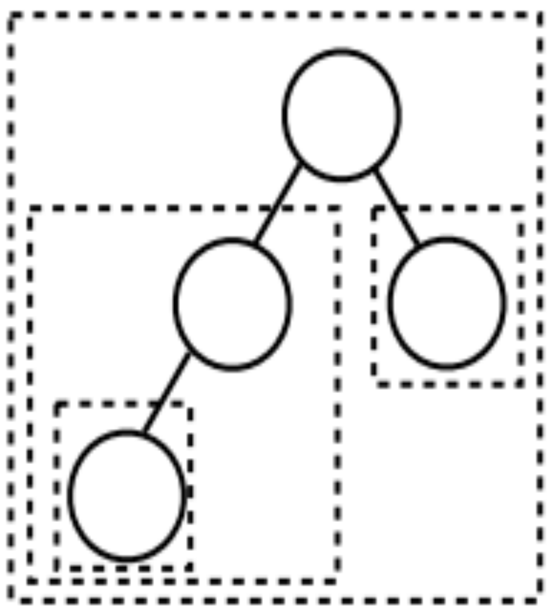
Lecture 18-19

Sub-trees

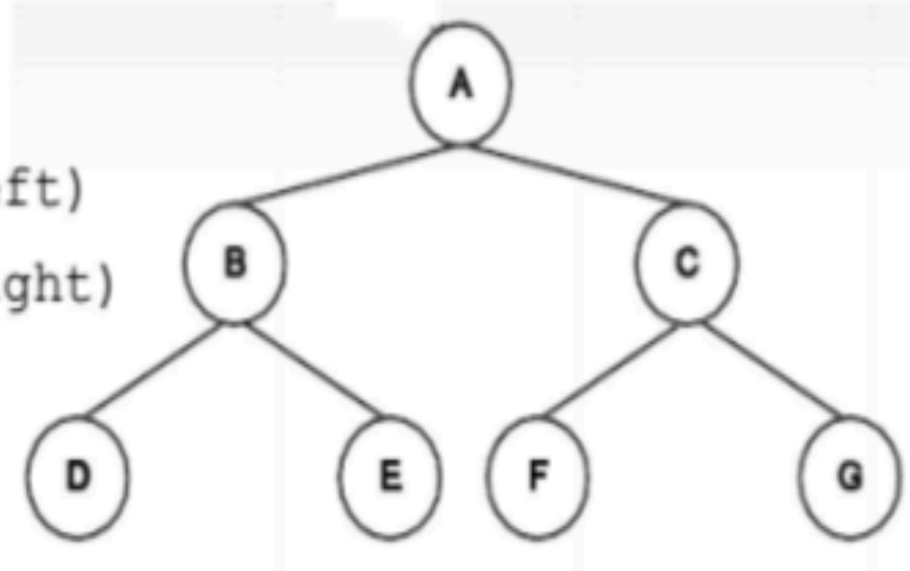
- Each node in a tree is the *root* of its own *sub-tree*.
- The gray boxes below show all possible sub-trees.



BINARY TREE TRAVERSALS

Pre-order traversal

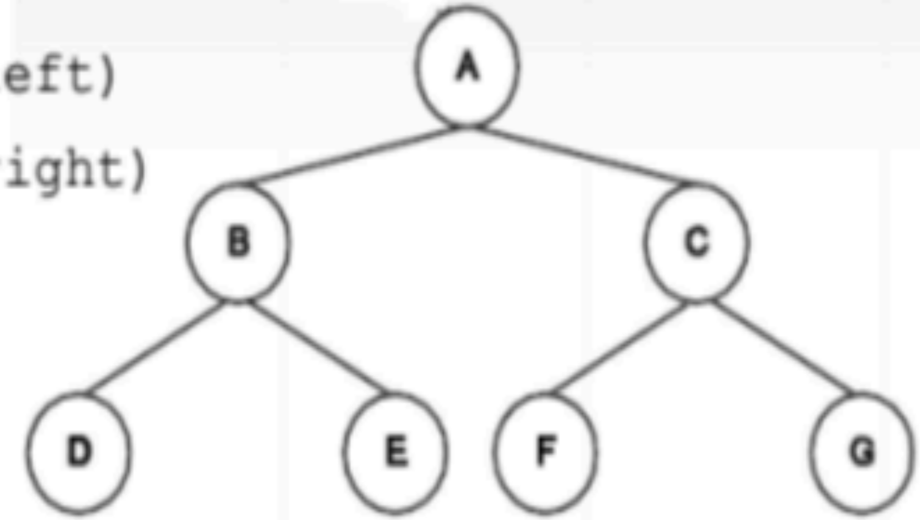
```
preorder(node) {
if (node != null){
      visit this node
      preorder(node.left)
      preorder(node.right)
}
}
```



A. D B E A F C G
B. A B D E C F G
C. A B C D E F G
D. D E B F G C A
E. Other/none/more

Post-order traversal

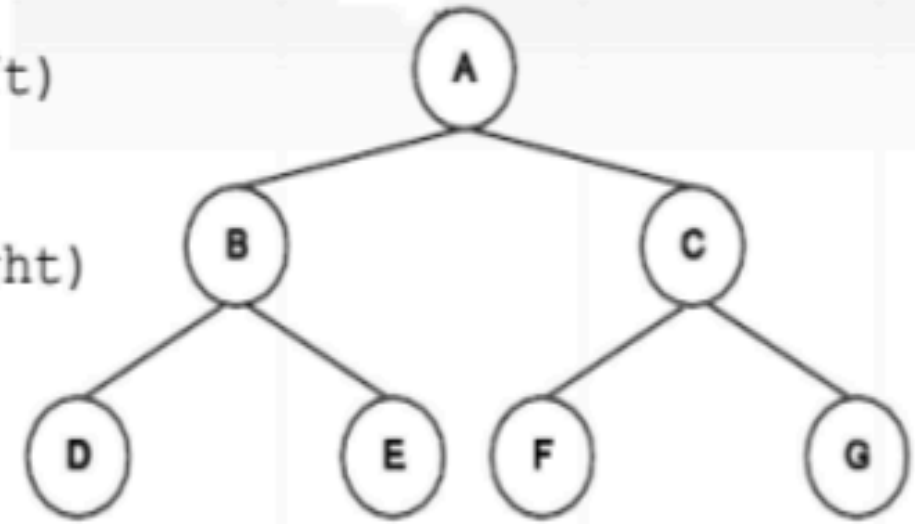
```
postorder(node) {
if (node != null){
      postorder(node.left)
      postorder(node.right)
      visit this node
}
}
```



A. D B E A F C G
B. A B D E C F G
C. A B C D E F G
D. D E B F G C A
E. Other/none/more

In-order traversal

```
inorder(node) {
if (node != null){
      inorder(node.left)
      visit this node
      inorder(node.right)
}
}
```



A. D B E A F C G
B. A B D E C F G
C. A B C D E F G
D. D E B F G C A
E. Other/none/more

Could you do it?

- Give a tree with at least 3 nodes (all nodes must have different keys) such that both its in-order read and its pre-order read are the same, or prove that there is no such tree.
- Give a tree with at least 3 nodes (all nodes must have different keys) such that both its pre-order read and its post-order read are the same, or prove that there is no such tree.

BINARY SEARCH TREES

Heaps are not enough!

- Heaps offer fast access to the largest element in a collection.
  - This is most useful in a priority queue.
- However, finding an arbitrary element is still slow -- $O(n)$ time.

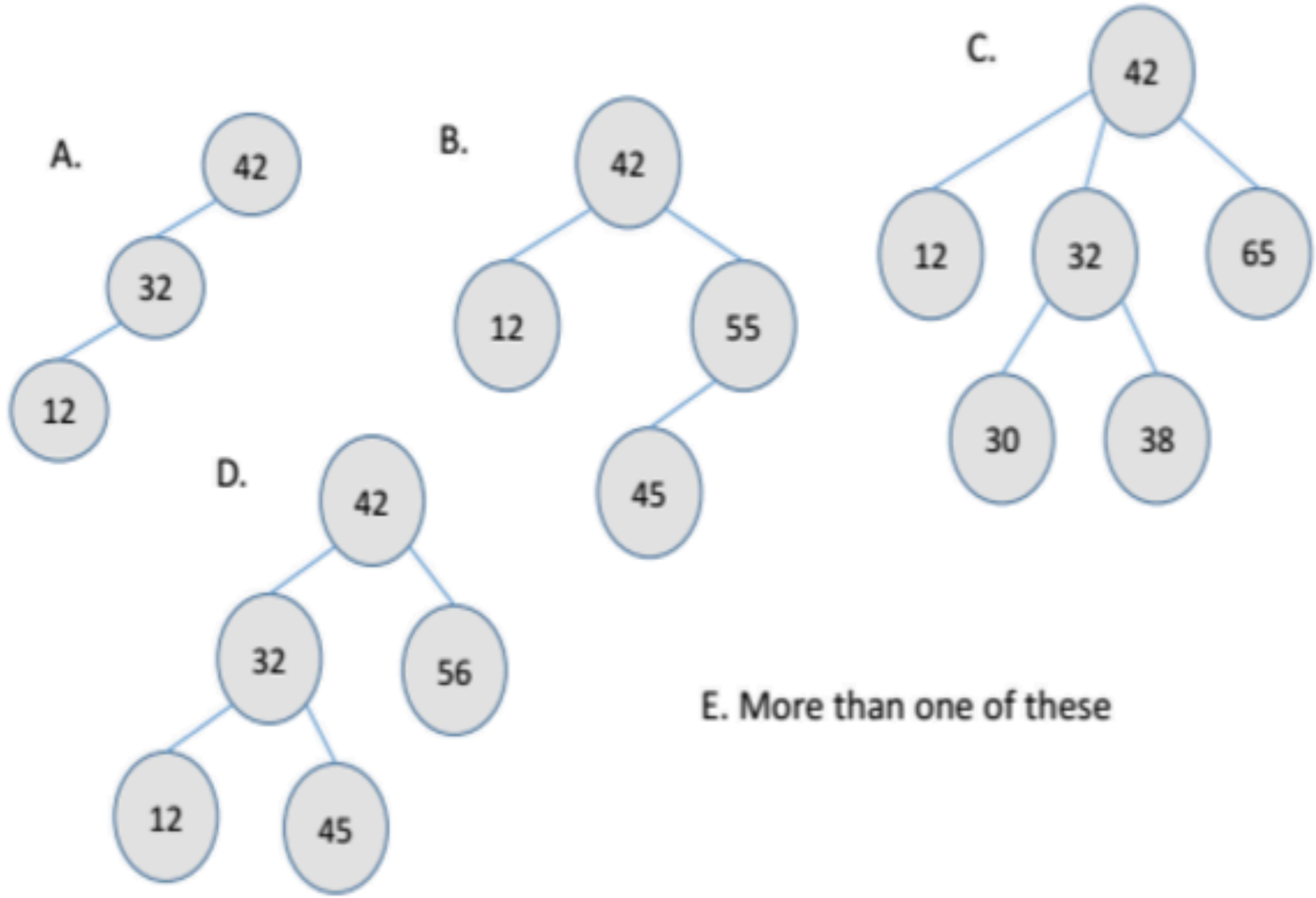
- We may want to sacrifice efficiency of getting the largest element in exchange for increased efficiency to access any *arbitrary* element.

Binary SEARCH tree

- A binary search tree (BST) is a binary-tree based data structure that offers O(log n) **average-case** time costs for:

  - Add (element)
  - Find (element)
  - Remove (element)
  - findLargest/removeLargest (element)

Which of the following is/are a binary search tree?



E. More than one of these

Binary SEARCH tree (BST)

- BST property? For each node n
  - A: all nodes in left subtree of n are “less than” node n, and all nodes in the right subtree of n are “greater than node n.
  - B: It must be complete
  - C: all nodes in left subtree of n are “greater than” node n, and all nodes in the right subtree of n are “less than node n.
  - D: answer A + tree must be complete.
  - E: answer C + tree must be complete

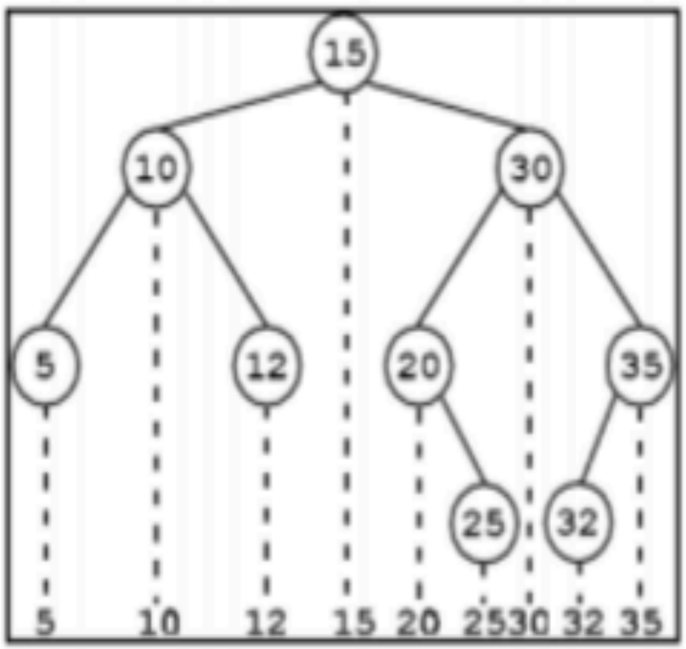
Binary Search Tree (BST)

FACT:

- The "in-order" traversal method, when applied to a BST, gives sorted order

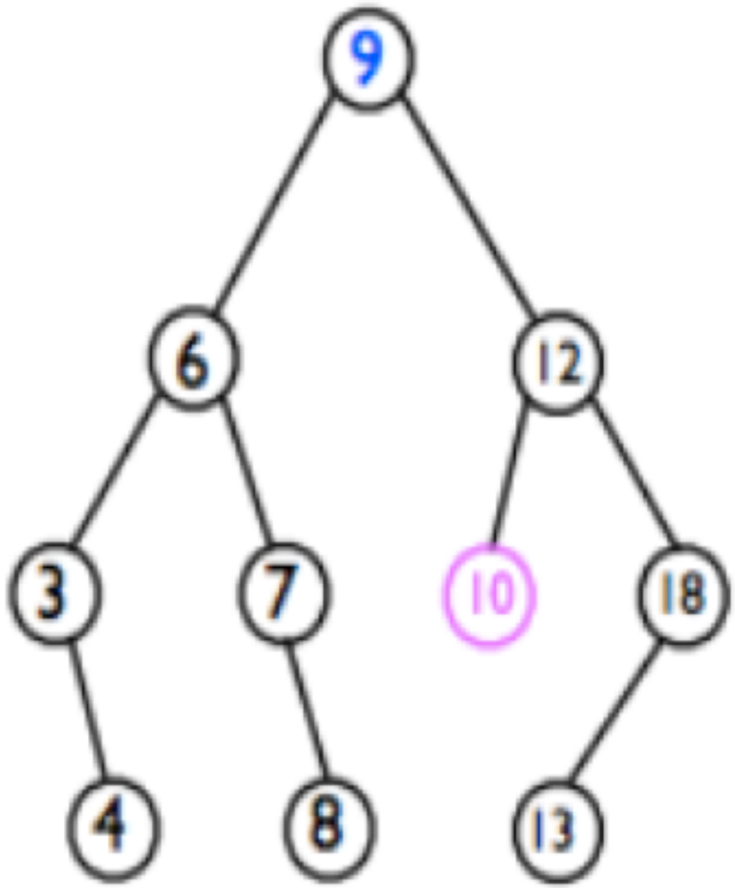
FACT:

- Easy to find things: just recursively check if you should go left or right based on > or <



Where to find a min element?

- In a given a binary tree?

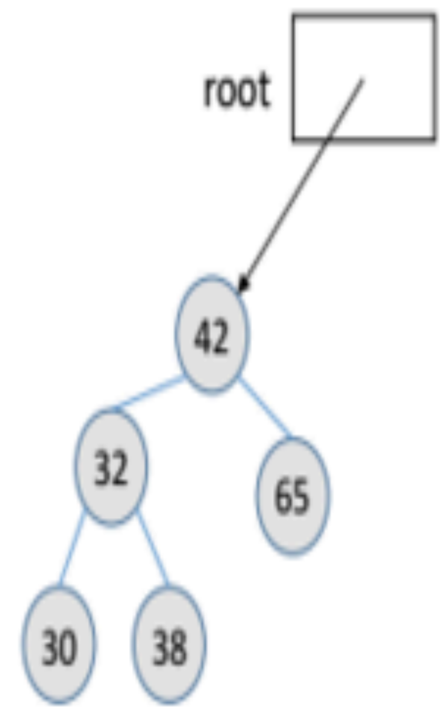


BST IMPLEMENTATION

The BST and BSTNode Classes

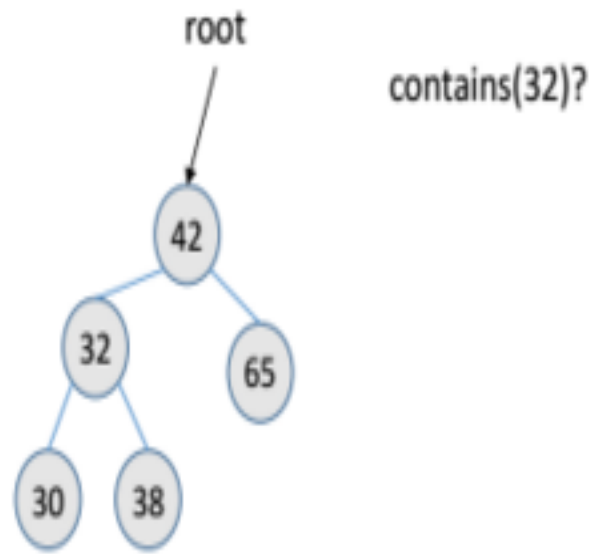
```
public class BST<E extends Comparable<? super E>>
{
    /** Inner class for the BSTNode */
    private class BSTNode {
        protected BSTNode leftChild;
        protected BSTNode rightChild;
        protected E element;

        public BSTNode(E elem)
        {
            element = elem;
        }
    }
    protected BSTNode root;
```



BST Contains: Let's write it!

```
// Return true if toFind is in the BST
public boolean contains(E toFind) {
```

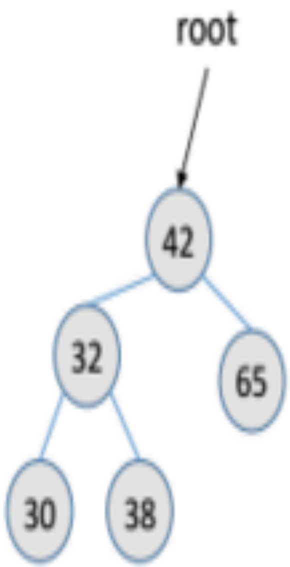


BST Contains: Let's write it!

```
// Return true if toFind is in the BST
public boolean contains(E toFind) {
    //RECURSION!
    return containsHelper(root, toFind);
}

// This recursive method returns true if toFind is in the
// tree rooted at currRoot, and false otherwise
private boolean containsHelper(BSTNode currRoot, E toFind)
{
   // To write!
}
```

contains(32)?

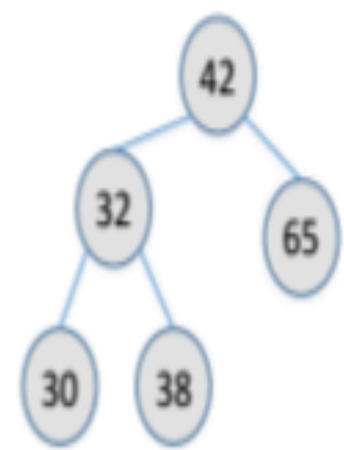


BST Contains: Let's write it!

```
// Return true if toFind is in the BST rooted at currRoot,
// false otherwise
boolean contains(BSTNode currRoot, E toFind) {
```

Base case(s): When do we know we are done?

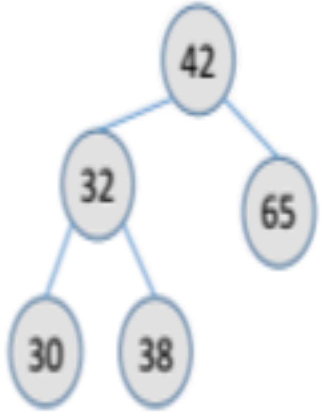
A. toFind is less than currRoot's element
B. toFind is greater than currRoot's element
C. toFind is equal to currRoot's element
D. currRoot is null
E. More than one of these



BST Contains: Let's write it!

```
// Return true if toFind is in the BST rooted at currRoot,
// false otherwise
boolean contains(BSTNode currRoot, E toFind) {
```

Base case 1: (sub)tree is empty, so we know currRoot is not in it



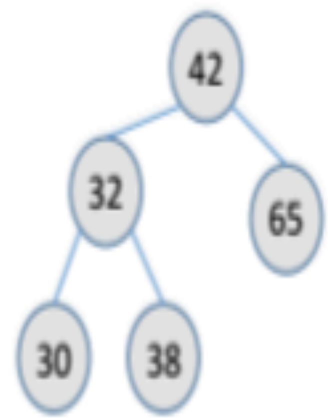
BST Contains: Let's write it!

```
// Return true if toFind is in the BST rooted at currRoot,
// false otherwise
boolean contains(BSTNode currRoot, E toFind) {
    if (currRoot == null) return false;
```

Base case 2: Element is found

We will roll this in with our recursive step

So what is our recursive step...?



contains(32)?
contains(65)?
contains(42)?
contains(40)?

BST Contains: Let's write it!

```
// Return true if toFind is in the BST rooted at currRoot,
// false otherwise
boolean contains(BSTNode currRoot, E toFind) {
    if (currRoot == null) return false;
```

Base case 2: Element is found

We will roll this in with our recursive step

So what is our recursive step...?

contains(32)?
contains(65)?
contains(42)?
contains(40)?

BST Contains: Let's write it!

```
// Return true if toFind is in the BST rooted at currRoot,
// false otherwise
boolean contains(BSTNode currRoot, E toFind) {
    if (currRoot == null) return false;
    if ( ______________________________ )
      return contains( _____________, toFind );
    else if ( ______________________________ )
      return contains( _____________, toFind );
    else
      return _____________;
```

Recursive step and base case 2: How do you know which way to go? Fill in the blanks above. Hint: use compareTo. If you need another hint, check out the next slide.

contains(32)?
contains(65)?
contains(42)?
contains(40)?

```
// Return true if toFind is in the BST rooted at root,
// false otherwise
boolean contains(BSTNode currRoot, E toFind) {
   if ( currRoot == null ) return false;
   if ( toFind.compareTo(currRoot.getElement()) < 0 )
      return _______1________________________;
   else if (toFind.compareTo(currRoot.getElement()) > 0 )
      return ________2_______________________;
   else
      return ________3_______________________;
```

contains(32)?
contains(65)?
contains(42)?
contains(40)?

```
// Return true if toFind is in the BST rooted at root,
// false otherwise
boolean contains(BSTNode currRoot, E toFind) {
   if ( currRoot == null ) return false;
   if ( toFind.compareTo(currRoot.getElement()) < 0 )
      return contains(currRoot.getLeft(), toFind);
   else if (toFind.compareTo(currRoot.getElement()) > 0 )
      return contains(currRoot.getRight(), toFind);
   else
      return true;
```

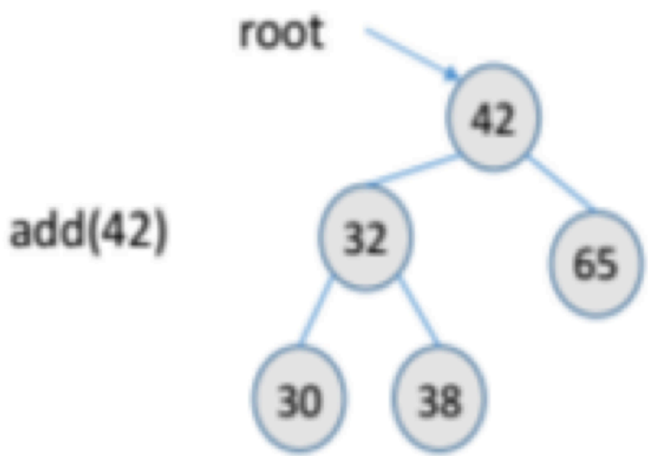
contains(32)?
contains(65)?
contains(42)?
contains(40)?

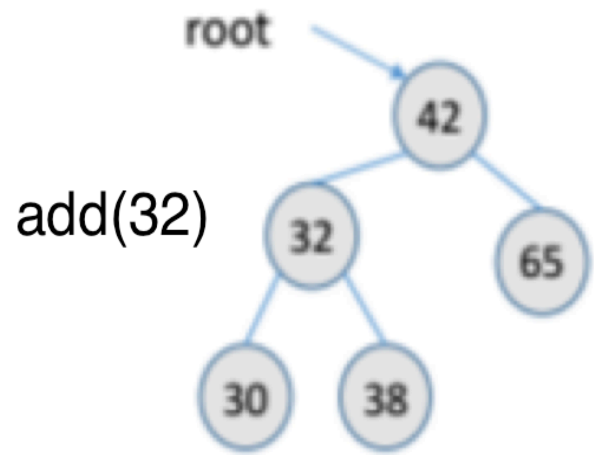
Plan

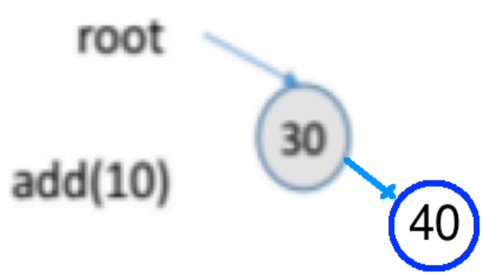
- Insert
- Delete
- Running time

BST Add: With recursion!

Consider the following:







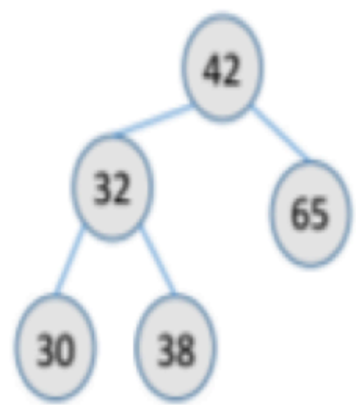
BST Add: Recursively

```
boolean add( E toAdd ) {
  if (root ==  null) {
    root = new BSTNode(toAdd);
    return true;
  }
   return addHelper( root, toAdd );
}

boolean addHelper( BSTNode currRoot, E toAdd )
{
     ...
}
```

Which of these is/are a base case for addHelper?

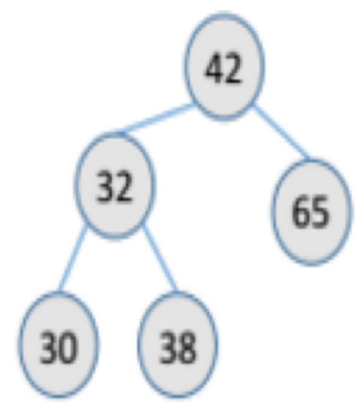
A. currRoot is null
B. currRoot's element is equal to toAdd
C. Both A & B
D. Neither of these



BST Add: Recursively

```
boolean add( E toAdd ) {
  if (root ==  null) {
    root = new BSTNode(toAdd);
    return true;
  }
   return addHelper( root, toAdd );
}
```

```
boolean addHelper( BSTNode currRoot, E toAdd )
{
    int compare = toAdd.compareTo(currRoot.getElement());
    if (compare == 0) {
        return                    ;
    }
}
```



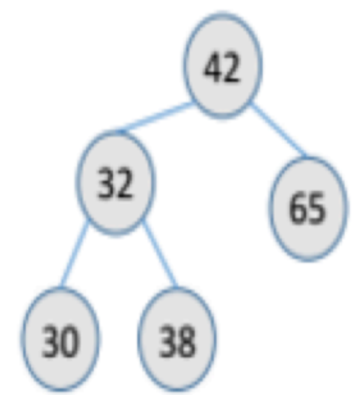
Which of these is/are a base case for addHelper?
currRoot's element is equal to toAdd

BST Add: Recursively

```
boolean add( E toAdd ) {
  if (root ==  null) {
    root = new BSTNode(toAdd);
    return true;
  }
   return addHelper( root, toAdd );
}

boolean addHelper( BSTNode currRoot, E toAdd )
{
    int compare = toAdd.compareTo(currRoot.getElement());
    if (compare == 0) {
        return false;
    }
    // Finish the code…

}
```



Which of these is/are a base case for addHelper?
currRoot's element is equal to toAdd

BST Add: Recursively

```
void addHelper( BSTNode currRoot, E toAdd )
{
   int value = toAdd.compareTo( currRoot.getElement() );
   if (value == 0) return false;
   if ( value < 0 ) {
      if ( currRoot.getLeftChild() == null ) {
         currRoot.setLeftChild(new BSTNode( toAdd ));
      }
      else {
         addHelper( currRoot.getLeftChild(), toAdd );
      }
   }
   else { // ( value > 0 )
      // Repeat for other side
   }
   return true;
}
```

Remove a node

- Real deletion (blackboard)
- Lazy deletion.

Duplicate keys

- Allow them and be consistent with the rule
- Node is a linked list. All duplicates are linked.

COMPLEXITY

What is the BEST CASE cost for doing find() in BST (tightest Big-O, on this and future questions)?

A. O(1)
B. O(log n)
C. O(n)
D. O(n log n)
E. $O(n^2)$

What is the WORST CASE cost for doing find() in a BST?

A. $O(1)$
B. $O(\log n)$
C. $O(n)$
D. $O(n \log n)$
E. $O(n^2)$

Balance

- The #1 issue to remember with BSTs is that they are great when balanced (O(log n) operations), and horrible when unbalanced (O(n) operations)
- Balance depends on order of insert/delete of elements
- Over the years, people have devised many ways of making sure BSTs stay balanced no matter what order the elements are inserted.
- We won't talk about these ways here, but stay tuned for CSE 100...

Fun Questions

- Find k-th minimum element in a BST.
- Given a binary tree, check whether it’s a binary search tree or not.