CSE 12 – Basic Data Structures and Object-Oriented Design
Lecture 18

Greg Miranda & Paul Cao, Winter 2021

This lecture is being recorded

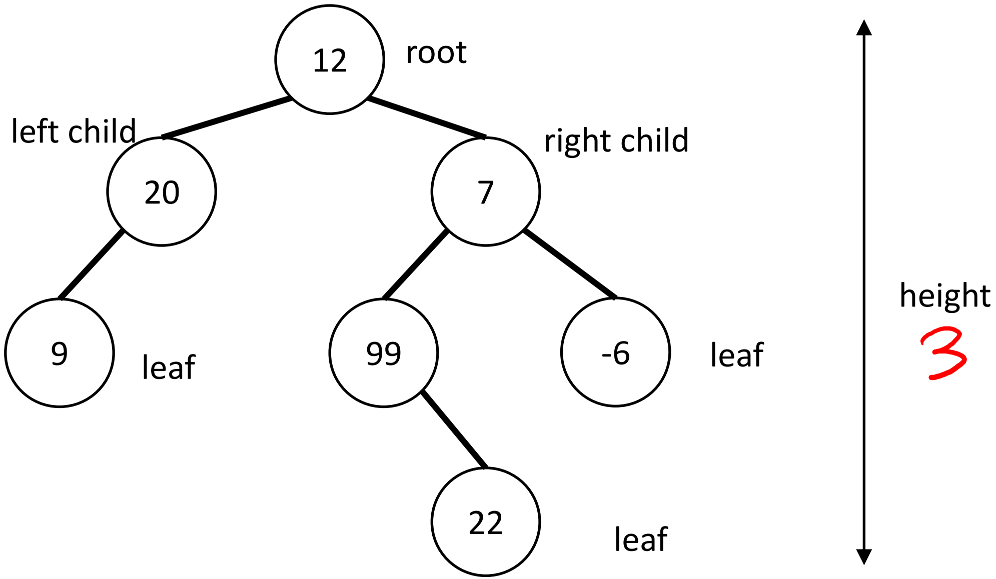
Announcements

- Quiz 18 due Monday @ 8am
- Survey 7 due tonight @ 11:59pm
- PA7 due Tuesday, March 2nd @ 11:59pm
- Exam 2 – Week 8
  - Released Friday 2/26 @ 8am
  - Due Saturday 2/27 @ 10am
  - Topics:
    - Cumulative
    - Big topics
      - Big O, Big Theta run-time analysis
      - Sorting algorithms
      - Hash tables/maps

Topics

- Questions on Lecture 18?
- Binary Search Trees

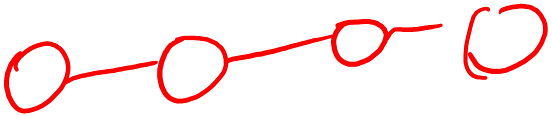
Tree



The **height** of a binary **tree** is the largest number of edges in a path from the root node to a leaf node.

Binary Tree: a node may have at most 2 children

Tree Node



```
class TNode{
   _________ left;
   _________ right;
   Integer  value;
}
```

What should be the type of left and right?

A. Integer
B. Object
C. TNode
D. Anything that implements Comparable interface
E. Something else

Tree

```
class TNode{
   TNode left;
   TNode right;
   Integer  value;
}
```

TNode parent

What methods we should NOT put into the TNode class?

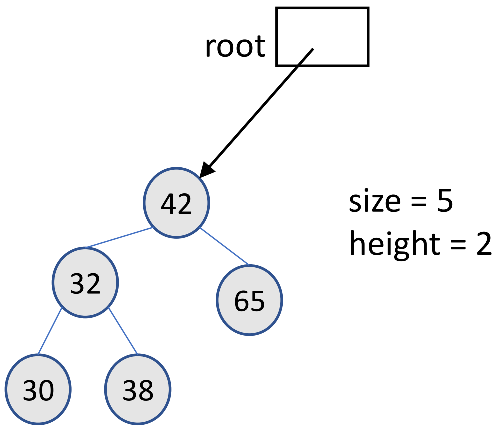
A. getLeftChild
B. getValue
C. setValue
D. setRightChild
E. getRoot

It is fairly similar to linked lists
except we have two children

BinaryTree Class

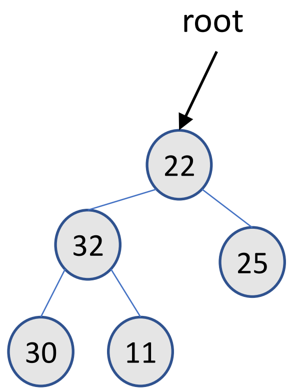
```
public class BST
{
    /** Inner class*/
    class TNode {
        TNode left;
        TNode right;
        Integer value;

        public BSTNode(Integer value)
        {
            this.value = value;
        }
    }
    BSTNode root;
    int size; //number of nodes in the tree
    int height; //height of the tree
```



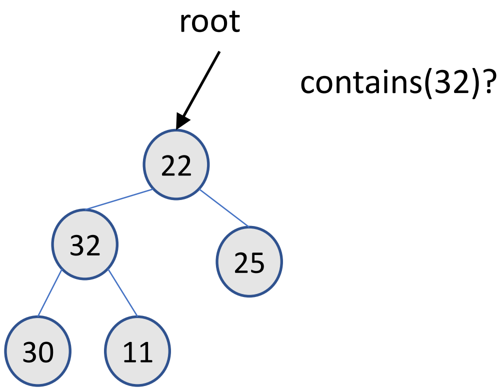
Binary Tree Contains: Let's write it!

```
// Return true if toFind is in the Tree. We will use recursion here
public boolean contains(Integer toFind) {
```



Binary Tree Contains: Let’s write it!

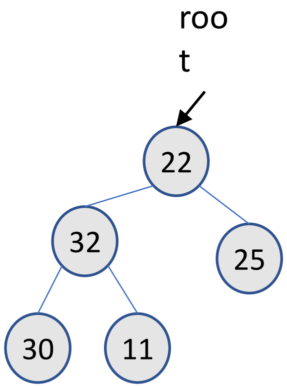
```
// Return true if toFind is in the Tree. We will use recursion here
public boolean contains(Integer toFind) {
```



Binary Tree Contains: Let’s write it!

```
// Return true if toFind is in the Tree
public boolean contains(Integer 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(TNode currRoot, Integer toFind)
{
   // To write!
}
```



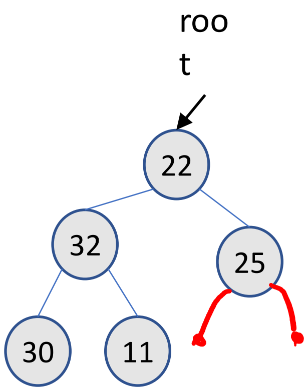
contains(32)?

Binary Tree Contains: Let's write it!

```
// Return true if toFind is in the Tree rooted at currRoot,
// false otherwise
boolean containsHelper(TNode currRoot, Integer toFind) {
```

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

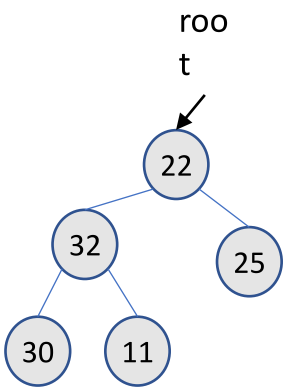
A. toFind is less than currRoot's element
B. toFind is greater than currRoot's element
C. toFind is equal to currRoot's element
D. currRoot is null
E. More than one of these



Binary Tree Contains: Let’s write it!

```
// Return true if toFind is in the Tree rooted at currRoot,
// false otherwise
boolean containsHelper(TNode currRoot, Integer toFind) {
```

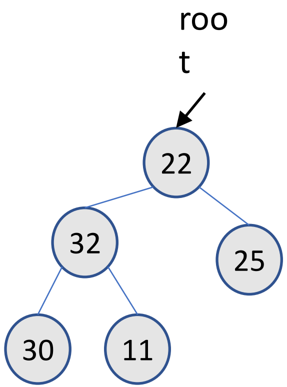
Base case 1: (sub)tree is empty, so we know toFind is not in it



Binary Tree Contains: Let's write it!

```
// Return true if toFind is in the Tree rooted at currRoot,
// false otherwise
boolean containsHelper(TNode currRoot, Integer toFind) {
    if (currRoot == null) return false;
```

Base case 2: toFind is found
We will roll this in with our recursive step
So what is our recursive step…?

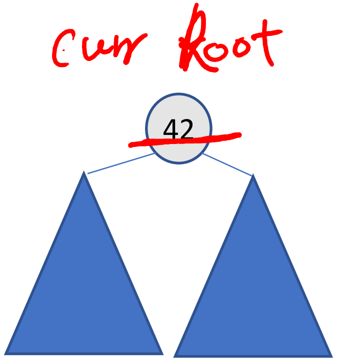


contains(32)?
contains(65)?
contains(42)?
contains(40)?

Binary Tree Contains: Let’s write it!

```
// Return true if toFind is in the Tree rooted at currRoot,
// false otherwise
boolean containsHelper(TNode currRoot, Integer 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…?



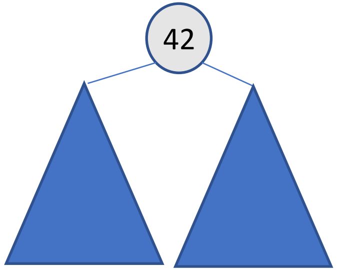
contains(32)?
contains(65)?
contains(42)?
contains(40)?

does the left subtree rooted at currRoot.left contain toFind?

Binary Tree Contains: Let’s write it!

```
// Return true if toFind is in the Tree rooted at currRoot,
// false otherwise
boolean containsHelper(TNode currRoot, Integer toFind) {
    if (currRoot == null) return false; // first base case   (leaf is reached)
    if (currRoot.getValue().equals(toFind)) //second base case
        return true
    return containsHelper(currRoot.getLeft(), toFind) || containsHelper
                                                         (right, toFind)
```

Recursive step and base case 2
Fill in the blanks above.
If you need another hint, check out the next slide.



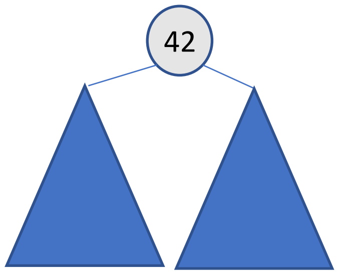
contains(32)?
contains(65)?
contains(42)?
contains(40)?

DFS approach using a stack

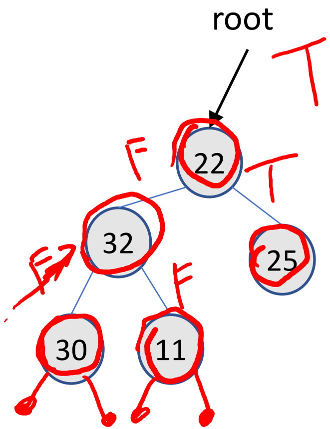
```
// Return true if toFind is in the Tree rooted at currRoot,
// false otherwise
boolean containsHelper(TNode currRoot, Integer toFind) {
    if (currRoot == null) return false;  // first base case
    if (_______________________) //second base case
       return ___________________

    return ____________________________________________
```

Recursive step and base case 2
Fill in the blanks above.
If you need another hint, check out the next slide.



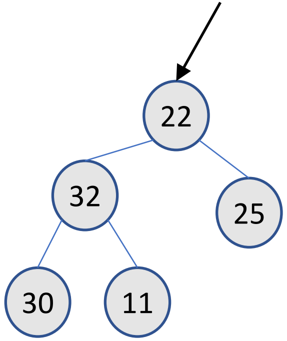
contains(32)?
contains(65)?
contains(42)?
contains(40)?



contains(25)?

```
// Return true if toFind is in the Tree rooted at currRoot,
// false otherwise
boolean containsHelper(TNode currRoot, Integer toFind) {
     if (currRoot == null) return false;  // first base case
    ✓if (currRoot.value.equals(toFind)) //second base case
        return true;
     return containsHelper(currRoot.left, toFind)
                             || containsHelper(currRoot.right, toFind);
}
```

root



contains(12)?

```
// Return true if toFind is in the Tree rooted at currRoot,
// false otherwise
boolean containsHelper(TNode currRoot, Integer toFind) {
     if (currRoot == null) return false;  // first base case
     if (currRoot.value.equals(toFind)) //second base case
        return true;
     return containsHelper(currRoot.left)
                            || containsHelper(currRoot.right);
}
```

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

A. O(1)

B. O(log n)

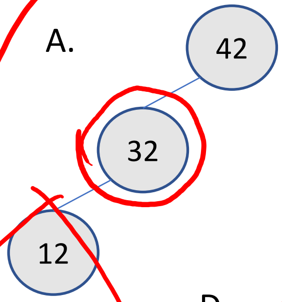
C. O(n)

D. O(n log n)

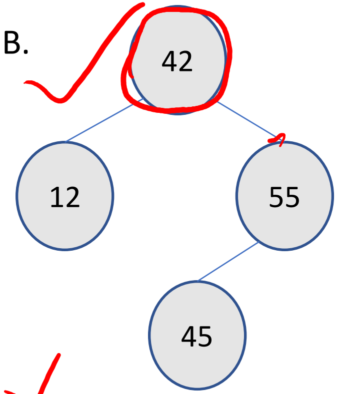
E. $O(n^2)$

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

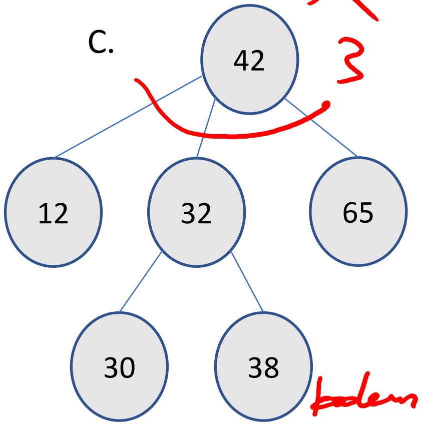
A.



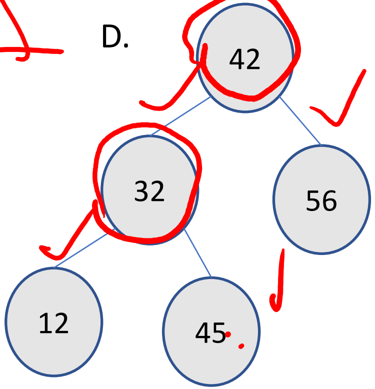
B.



C.



D.



E. More than one of these

verify if a BT is a BST?

use recursion

boolean verify(Node root)
{ verify(root.left) && (...)

```
class Node<K,V> {
  K key;
  V value;
  Node<K,V> left;
  Node<K,V> right;
  public Node(K key, V value,
              Node<K,V> left,
              Node<K,V> right) {
    this.key = key;
    this.value = value;
    this.left = left;
    this.right = right;
  }
}
```

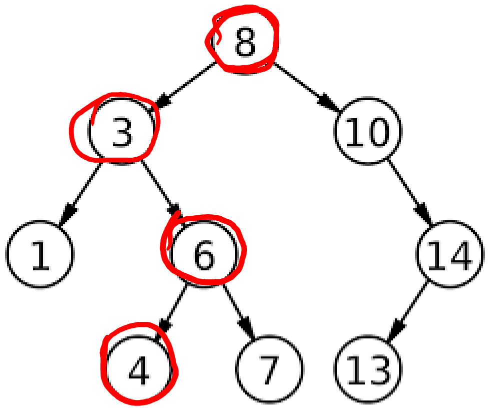
```
class BST<K, V> {
  Node<K, V> root;
  BST() (this.root = null);
  BST(Node<K, V> root) { this.root = root; }

  V get(Node<K, V> node, K key) {
    if (node == null) { //throw error }      Can't find it
    if (node.key.equals(key)) {
      return node.value;
    }                 .compareTo(key) > 0
    if (node.key > key) {
      return get(node.left, key);
    }
    else {
      return get(node.right, key);
    }
  }

  V get(Key key) {
    return this.get(root, key);
  }
}
```

Binary Search Tree

- Assume the key and value are identical for this example
- Trace the path for get(4)
  - How many nodes does it touch?
- Trace the path for get(2)
  - How many nodes does it touch?
  - What happens when the nodes isn't found?



Binary Search Tree

- Assume the key and value are identical for this example
- Trace the path for get(40)
  - How many nodes does it touch?
- Trace the path for get(4)
  - How many nodes does it touch?