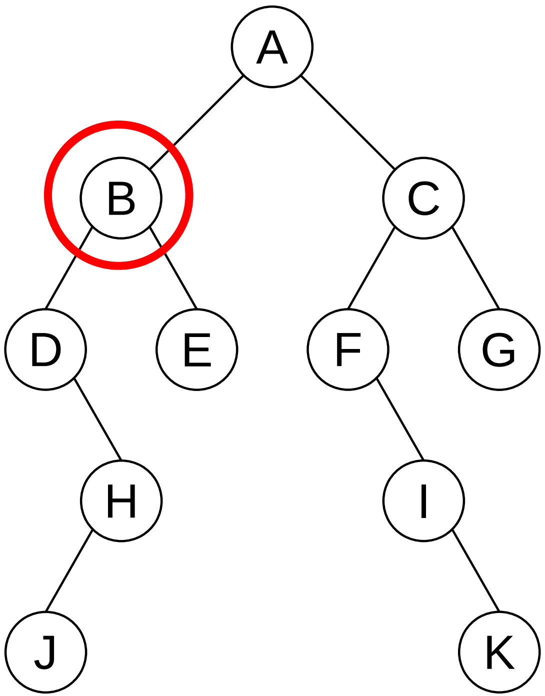
inorderVisit(A)

inorderVisit(B)

inorderVisit(D)

D, J, H

Tree Iteration: In-Order



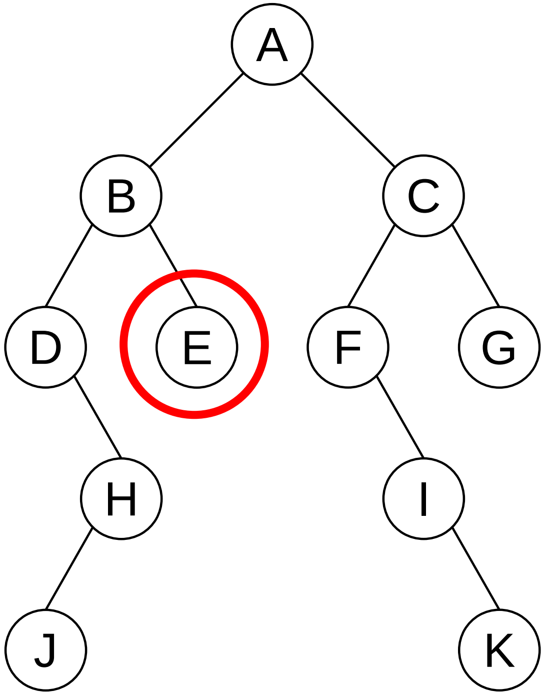
inorderVisit(A)

inorderVisit(B)

visit(B)

D, J, H, B

Tree Iteration: In-Order



inorderVisit(A)

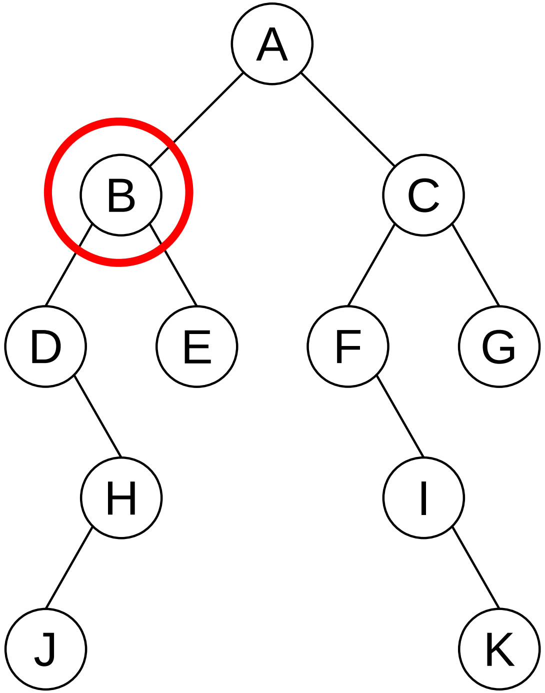
inorderVisit(B)

inorderVisit(E)

visit(E)

D, J, H, B, E

Tree Iteration: In-Order

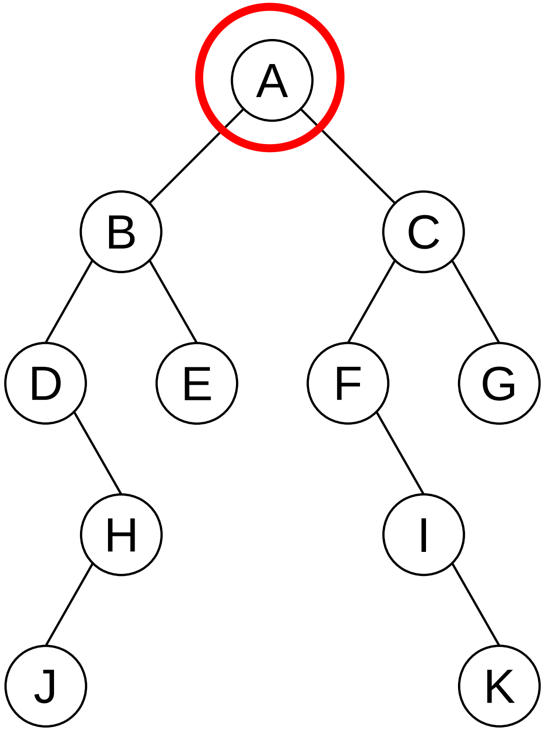


inorderVisit(A)

inorderVisit(B)

D, J, H, B, E

Tree Iteration: In-Order

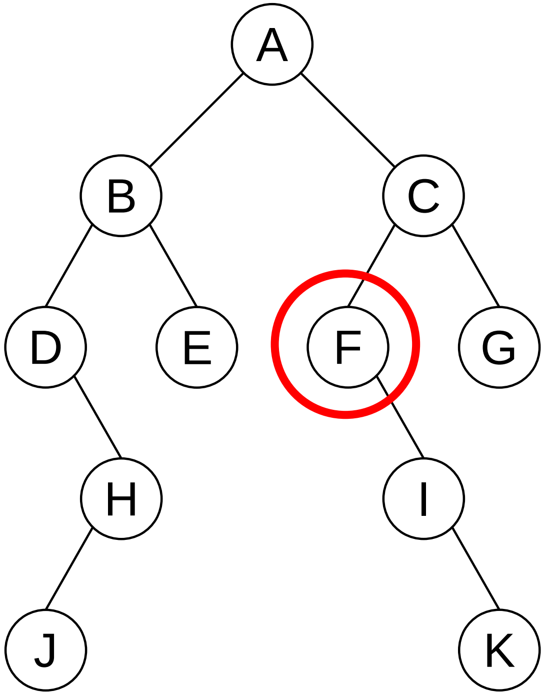


inorderVisit(A)

visit(A)

D, J, H, B, E, A

Tree Iteration: In-Order



inorderVisit(A)

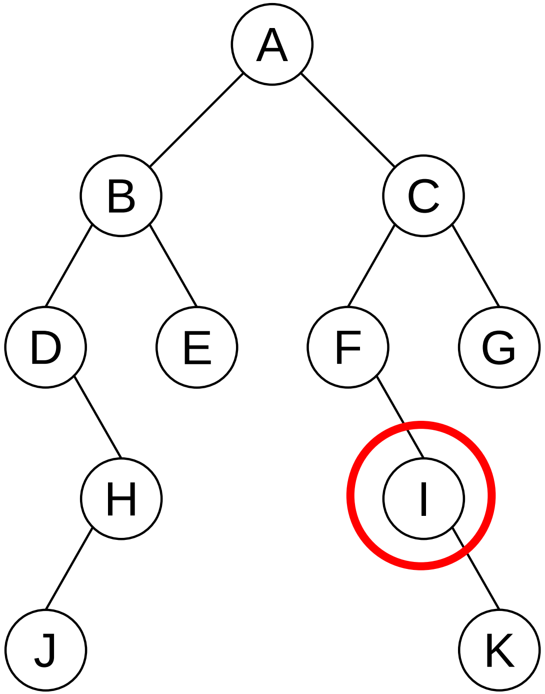
inorderVisit(C)

inorderVisit(F)

visit(F)

D, J, H, B, E, A, F

Tree Iteration: In-Order



inorderVisit(A)

inorderVisit(C)

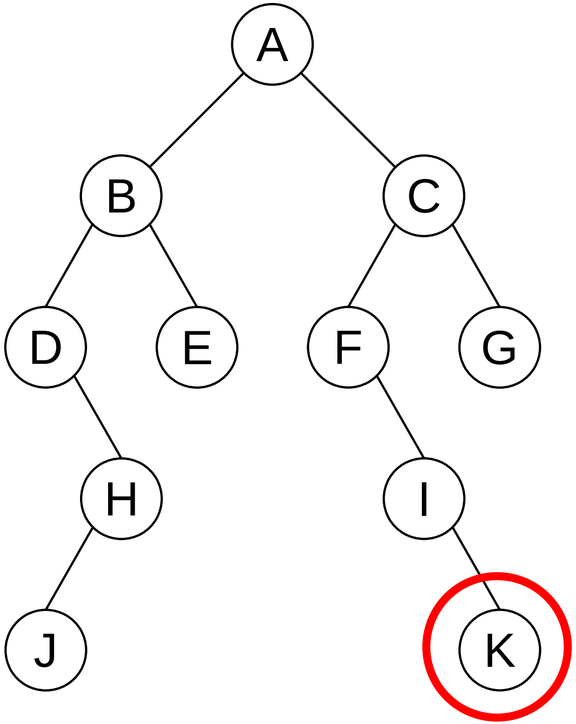
inorderVisit(F)

inorderVisit(I)

visit(I)

D, J, H, B, E, A, F, I

Tree Iteration: In-Order



inorderVisit(A)

inorderVisit(C)

inorderVisit(F)

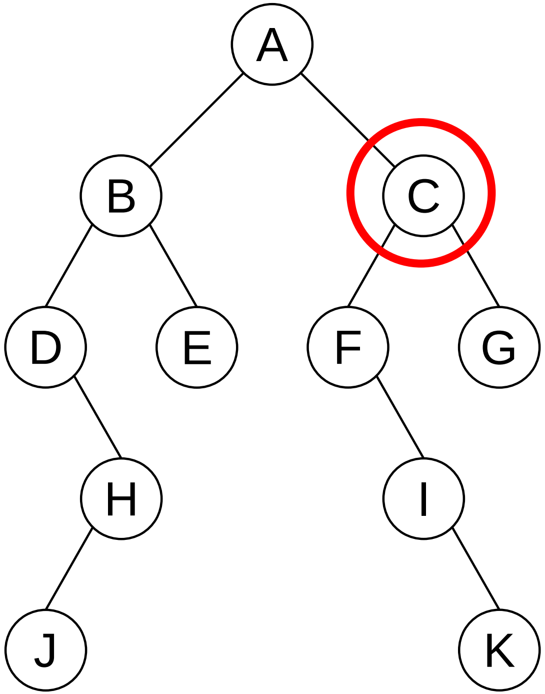
inorderVisit(I)

inorderVisit(K)

visit(K)

D, J, H, B, E, A, F, I, K

Tree Iteration: In-Order



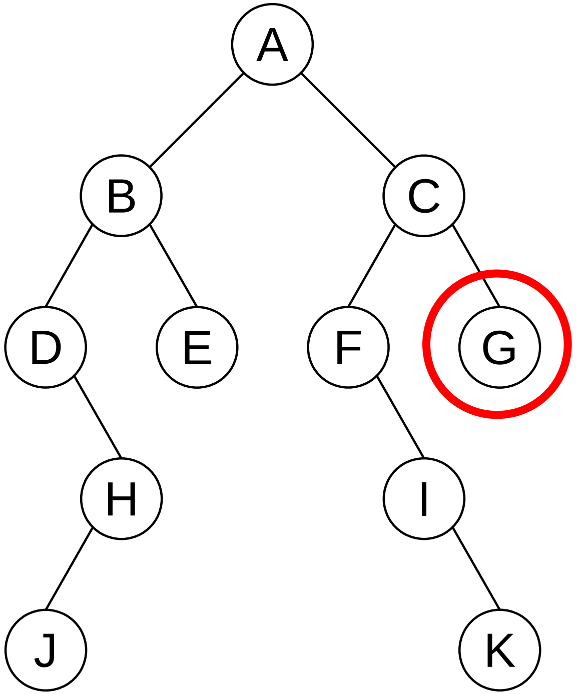
inorderVisit(A)

inorderVisit(C)

visit(C)

D, J, H, B, E, A, F, I, K, C

Tree Iteration: In-Order



inorderVisit(A)

inorderVisit(C)

inorderVisit(G)

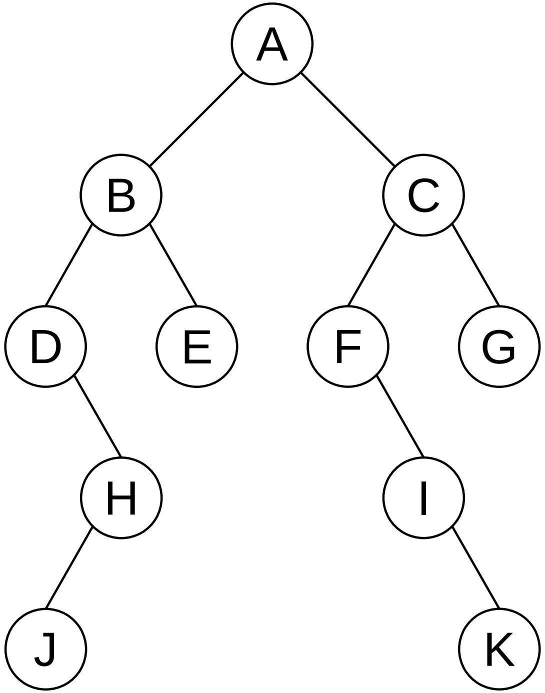
visit(G)

D, J, H, B, E, A, F, I, K, C, G

Tree Iteration: In-Order

```
class ImmutableTreeIterator[T](root: ImmutableTree[T]){
  /*** Initialize the Iterator ***/
  val toVisit = mutable.Stack[ImmutableTree[T]]
  pushLeft(root)

  def pushLeft(node: ImmutableTree[T]): Unit =
    node match {  case EmptyTree => ()
                  case t: ImmutableTree =>
                    toVisit.push(t)
                    pushLeft(t.left)    }

  def isEmpty = toVisit.isEmpty

  def next: T = {
    val nextNode = toVisit.pop
    pushLeft(nextNode.right)
    return nextNode.value
  }
}
```

Tree Iteration: In-Order



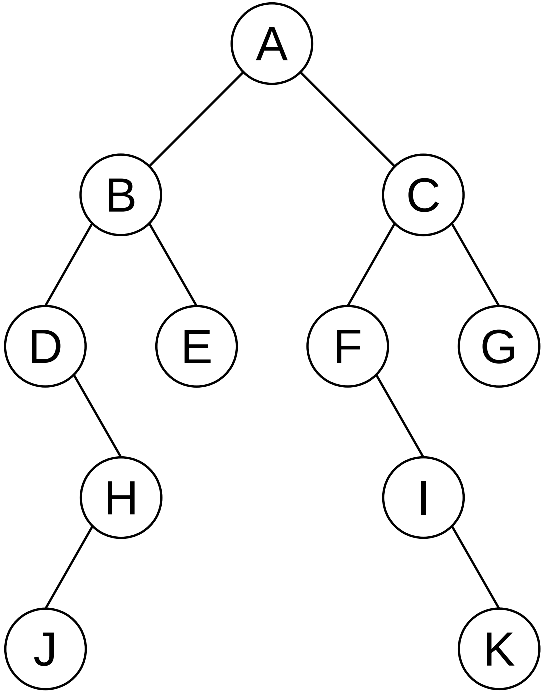
A

B

D

—

Tree Iteration: In-Order



A

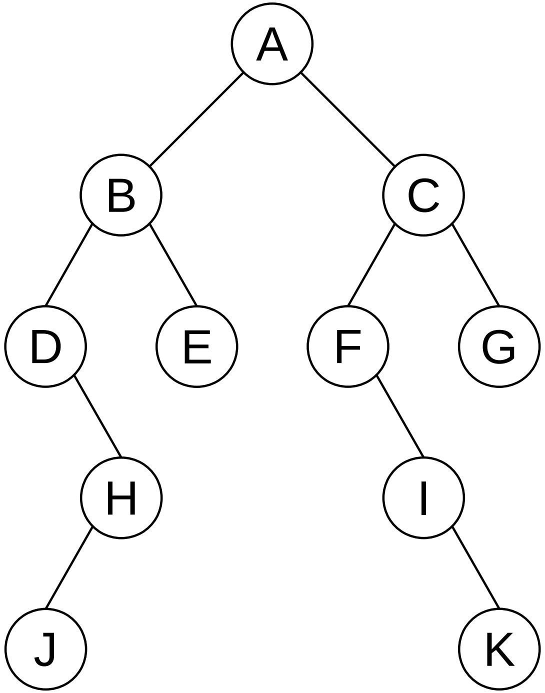
B

DH

J

D

Tree Iteration: In-Order



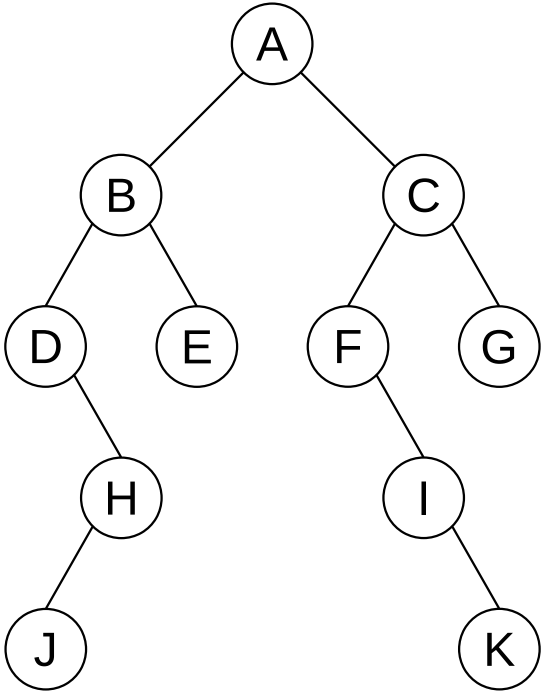
A

B

H

D, J

Tree Iteration: In-Order

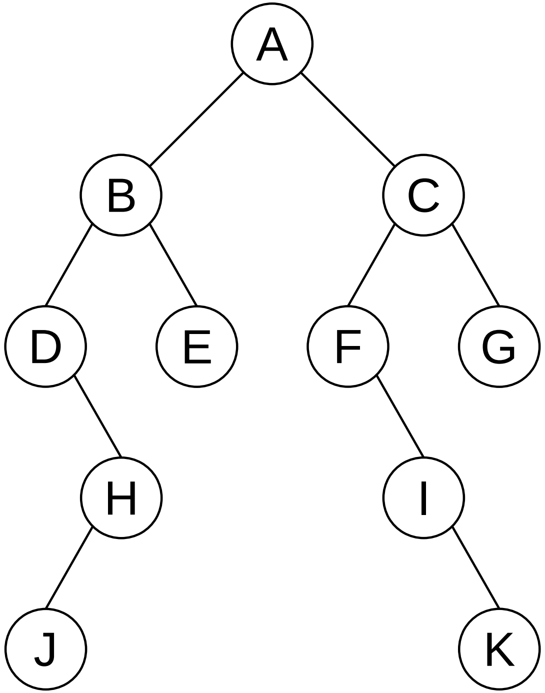


A

B

D, J, H

Tree Iteration: In-Order

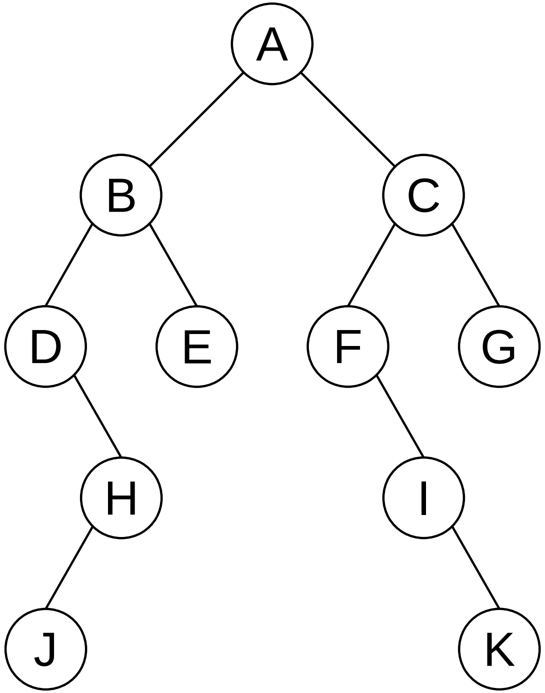


A

E

D, J, H, B

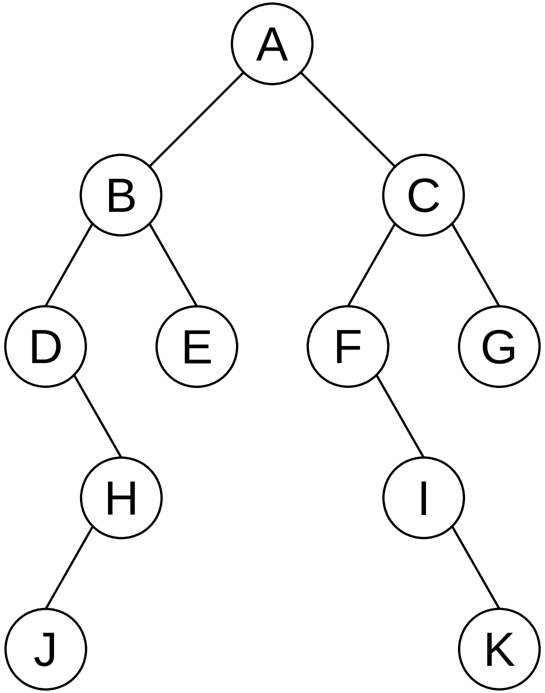
Tree Iteration: In-Order



A

D, J, H, B, E

Tree Iteration: In-Order

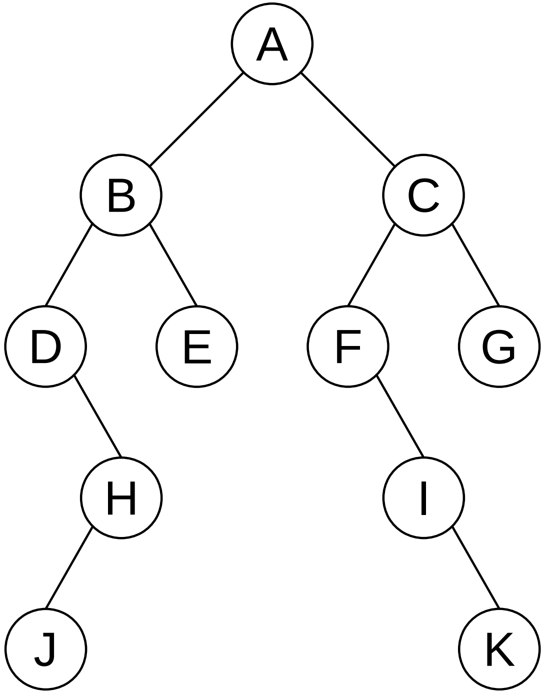


C

F

D, J, H, B, E, A

Tree Iteration: In-Order

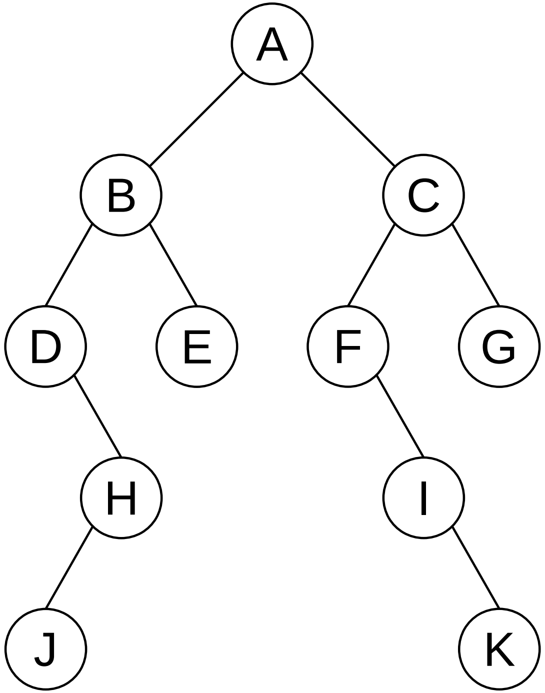


C

I

D, J, H, B, E, A, F

Tree Iteration: In-Order

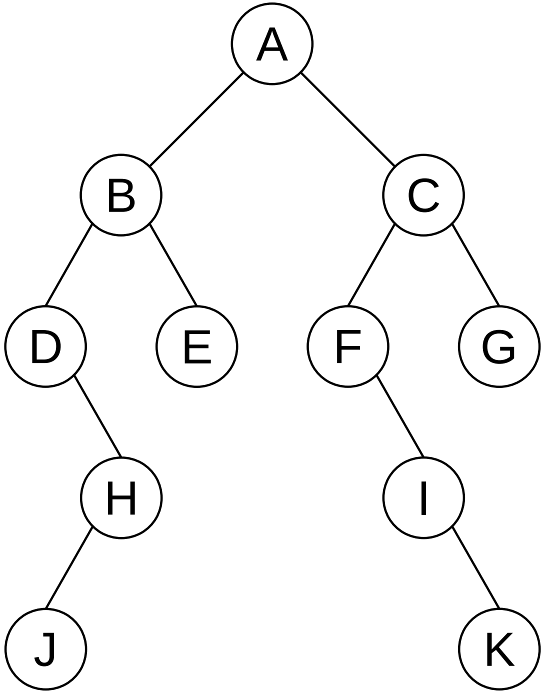


C

K

D, J, H, B, E, A, F, I

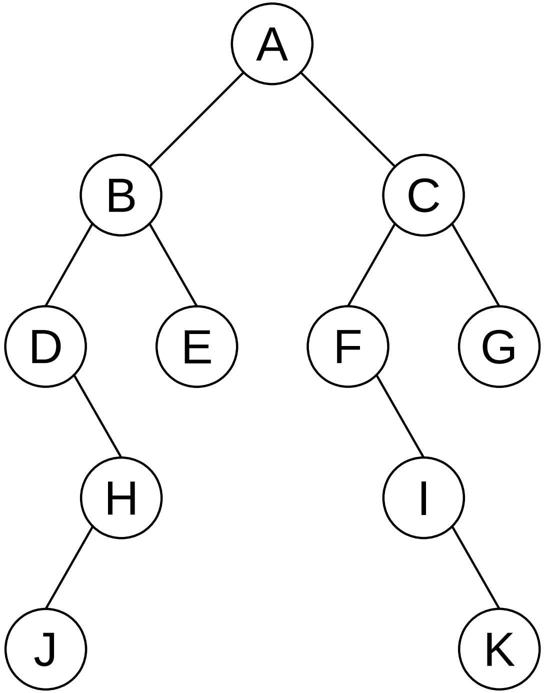
Tree Iteration: In-Order



C

D, J, H, B, E, A, F, I, K

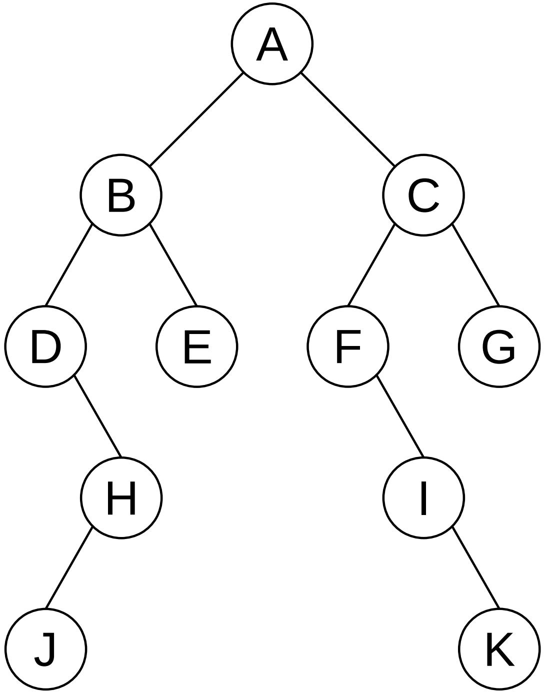
Tree Iteration: In-Order



G

D, J, H, B, E, A, F, I, K, C

Tree Iteration: In-Order



D, J, H, B, E, A, F, I, K, C, G

Tree Iteration: In Order

- Worst-Case runtime to initialize the iterator

```
/*** Initialize the Iterator ***/
val toVisit = mutable.Stack[ImmutableTree[T]]
pushLeft(root)
```

```
def pushLeft(node: ImmutableTree[T]): Unit =
  node match {  case EmptyTree => ()
                case t: ImmutableTree =>
                  toVisit.push(t)
                  pushLeft(t.left)    }
```

O(d)

Tree Iteration: In Order

- Worst-Case runtime to call next

```
def next: T = {
  val nextNode = toVisit.pop
  pushLeft(nextNode.right)
  return nextNode.value
}
```

O(d)

Tree Iteration: In Order

- Worst-Case runtime to visit all nodes
  - Each node is at the top of the stack exactly once
    - One push
    - One pop
    - One visit

O(n)

Balanced Trees

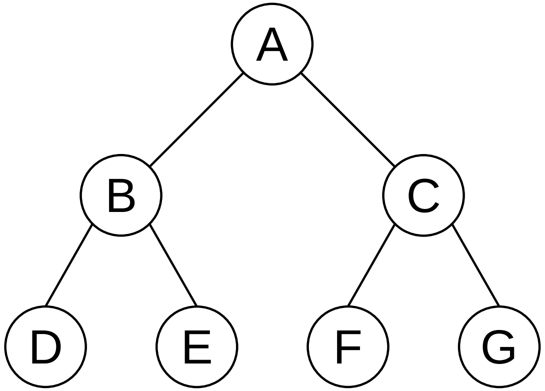
BST Operation Costs

$\log(n) \leq d \leq n$

Operation	Runtime
find	O(d)
insert	O(d)
remove	O(d)

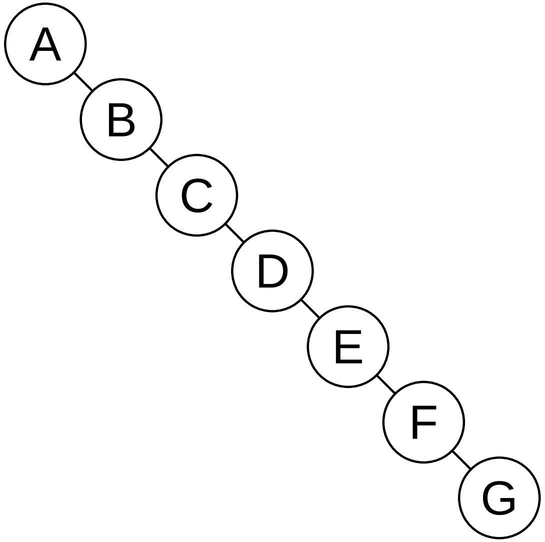
Tree Depth vs Size

height(left) ≈ height(right)



d = O(log(n))

height(left) ⩽ height(right)



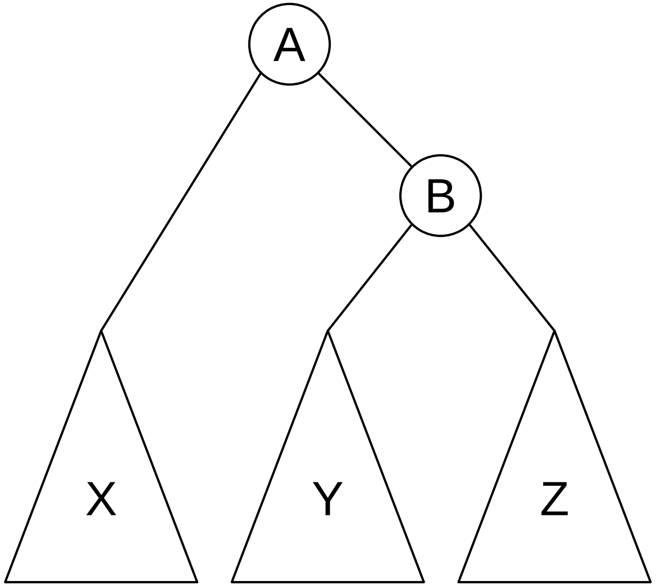
d = O(n)

“Balanced” Trees

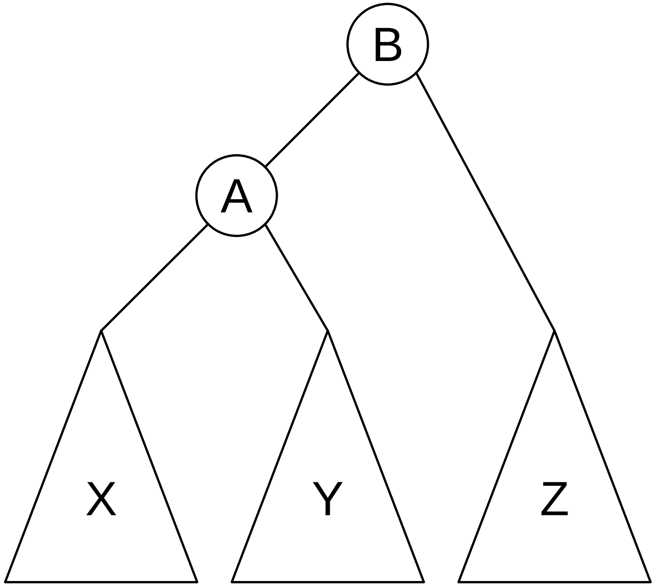
- Faster search: Want height(left) ≈ height(right)
  - Make it more precise: |height(left) - height(right)| ≤ 1
  - (left, right height differ by at most 1)
- **Question**: How do we keep the tree balanced?
  - Option 1: Keep left/right subtrees within **+/- 1** of each other
    - Add a field to track the “imbalance factor”
  - Option 2: Ensure leaves are at a minimum depth of **d / 2**
    - Add a designation marking each node as red or black

Subtree Rotation

Rebalancing Trees

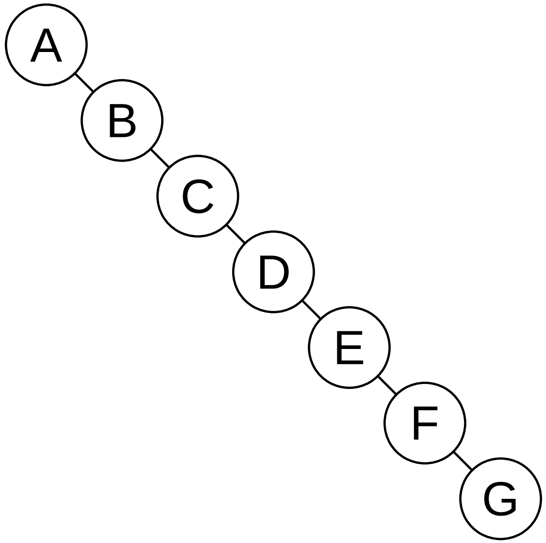


Rebalancing Trees



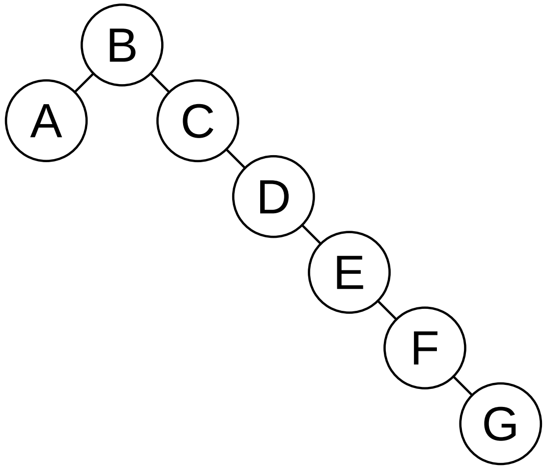
Rotate(A, B)

Rebalancing Trees



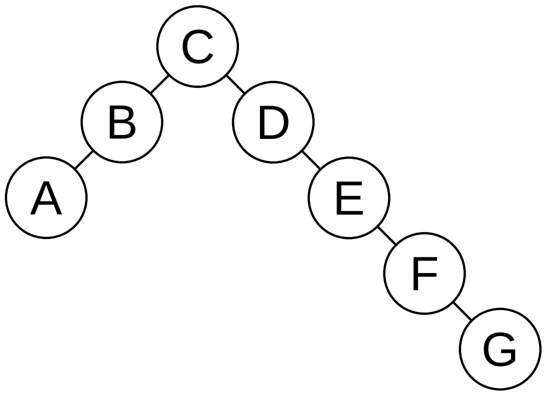
Rotate(A, B)

Rebalancing Trees



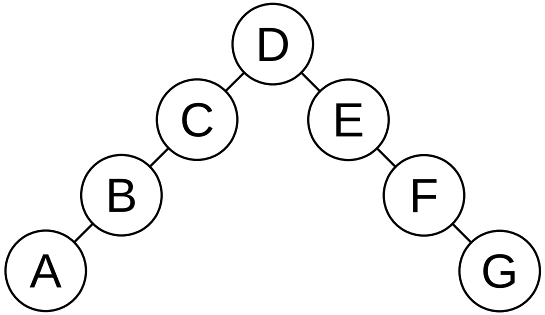
Rotate(B, C)

Rebalancing Trees



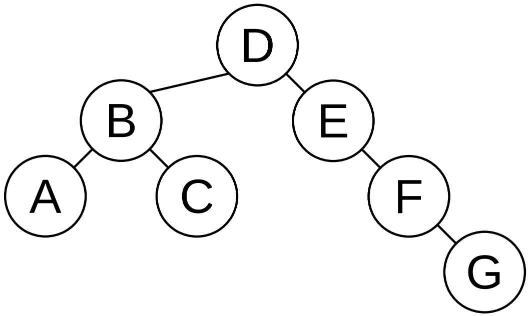
Rotate(C, D)

Rebalancing Trees



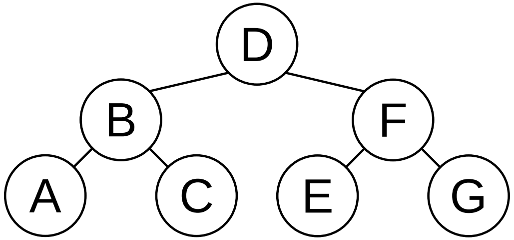
Rotate(C, B)

Rebalancing Trees



Rotate(E, F)

Rebalancing Trees



AVL Trees

AVL Trees

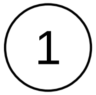
- An AVL tree (Adelson-Velsky and Landis) is a BST where every node is "depth-balanced"
  - |depth(left subtree) - depth(right subtree)| < 1
- define **`balance(v) = height(v.right) - height(v.left)`**
  - Maintain balance(v) ∈ { -1, 0, 1 }
    - balance(v) = 0 → "v is balanced"
    - balance(v) = -1 → "v is left-heavy"
    - balance(v) = 1 → "v is right-heavy"

AVL Trees

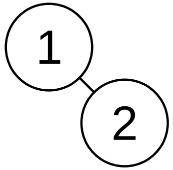
- **Goal**: AVL tree property maintains a nearly balanced tree
  - Depth balance forces a maximum possible depth $d \ll n$
    - ($d \ll n$ means $d \leq c \log(n)$ for some constant $c > 0$)
- **Proof idea**: An AVL tree with depth d has “enough” nodes

AVL Trees

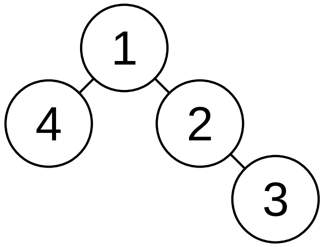
- Let minNodes(d) be the minimum number of nodes in an AVL tree of depth d



minNodes(0) = 1



minNodes(1) = 2



minNodes(2) = 4

Enough Nodes?

- For d > 1
  - minNodes(d) = 1 + minNodes(d-1) + minNodes(d-2)
  - This is the Fibbonacci Sequence!
    - minNodes(d) = Fib(d+3)-1
    - Fib(0), Fib(1), Fib(2), ... = 0, 1, 1, 2, 3, 5, 8, ...
  - minNodes(d) = $\Omega(1.5^d)$

$$n \geq c1.5^d \qquad \frac{n}{c} \geq 1.5^d$$

$$\log_2\left(\frac{n}{c}\right) \geq \log_2\left(1.5^d\right)$$

Enough Nodes?

- minNodes(d) = Ω(1.5^d)

$$n \geq c1.5^d$$

$$\frac{n}{c} \geq 1.5^d$$

$$\log_2\left(\frac{n}{c}\right) \geq \log_2\left(1.5^d\right)$$

$$\log_2\left(\frac{n}{c}\right) \geq \log_{1.5}(1.5^d)\log_2 1.5$$

$$\log_2\left(\frac{n}{c}\right) \geq d\log_2(1.5)$$

$$\frac{\log_2\left(\frac{n}{c}\right)}{\log_2(1.5)} \geq d$$

$$\frac{\log_2(n)}{\log_2(1.5)} - \frac{\log_2(c)}{\log_2(1.5)} \geq d$$

$$d \leq O\left(\log_2(n)\right)$$