CSE 326: Data Structures
Binary Search Trees

James Fogarty

Autumn 2007

Lecture 8

Today's Outline

- Dictionary ADT / Search ADT
- Quick Tree Review
- Binary Search Trees

ADTs Seen So Far

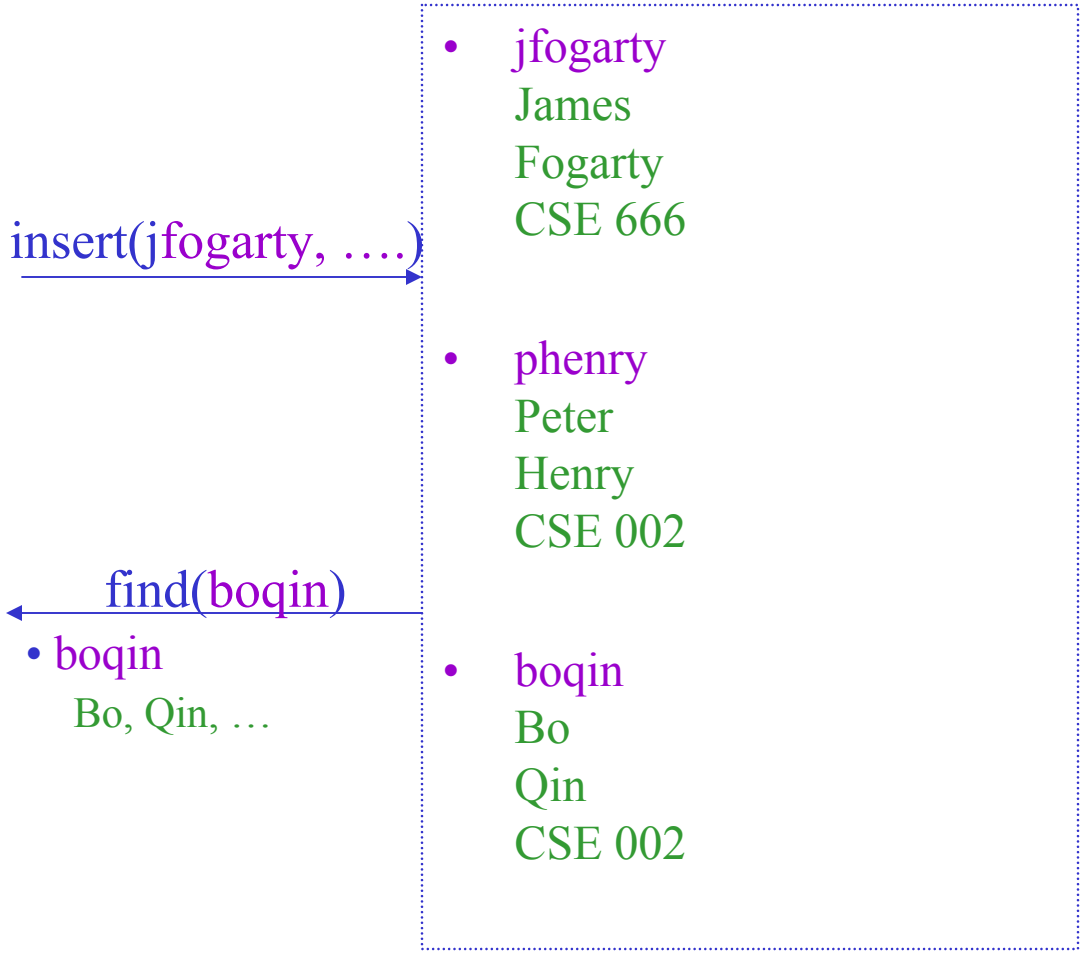
- Stack
  - Push
  - Pop
- Queue
  - Enqueue
  - Dequeue
- Priority Queue
  - Insert
  - DeleteMin

Then there is decreaseKey…

Need *pointer*! Why?
Because *find* not efficient.

The Dictionary ADT

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



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

A Modest Few Uses

- Sets
- Dictionaries
- Networks : Router tables
- Operating systems : Page tables
- Compilers : Symbol tables

Probably the most widely used ADT!

Implementations

	insert	find	delete
• Unsorted Linked-list	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
• Unsorted array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
• Sorted array	$\log n + n$	$\Theta(\log n)$	$\log n + n$

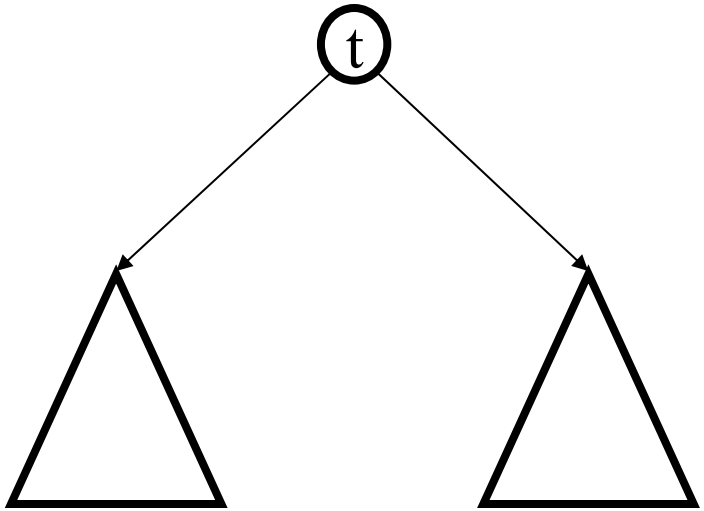
SO CLOSE!

What limits the performance?

Time to move elements, can we mimic BinSearch with BST?

Tree Calculations

Recall: height is max number of edges from root to a leaf



Find the height of the tree...

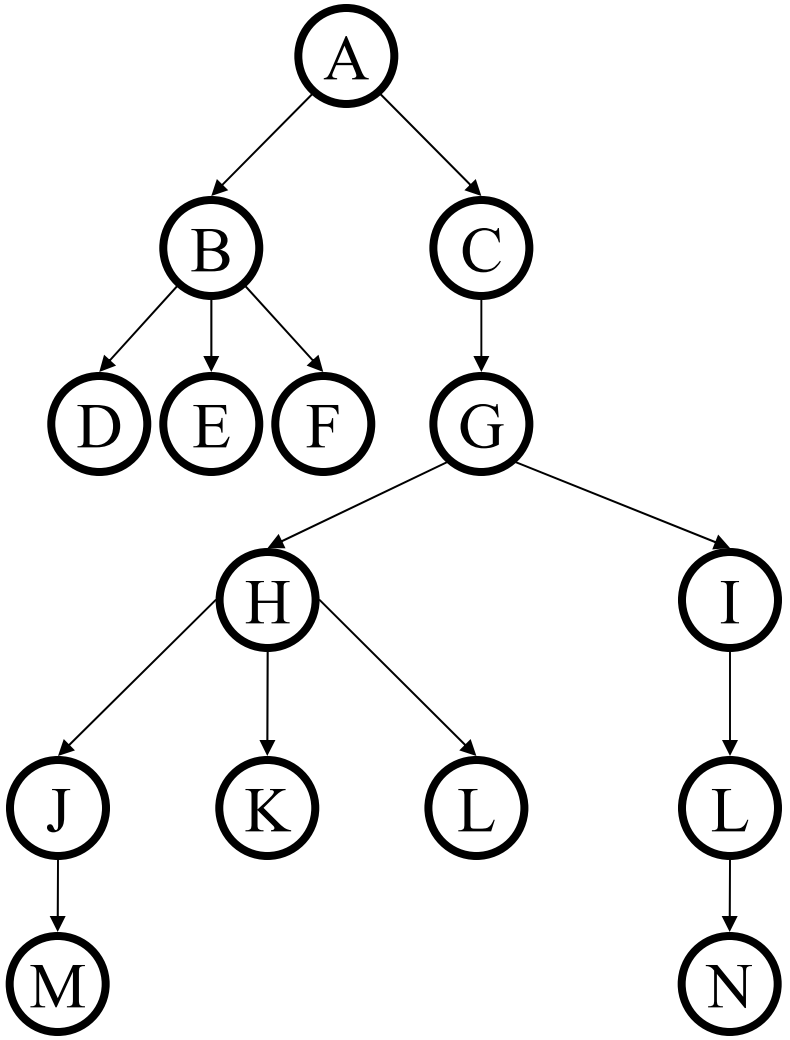
height(t) = 1 + max {height(t.left), height(t.right)}

runtime:

Θ(N) (constant time for each node; each node visited twice)

Tree Calculations Example

How high is this tree?



height(B) = 1
height(C) = 4

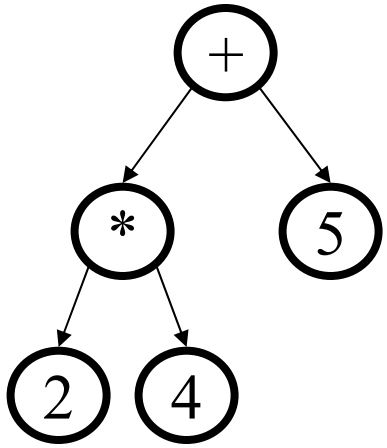
so height(A) = 5

More Recursive Tree Calculations: 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

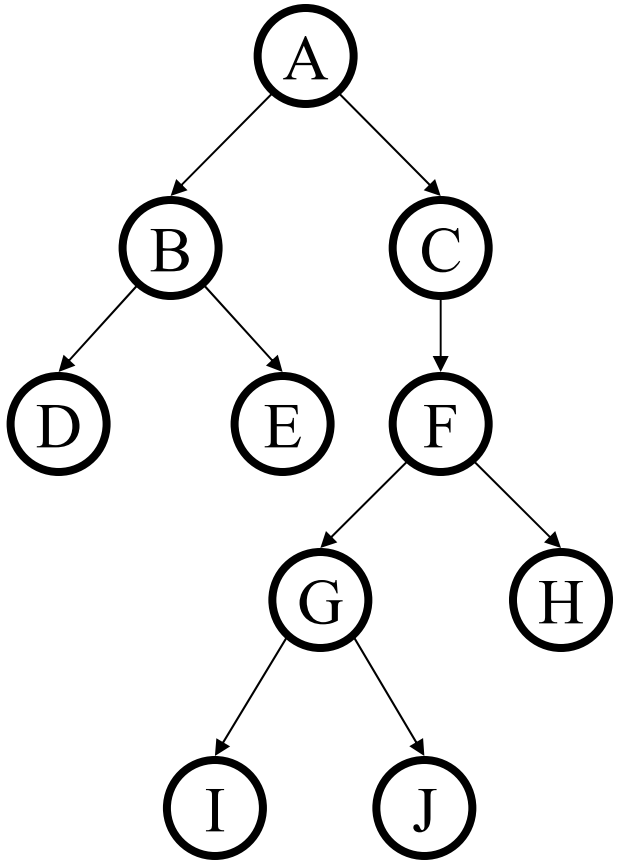
```
void traverse(BNode t){
  if (t != NULL)
    traverse (t.left);
    process t.element;
    traverse (t.right);
  }
}
```

Binary Trees

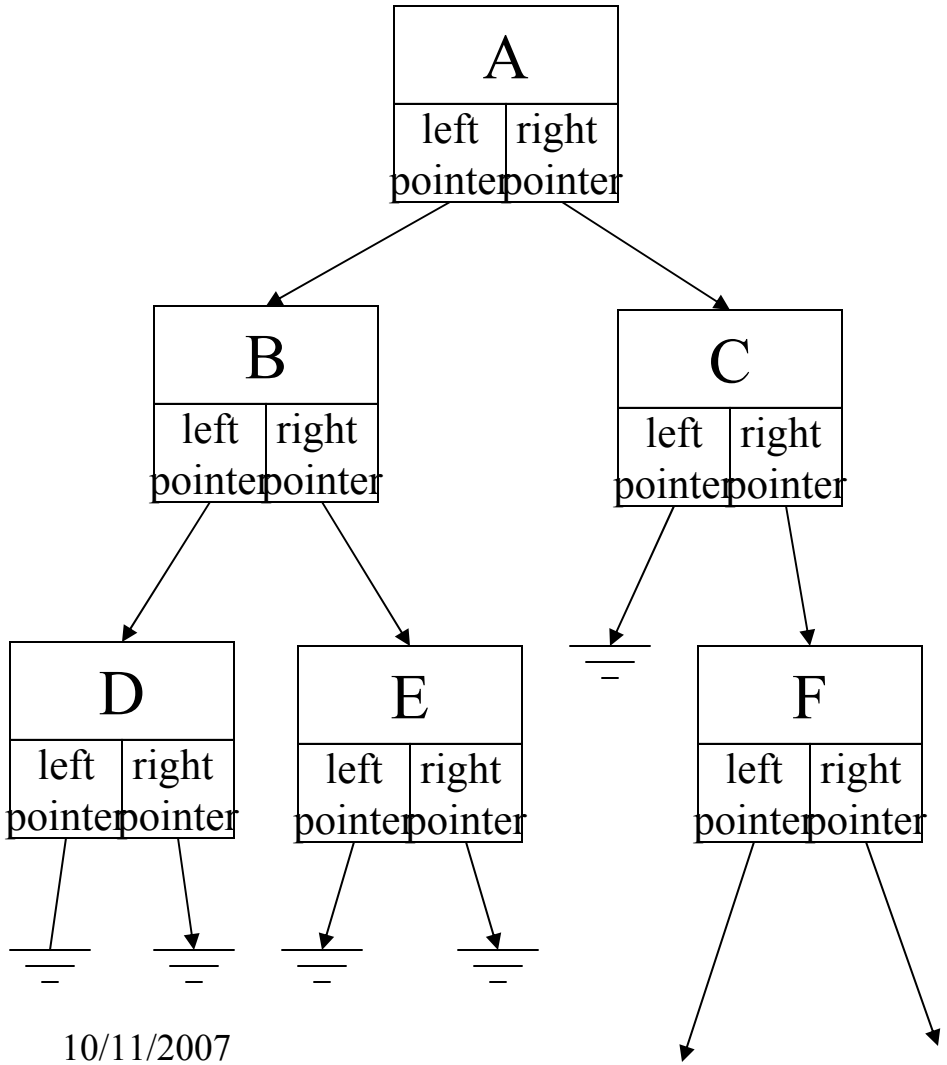
- Binary tree is
  - a root
  - left subtree *(maybe empty)*
  - right subtree *(maybe empty)*

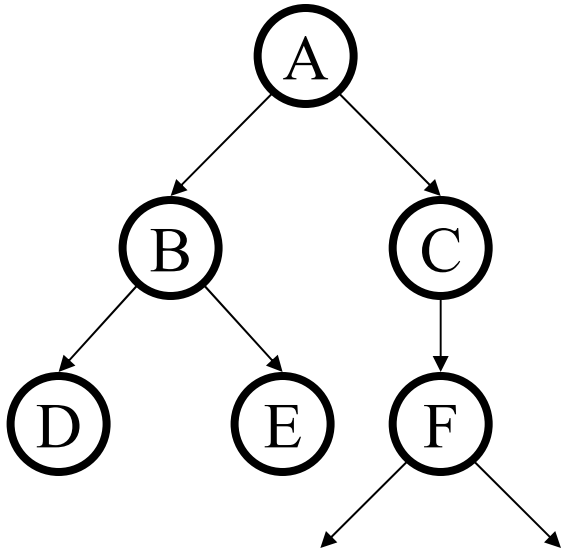
- Representation:

Data	
left pointer	right pointer

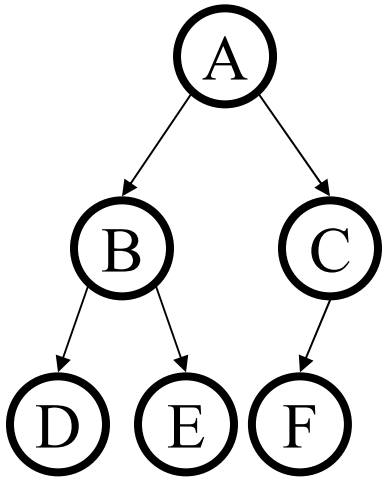


Binary Tree: Representation

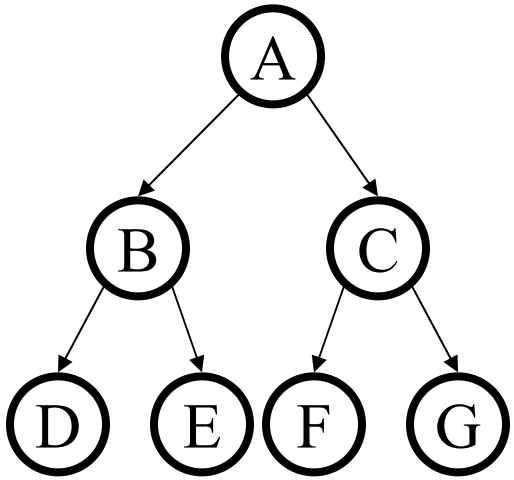




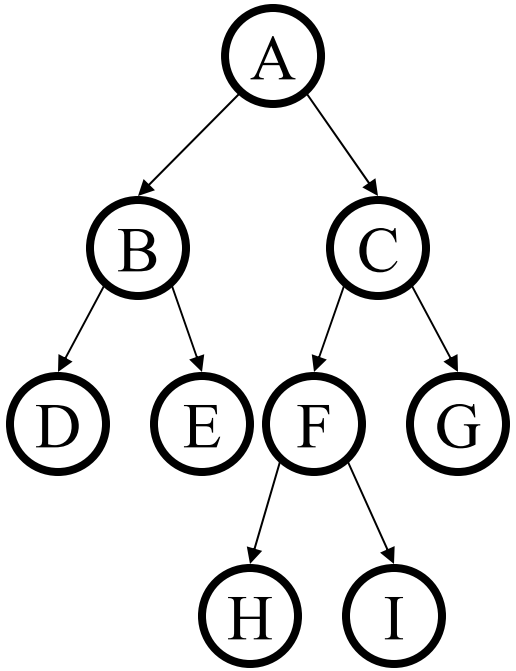
Binary Tree: Special Cases



Complete Tree



Perfect Tree



Full Tree

Binary Tree: Some Numbers!

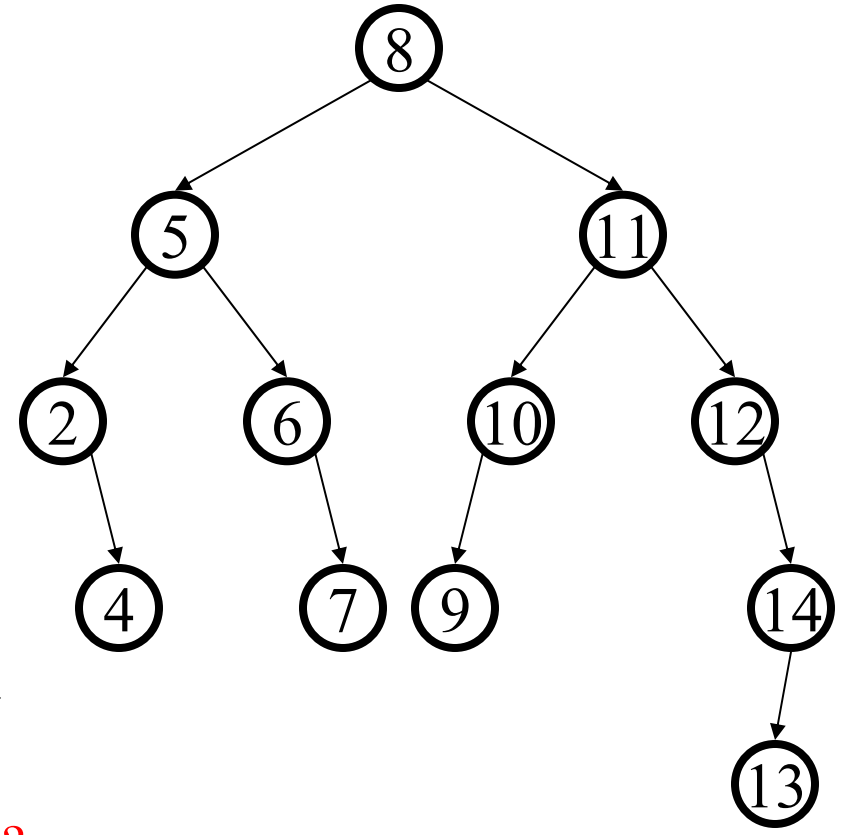
For binary tree of height h:

- max # of leaves: 2^h, for perfect tree
- max # of nodes: $2^{h+1} - 1$, for perfect tree
- min # of leaves: 1, for "list" tree
- min # of nodes: $h+1$, for "list" tree

Average Depth for N nodes?

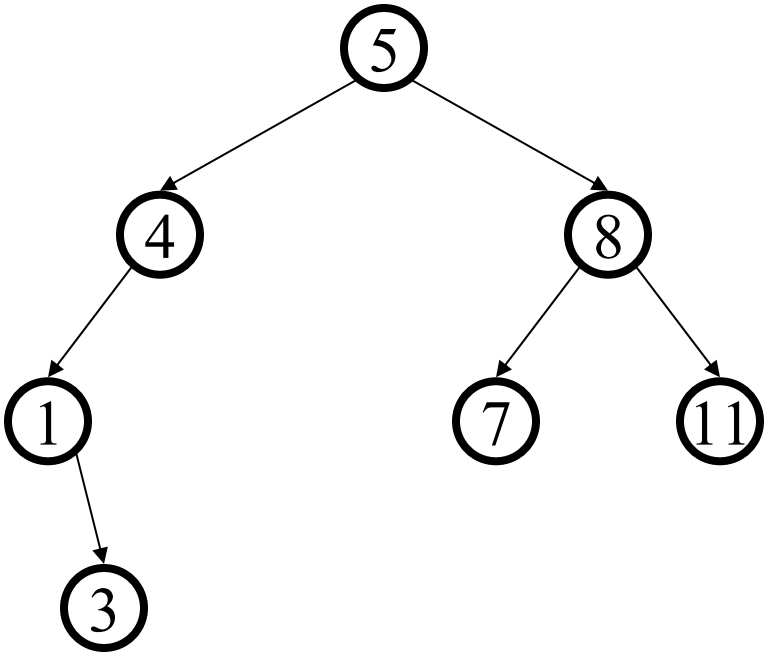
Binary Search Tree Data Structure

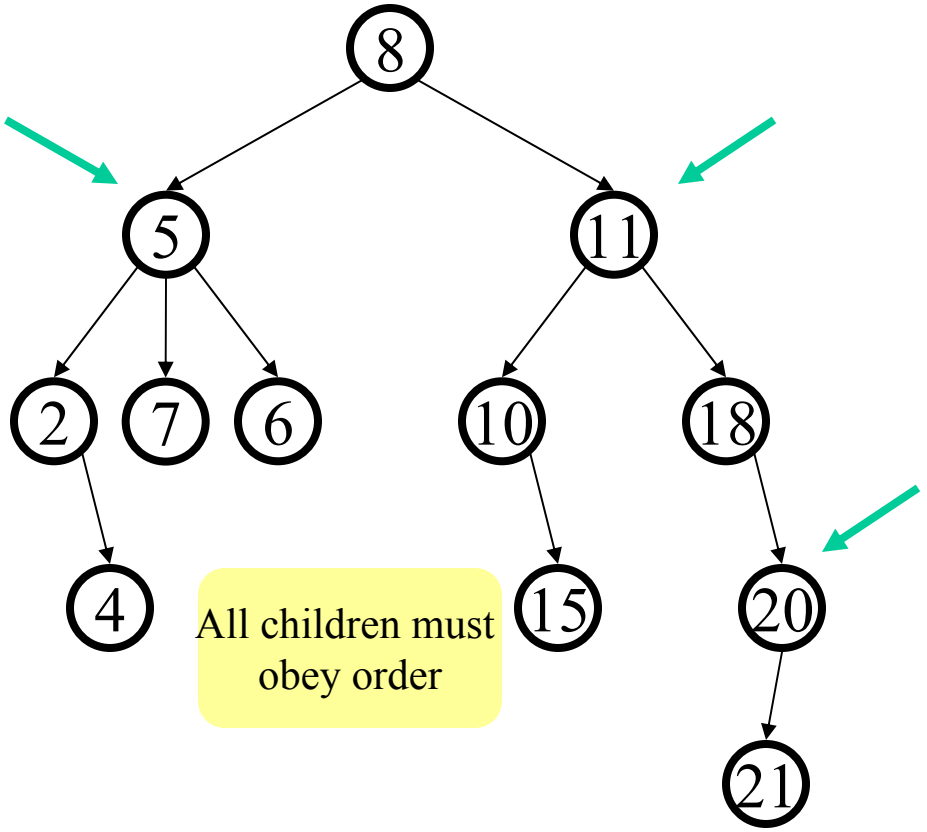
- Structural property
  - each node has ≤ 2 children
  - result:
    - storage is small
    - operations are simple
    - average depth is small
- Order property
  - all keys in left subtree smaller than root's key
  - all keys in right subtree larger than root's key
  - result: easy to find any given key
- What must I know about what I store?



Comparison, equality testing

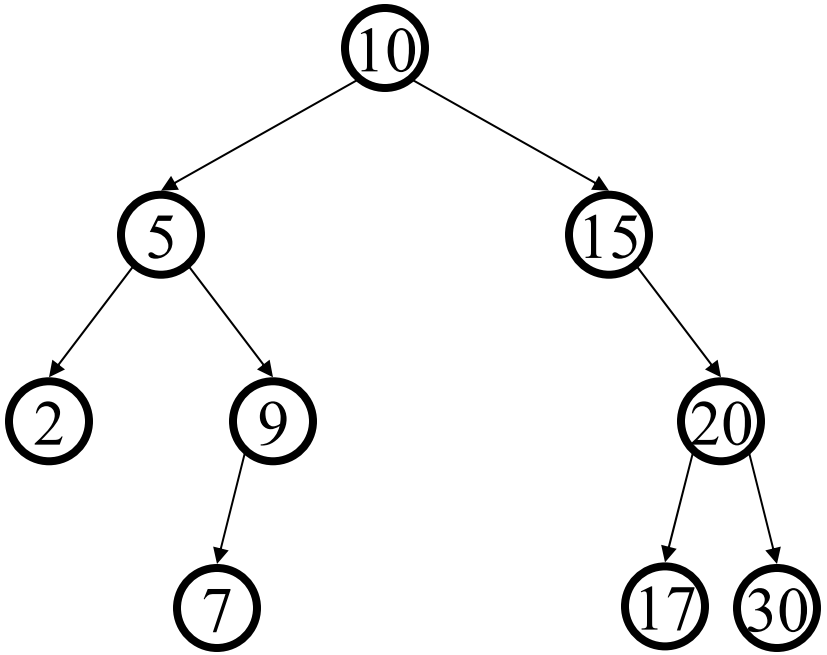
Example and Counter-Example





BINARY SEARCH TREES?

Find in BST, Recursive

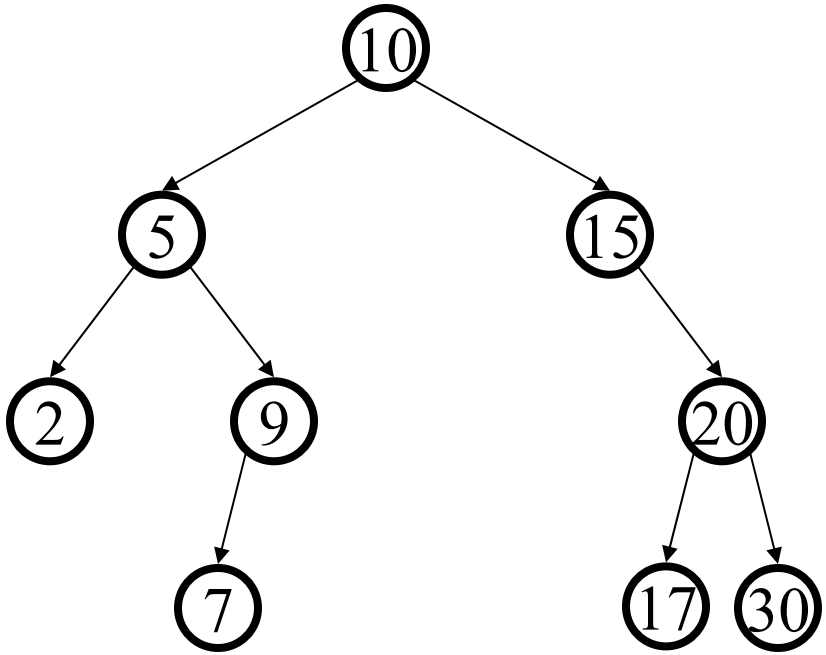


Runtime:

```
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;
}
```

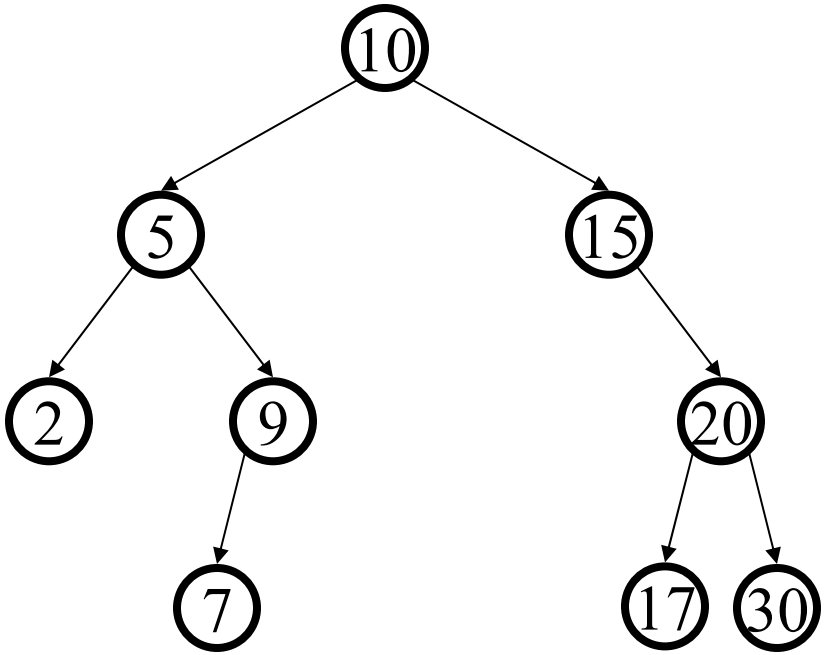
Find in BST, Iterative

```
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:

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.

Runtime depends on the order!

  - in given order

$\Theta(n^2)$

  - in reverse order

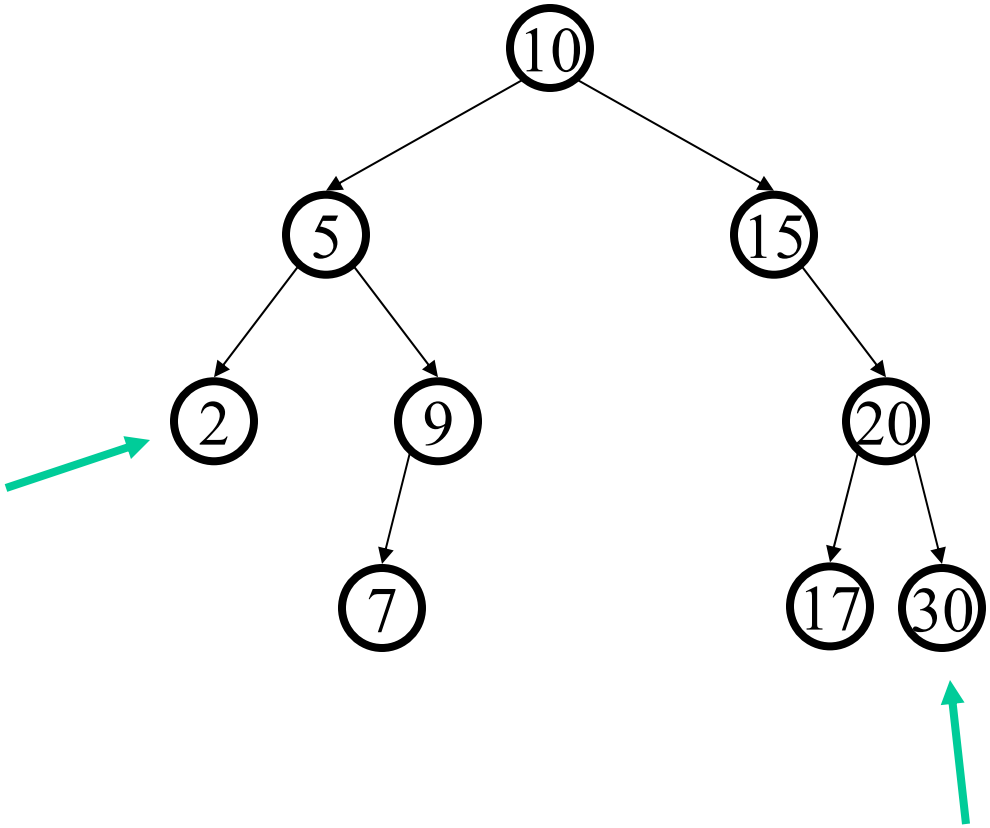
$\Theta(n^2)$

  - median first, then left median, right median, etc.

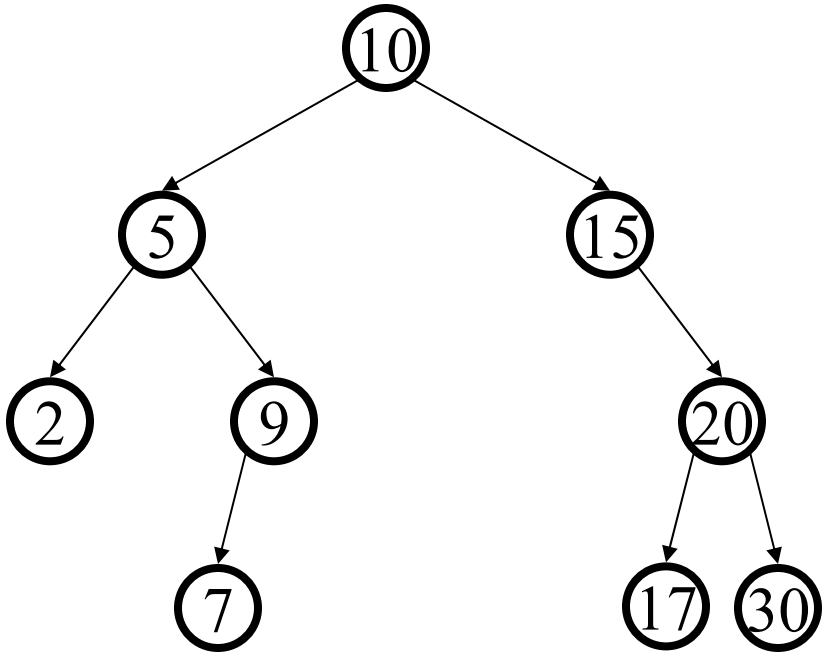
5, 3, 7, 2, 1, 6, 8, 9 better: $n \log n$

Bonus: FindMin/FindMax

- Find minimum
- Find maximum



Deletion in BST



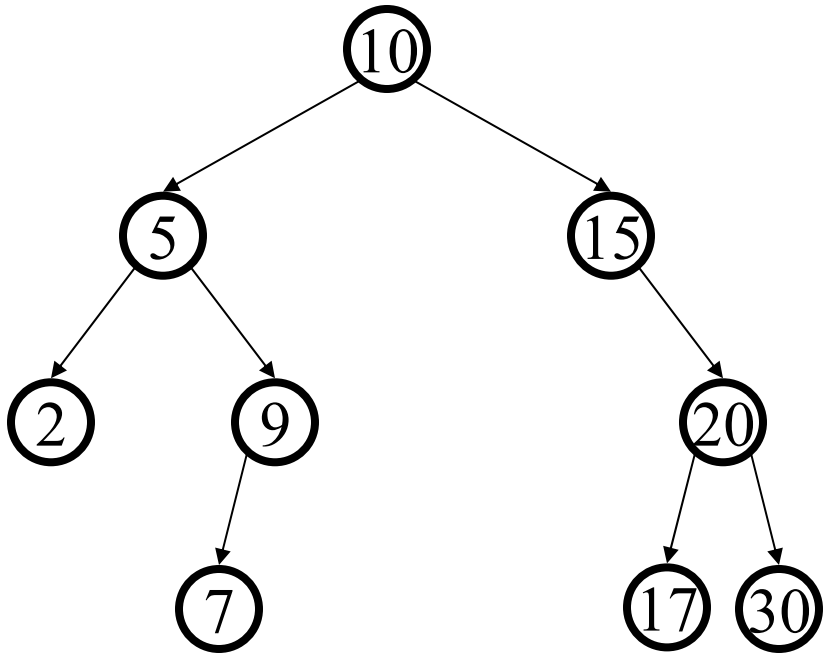
Why might deletion be harder than insertion?

Lazy Deletion

Instead of physically deleting nodes, just mark them as deleted

+ simpler
+ physical deletions done in batches
+ some adds just flip deleted flag

– extra memory for "deleted" flag
– many lazy deletions = slow finds
– some operations may have to be modified (e.g., min and max)

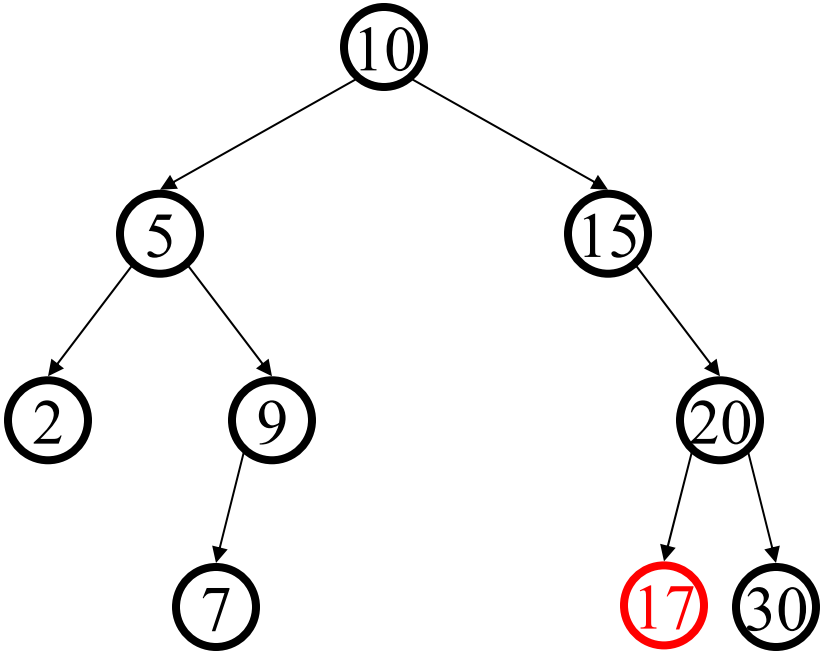


Non-lazy Deletion

- 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

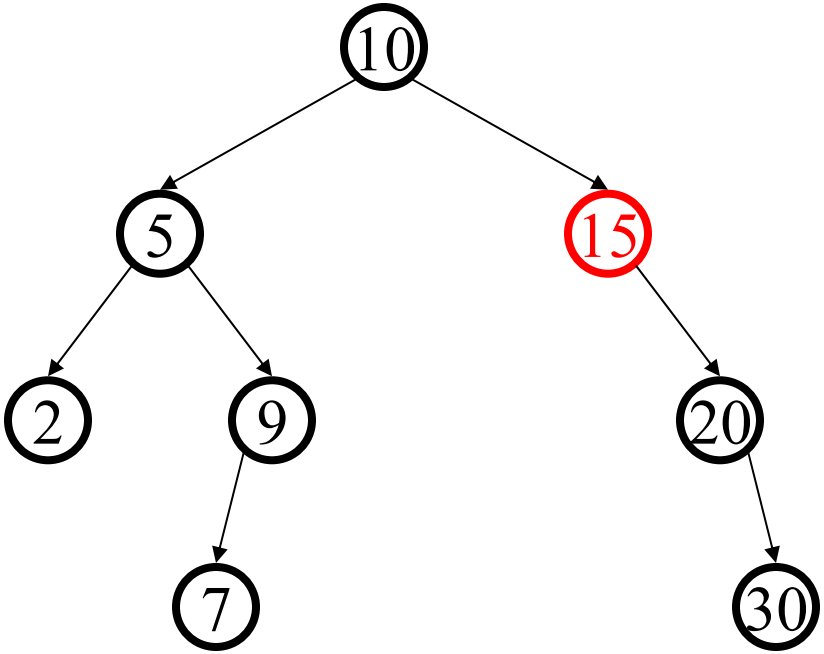
Non-lazy Deletion – The Leaf Case

Delete(17)



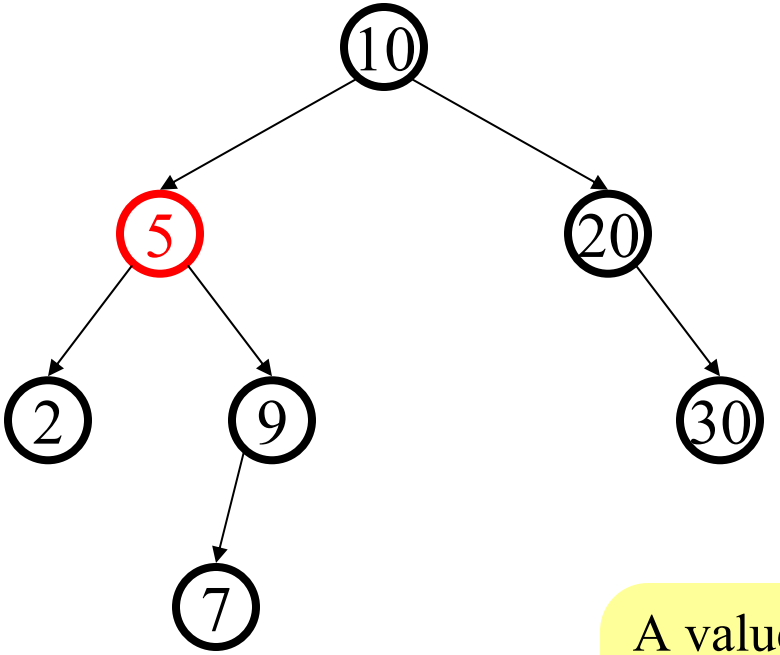
Deletion – The One Child Case

Delete(15)



Deletion – The Two Child Case

Delete(5)



What can we replace 5 with?

A value guaranteed to be between the two subtrees!
- *succ* from right subtree
- *pred* from left subtree

Deletion – The Two Child Case

Idea: Replace the deleted node with a value guaranteed to be between the two child subtrees

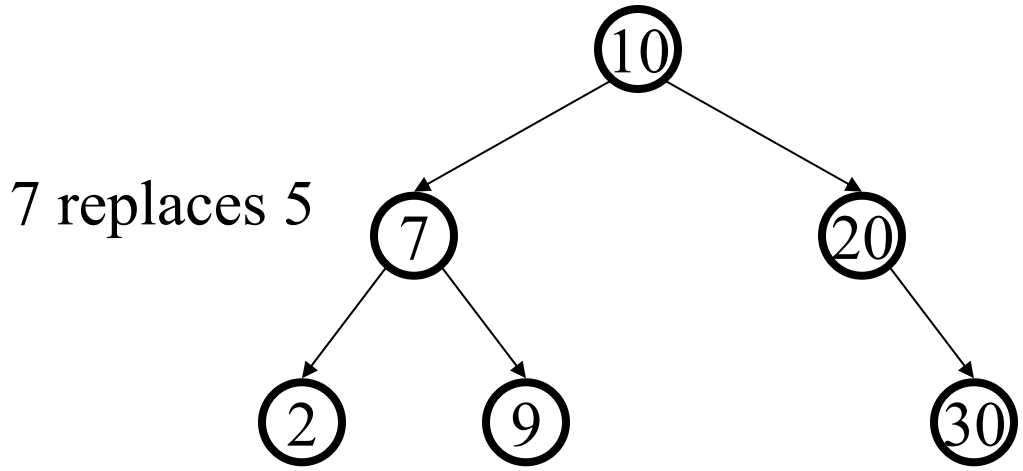
Options:

- *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

Observation

- BST: the shallower the better!
- For a BST with n nodes
  - Average height is O(log n)
  - Worst case height 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!

Potential Balance Conditions

1. Left and right subtrees of the root have equal number of nodes

Too weak!
Do height mismatch example

2. Left and right subtrees of the root have equal *height*

Too weak!
Do example where there's a left chain and a right chain, no other nodes

Potential Balance Conditions

3. Left and right subtrees of *every node* have equal number of nodes

Too strong!
Only perfect trees

4. Left and right subtrees of *every node* have equal *height*

Too strong!
Only perfect trees