CSE 332: Data Abstractions
Binary Search Trees

Richard Anderson

Spring 2016

Announcements

Fun with sums

$$\sum_{i=1}^{\infty} \frac{i}{2^i} = \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \cdots$$

$$= (\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots) + (\frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots) + (\frac{1}{8} + \frac{1}{16} + \cdots) + \cdots$$

$$= 1 + \frac{1}{2} + \frac{1}{4} + \cdots$$

$$= 2$$

ADTs Seen So Far

- **Stack**
  - Push
  - Pop
- **Queue**
  - Enqueue
  - Dequeue
- **Priority Queue**
  - Insert
  - DeleteMin

None of these support “find”

The Dictionary ADT

- Data:
  - a set of (key, value) pairs
- Operations:
  - Insert (key, value)
  - Find (key)
  - Remove (key)

insert(seitz, ….)

find(anderson)

- anderson Richard, Anderson,…

- seitz
  Steve
  Seitz
  CSE 592
- anderson
  Richard
  Anderson
  CSE 582
- kainby87
  HyeIn
  Kim
  CSE 220
- …

*The Dictionary ADT is also called the "**Map ADT**"*

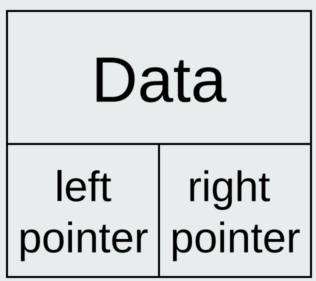
Implementations

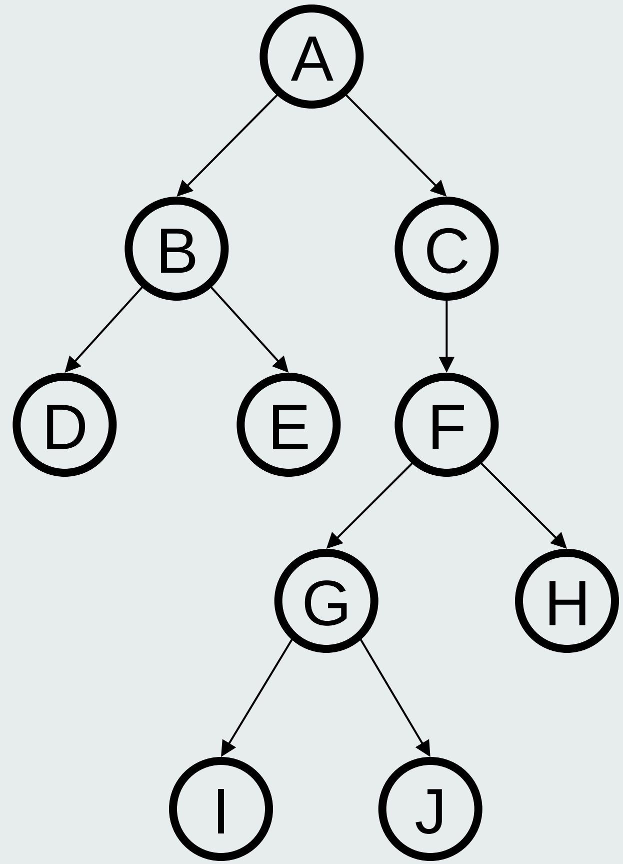
	insert	find	delete
• Unsorted Linked-list			
• Unsorted array			
• Sorted array			

Binary Trees

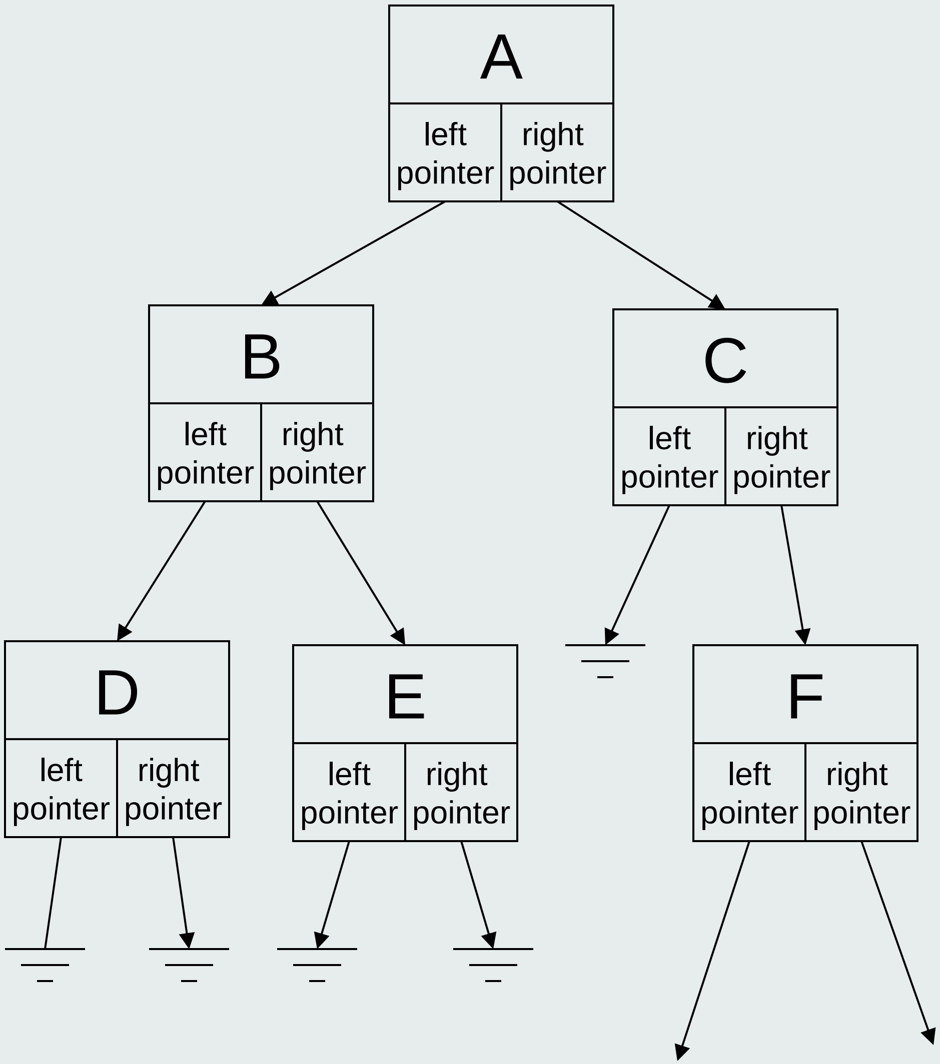
- Binary tree is
  - a root
  - left subtree *(maybe empty)*
  - right subtree *(maybe empty)*

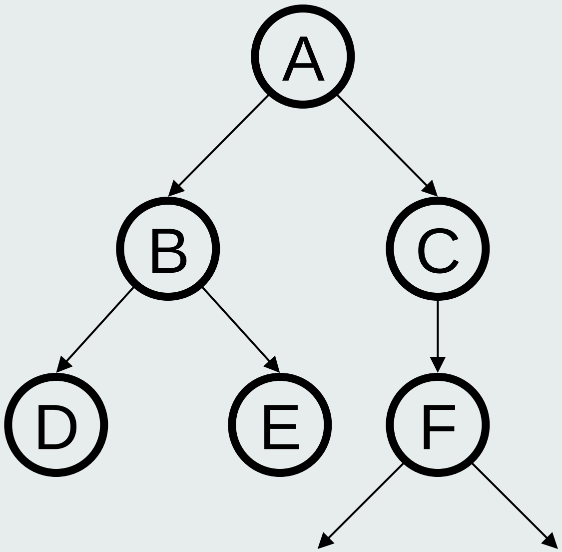
- Representation:





Binary Tree: Representation



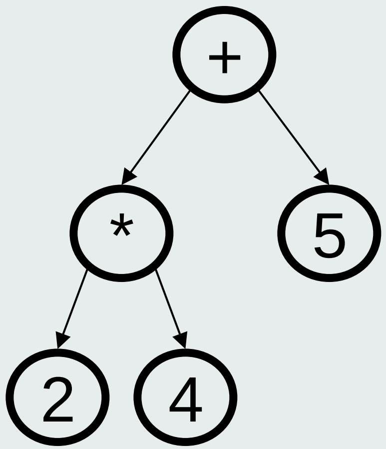


Tree Traversals

A *traversal* is an order for visiting all the nodes of a tree

Three types:

- Pre-order: Root, left subtree, right subtree
- In-order: Left subtree, root, right subtree
- Post-order: Left subtree, right subtree, root

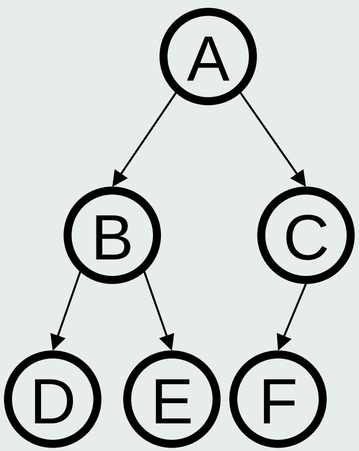


(an expression tree)

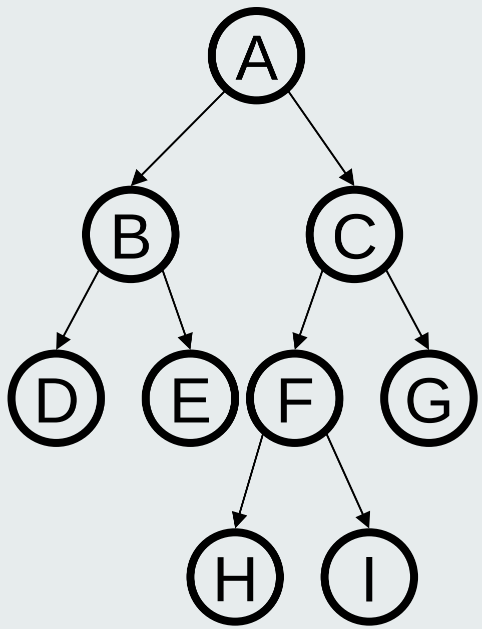
Inorder Traversal

```
void traverse(BNode t){
  if (t != NULL)
    traverse (t.left);
    process t.element;
    traverse (t.right);
  }
}
```

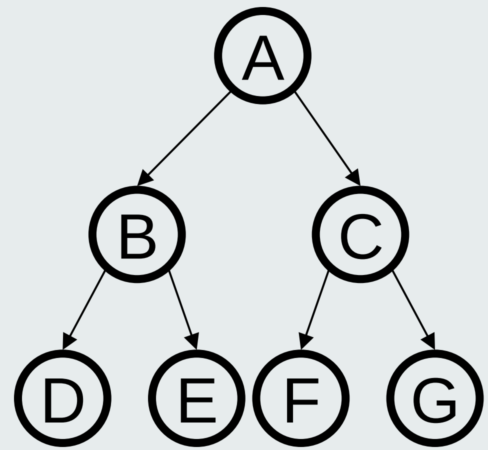
Binary Tree: Special Cases



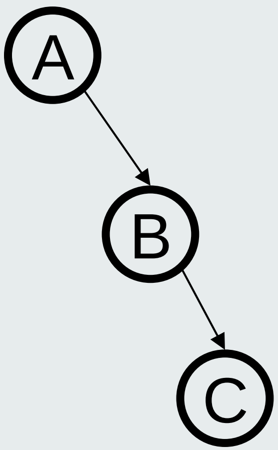
Complete Tree



Full Tree



Perfect Tree



“List” Tree

Binary Tree: Some Numbers…

Recall: height of a tree = longest path from root to leaf.

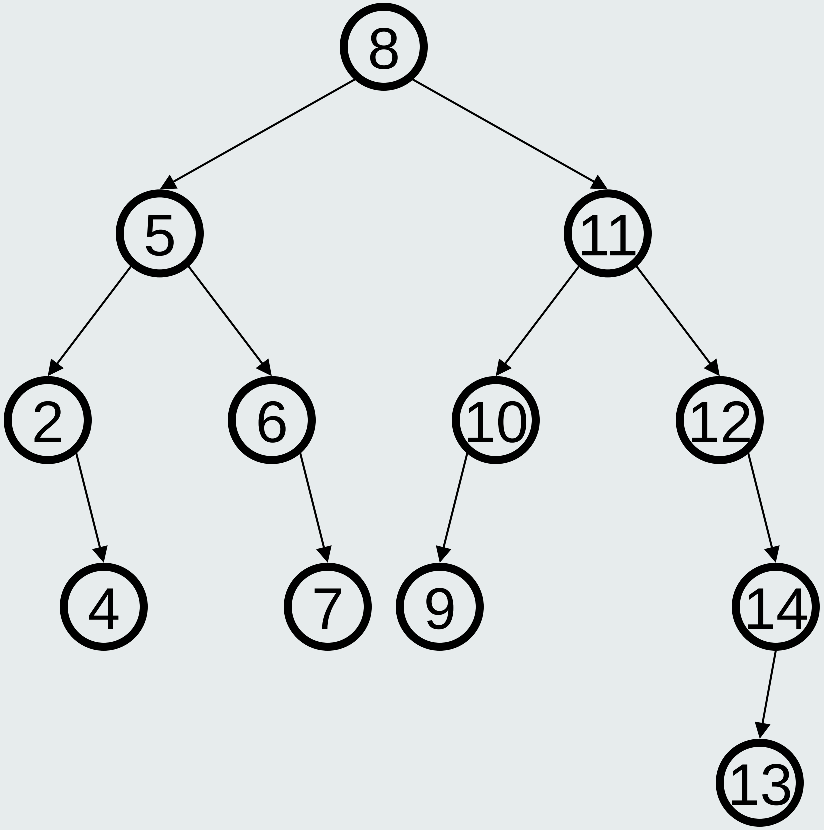
For binary tree of height *h*:

- max # of leaves:
- max # of nodes:
- min # of leaves:
- min # of nodes:

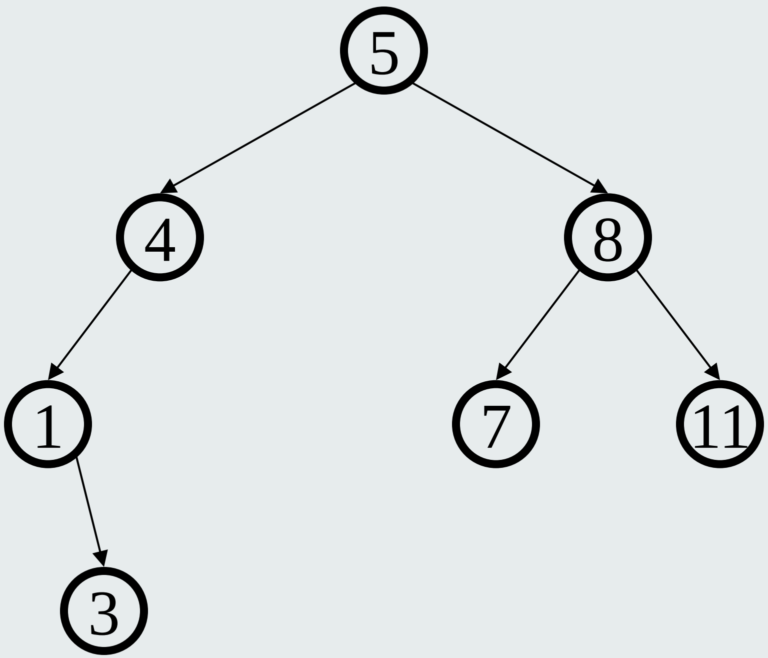
Binary Search Tree Data Structure

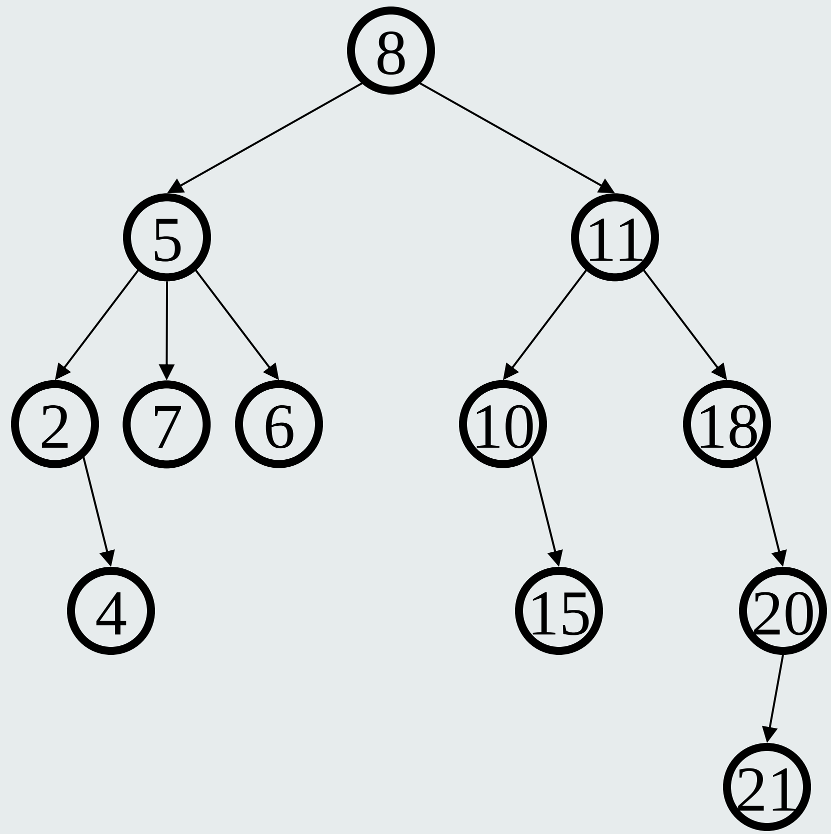
- Structural property
  – each node has ≤ 2 children

- Order property
  – all keys in left subtree smaller than root's key
  – all keys in right subtree larger than root's key



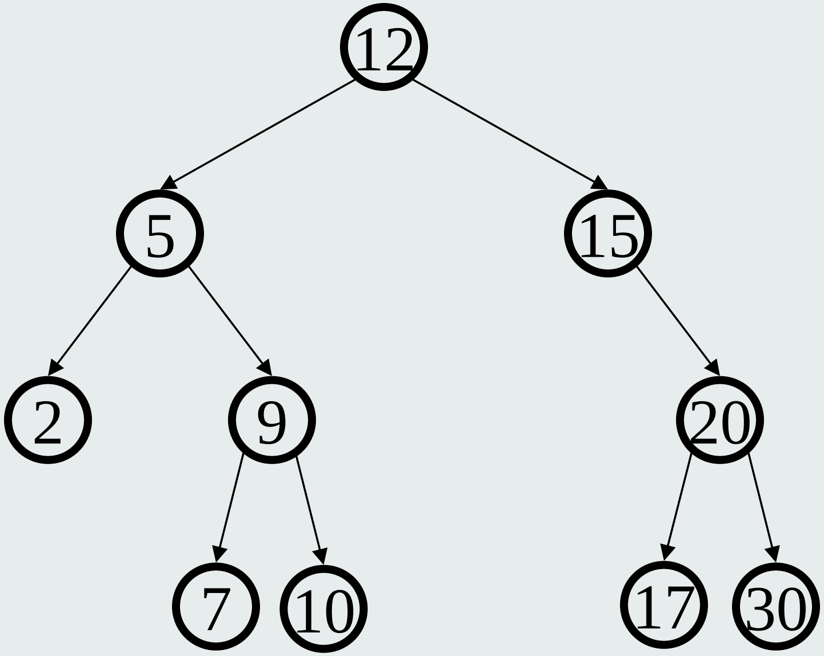
Example and Counter-Example





BINARY SEARCH TREES?

Find in BST, Recursive




```
Node Find(Object key,
                Node root) {
  if (root == NULL)
    return NULL;

  if (key < root.key)
    return Find(key,
                    root.left);
  else if (key > root.key)
    return Find(key,
                    root.right);
  else
    return root;
}
```

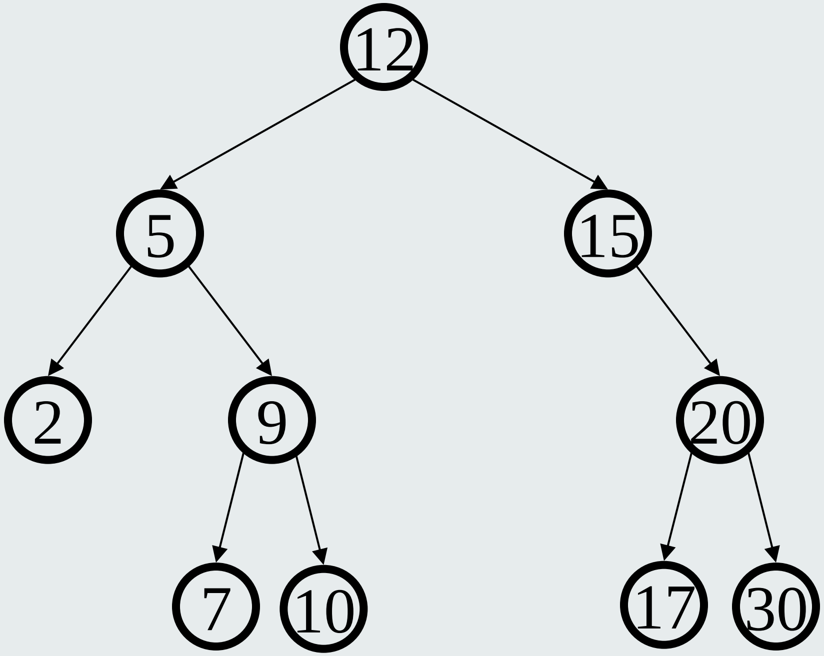
Runtime:

Find in BST, Iterative

```
Node Find(Object key,
            Node root) {

  while (root != NULL &&
         root.key != key) {
    if (key < root.key)
      root = root.left;
    else
      root = root.right;
  }

  return root;
}
```

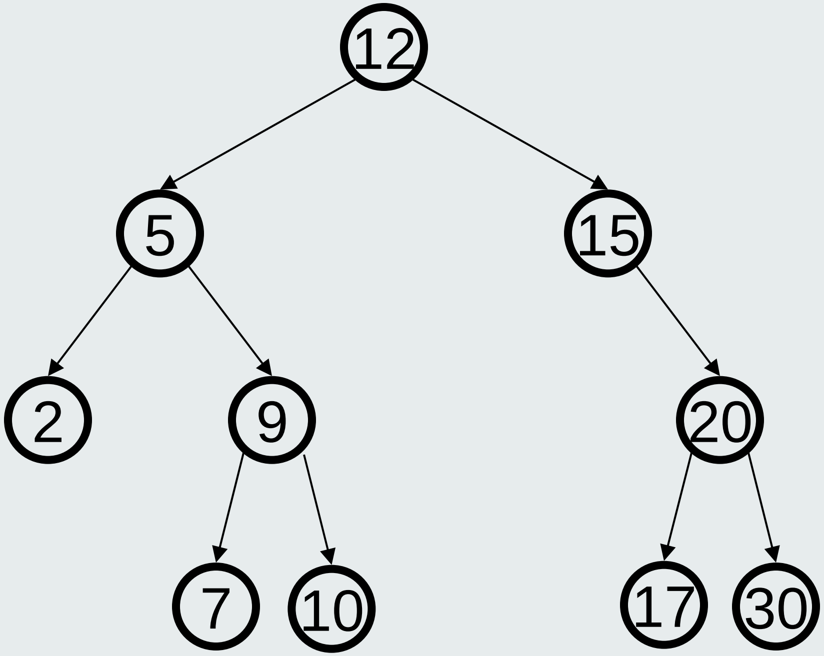


Runtime:

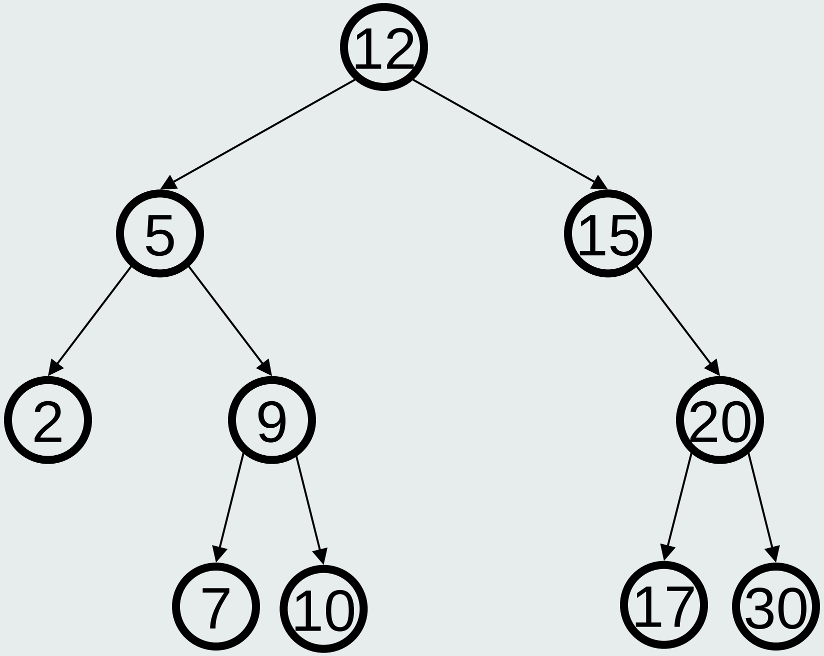
Bonus: FindMin/FindMax

- Find minimum

- Find maximum



Insert in BST



Insert(13)
Insert(8)
Insert(31)

Insertions happen only at the leaves – easy!

Runtime:

BuildTree for BST

- Suppose keys 1, 2, 3, 4, 5, 6, 7, 8, 9 are inserted into an initially empty BST.

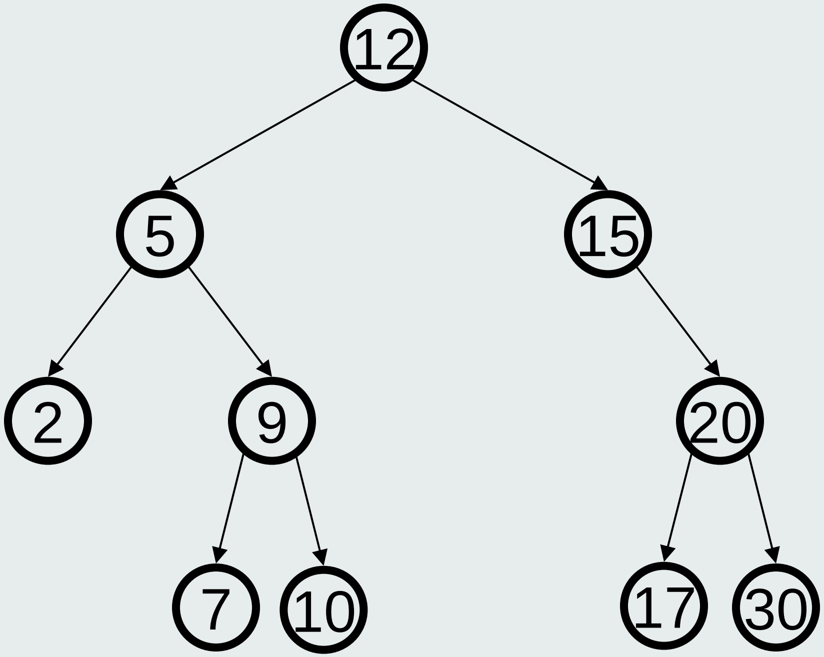
If inserted in given order, what is the tree? What big-O runtime for this kind of sorted input?

If inserted in reverse order, what is the tree? What big-O runtime for this kind of sorted input?

BuildTree for BST

- Suppose keys 1, 2, 3, 4, 5, 6, 7, 8, 9 are inserted into an initially empty BST.
  - If inserted median first, then left median, right median, etc., what is the tree? What is the big-O runtime for this kind of sorted input?

Deletion in BST



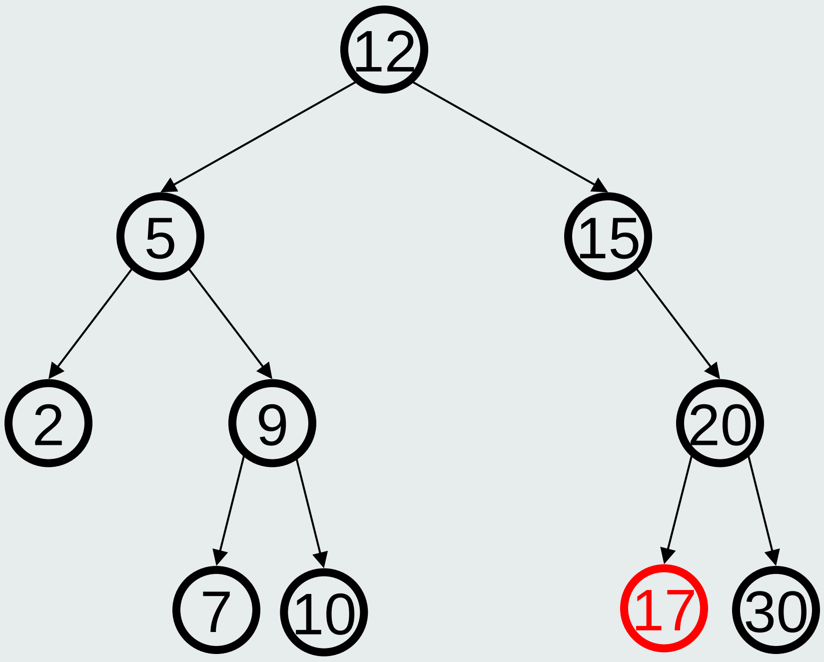
Why might deletion be harder than insertion?

Deletion

- Removing an item disrupts the tree structure.
- Basic idea: find the node that is to be removed. Then “fix” the tree so that it is still a binary search tree.
- Three cases:
  - node has no children (leaf node)
  - node has one child
  - node has two children

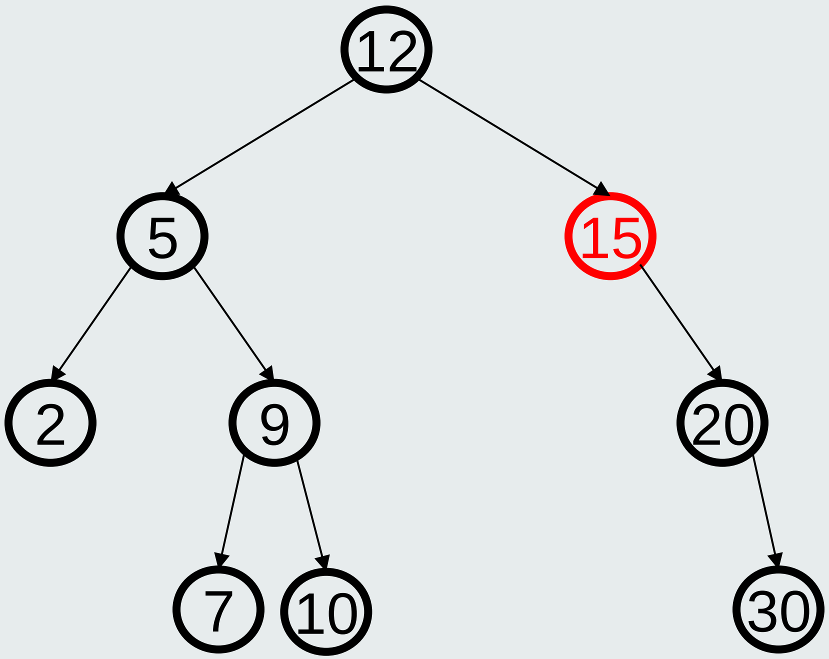
Deletion – The Leaf Case

Delete(17)



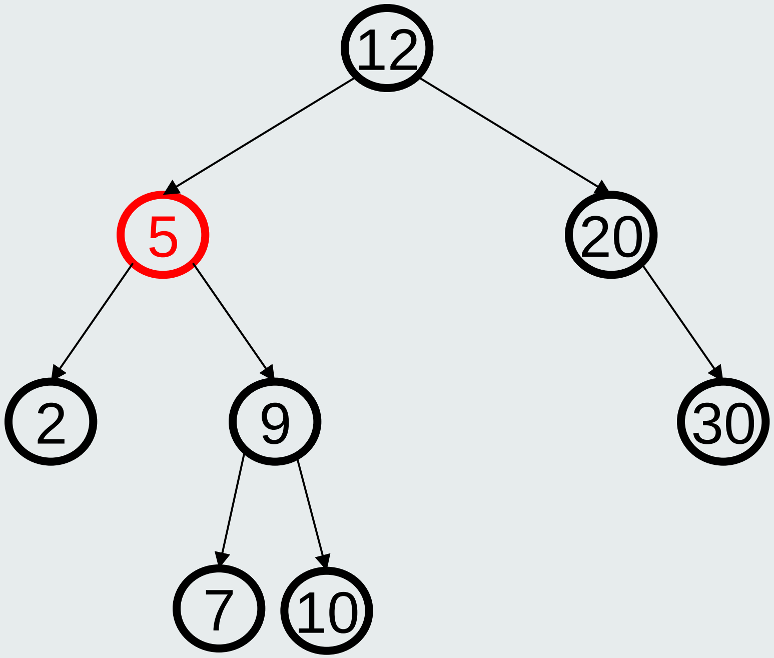
Deletion – The One Child Case

Delete(15)



Deletion: The Two Child Case

Delete(5)



What can we replace 5 with?

Deletion – The Two Child Case

Idea: Replace the deleted node with a value *between* the two child subtrees

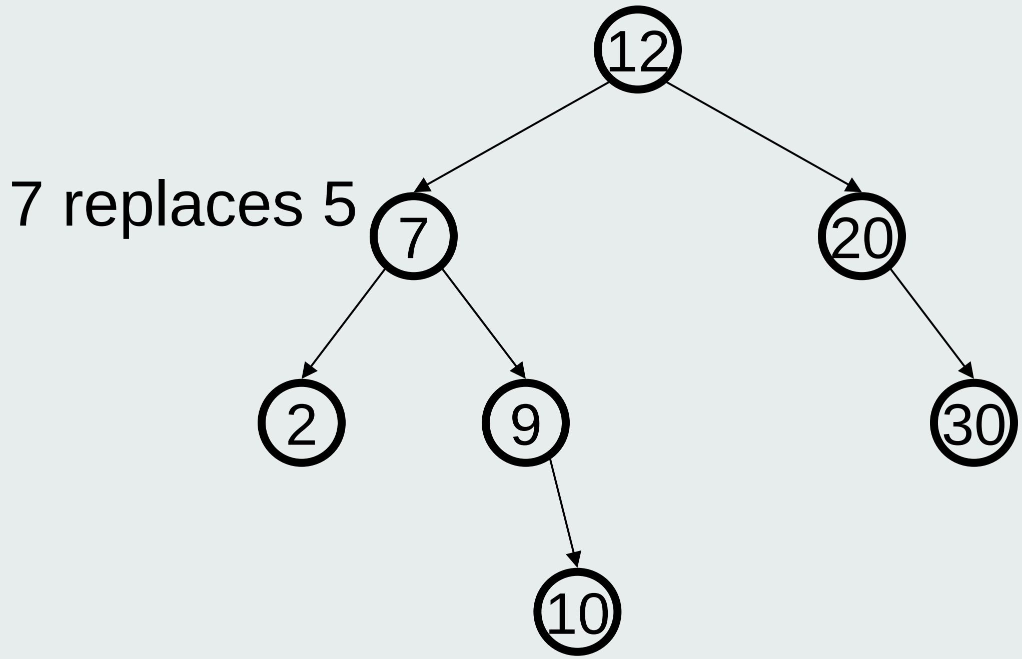
Options:

- *succ* from right subtree: findMin(t.right)
- *pred* from left subtree: findMax(t.left)

Now delete the original node containing *succ* or *pred*

- Leaf or one child case – easy!

Finally…



Original node containing
7 gets deleted

Balanced BST

Observations

- BST: the shallower the better!
- For a BST with n nodes
  - Average depth (averaged over all possible insertion orderings) is O(log n)
  - Worst case maximum depth is O(n)
- Simple cases such as insert(1, 2, 3, ..., n) lead to the worst case scenario

Solution: Require a **Balance Condition** that

1. ensures depth is O(log n) – strong enough!
2. is easy to maintain – not too strong!