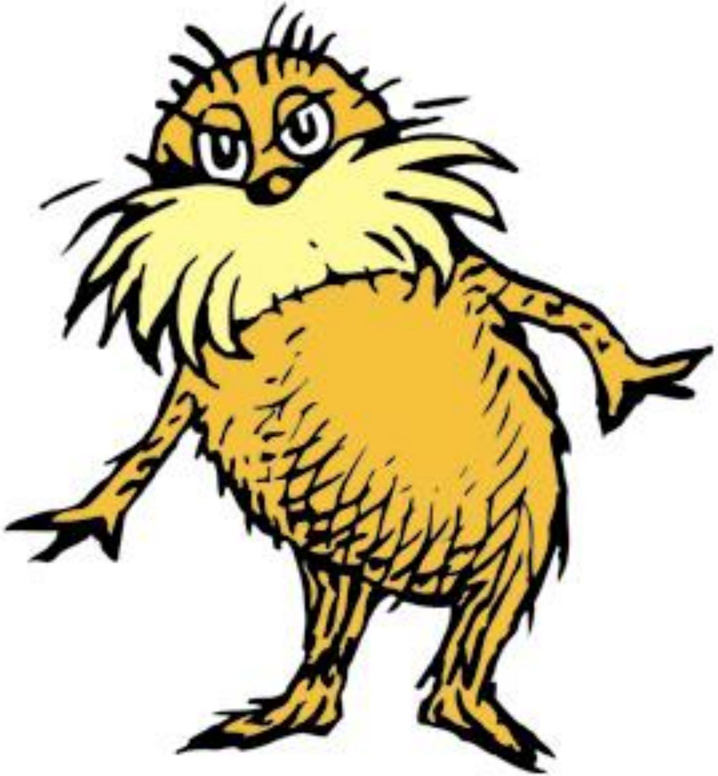
Lecture 10: Ordered Collections with Binary Search Trees



These slides include material originally prepared by Dr. Ron Cytron, Dr. Jeremy Buhler, and Dr. Steve Cole.

Announcements

- Lab 11 out next week – balanced binary trees (with coding)
- **Exam 2** is Wednesday, April 3rd
  - Same ground rules and procedures as Exam 1 (2-sided crib sheet and nothing else, Piazza post will specify room location)
  - Will cover material since Exam 1 (Master Method)
    - Lectures/studios 5-10 inclusive
  - Review on Sunday 2-5 pm in Louderman 458

Motivation – Limitations of Dictionaries

- We developed hashing to permit efficient **dictionaries**
  - **Insert()**
  - **Remove()**
  - **Find()**
- But hash tables are unsatisfactory in two ways
  1. Worst-case op performance is **$\Theta(n)$** (only *average* case is good)
  2. Does not adequately represent naturally **ordered** collections.

Ordered Dynamic Set Operations

- Besides the usual dictionary operations, ordered sets support
  - **min / max** – what is smallest/largest item in collection?
  - **iterator** – list collection's items in order *from smallest to largest*

- *See, e.g., Java* ***SortedSet*** *interface*

- Many data types are naturally ordered (strings, ID #'s), even if we don't always use this fact.

Ordered Dynamic Set Operations

- Besides th … pport
  - **min** … ?
  - **itera** … *st to largest*
- *See, e.g* …
- Many da … en if we don't alway …

"**Dynamic**" means that a query of the set must return the correct answer at *any* point during a sequence of insertions and deletions.

Candidate Implementations?

- *Sorted* Array
  - Θ(log n) find, O(1) min/max, O(1) iteration/item
  - Θ(n) insert/remove
- *Sorted* List
  - Much like array, except for Θ(n) find

- (Hash table does not support ordering – must iterate through all items to find min/max or next item in order)

What We Would Like from Our Ordered Sets

- Sub-linear time insert/remove/find
    - (what does sub-linear mean again?)

What We Would Like from Our Ordered Sets

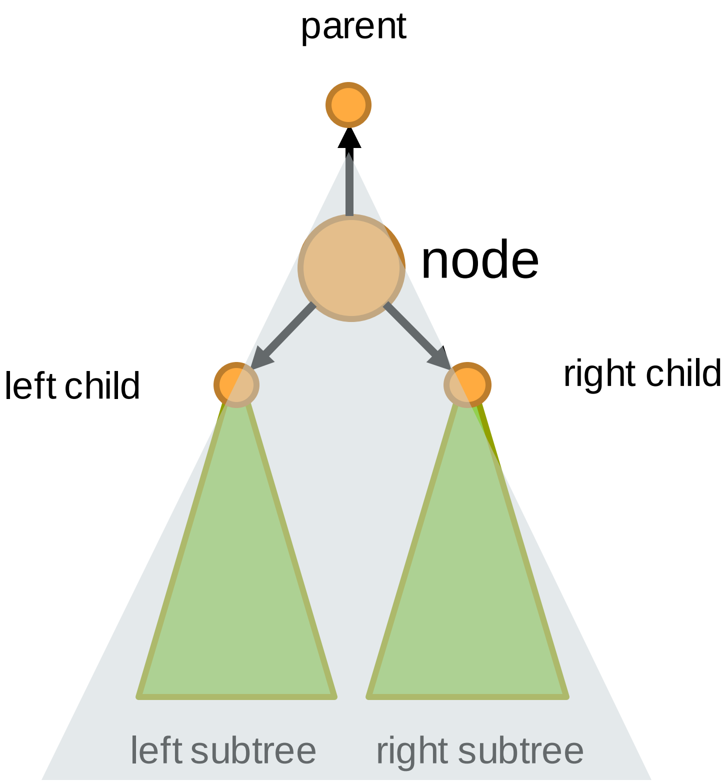
- Sub-linear time insert/remove/find
    - (what does sub-linear mean again?)
- Sub-linear time min/max
- Iteration in sub-linear time per element
- All times **worst-case** (*unlike a hash table*)

How We'll Get It

- New data structure – **binary search tree (BST)**
- Can do all operations in time proportional to **height** of tree
- But height isn't *necessarily* sub-linear in size (unlike a heap)
- So we'll consider how to *force* BSTs to have small height

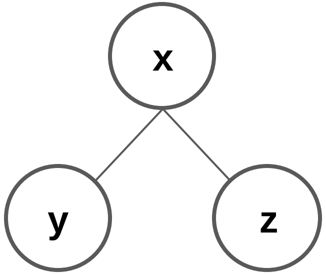
Binary Trees, Revisited

- A BST is a type of binary tree.
- Tree is made of **nodes**, each of which is root of a **subtree**
- Each node has left and right children, and a parent (any may be *null*)
- Unlike heaps, trees used as BSTs *need not be compact*.



What Makes a Binary Tree a BST?

- Every node *x* contains a **key** value *x.key*
- Every node satisfies the following **invariant** (“**BST property**”):
- For every node y in x’s *left* subtree, y.key ≤ x.key
- For every node z in x’s *right* subtree, x.key ≤ z.key
- (If each key in BST is *unique*, these inequalities are *strict* <)



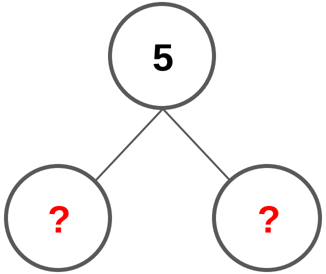
What Makes a Binary Tree a BST?

- Every node *x* contains a **key** value *x.key*

- Every node satisfies the following **invariant** (“**BST property**”):

- For every node y in x’s *left* subtree, y.key ≤ x.key

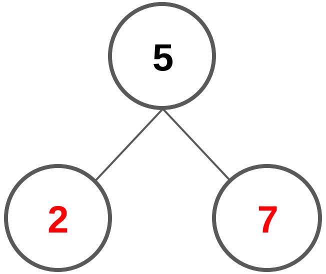
- For every node z in x’s *right* subtree, x.key ≤ z.key



- (If each key in BST is *unique*, these inequalities are *strict* <)

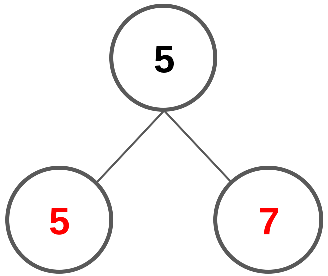
What Makes a Binary Tree a BST?

- Every node *x* contains a **key** value *x.key*
- Every node satisfies the following **invariant** (“**BST property**”):
- For every node y in x’s *left* subtree, y.key ≤ x.key
- For every node z in x’s *right* subtree, x.key ≤ z.key
- (If each key in BST is *unique*, these inequalities are *strict* <)



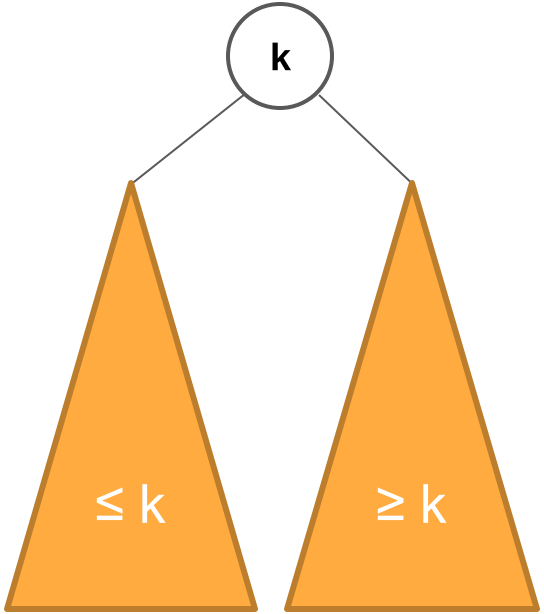
What Makes a Binary Tree a BST?

- Every node *x* contains a **key** value *x.key*
- Every node satisfies the following **invariant** (“**BST property**”):
- For every node y in x’s *left* subtree, y.key **≤** x.key
- For every node z in x’s *right* subtree, x.key **≤** z.key
- (If each key in BST is *unique*, these inequalities are *strict* <)



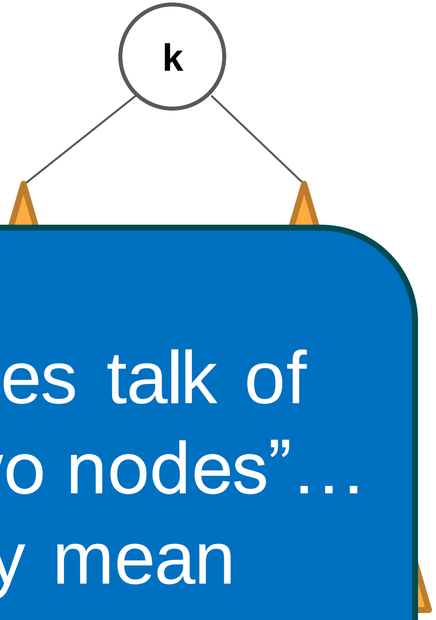
BST Property in Brief

- Node x is ≥ every node in its *left* subtree
- Node x is ≤ every node in its *right* subtree
- [Note that this is a different, stronger tree invariant than heap property]



BST Property in Brief

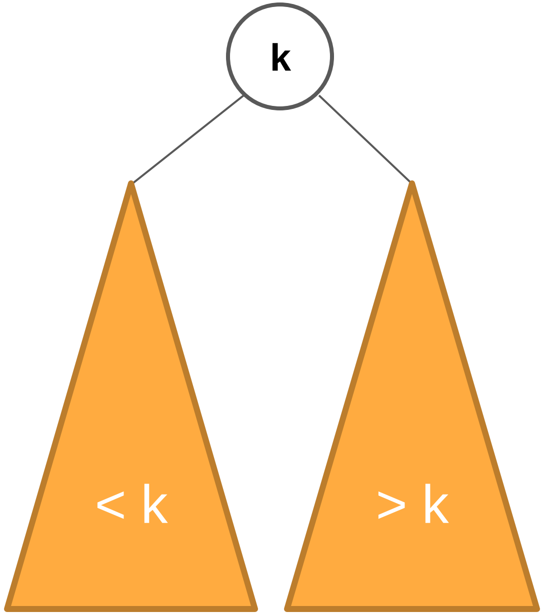
- Node x is ≥ every node in its *left* subtree
- Node x is ≤ every node in its *right* subtree
- [Note that this is a different, stron invariant than heap property]



We sometimes talk of "comparing two nodes"… we actually mean comparing their keys.

BST Property in Brief (**With Unique Keys**)

- Node x is **>** every node in its *left* subtree
- Node x is **<** every node in its *right* subtree
- [Note that this is a different, stronger tree invariant than heap property]



Using a BST, How Do We Implement…

- Find?
- Min/Max?
- Insert?
- Iterate?
- Remove?

Caveat - Uniqueness

- In what follows, we assume that keys in tree are all unique
- Still possible to have an efficient BST with duplicate keys…
- (E.g. if we must store two records with same key)
- …but it adds complexity to the ops and/or their correctness proofs.

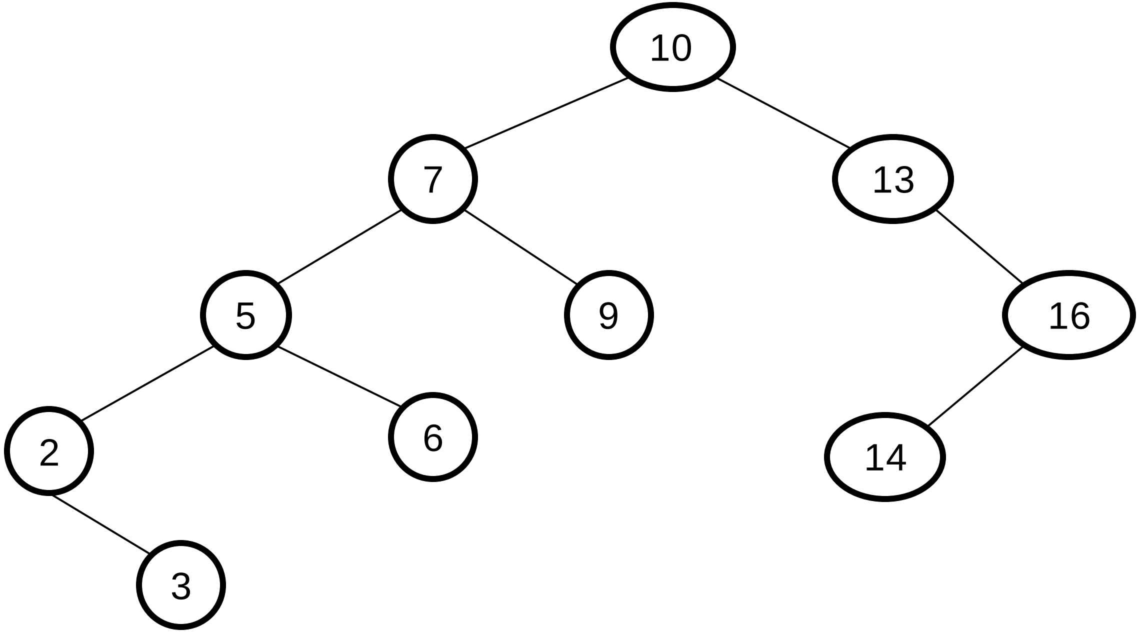
Find: Use the BST Property

- Suppose we search tree rooted at node x for key k
- If x.key = k, we are done!
- If x.key > k, search for k in ???
- If x.key < k, search for k in ???

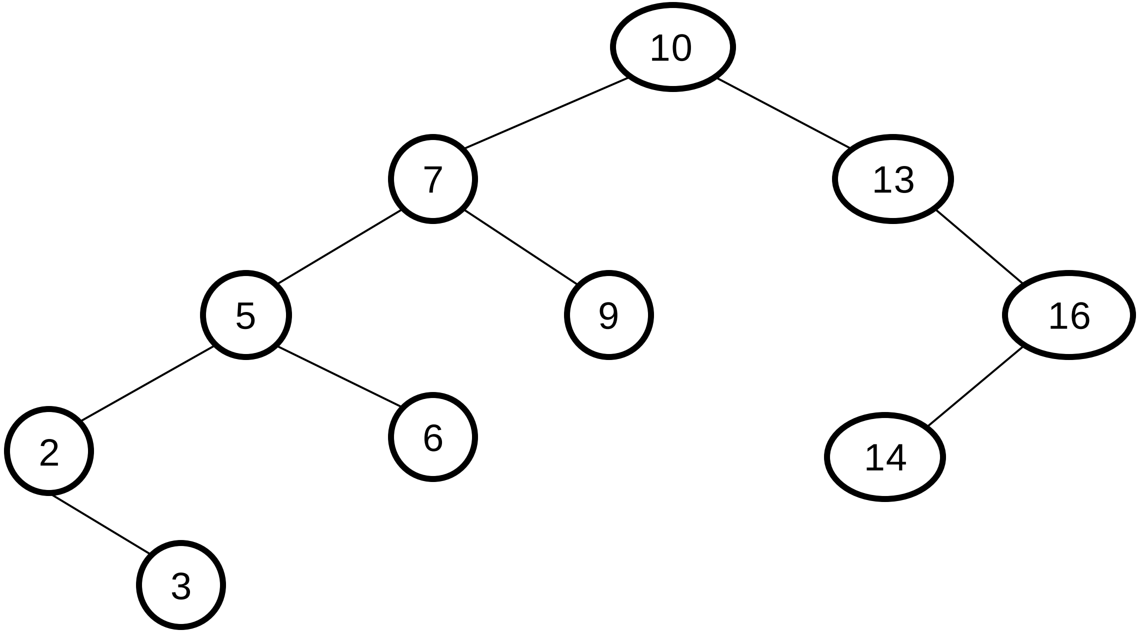
Find: Use the BST Property

- Suppose we search tree rooted at node x for key k
- If x.key = k, we are done!
- If x.key > k, search for k in subtree rooted at x.left
- If x.key < k, search for k in subtree rooted at x.right
- (If desired subtree is null, k is not found)

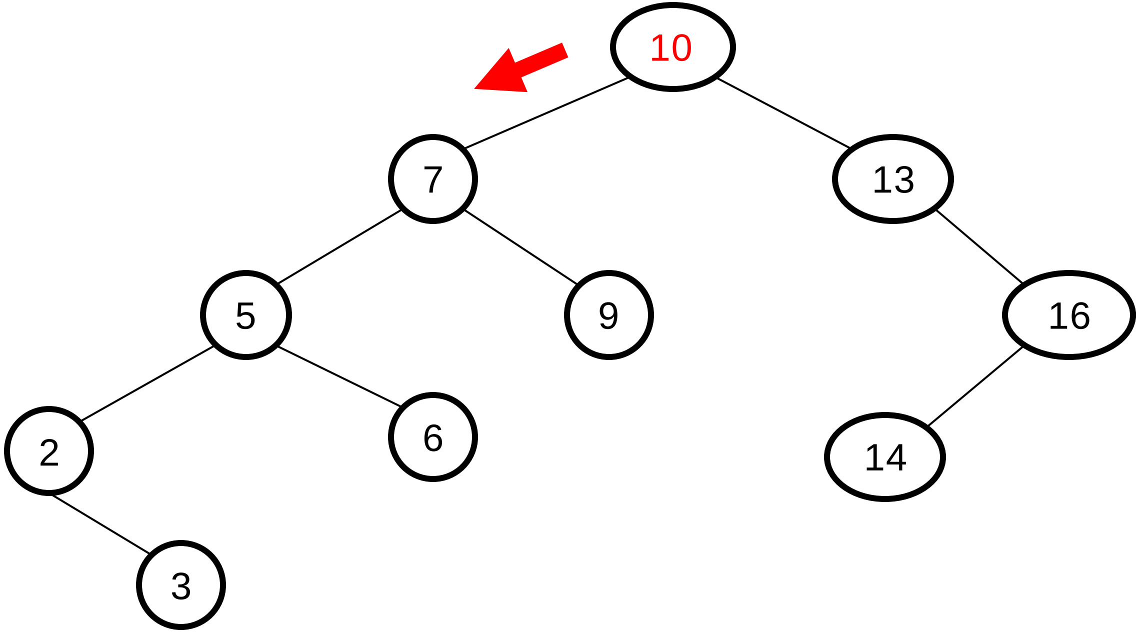
Find Examples



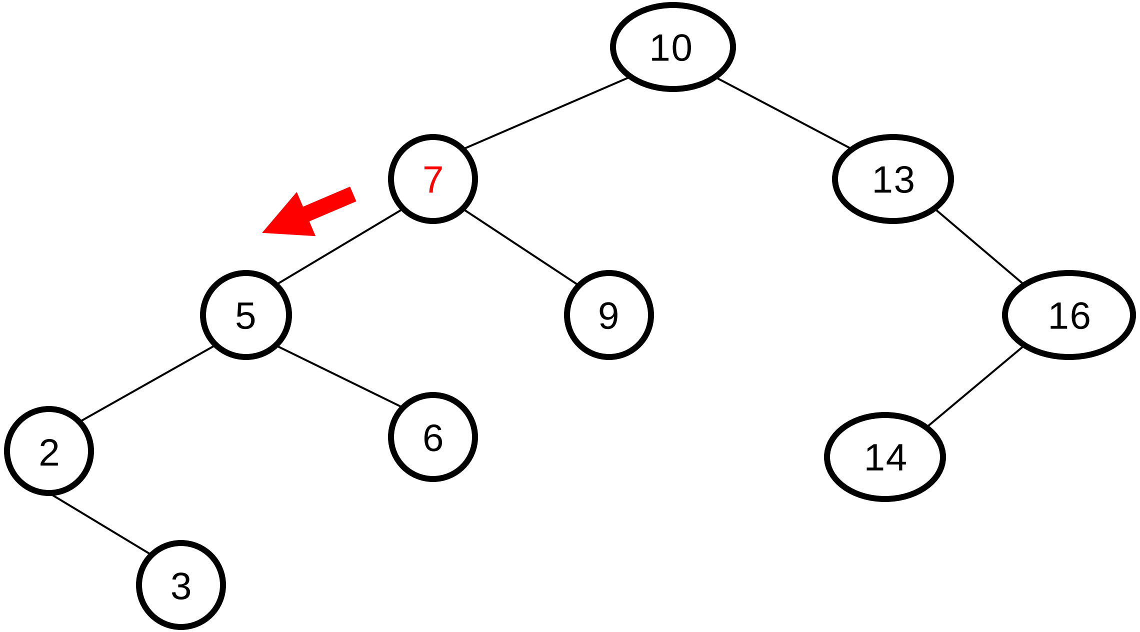
Find 6?



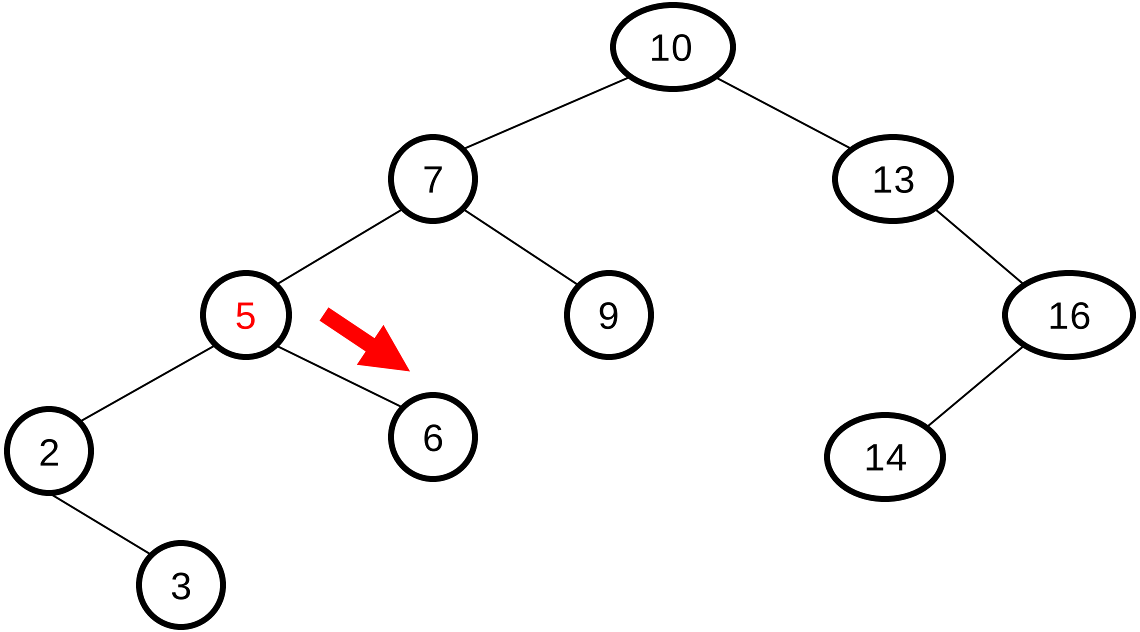
Find 6?



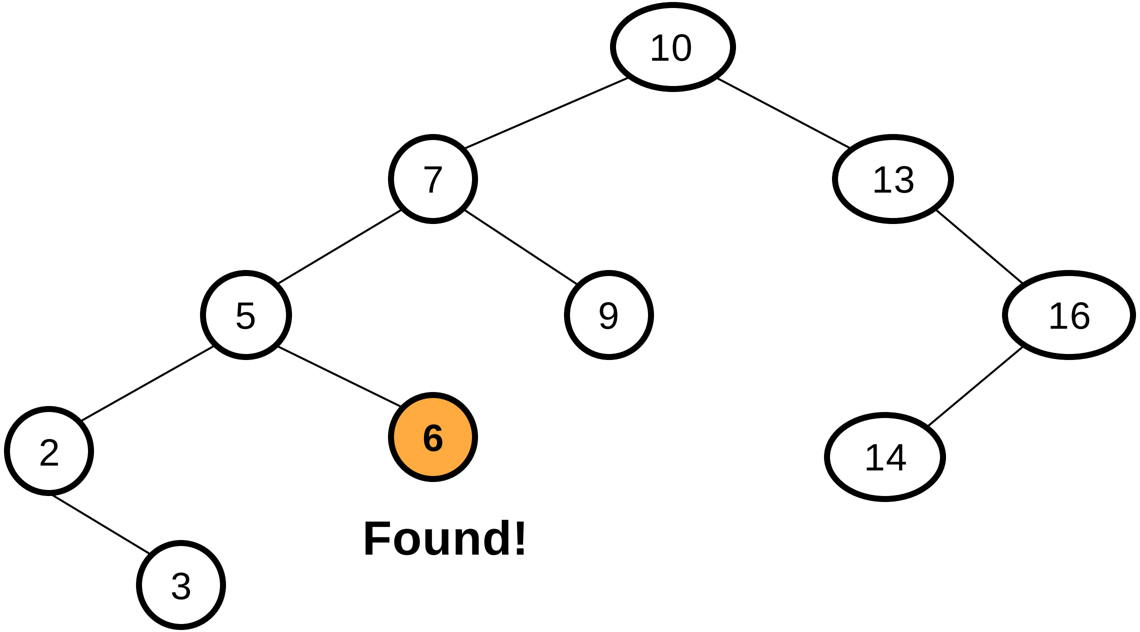
Find 6?



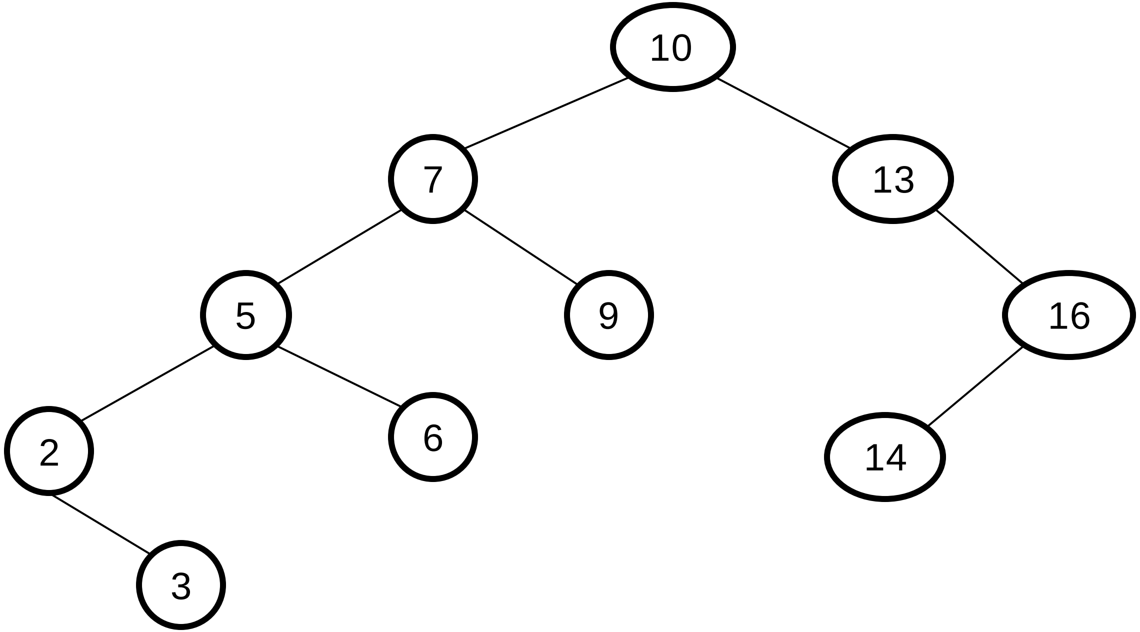
Find 6?



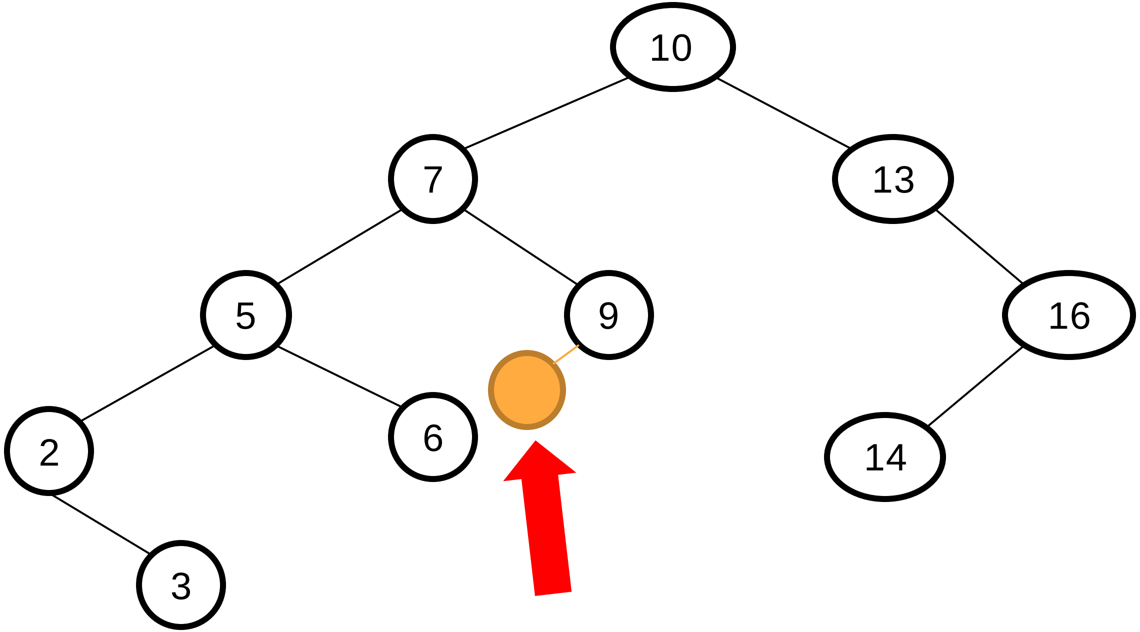
Find 6?



Find 8?



Find 8? Not Found!



(If it existed, it would be here)

Min and Max

- Thanks to BST property, we can easily find min key in tree…
- *Remember, we assume unique keys*
- Min node can’t have other nodes in its **left** subtree
- Min node can’t be in the **right** subtree of any other node
- So where is it?

Min and Max

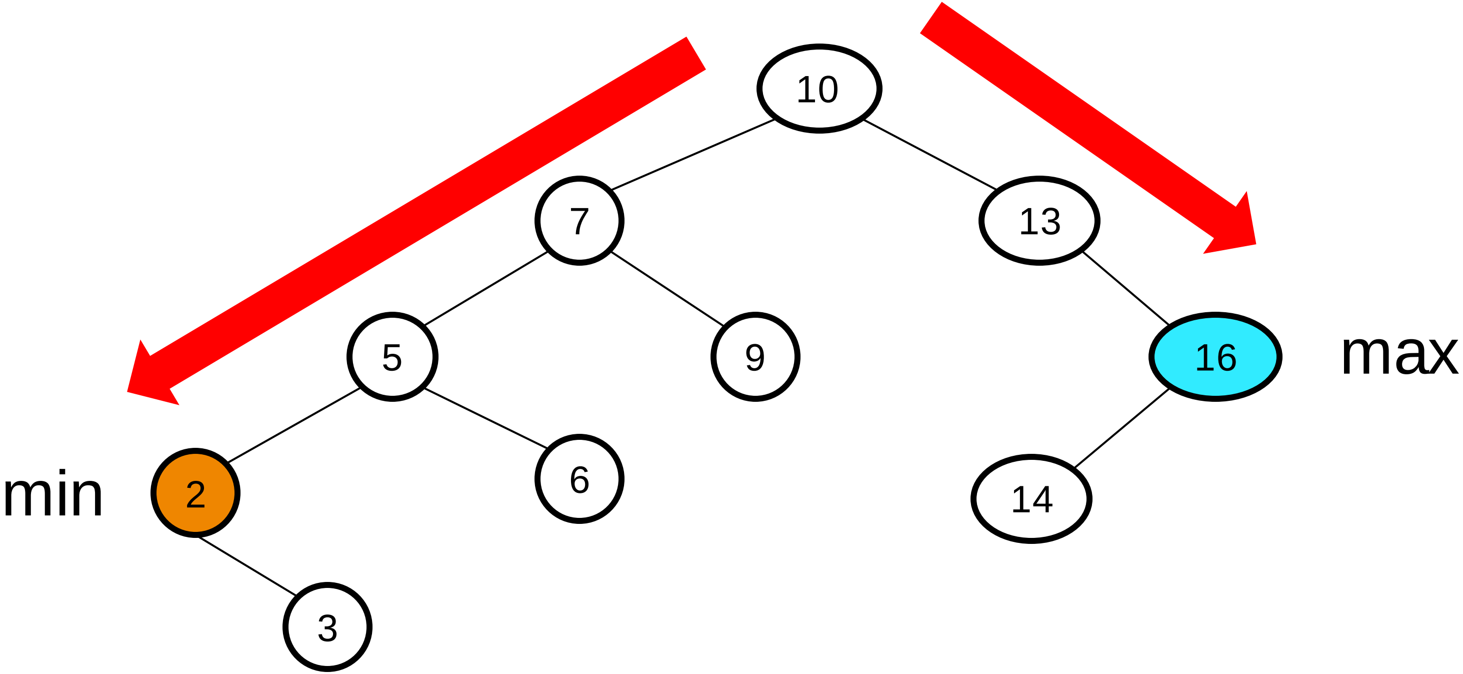
- Thanks to BST property, we can easily find min key in tree…
- *Remember, we assume unique keys*
- Min node can’t have other nodes in its **left** subtree
- Min node can’t be in the **right** subtree of any other node
- Start at root, go left until no longer possible. Final node is min.

Min and Max

- Thanks to BST property, we can easily find min key in tree…
- *Rememb*
- Min node ee
- Min node er node
- Start at root, go left until no longer possible. Final node is min.

Max is found by "opposite" rule (keep going right), for similar reasons.

Min/Max Examples

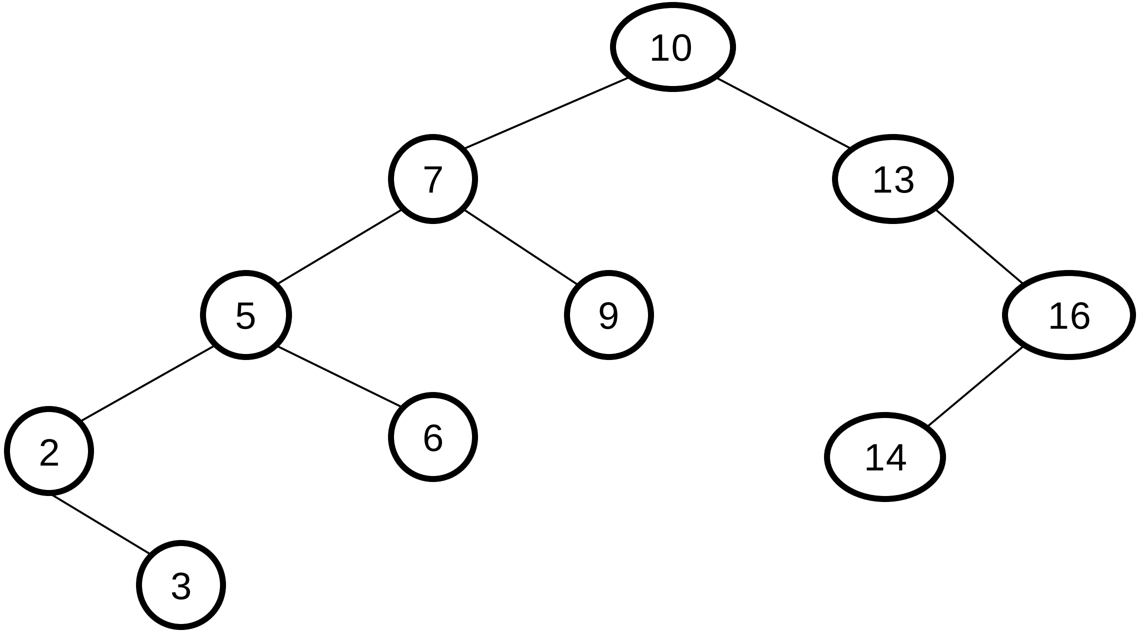


How to Insert a Key into a BST

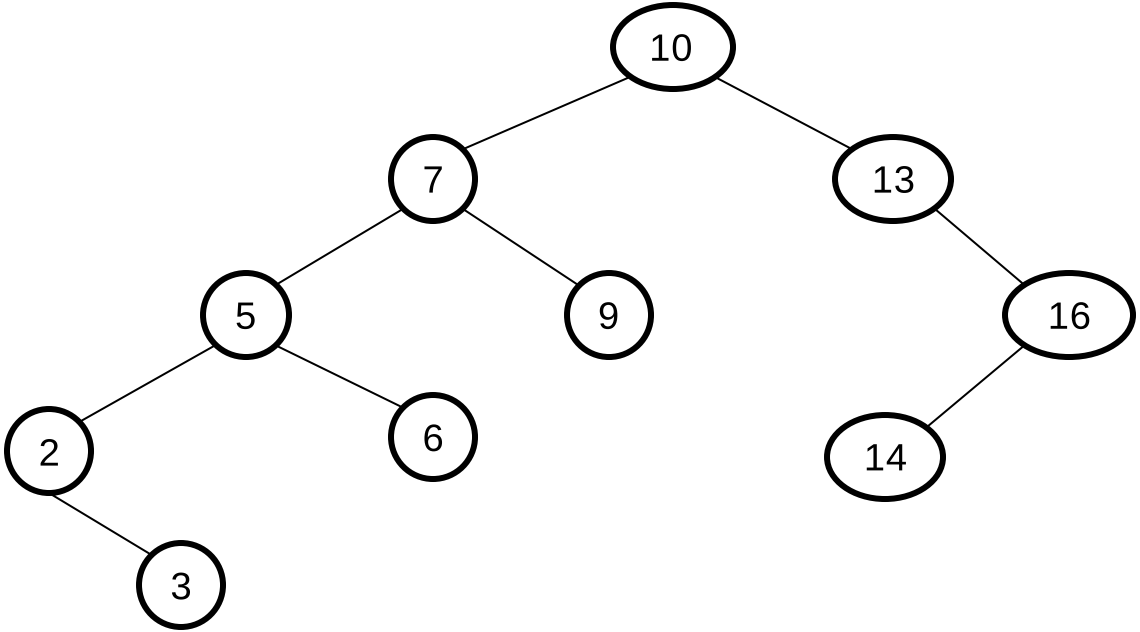
- An unsuccessful find() ends at null subtree where node containing key would be if it existed.

- → Create a new leaf node there and put the key in it!

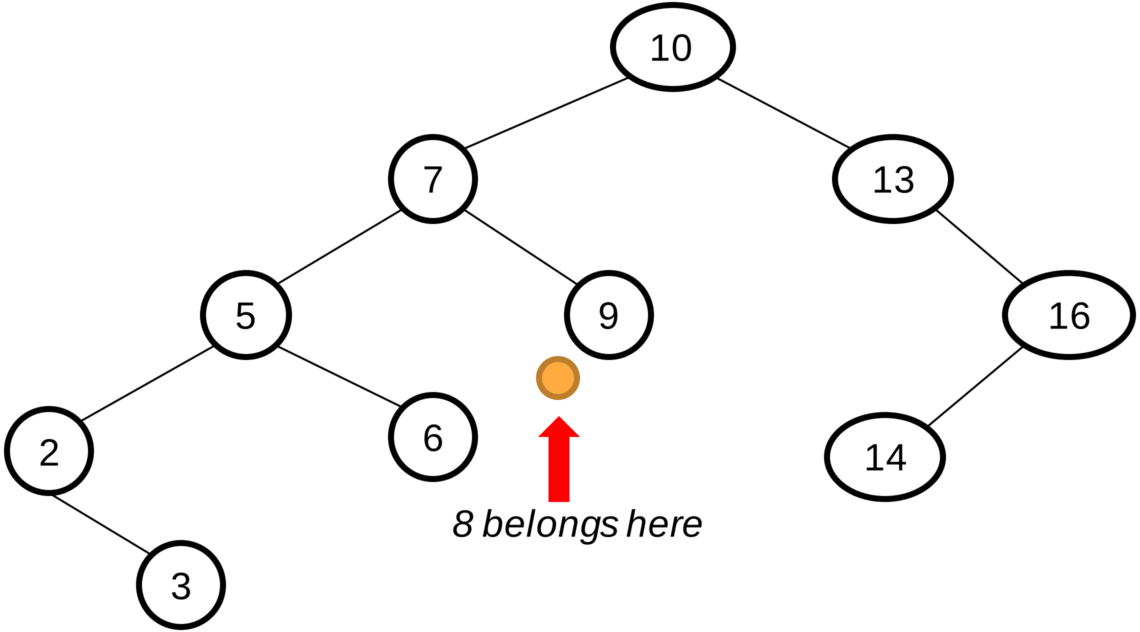
Insert Examples



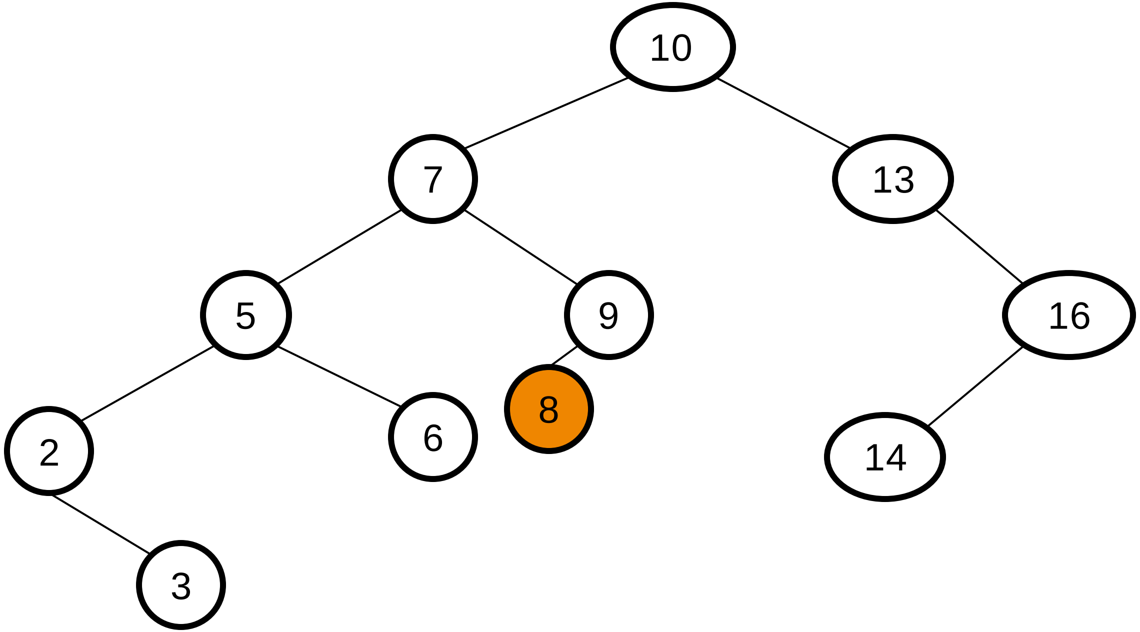
Insert 8



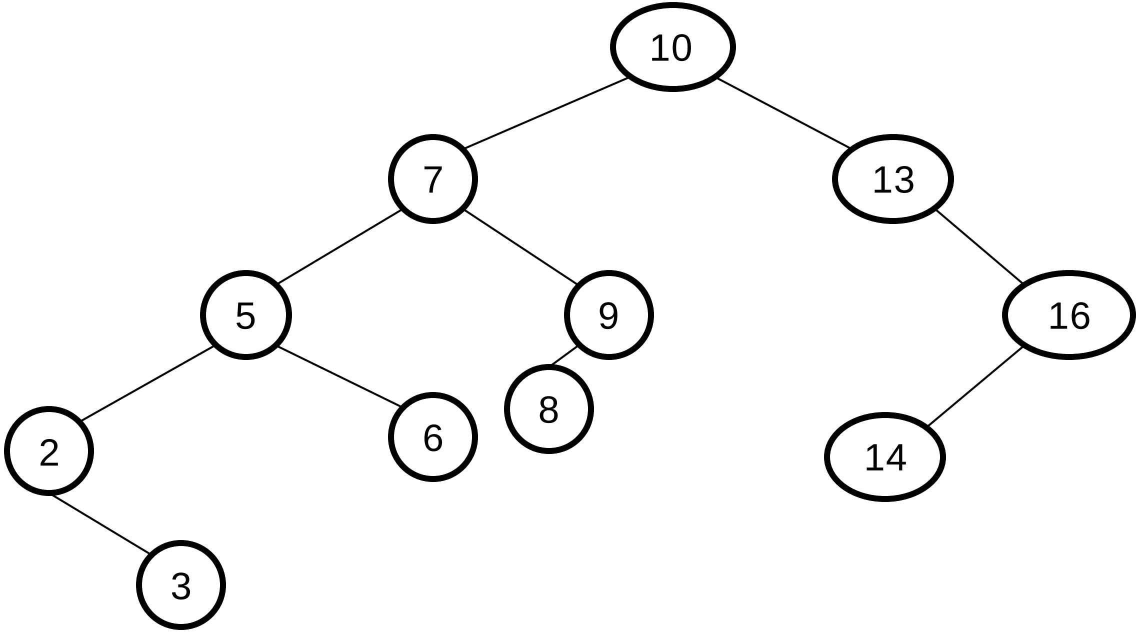
Insert 8



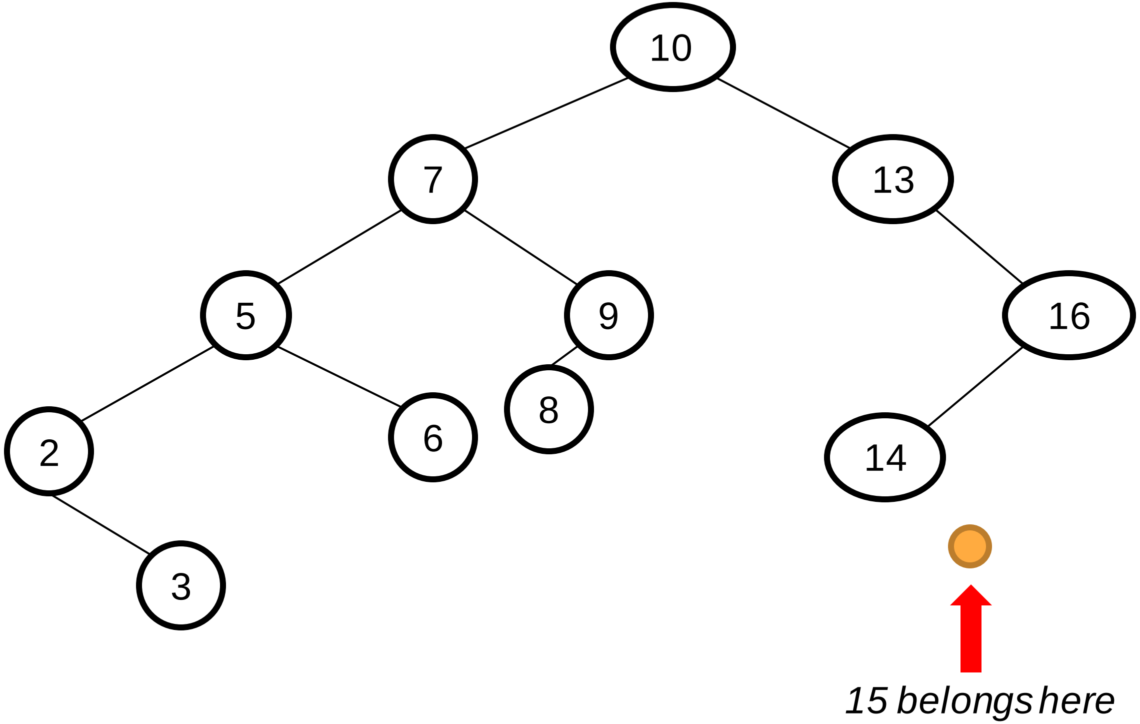
Insert 8



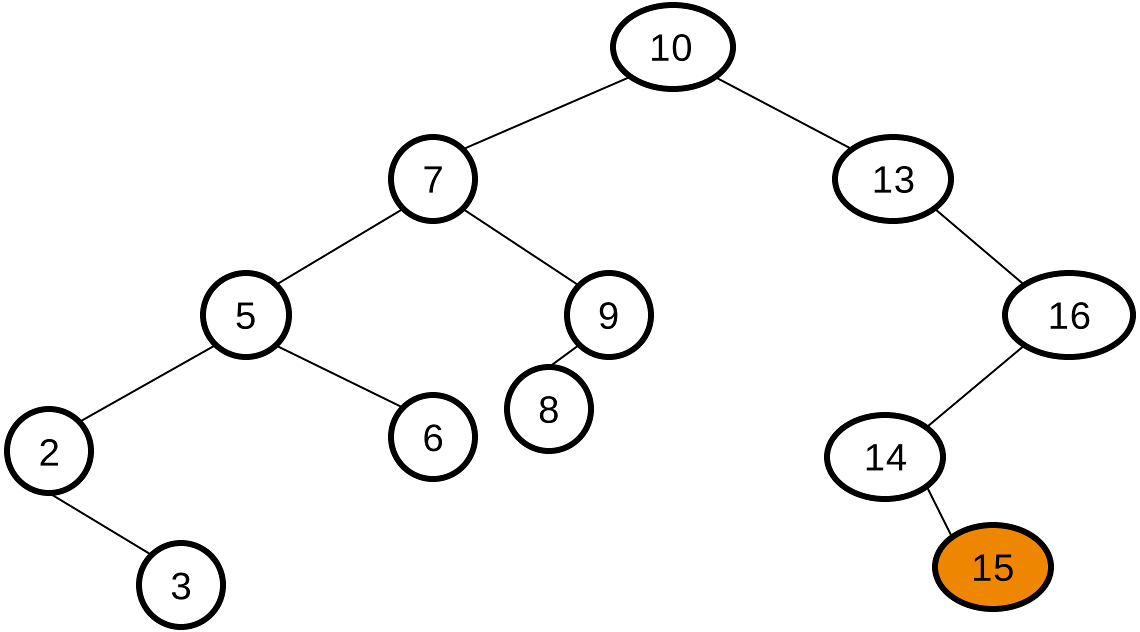
Insert 15



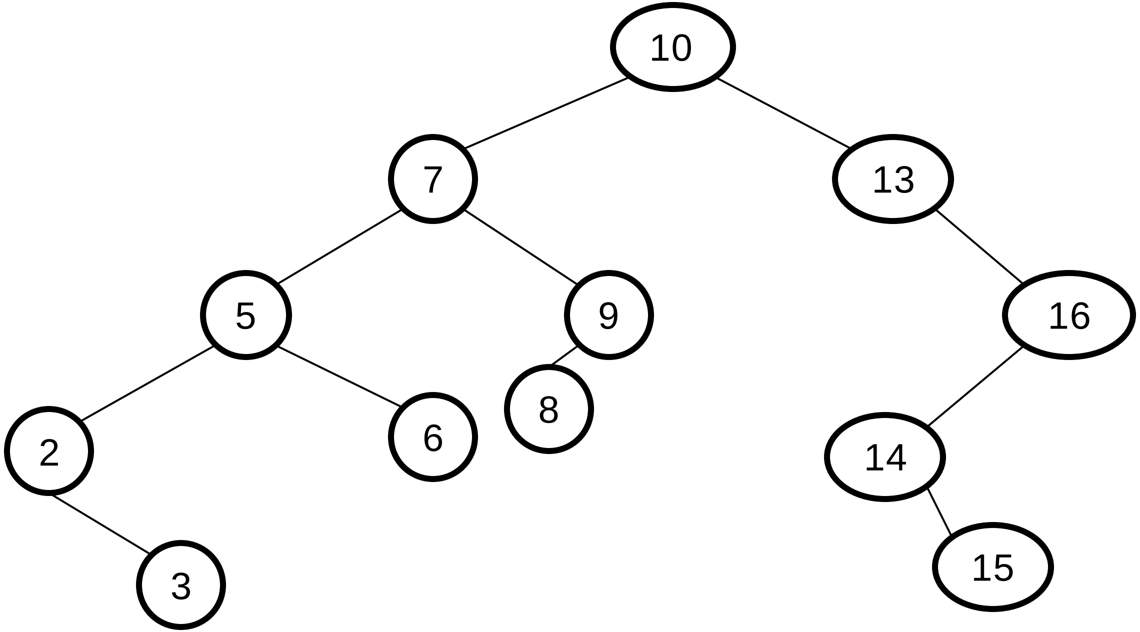
Insert 15



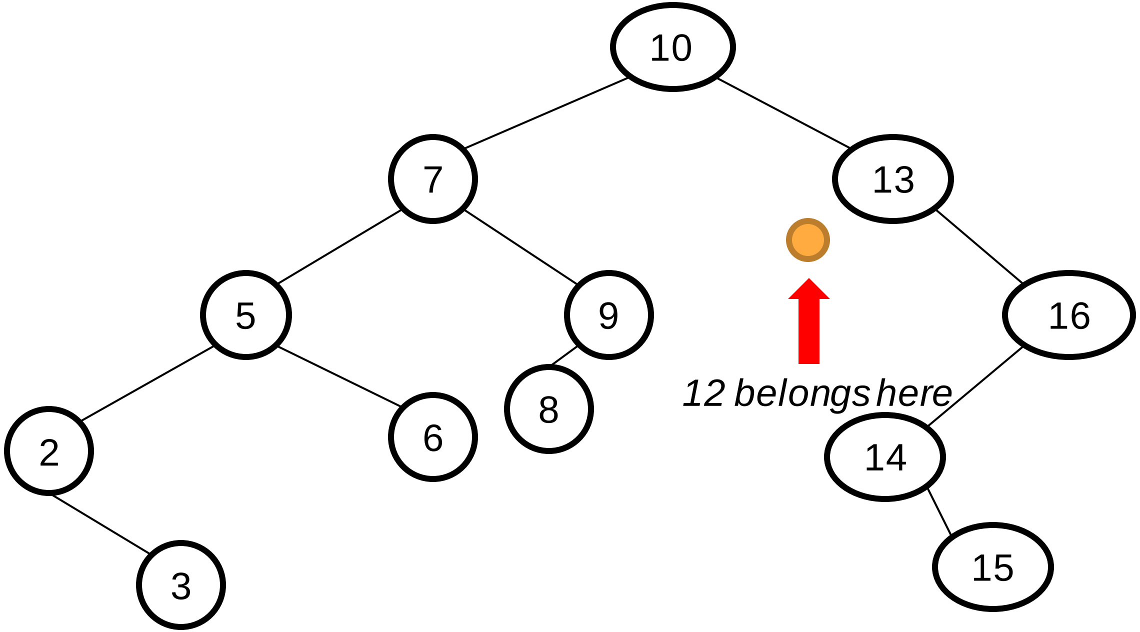
Insert 15



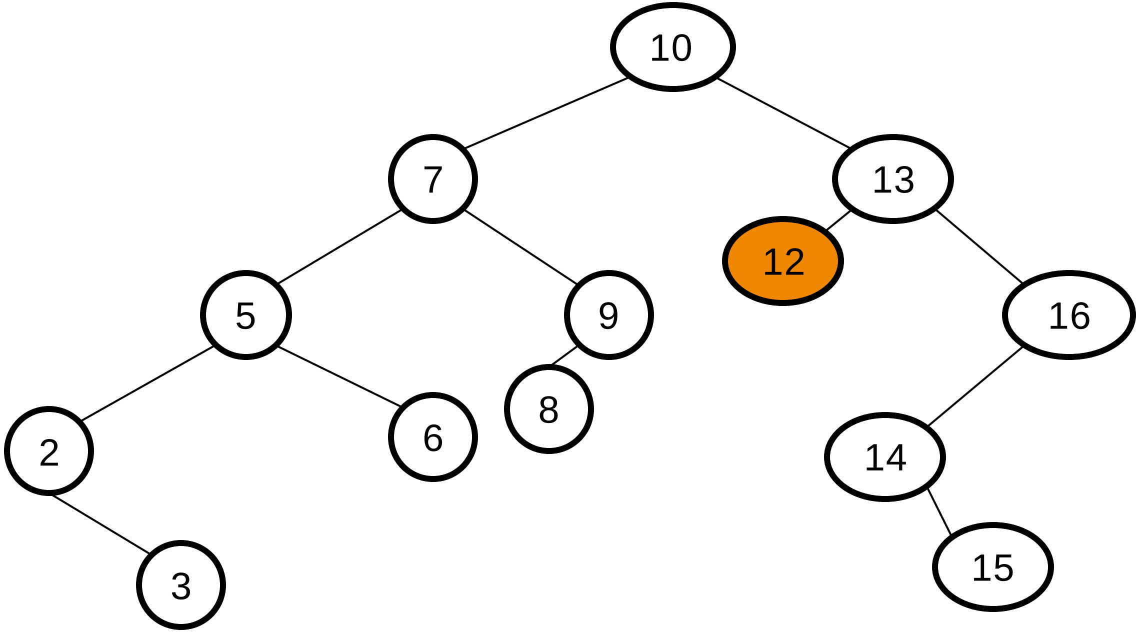
Insert 12



Insert 12



Insert 12



The Story So Far

- Find ✓
- Min/Max ✓
- Insert ✓
- Iterate?
- Remove?

Worst-Case Cost of Operations

- Find – might have to walk from root to deepest leaf of tree
- Min/Max – same
- Insert – same
- Iterate?
- Remove?

Worst-Case Cost of Operations

- Find – $\Theta(h)$ for tree of height h
- Min/Max – same
- Insert – same
- Iterate?
- Remove?

And Now, Some Slightly Less Trivial Methods

Iteration

- As we saw in Lab 7, a collection can provide an iterator
- An iterator for a BST starts out pointing to the min node (by key)
- Each call to iterator.next() must move from current node to next largest
- *This operation is called finding the successor of a node*
- We write it as “succ(x)” for a node x

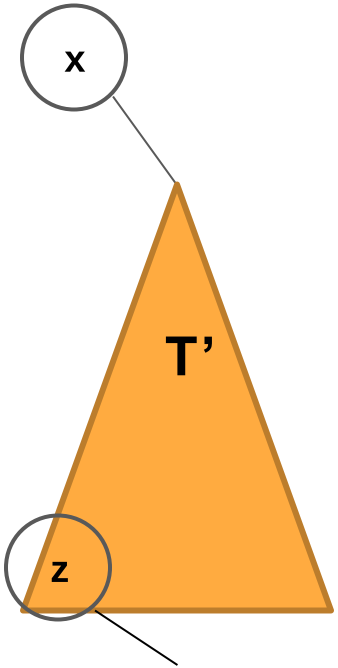
Where is Successor of Node x?

- If x has a right subtree T'…

- *Leftmost (minimum)* node z in T' is > x.

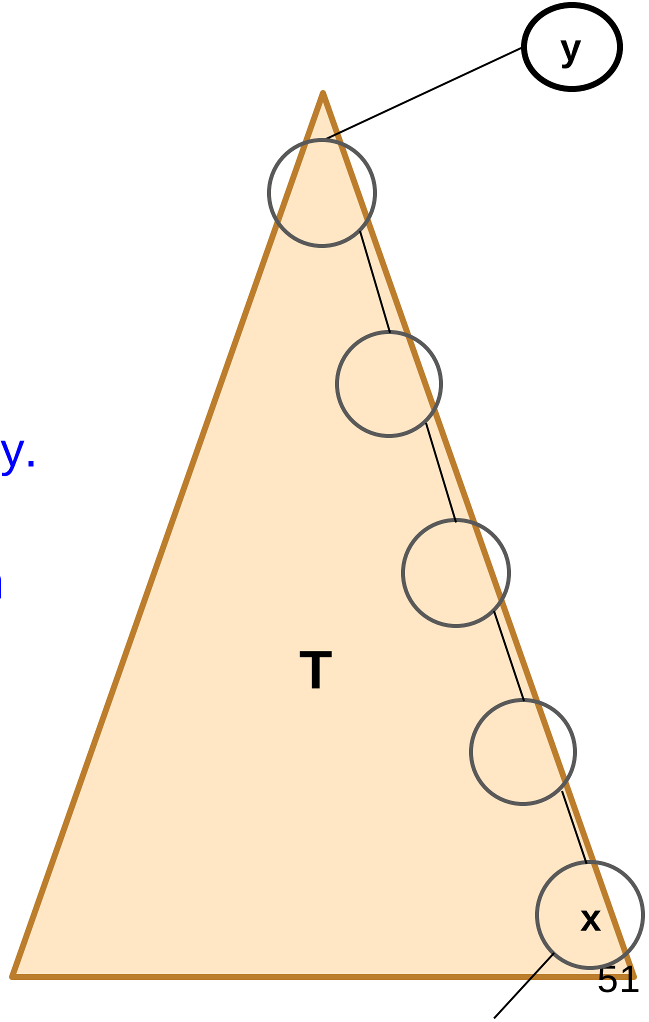
- Every node > x that is not in T' is > every node in T', hence is also > z.

- Conclude that succ(x) = z.



Where is Successor of Node x?

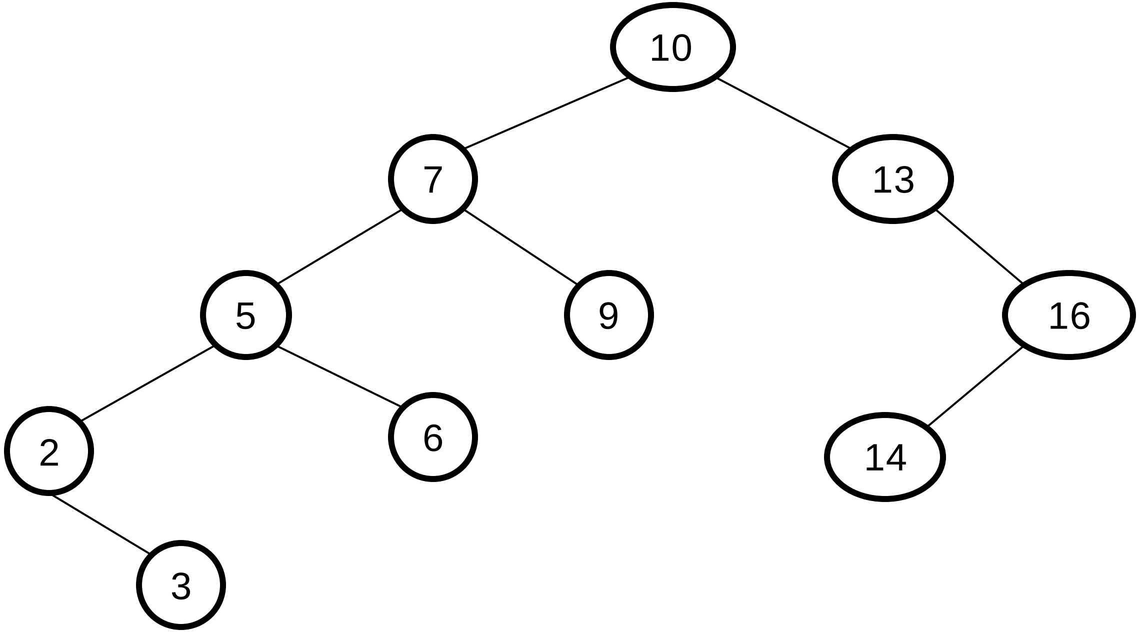
- If x has no right subtree…
- If any node of tree is > x, then x is *rightmost (maximum) node in left subtree T* of some node y.
- Every node < y that is not in T is < every node in T, hence is also < x.
- Conclude that succ(x) = y.



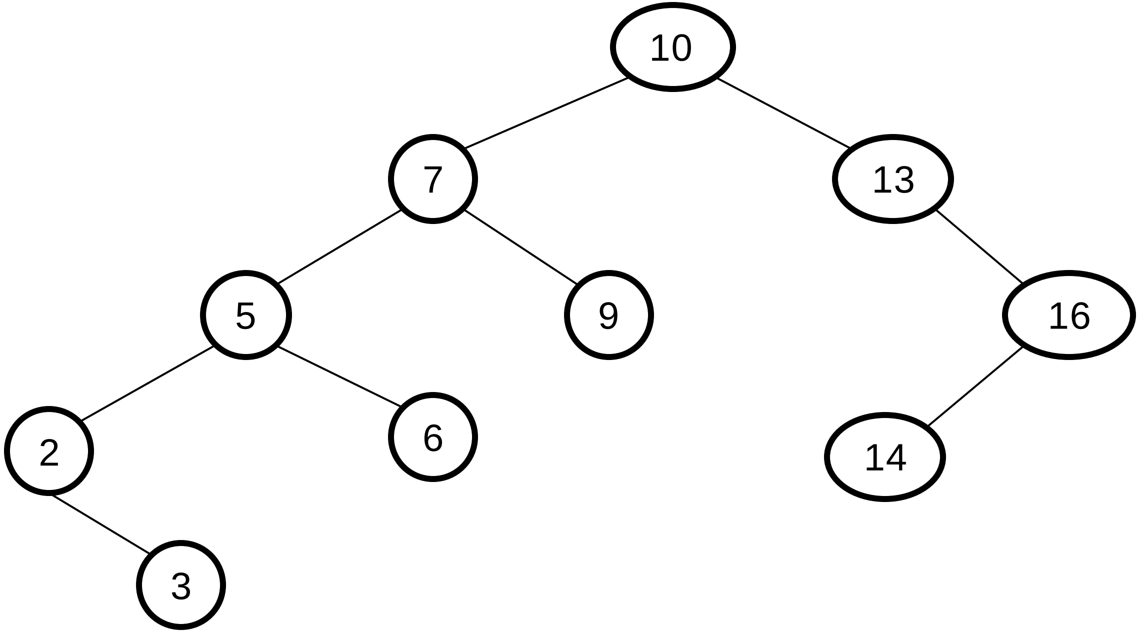
How to Compute succ(x)

- If x has a right subtree T’
- return min(T’)
- Else
- follow parent pointers from x until some node y is a right parent
- return y

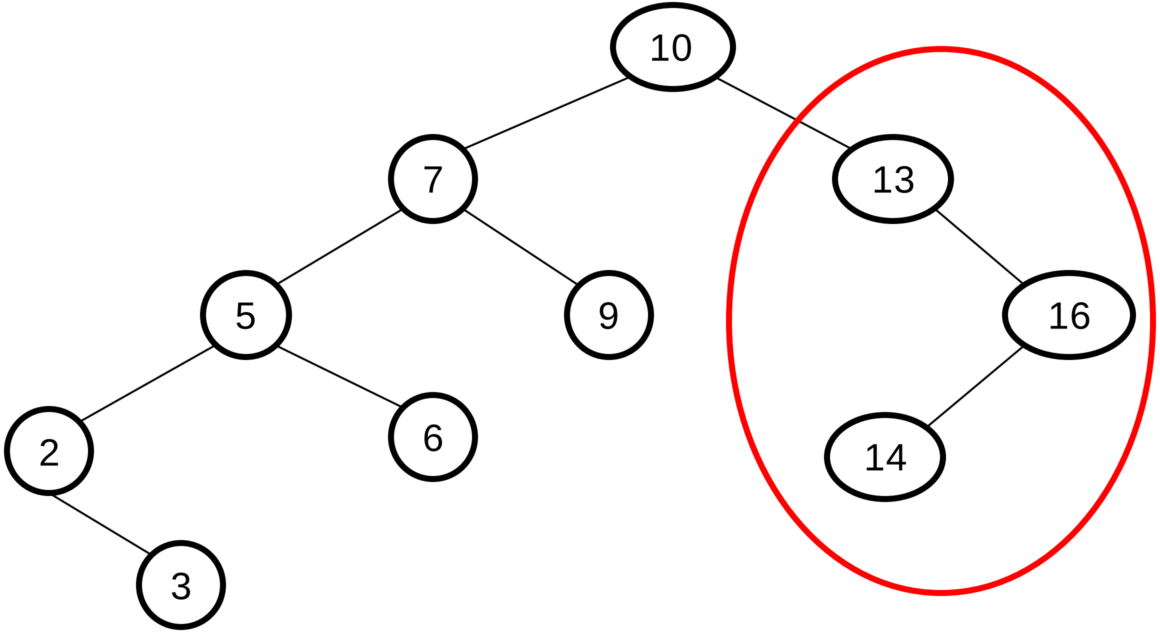
Successor Examples



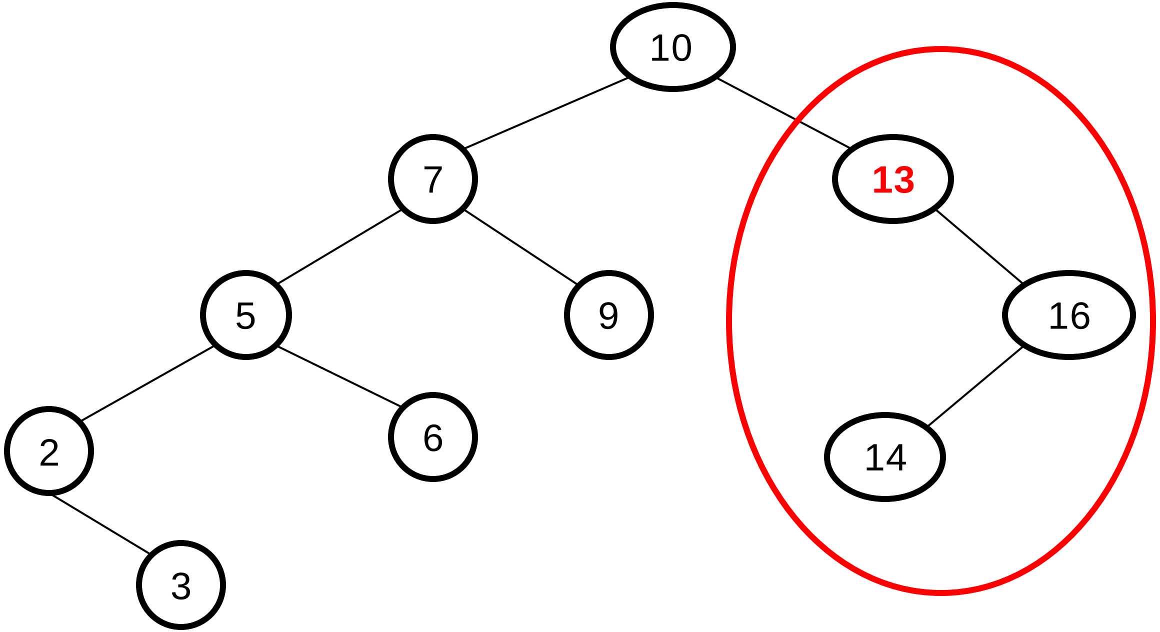
Succ(10)



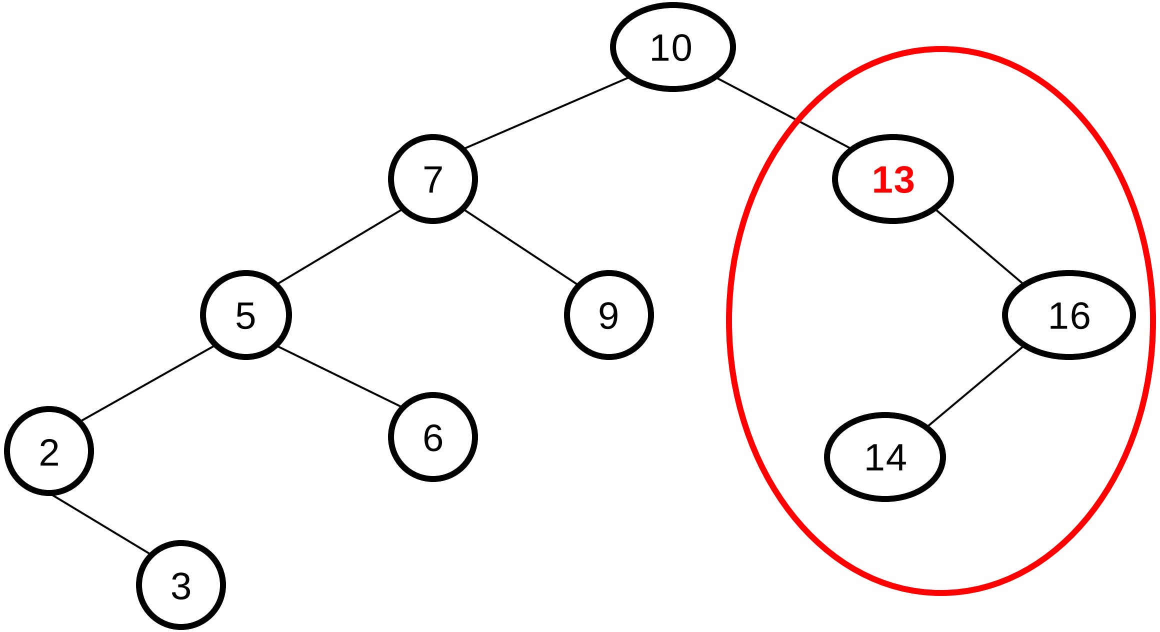
Succ(10) – 10 has a right subtree



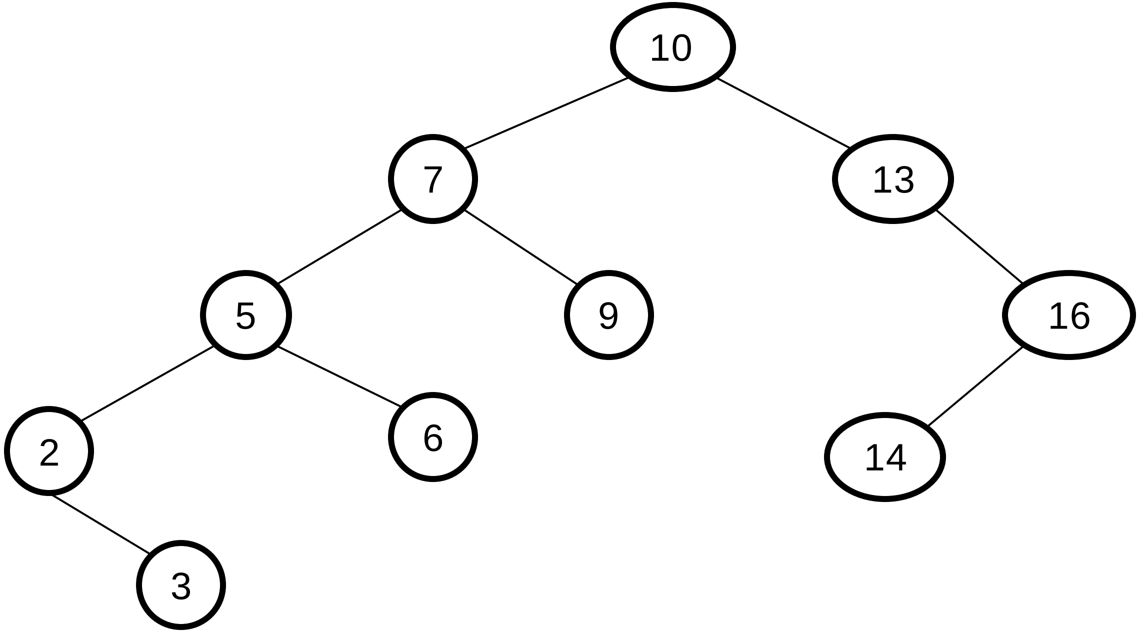
Succ(10) – min of right subtree of 10 is 13



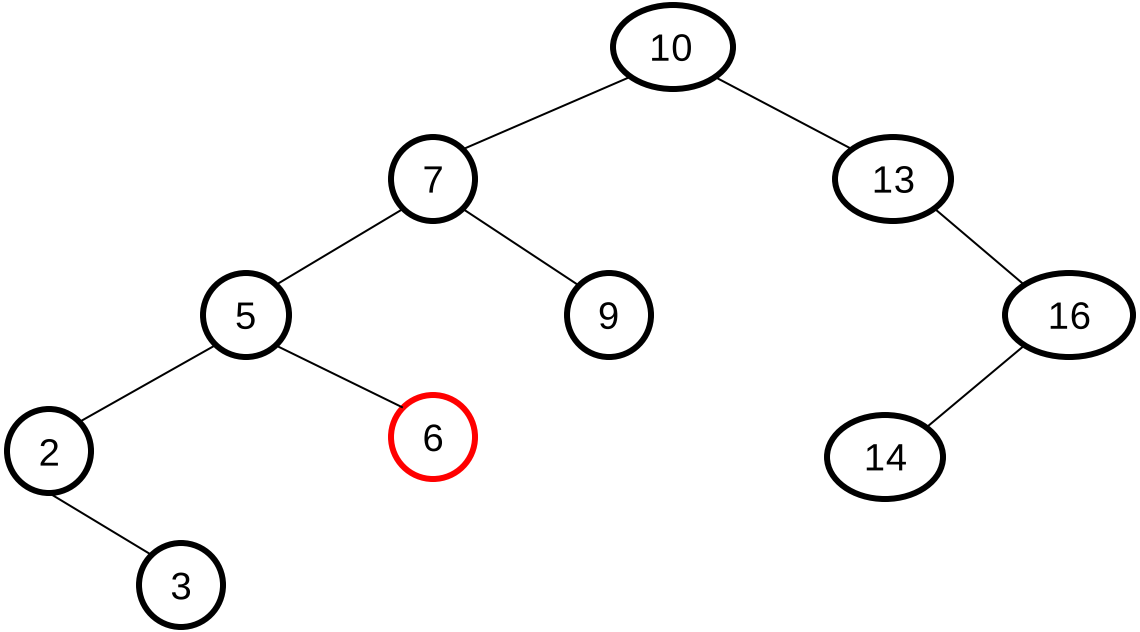
Succ(10) = 13



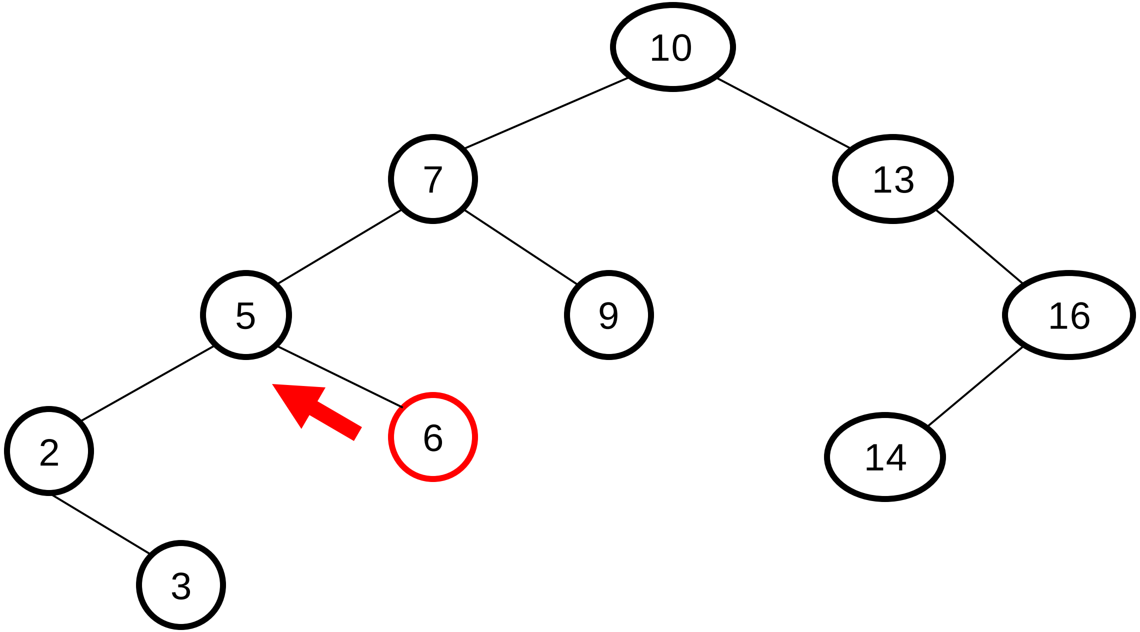
Succ(6)



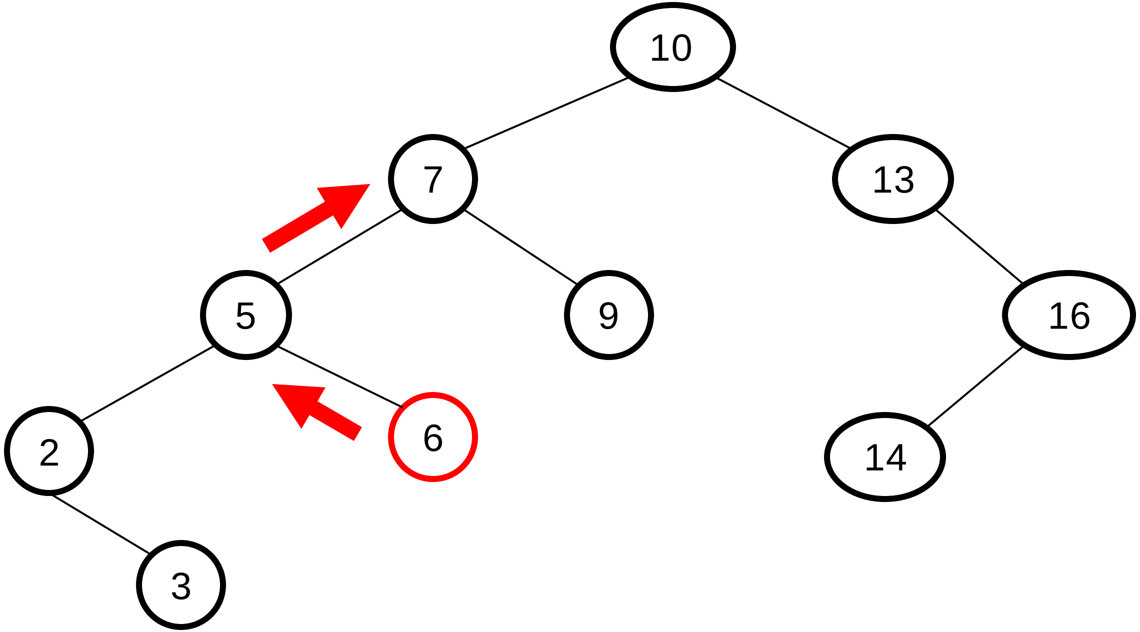
Succ(6) – 6 has *no* right subtree



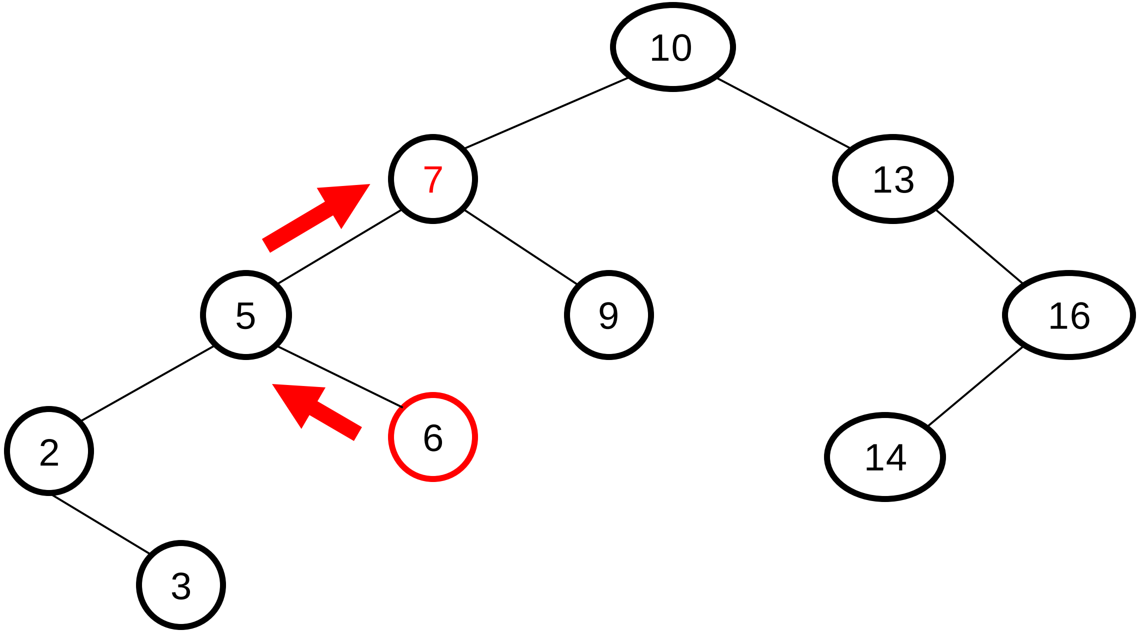
Succ(6) – Follow parents to first right parent



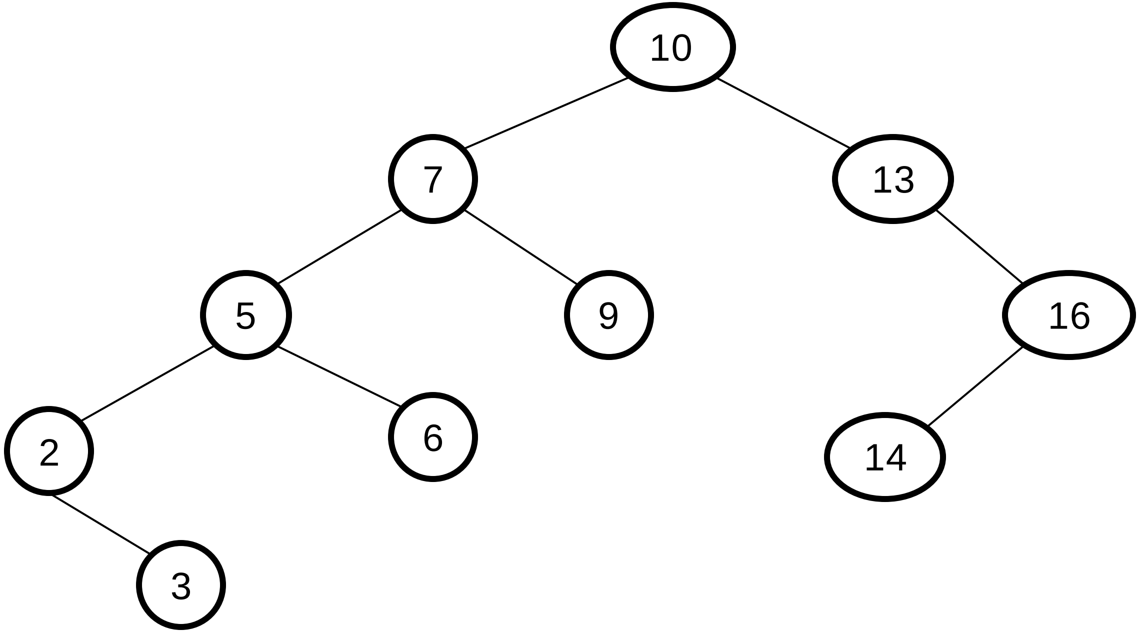
Succ(6) – Follow parents to first right parent



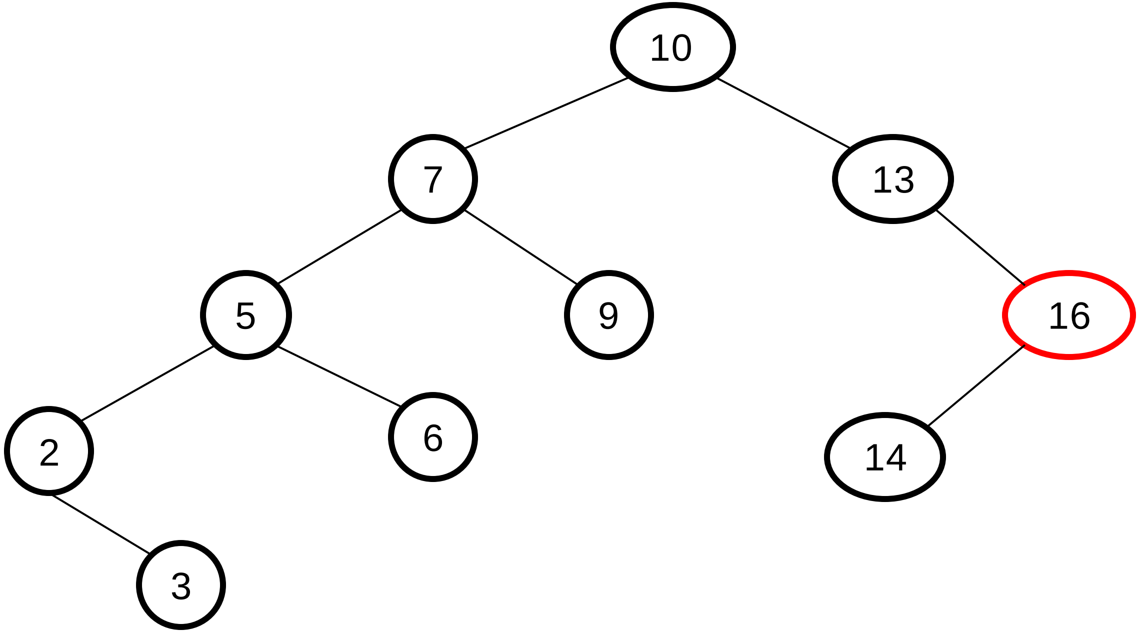
Succ(6) = 7



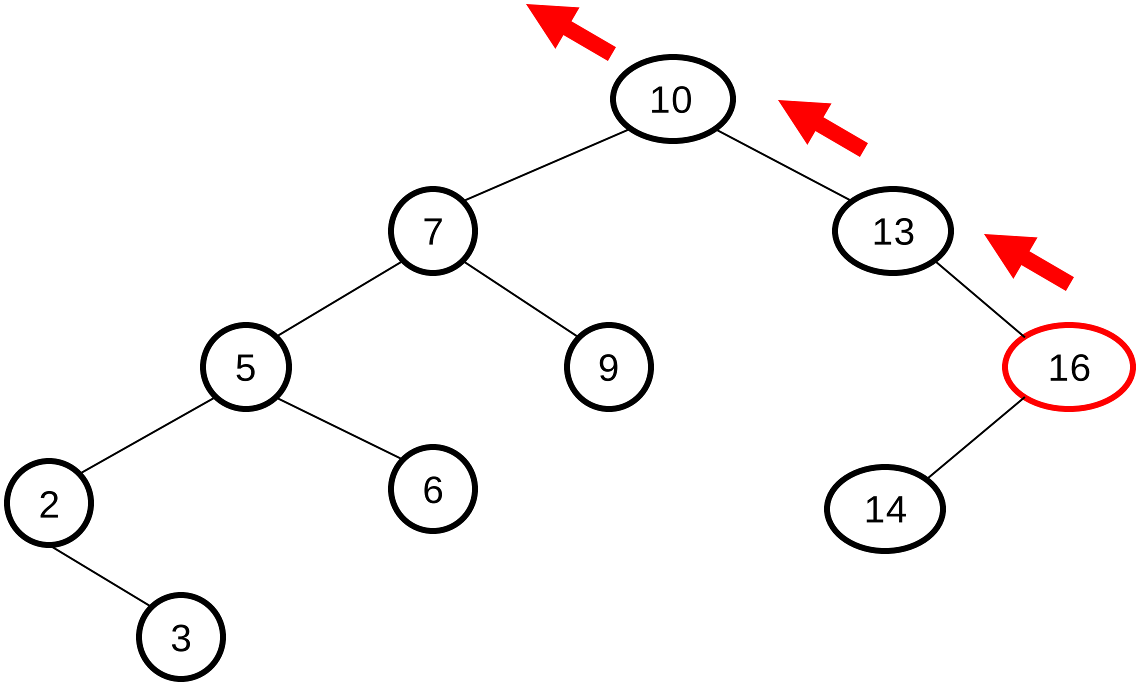
Succ(16)



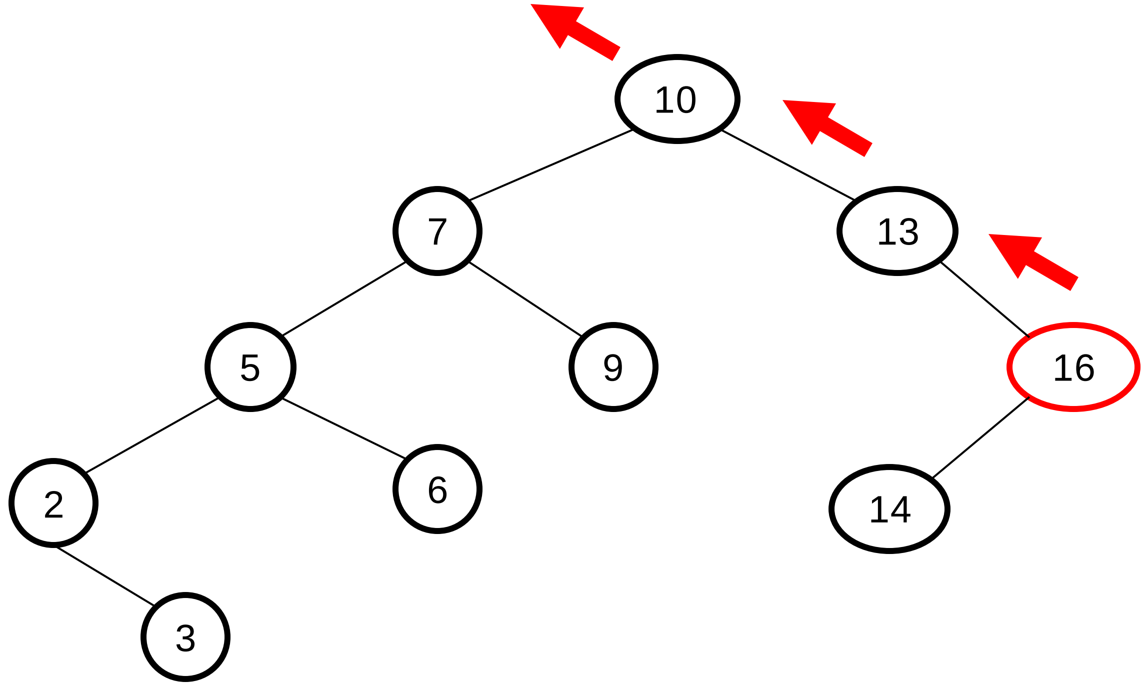
Succ(16) – 16 has no right subtree



Succ(16) – follow parents to first right parent?



Succ(16) *does not exist* (16 is max!)



Worst-Case Cost of Operations

- Find – might have to walk from root to deepest leaf of tree
- Min/Max – same
- Insert – same
- Iterate – might have to walk from root to deepest leaf *or* vice versa
- Remove?

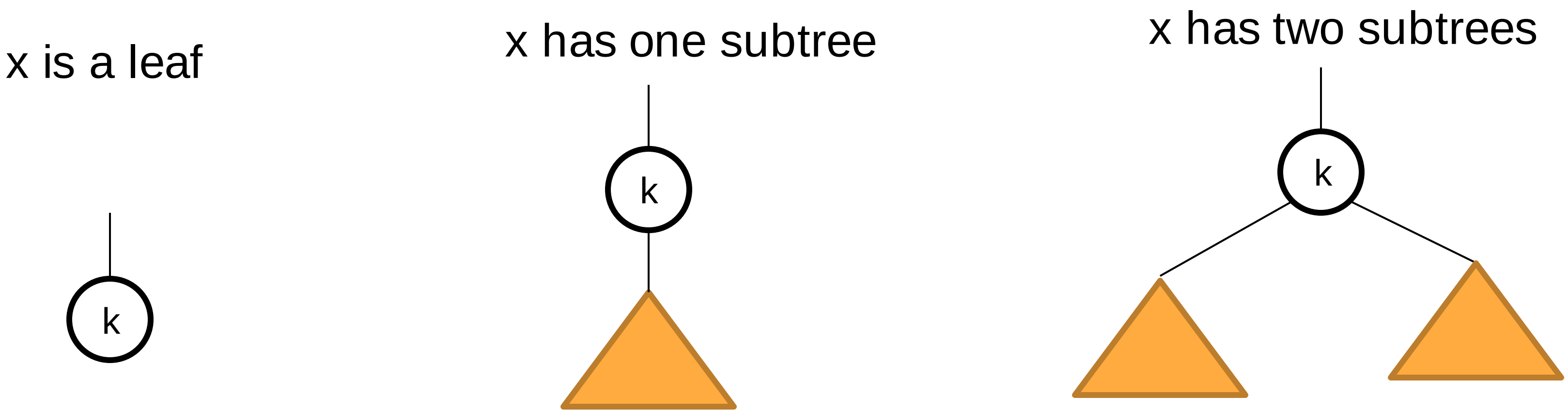
Worst-Case Cost of Operations

- Find – Θ(h) for tree of height h
- Min/Max – same
- Insert – same
- Iterate – same
- Remove?

Last But Not Least, Remove(k)

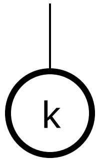
- First, walk down from root to locate node x with key k, as for find().

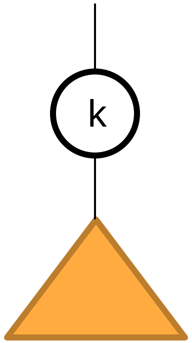
- Three possibilities for node x to be removed:



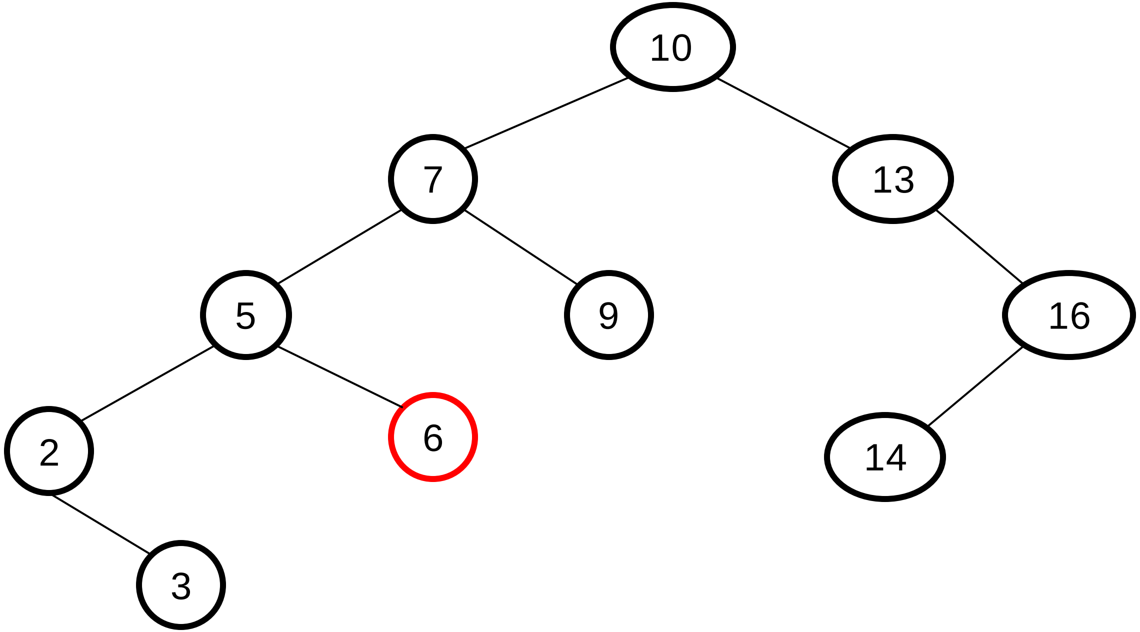
Easy Cases for Removal (*Verify BST Property*)

- If x is a leaf, removing x does not impact remaining tree at all.
- If x has one subtree, remove x and link subtree's root to x's parent.
- (BST property holds between x's parent and its *entire subtree*)

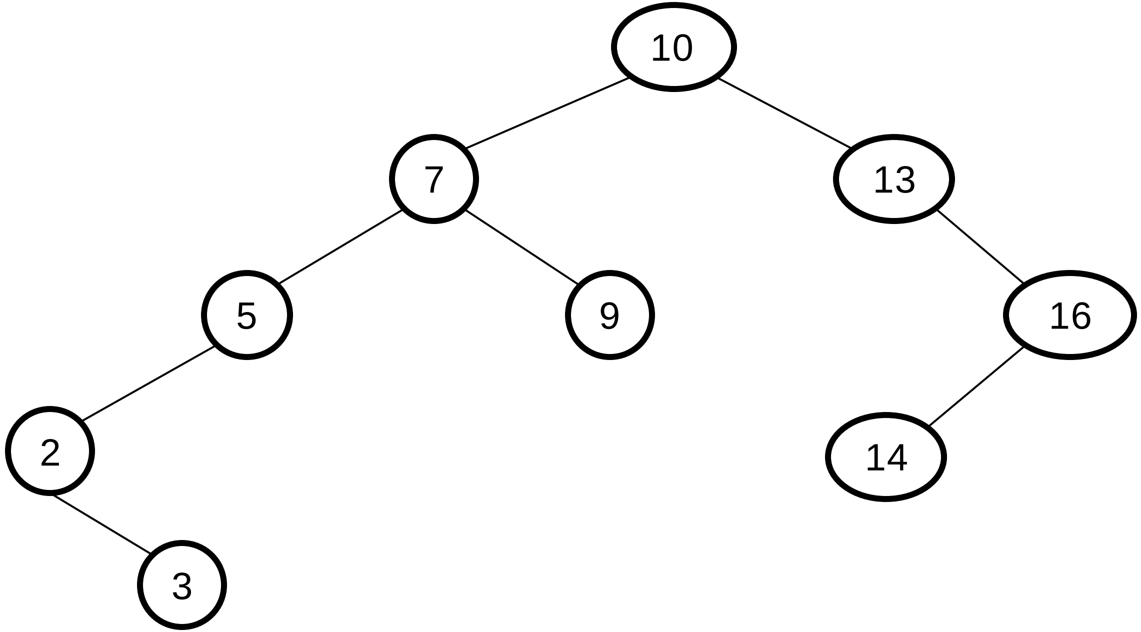




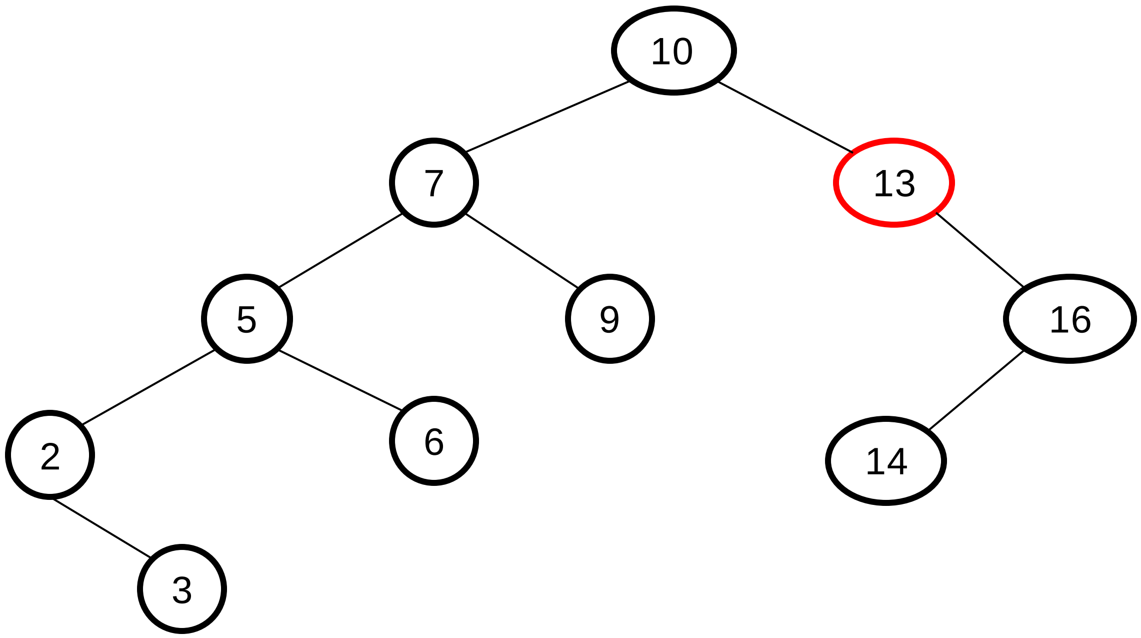
Remove(6)



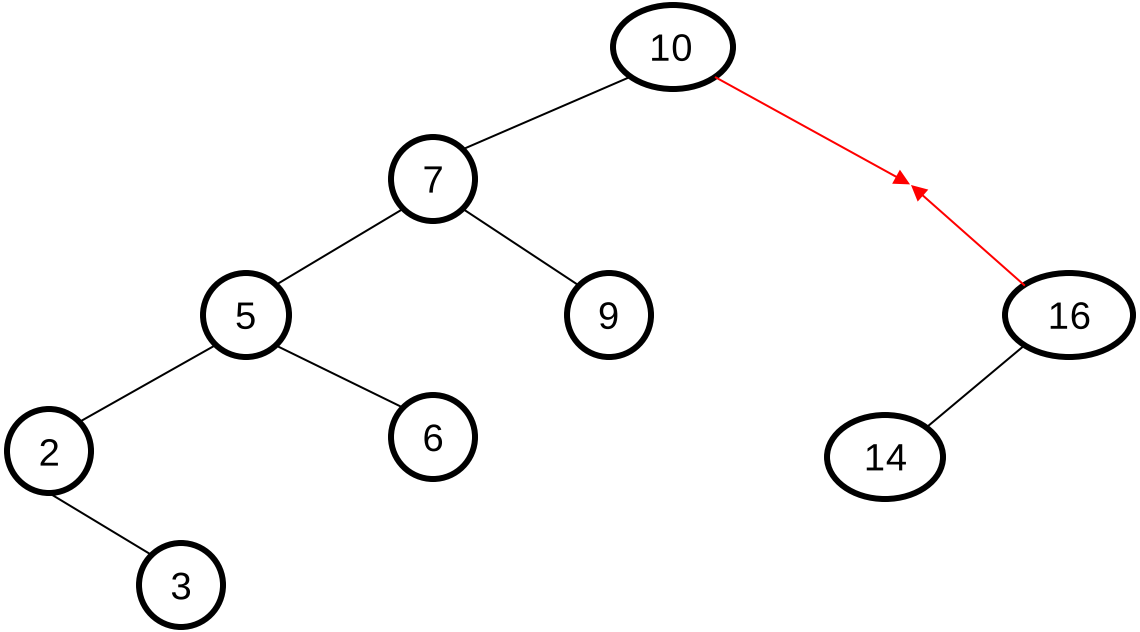
Remove(6)



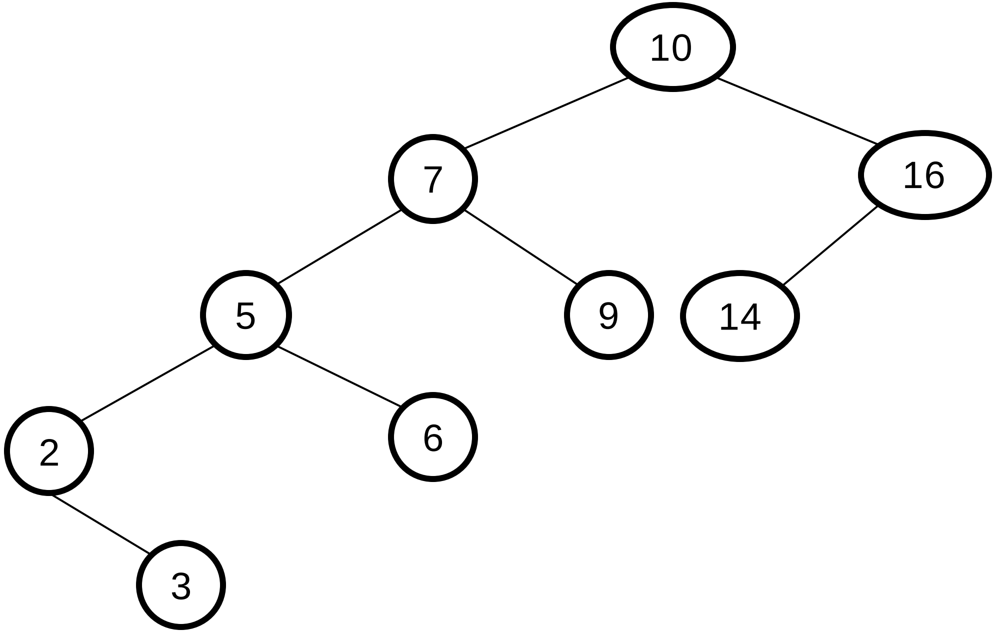
Remove(13)



Remove(13)

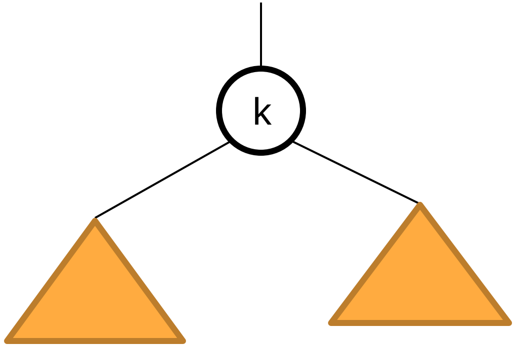


Remove(13)



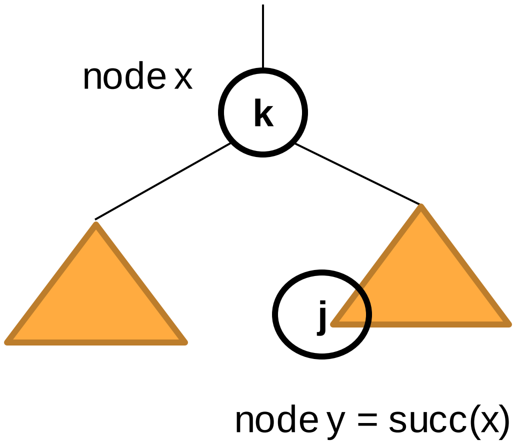
Removing a Node With Two Subtrees

- **We cannot just delete the node!**
- One parent, two subtrees – *no place to put one of the subtrees*
- Instead, will preserve tree structure by “stealing” key from a subtree



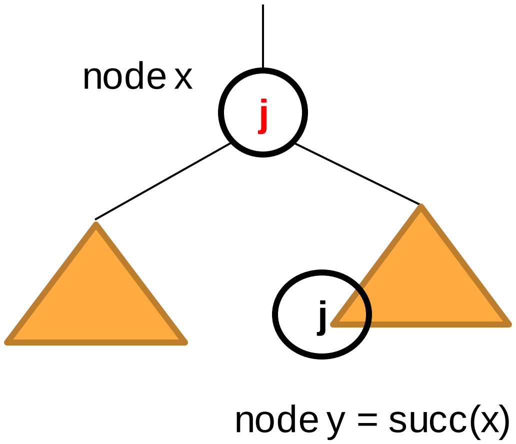
Removing a Node With Two Subtrees

- Let x be node to be deleted, and let y = succ(x).
- Replace x.key by y.key
- This is safe for BST property – **why**?
- Now delete duplicate copy of y.key by removing y



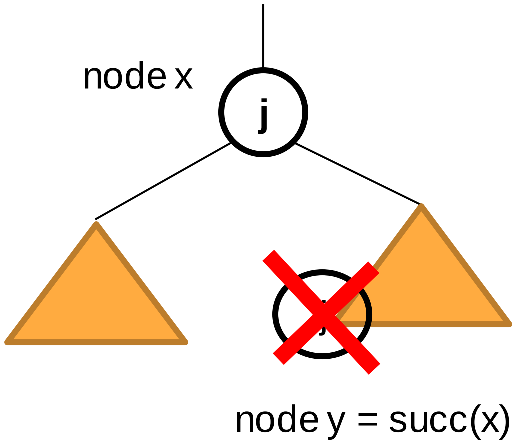
Removing a Node With Two Subtrees

- Let x be node to be deleted, and let y = succ(x).
- Replace x.key by y.key
- This is safe for BST property – **why**?
- Now delete duplicate copy of y.key by removing y

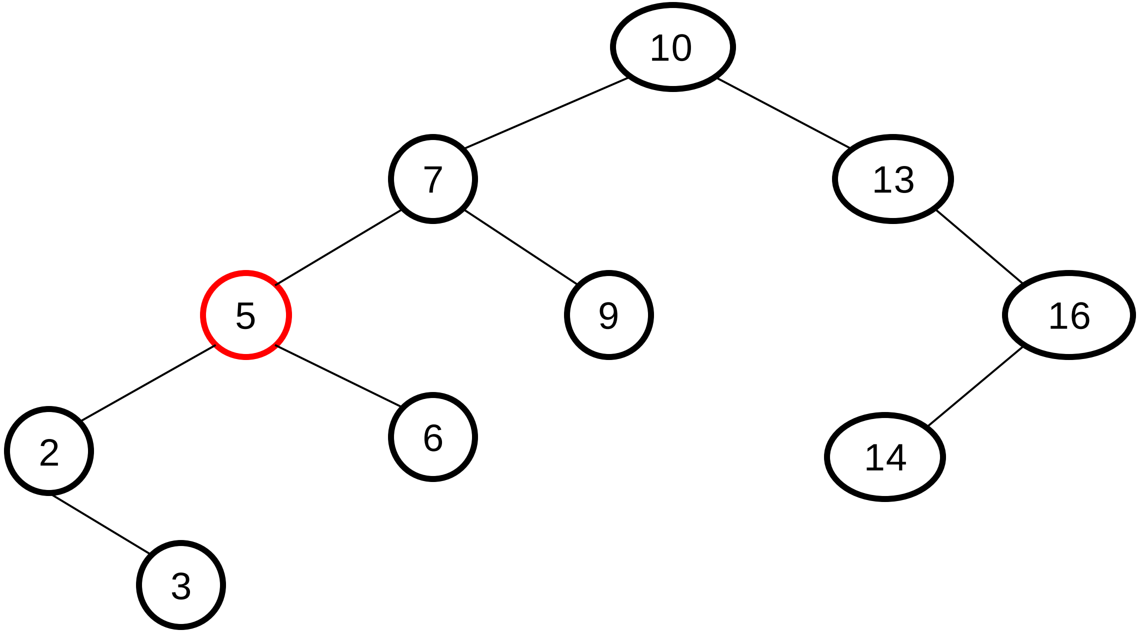


Removing a Node With Two Subtrees

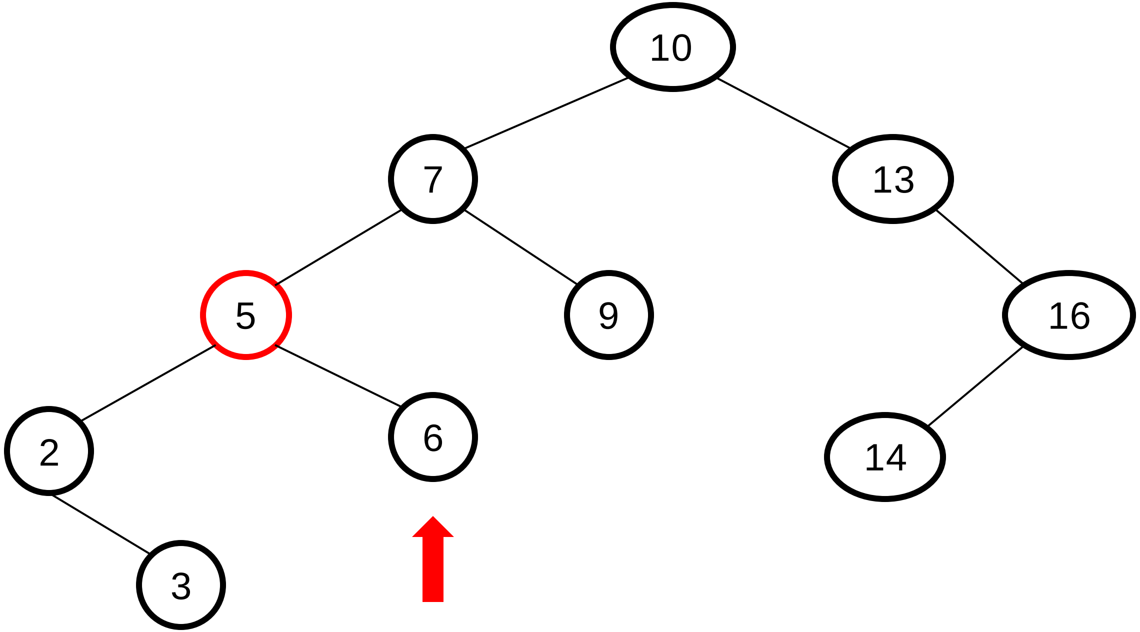
- Let x be node to be deleted, and let y = succ(x).
- Replace x.key by y.key
- This is safe for BST property – **why**?
- Now delete duplicate copy of y.key by removing y



Remove 5

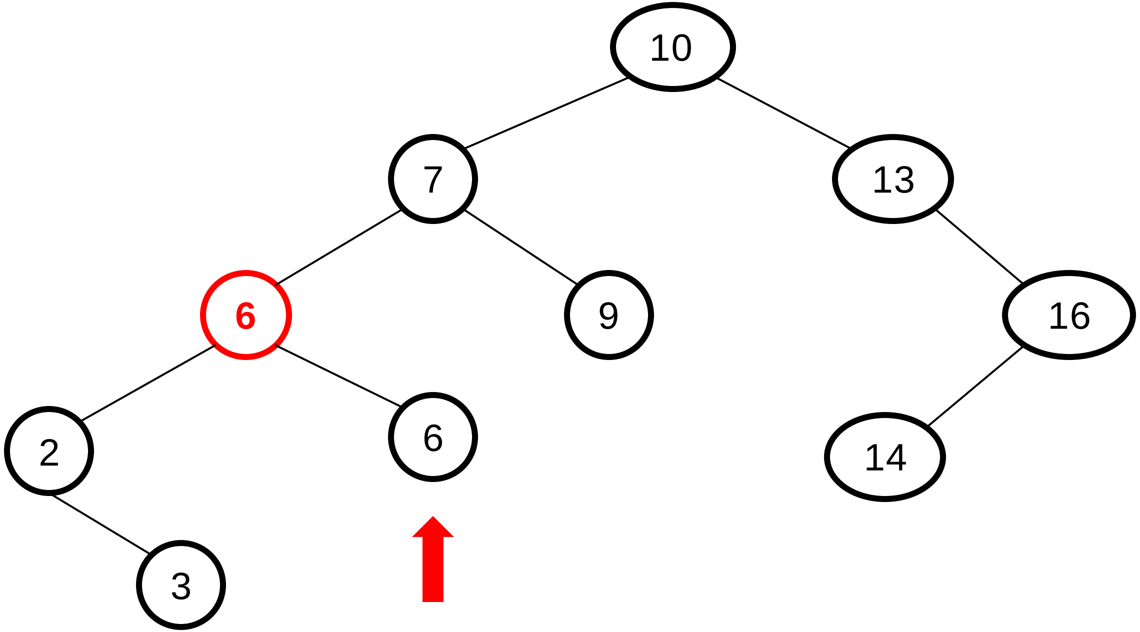


Remove 5



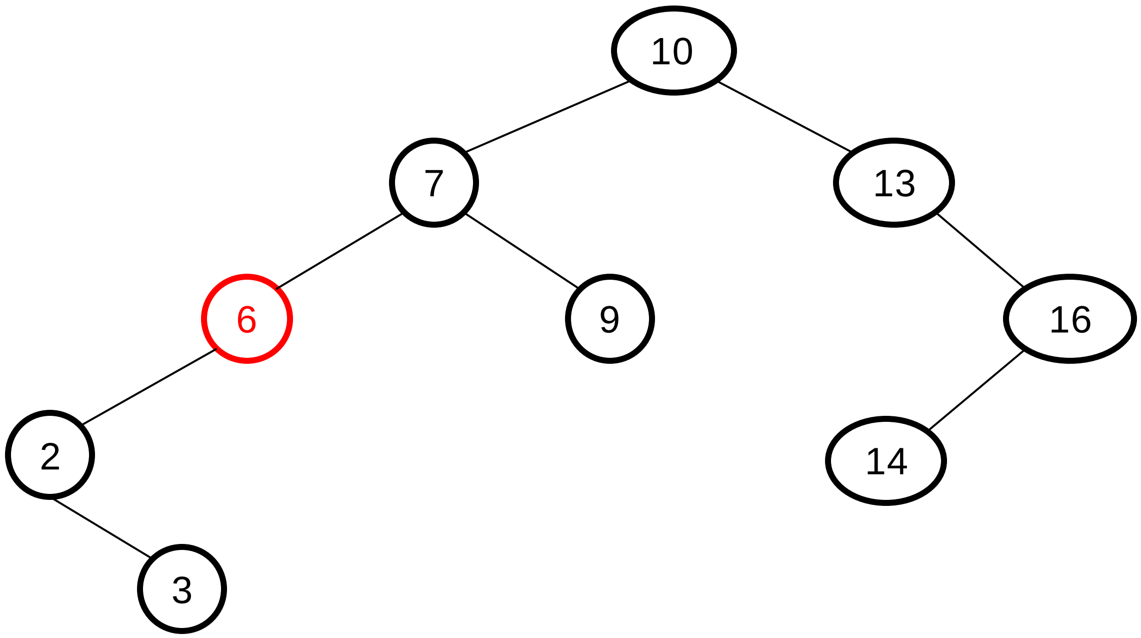
succ(5) = 6

Remove 5

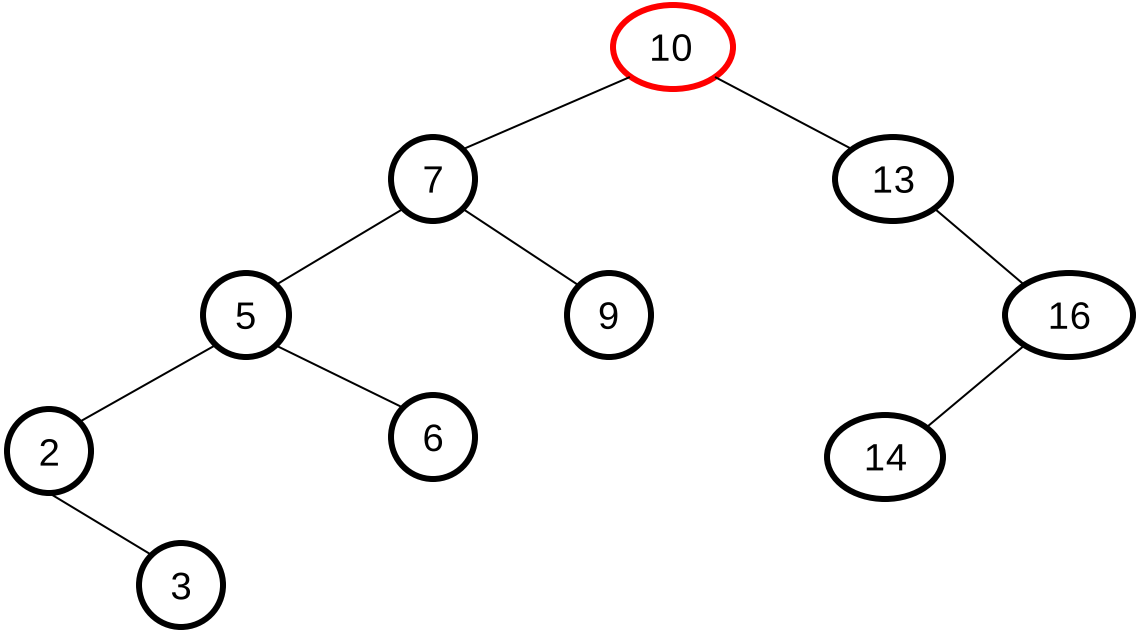


$succ(5) = 6$

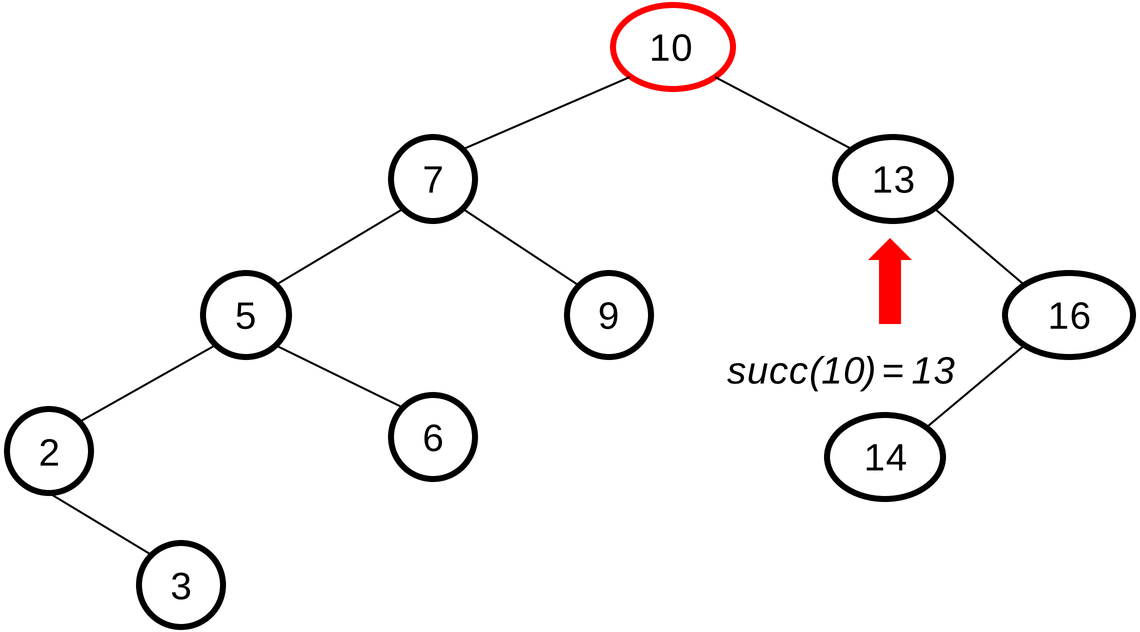
Remove 5



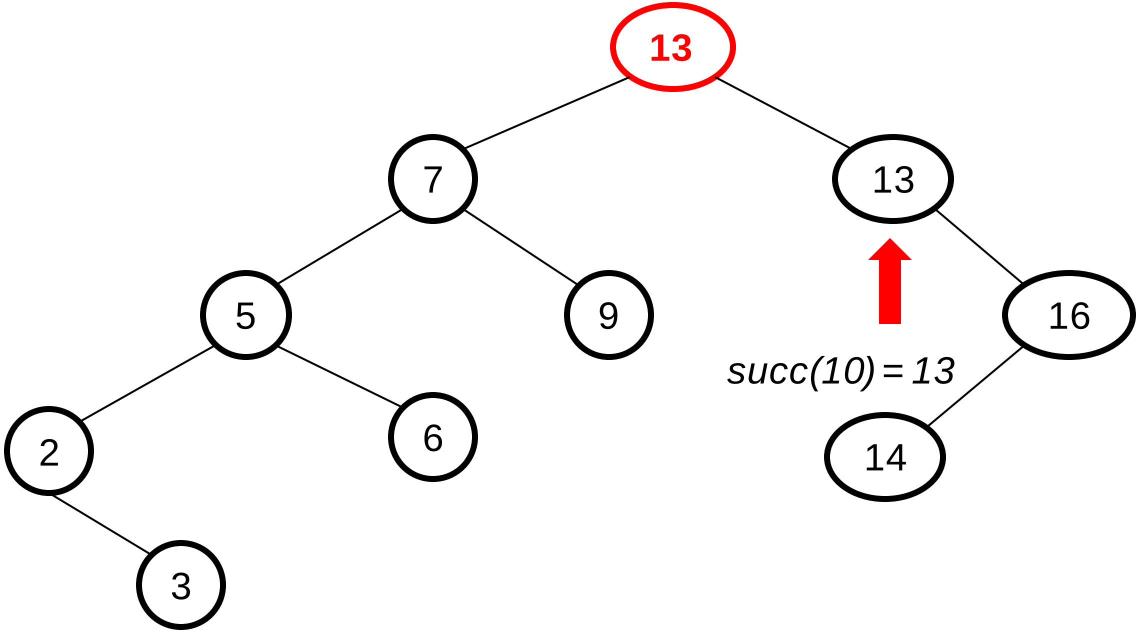
Remove 10



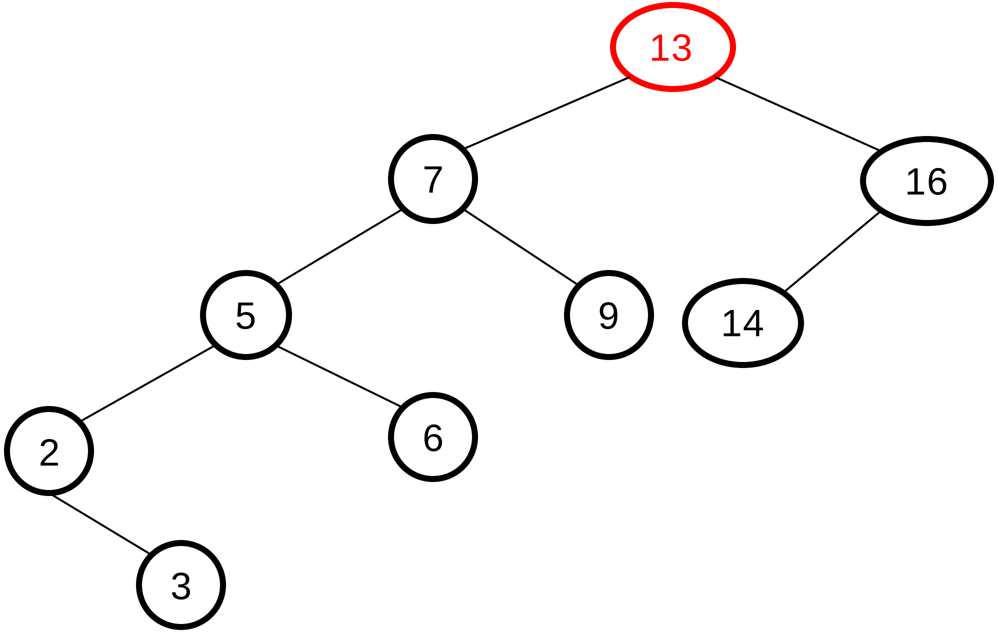
Remove 10



Remove 10



Remove 10



Sanity Check – Is Recursive Remove Safe?

- If we remove a node with two subtrees…
- Its successor is leftmost node of its right subtree.
- Leftmost node has no left subtree.
- Hence, “recursive” remove always removes node with 0 or 1 subtrees – easy cases!

Worst-Case Cost of Operations

- Find – might have to walk from root to deepest leaf of tree
- Min/Max – same
- Insert – same
- Iterate – might have to walk from root to deepest leaf *or* vice versa
- Remove – might have to walk from root to deepest leaf

Worst-Case Costs for BST Operations

- Find – Θ(h) for tree of height h
- Min/Max – Θ(h) for tree of height h
- Insert – Θ(h) for tree of height h
- Iterate – Θ(h) for tree of height h
- Remove – Θ(h) for tree of height h

Worst-Case Costs for BST Operations

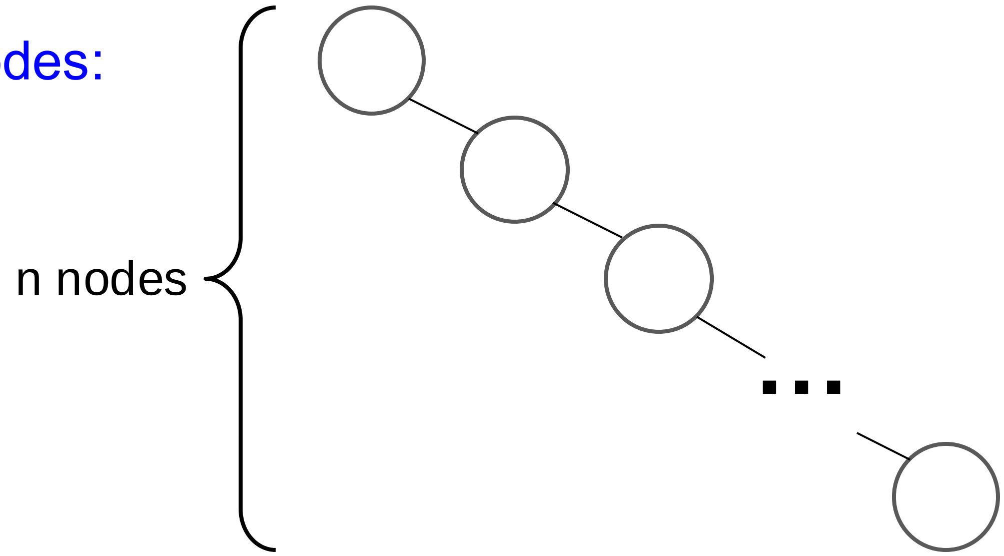
- Find – Θ
- Min/Max
- Insert – Θ
- Iterate –
- Remove – Θ(h) for tree of height h

Are these costs sublinear in n, the # of nodes in the tree? *Depends how # nodes relates to height.*

How Tall Can a BST with n Nodes Be?

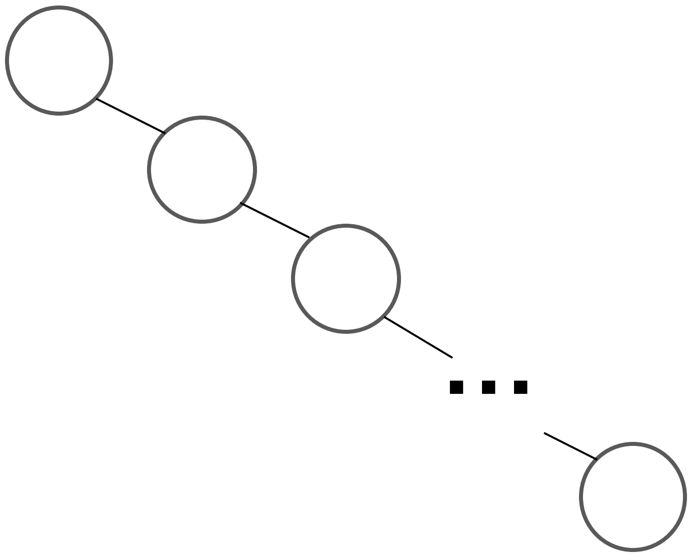
- Here’s a binary tree with n nodes:

- This tree has height **???**.



How Tall Can a BST with n Nodes Be?

- Here’s a binary tree with n nodes:

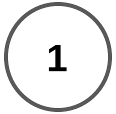


- This tree has height **n-1**.

- **Can we realize this tree as a BST by some sequence of insertions?**

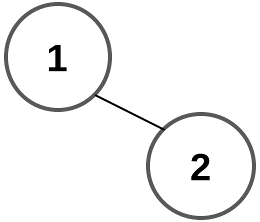
How Tall Can a BST with n Nodes Be?

- Insert keys 1..n in order



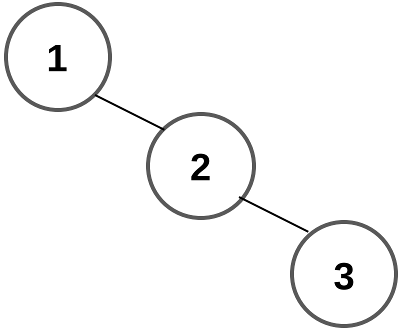
How Tall Can a BST with n Nodes Be?

- Insert keys 1..n in order



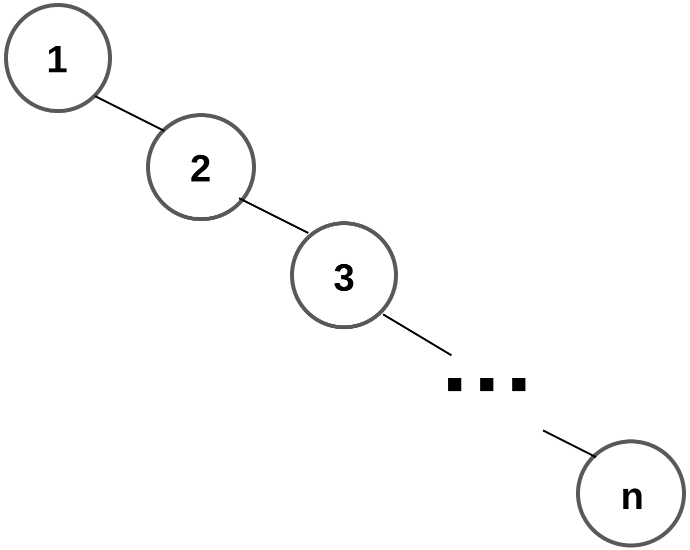
How Tall Can a BST with n Nodes Be?

- Insert keys 1..n in order



How Tall Can a BST with n Nodes Be?

- Insert keys 1..n in order



Bad News…

- Given the right sequence of insertions, a BST with n nodes can have height $\Theta(n)$

- That means that all our BST operations are worst-case $\Theta(n)$

- This is no better in the worst case than a list or array. In fact, it's *worse* for some operations (e.g. min/max).

Can We Overcome Worst-Case $\Theta(n)$ Costs for Tree Operations?

What If Our Trees Were Never Too Tall?

- Defn: a binary tree with n nodes is said to be balanced if it has height O(log n).

- *Example:* a complete binary tree with 2^n-1 nodes has height n – 1, so is balanced.

- *In a balanced BST, all BST ops are worst case O(log n).*

What If Our Trees Were Never Too Tall?

- Defn: a bina… …o be balanced if it has height …
- *Example:* a… … nodes has height n – …
- *In a balanc… …t case O(log n).*

Really, we can write Θ(log n) here – *all* binary trees have height Ω(log n).

Strategy for Balancing Trees

1. Define a structural property P that applies to only *some* BSTs

2. Prove that BSTs satisfying property P are balanced

3. Make sure a trivial BST (one node) satisfies P

4. Show how to insert, remove while maintaining P
    - i.e. show that P is an *invariant* of the BST

An Example of a Balance Property

- **AVL** Property

- Described 1962 by **A**delson-**V**elsky and **L**andis

- A tree T satisfies the AVL property if for each node in T, its *left and right subtrees differ in height by at most 1.*

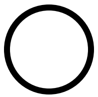
- Intuitively, prevents very lopsided trees.

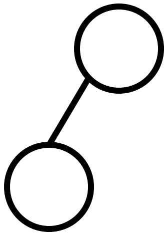
AVL Property for Binary Trees – Formal Defn

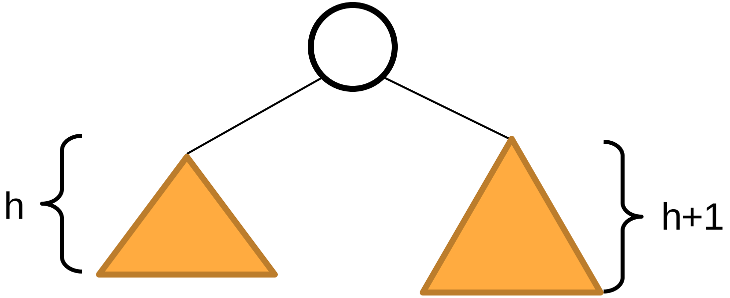
- Let H(r) be the height of a binary tree rooted at r

- Defn: T is an **AVL tree** iff, for every node x in T, *one of these is true*:

1. x is a leaf.
2. x has one child, which is a leaf.
3. x has two children, and |H(x.right) – H(x.left)| ≤ 1.





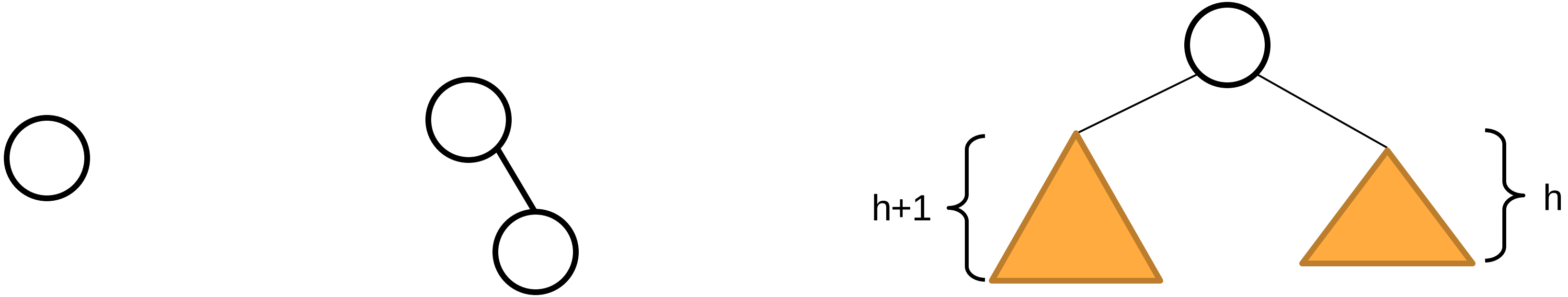


AVL Property for Binary Trees – Formal Defn

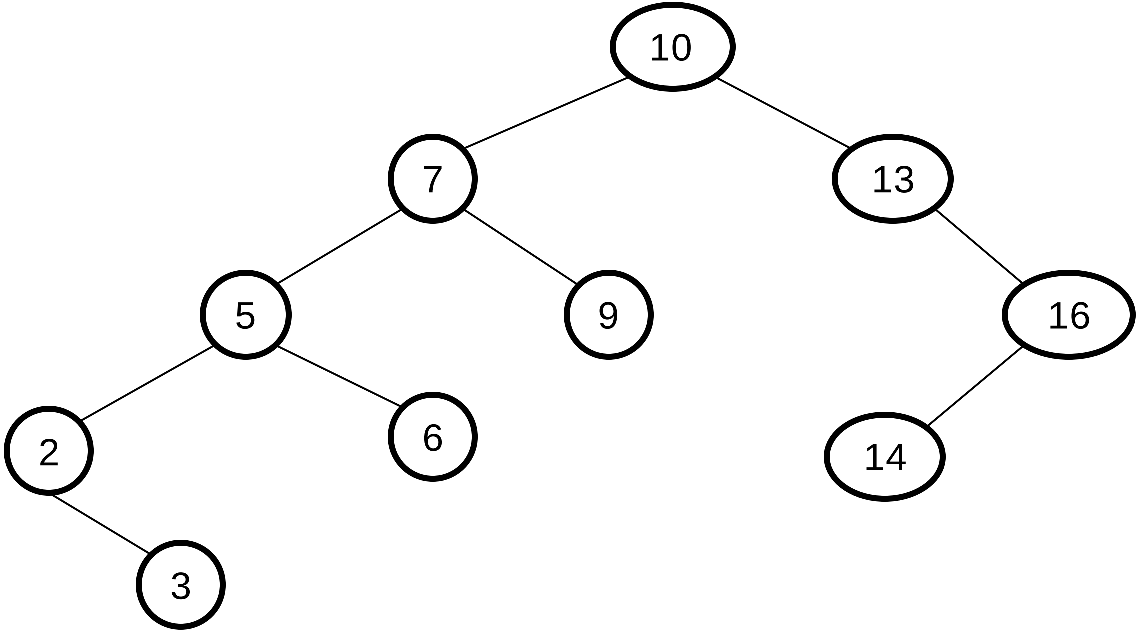
- Let $H(r)$ be the height of a binary tree rooted at r

- Defn: T is an **AVL tree** iff, for every node x in T, *one of these is true*:

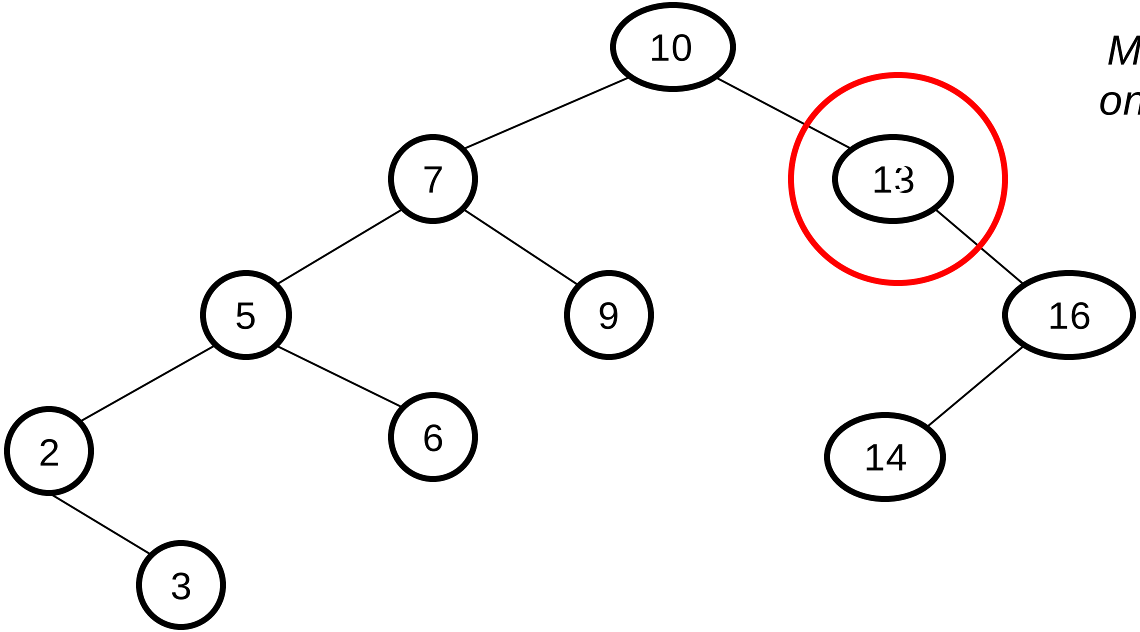
1. x is a leaf.
2. x has one child, which is a leaf.
3. x has two children, and $|H(x.right) - H(x.left)| \leq 1$.



Is This an AVL Tree?

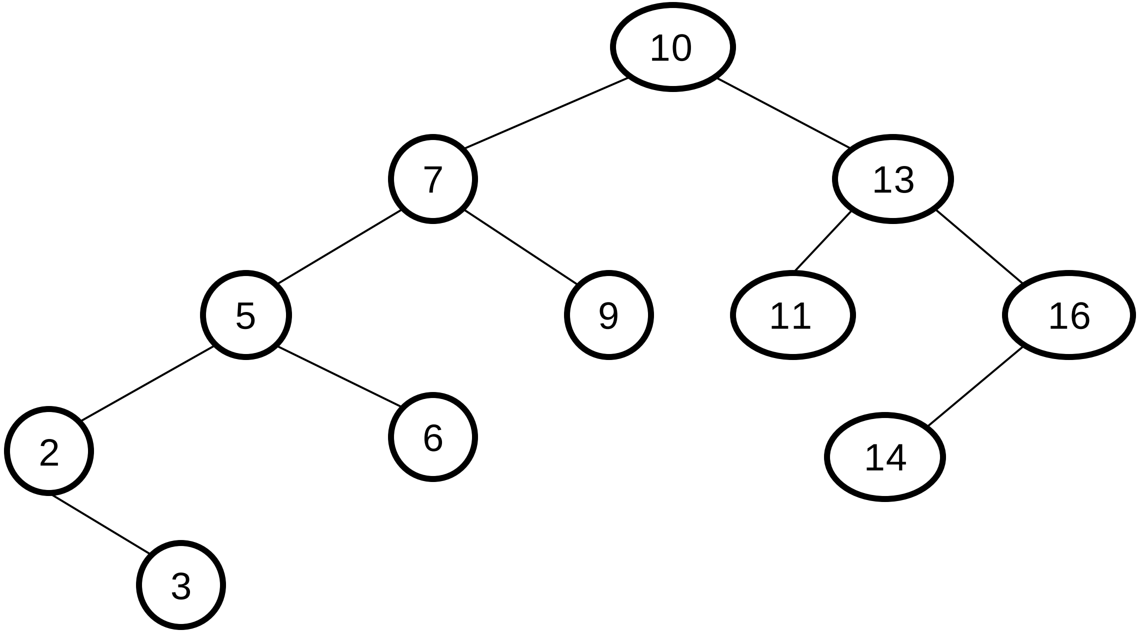


Is This an AVL Tree? NO!



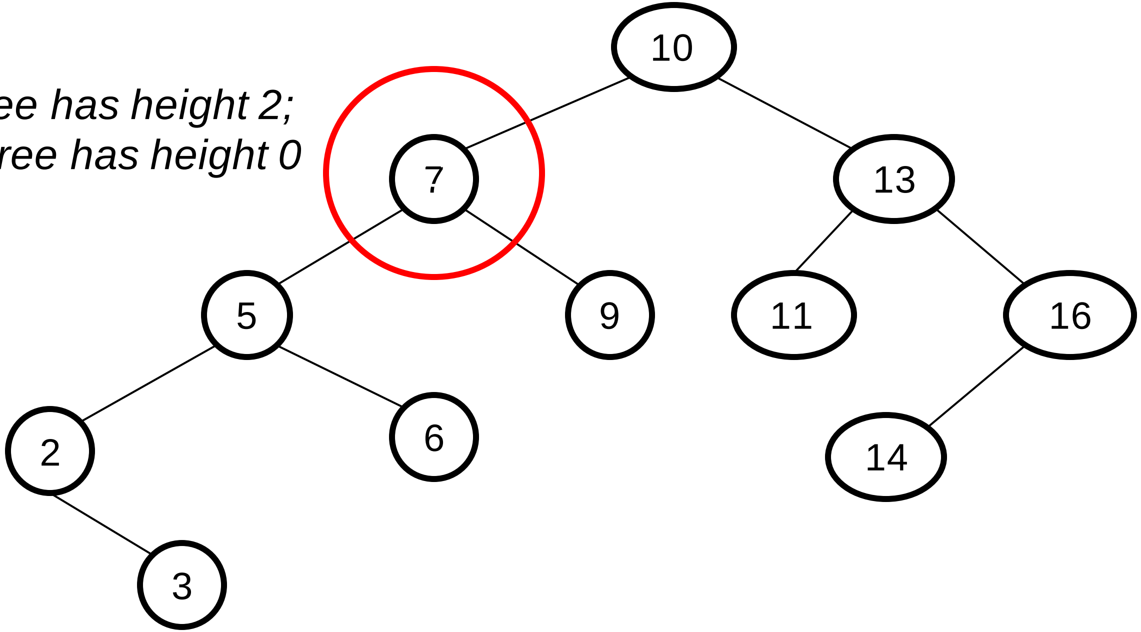
May not have a node with one child that is not a leaf.

Is This an AVL Tree?

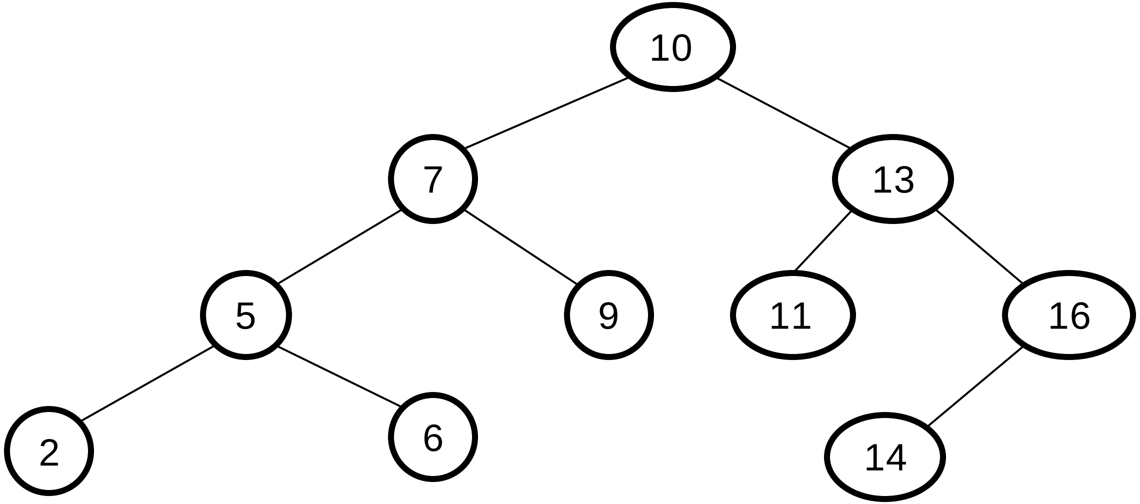


Is This an AVL Tree? NO!

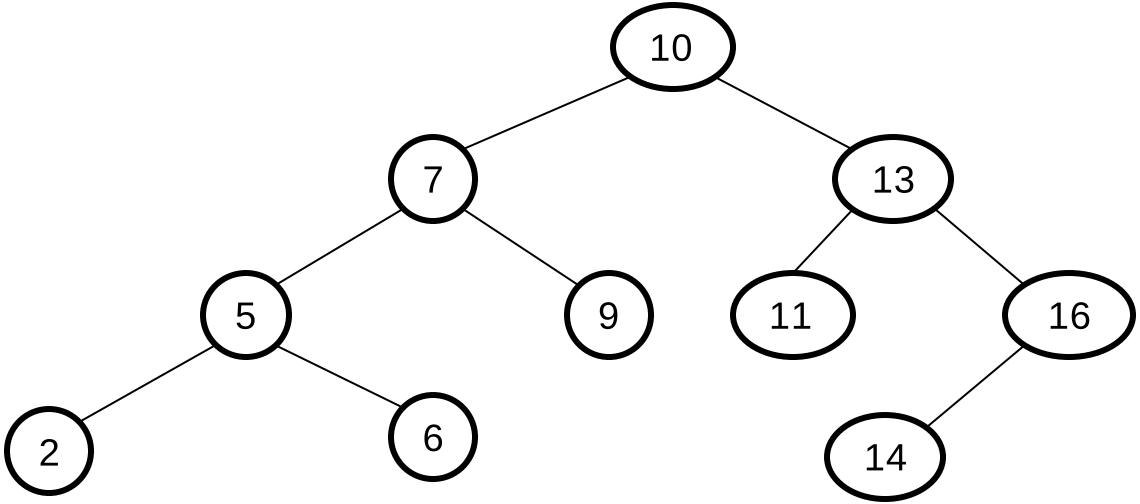
Left subtree has height 2;
Right subtree has height 0



Is This an AVL Tree?

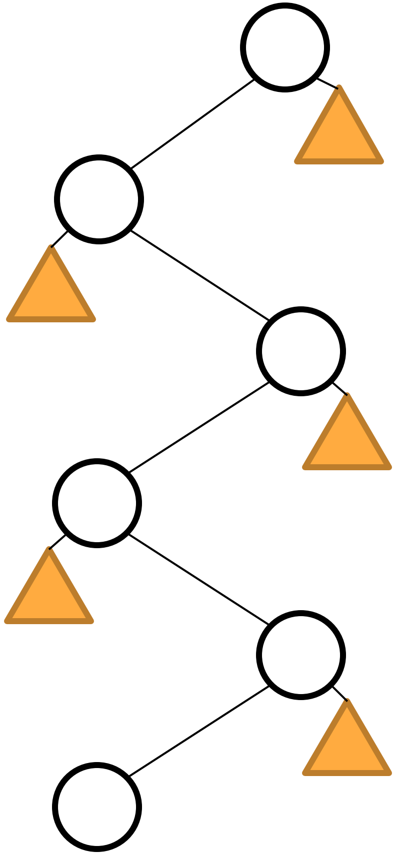


Is This an AVL Tree? YES!



Why Are AVL Trees Balanced?

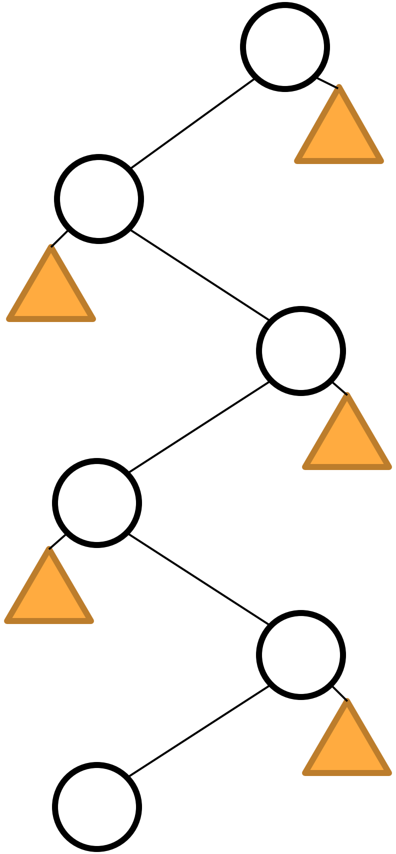
- Intuitively, a tall tree with few nodes is "skinny"
- Long path to its deepest leaf cannot have many nodes branching off it.
- Skinny trees have subtrees with very different heights
- AVL property prevents skinny trees



Why Are AVL Trees Balanced?

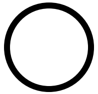
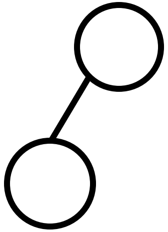
- Intuitively, a tall tree with few nodes is “skinny”
- Long path to have many r
- Skinny trees different heig
- AVL property prevents skinny trees

Let’s formalize this idea to prove that an AVL tree is balanced.



What is “Skinniest” AVL Tree We Can Build?

- Let N(h) be minimum # of nodes in any AVL tree with height h.

- N(0) = 1  N(1) = 2 

- *Can we find a formula for N(h) for h > 1?*

What is “Skinniest” AVL Tree We Can Build?

- If tree has height h, root’s tallest subtree has height ???.

What is “Skinniest” AVL Tree We Can Build?

- If tree has height h, root’s tallest subtree has height h-1.
- By AVL property, other subtree must have height ≥ ???.

What is “Skinniest” AVL Tree We Can Build?

- If tree has height h, root’s tallest subtree has height h-1.
- By AVL property, other subtree must have height ≥ h-2.
- Both subtrees are also AVL trees.
- Hence, $N(h) = N(h-1) + N(h-2) + 1$ ⟵ 2 subtrees, plus 1 node for root.

What is "Skinniest" AVL Tree We Can Build?

- If tree has he has height h-1.
- By AVL prop height ≥ h-2.
- Both subtree
- Hence, $N(h) = N(h-1) + N(h-2) + 1$

Let's guess a solution to recurrence for N(h) and check our guess.

Lower Bound on AVL Tree Size vs Height

- Let $\Phi = \frac{\sqrt{5}+1}{2} \approx 1.618$. [*Yes, the golden ratio again]*

- **Claim**: $N(h) \geq \Phi^h$

- → *Every* AVL tree with height h has $\geq \Phi^h$ nodes

- → *Every AVL tree with n nodes has height $\leq \log_\Phi(n)$, hence is balanced.*

Lower Bound Proof, 1/2

- Claim: $N(h) \geq \Phi^h$
- Pf: by induction on h
- Base 1: $N(0) = 1 \geq \Phi^0$
- Base 2: $N(1) = 2 \geq \Phi^1$

Lower Bound Proof, 2/2

- Ind: $N(h) = N(h-1) + N(h-2) + 1$
- $\geq N(h-1) + N(h-2)$
- $\geq \Phi^{h-1} + \Phi^{h-2}$
- $= \Phi^{h-2} (\Phi + 1)$

Apply inductive hypothesis.

Lower Bound Proof, 2/2

- Ind: $N(h) = N(h-1) + N(h-2) + 1$
- $\geq N(h-1) + N(h-2)$
- $\geq \Phi^{h-1} + \Phi^{h-2}$
- $= \Phi^{h-2} (\Phi + 1)$

Fact:

$\Phi^2 = \Phi+1$

Lower Bound Proof, 2/2

- Ind: $N(h) = N(h-1) + N(h-2) + 1$
- $\geq N(h-1) + N(h-2)$
- $\geq \Phi^{h-1} + \Phi^{h-2}$
- $= \Phi^{h-2} (\Phi + 1)$
- $= \Phi^{h-2}\, \Phi^2$
- $= \Phi^h$. QED

Fact:
$\Phi^2 = \Phi+1$

Next Time

How can we modify BST insertion and deletion to ensure that the trees they create are always AVL trees?