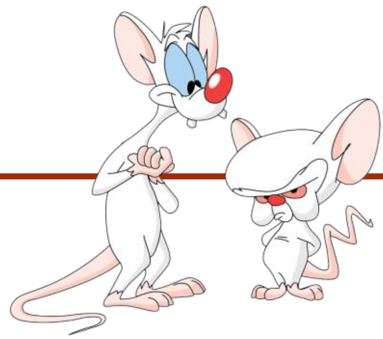
COMP 250
INTRODUCTION TO COMPUTER SCIENCE

Week 12-1 : Binary Search Trees

Giulia Alberini, Fall 2020

Slides adapted from Michael Langer's

WHAT ARE WE GOING TO DO IN THIS VIDEO?



- Binary Search Trees

BINARY SEARCH TREES

BSTNode

- The keys are "comparable"   <, =, >
  e.g. numbers, strings.

```
class BSTNode<K>{
    K key;
    BSTNode<K> leftchild;
    BSTNode<K> rightchild;
        :
}
```

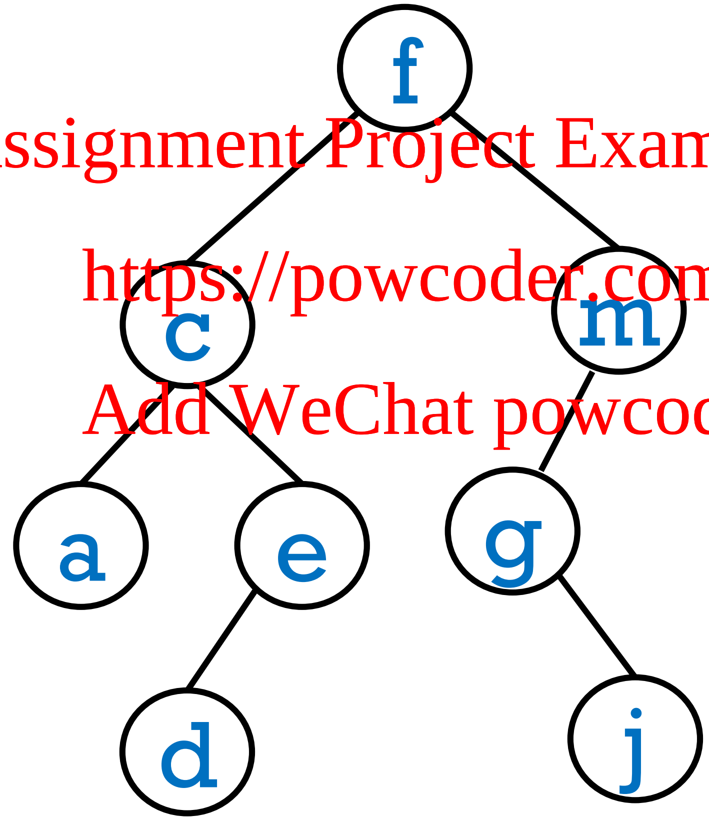
BINARY SEARCH TREE DEFINITION

- binary tree

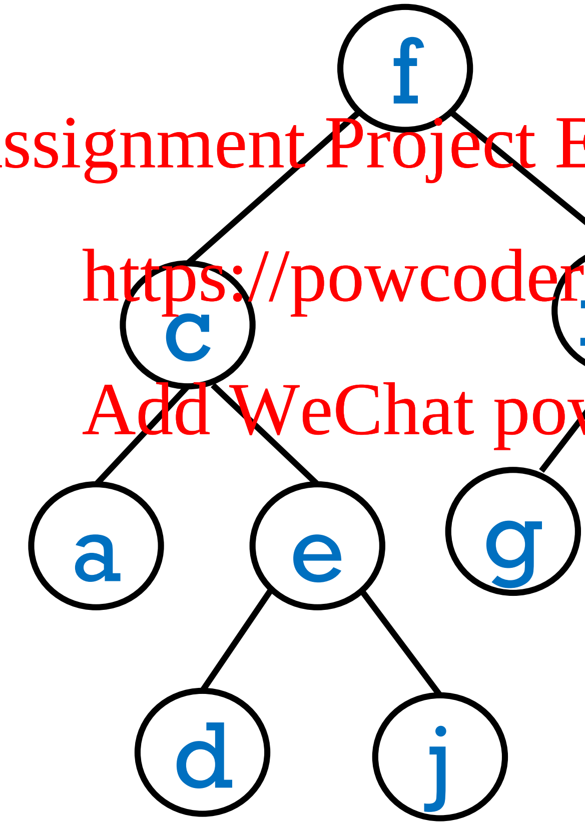
- keys are comparable, and unique (no duplicates)

- for each node, all descendents in left subtree are less than the node, and all descendents in the node’s right subtree are greater than the node (comparison is based on node key)

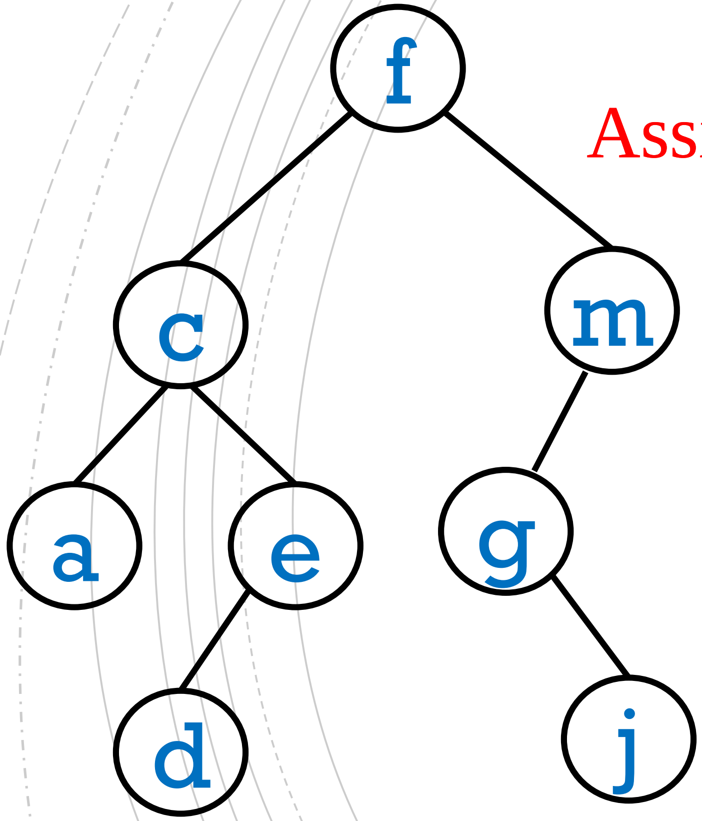
EXAMPLE



THIS IS NOT A BST. WHY NOT?



BST - TRAVERSALS



An in-order traversal on a BST visits the nodes in the natural order defined by the key.

a c d e f g j m

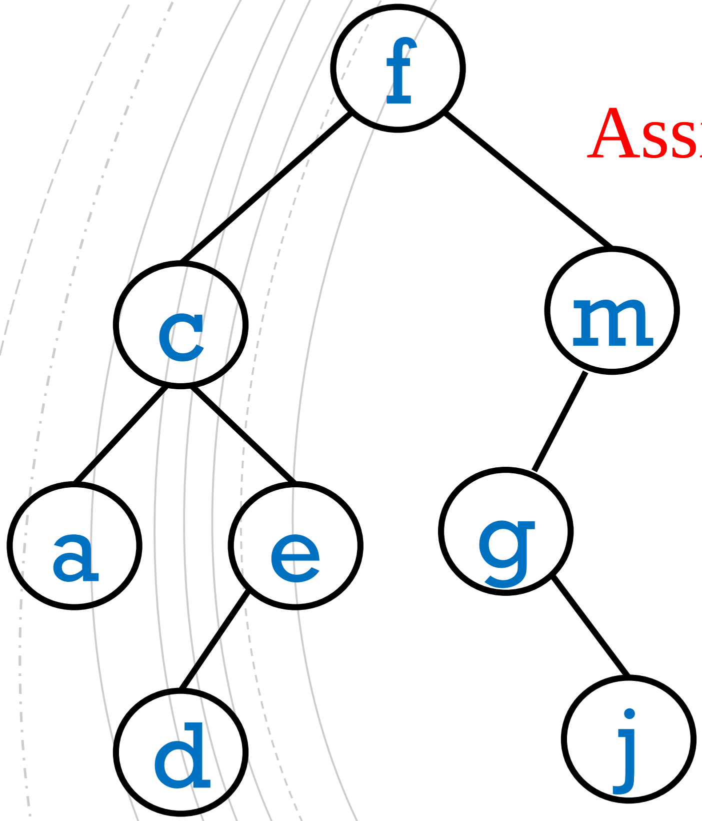
BINARY SEARCH TREE ADT

- find(key)
- findMin()
- findMax()
- add(key)
- remove(key)

We can define the operations of a BST without knowing how they are implemented. (ADT)

Let's next look at some recursive algorithms for implementing them.

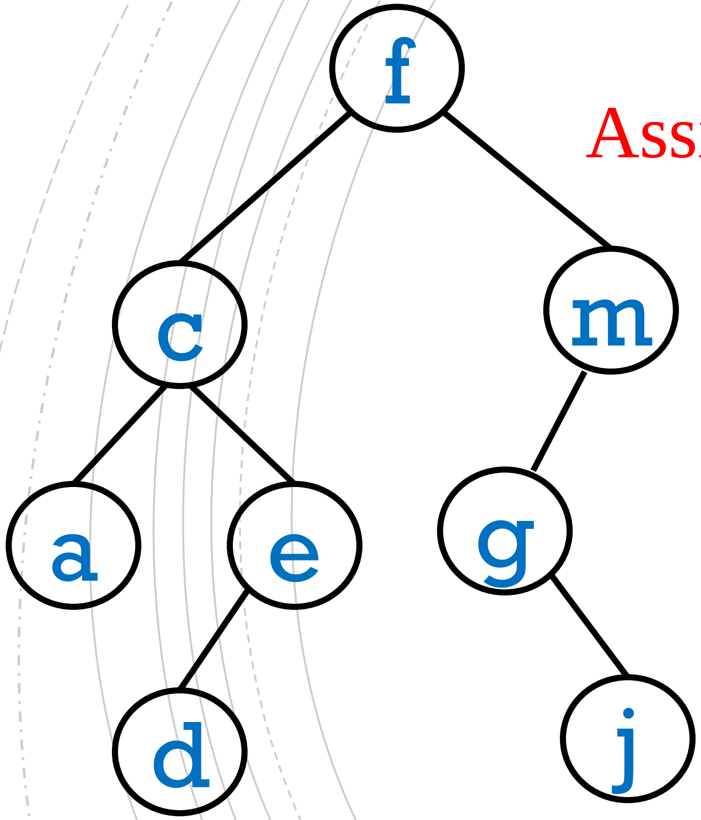
FIND()



Desired behaviour:

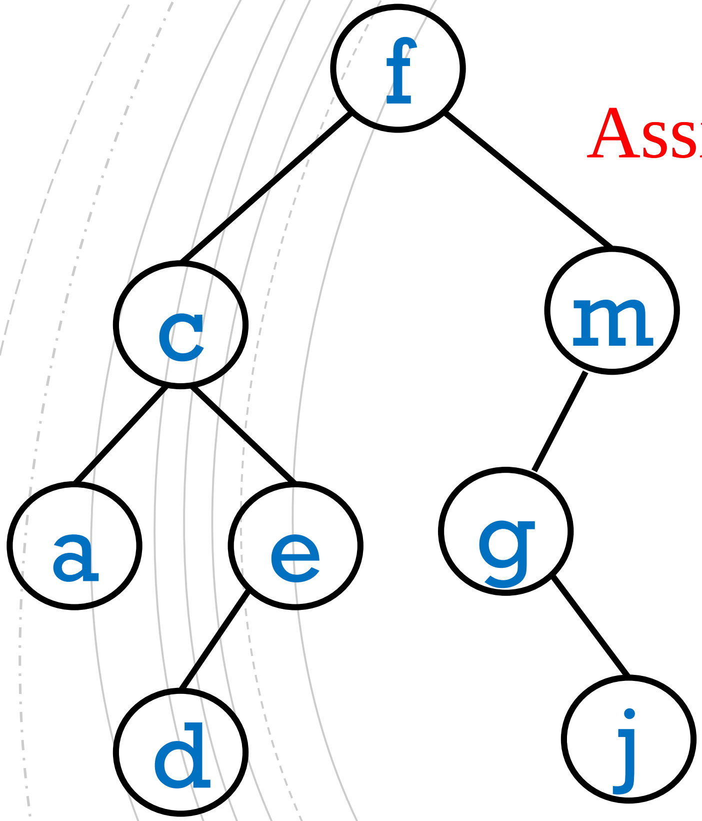
- `find(root, g)` returns the `g` node
- `find(root, s)` returns `null`.

FIND() – IMPLEMENTATION



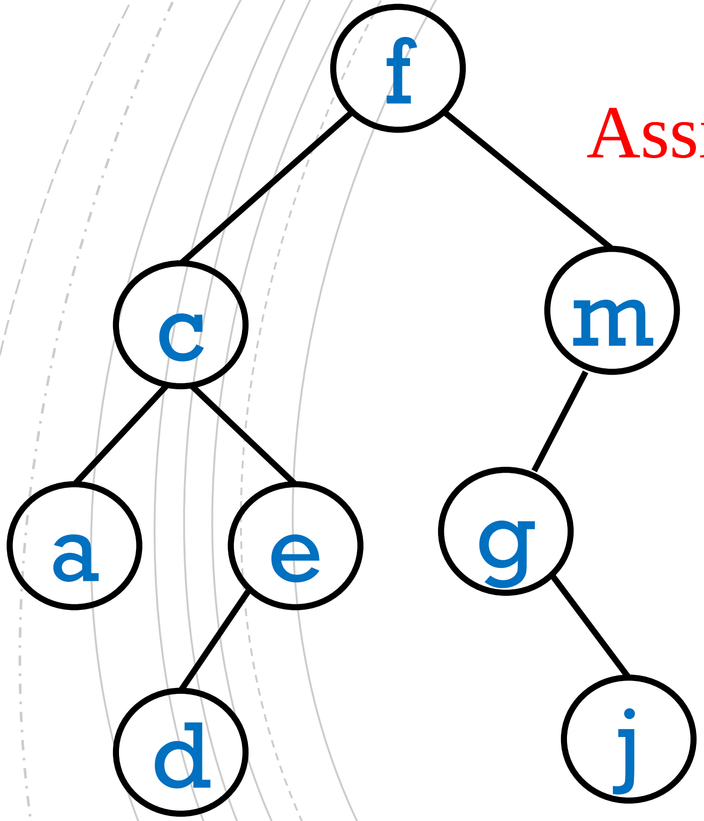
```
find(root, key){ // returns a node
    if (root == null)
            return null
    else if (root.key == key))
            return root



}
```

FIND() – IMPLEMENTATION



```
find(root, key){ // returns a node
    if (root == null)
        return null
    else if (root.key == key))
        return root
    else if (key < root.key)
        return find(root.left, key)
    else
        return find(root.right, key)
}
```

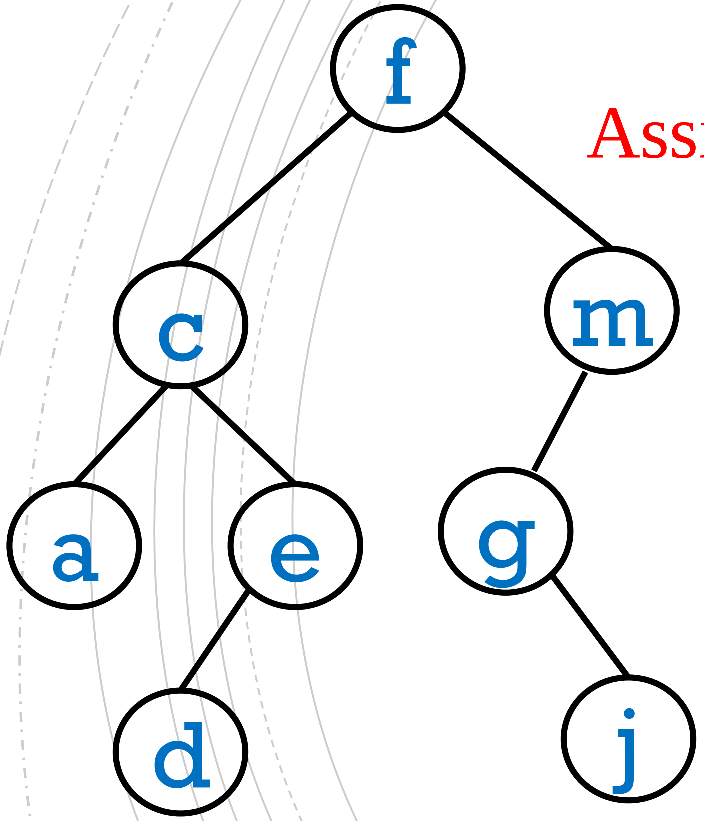
FINDMIN()



findMin() should return the node with the smallest key. So, for example given the BST on the left,

- `findMin(root)` returns … ?

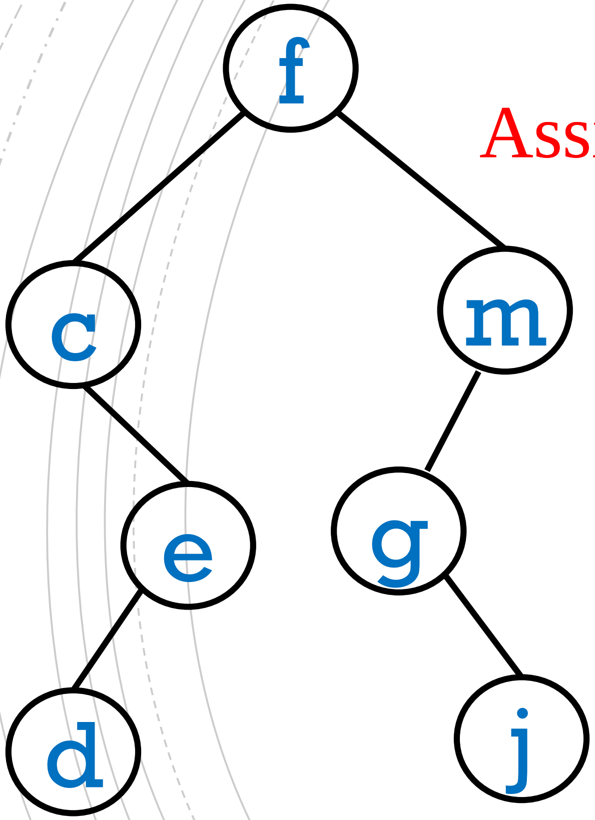
FINDMIN()



findMin() should return the node with the smallest key. So, for example given the BST on the left,

- `findMin(root)` returns the **a** node

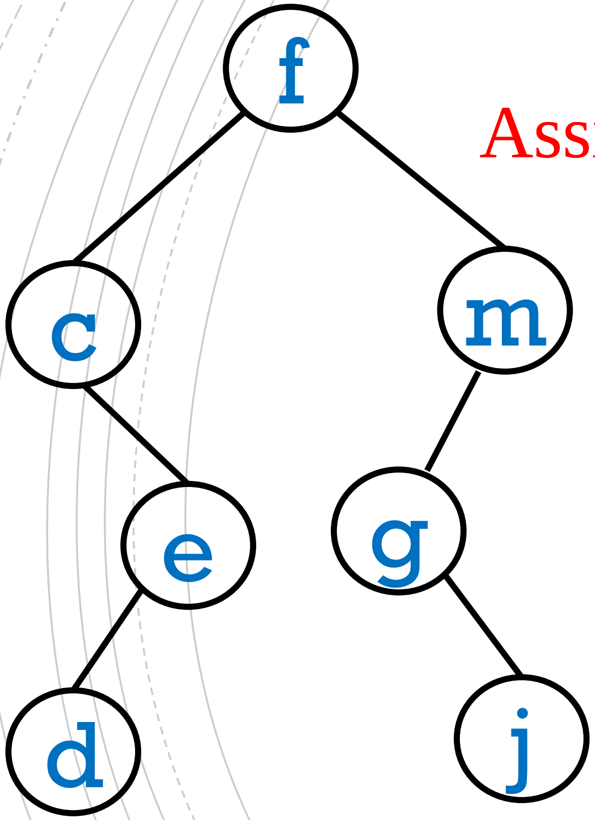
FINDMIN()



findMin() should return the node with the smallest key. So, for example given the BST on the left,

- `findMin(root)` returns … ?

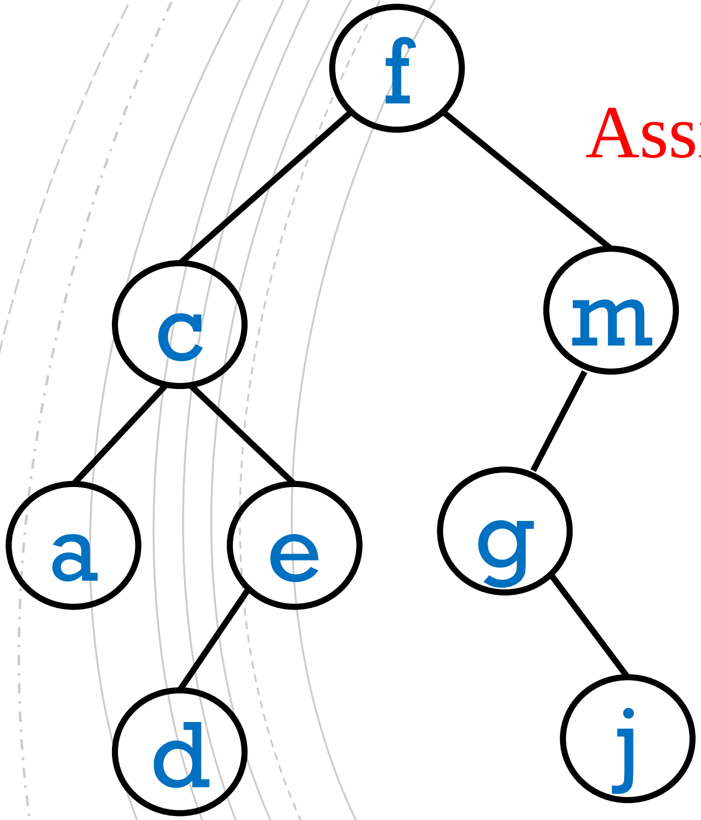
FINDMIN()



findMin() should return the node with the smallest key. So, for example given the BST on the left,

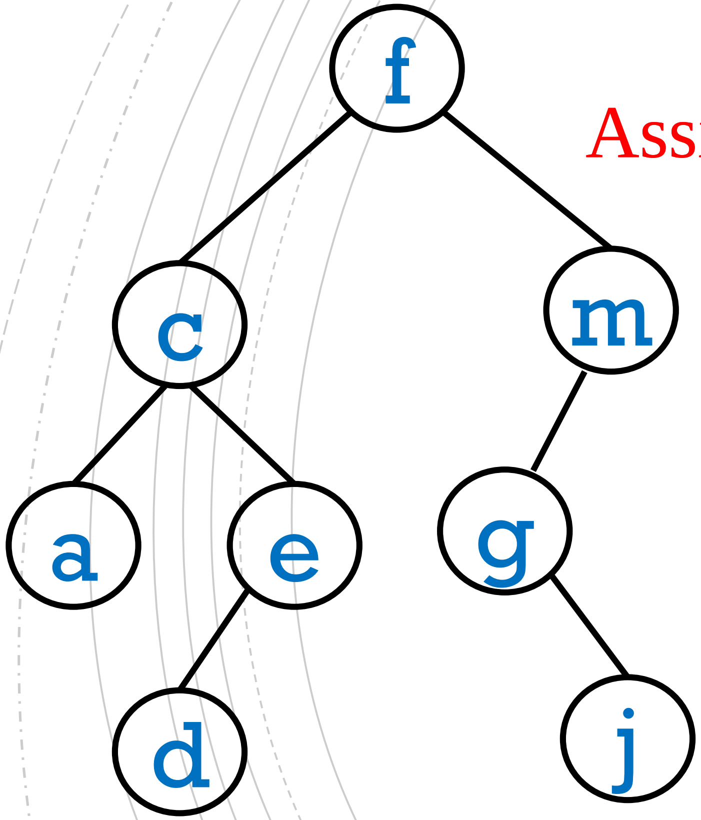
- `findMin(root)` returns the **c** node

FINDMIN() - IMPLEMENTATION



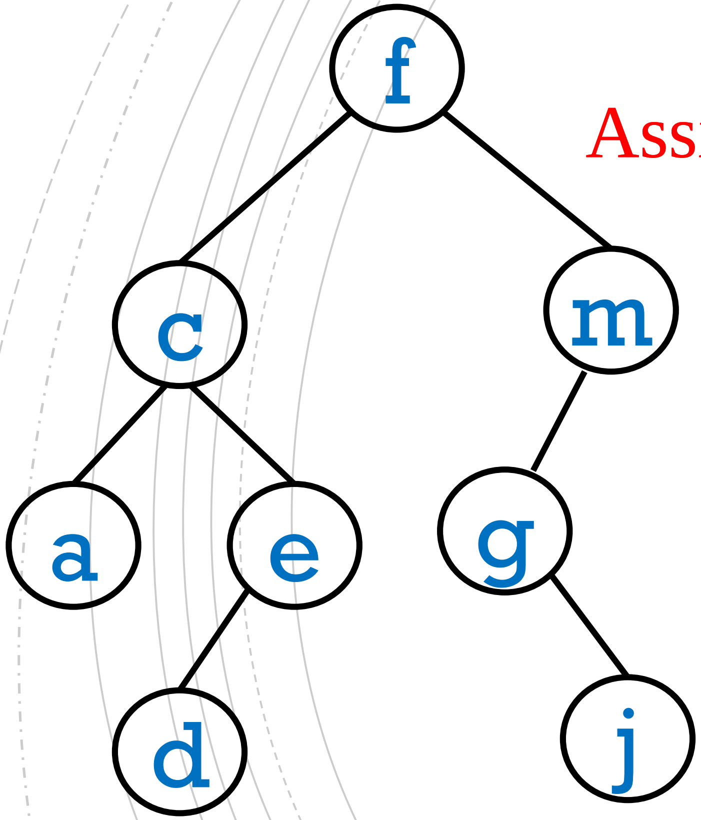
```
findMin(root){ // returns a node
        if (root == null)
            return null





}
```

FINDMIN() - IMPLEMENTATION



```
findMin(root){ // returns a node
        if (root == null)
                return null
        else if (root.left == null)
                return root
        else
                return findMin( root.left )
}
```

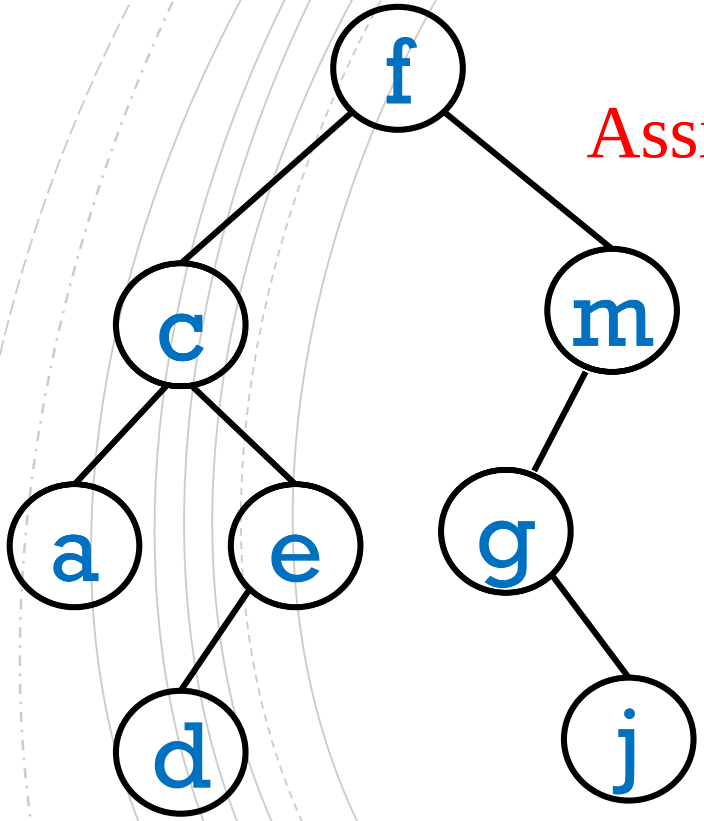
FINDMAX()



findMax() should return the node with the greatest key. So, for example given the BST on the left,

- `findMax(root)` **returns … ?**

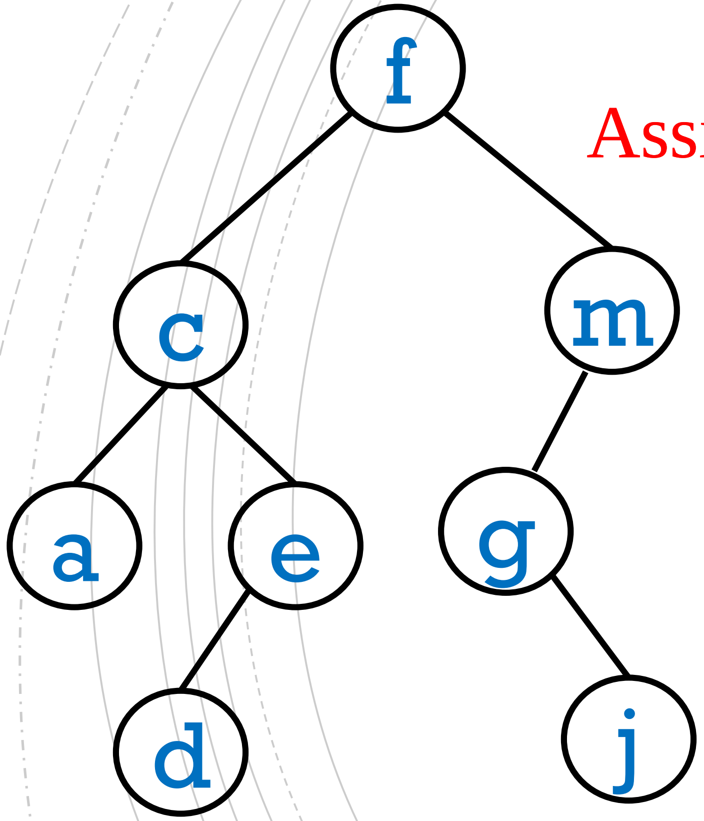
FINDMAX()



findMax() should return the node with the greatest key. So, for example given the BST on the left,

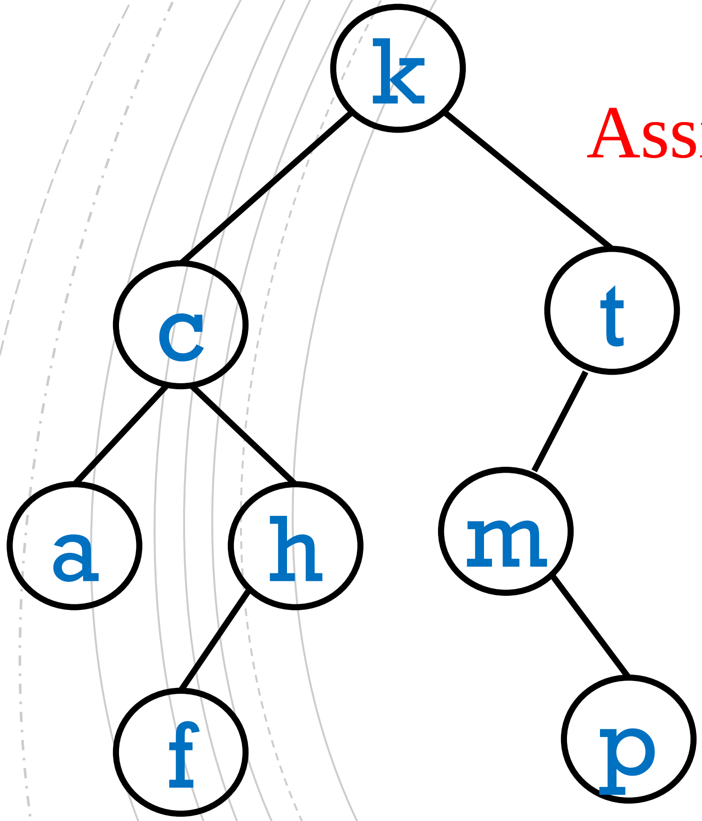
- `findMax(root)` returns the m node.

FINDMAX() – IMPLEMENTATION



```
findMax (root){ // returns a node
   if (root == null)
      return null
   else if (root.right == null))
      return root
   else
      return findMax (root.right)
}
```

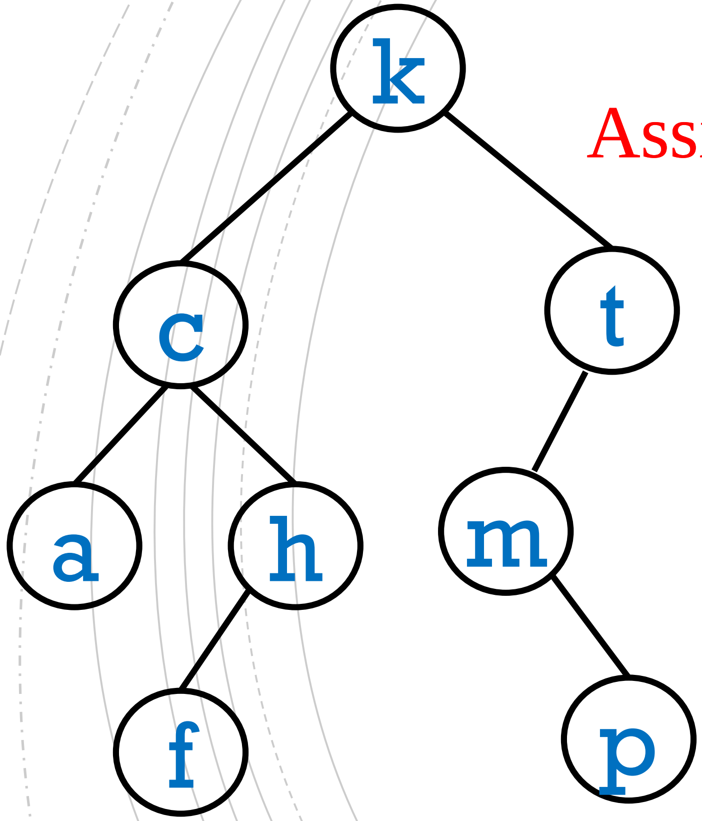
ADD()



add(key) should add a BSTNode to the tree.

- Where?
- How?

ADD()

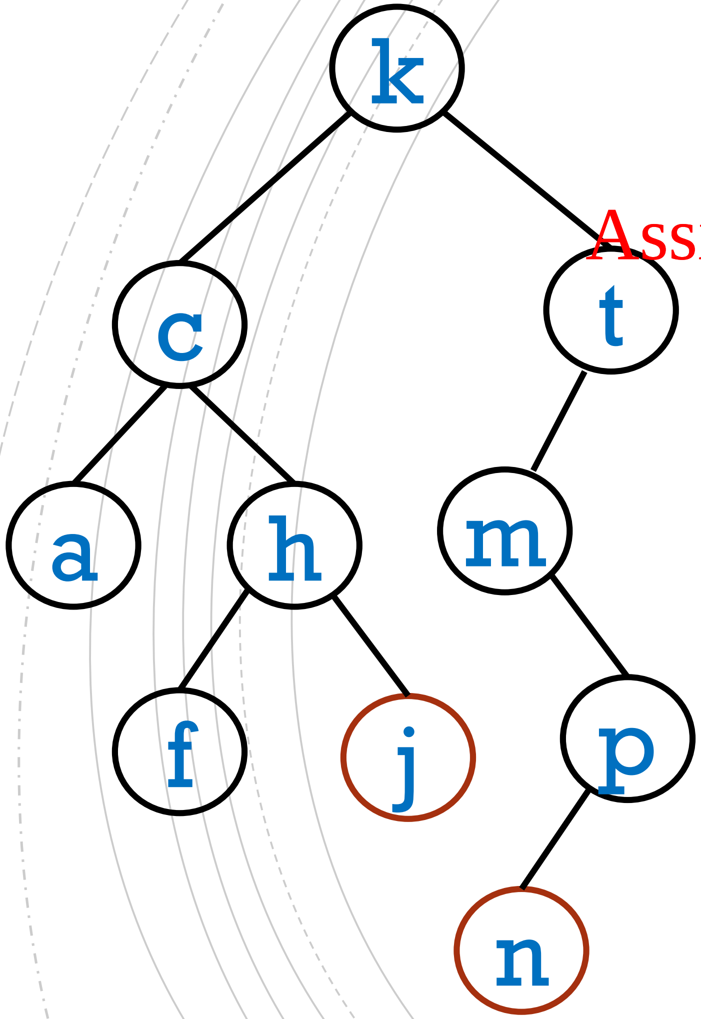


add(key) should add a BSTNode to the tree.

- `add(j)` ?
- `add(n)` ?

A new node is always a leaf.

ADD()

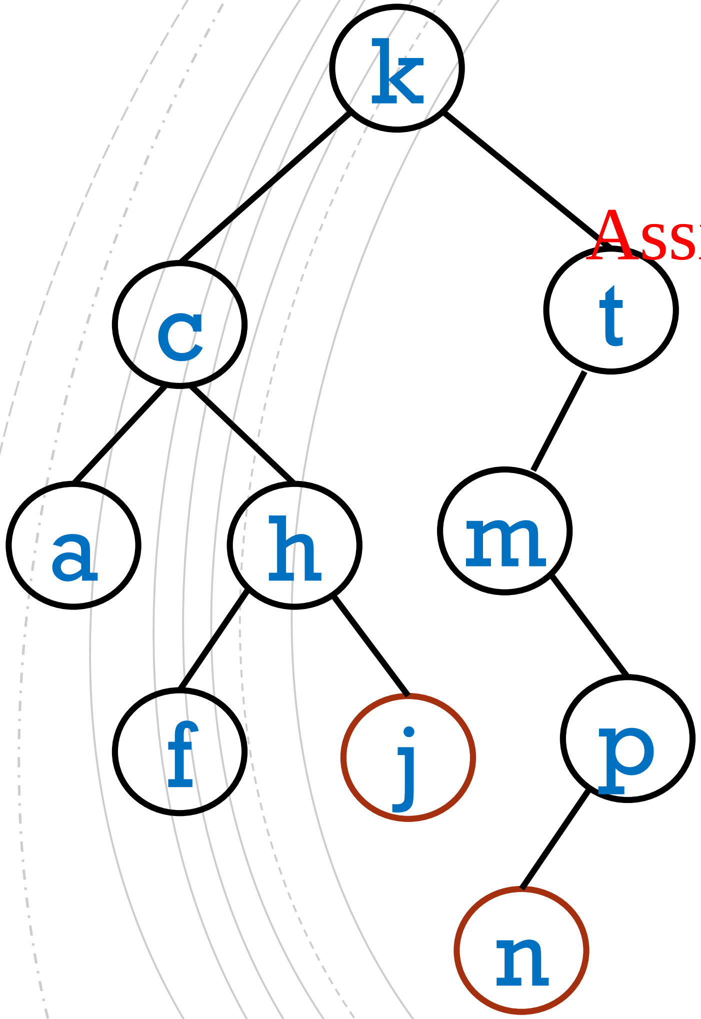


add(key) should add a BSTNode to the tree.

- `add(j)` ?
- `add(n)` ?

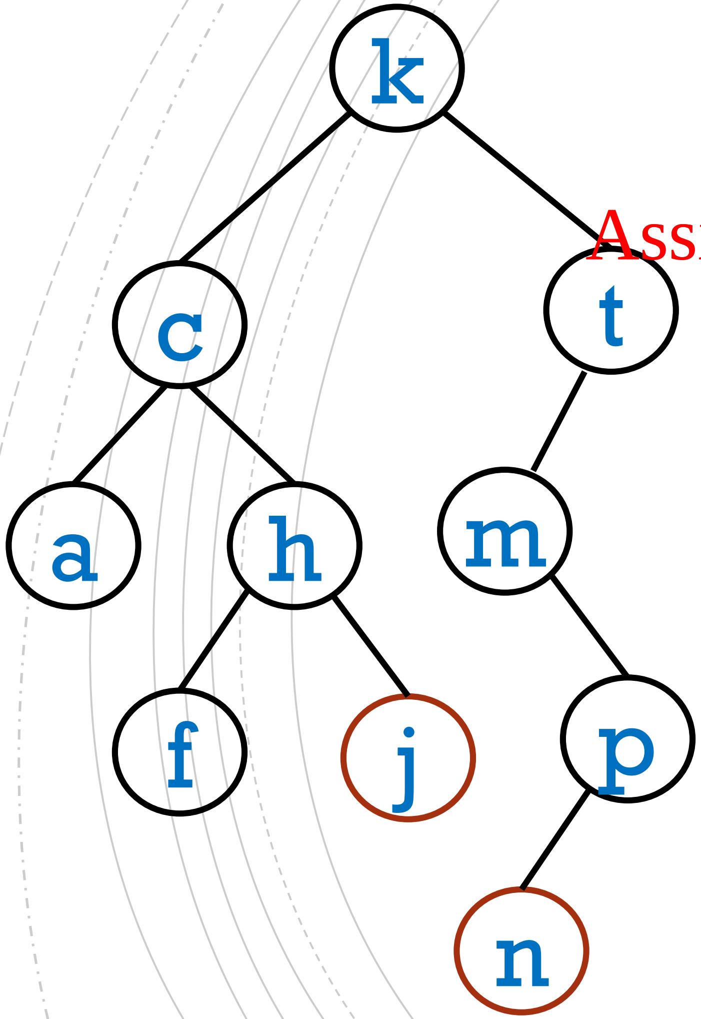
A new node is always a leaf.

ADD() - IMPLEMENTATION



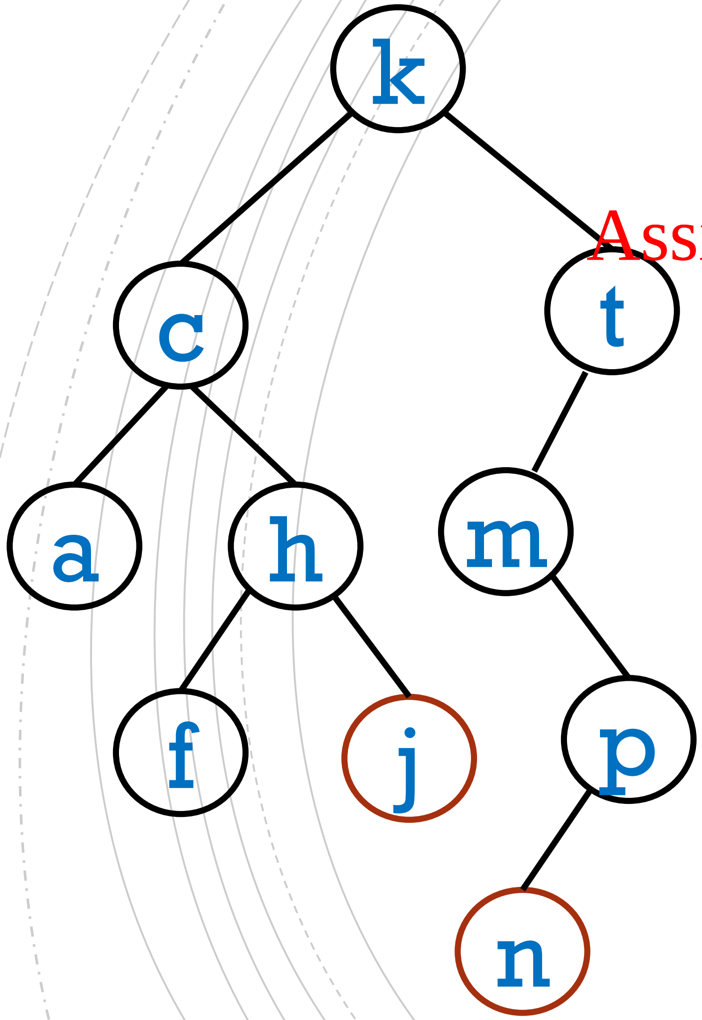
```
add(root, key){ // returns root node




    return root
}
```

ADD() - IMPLEMENTATION



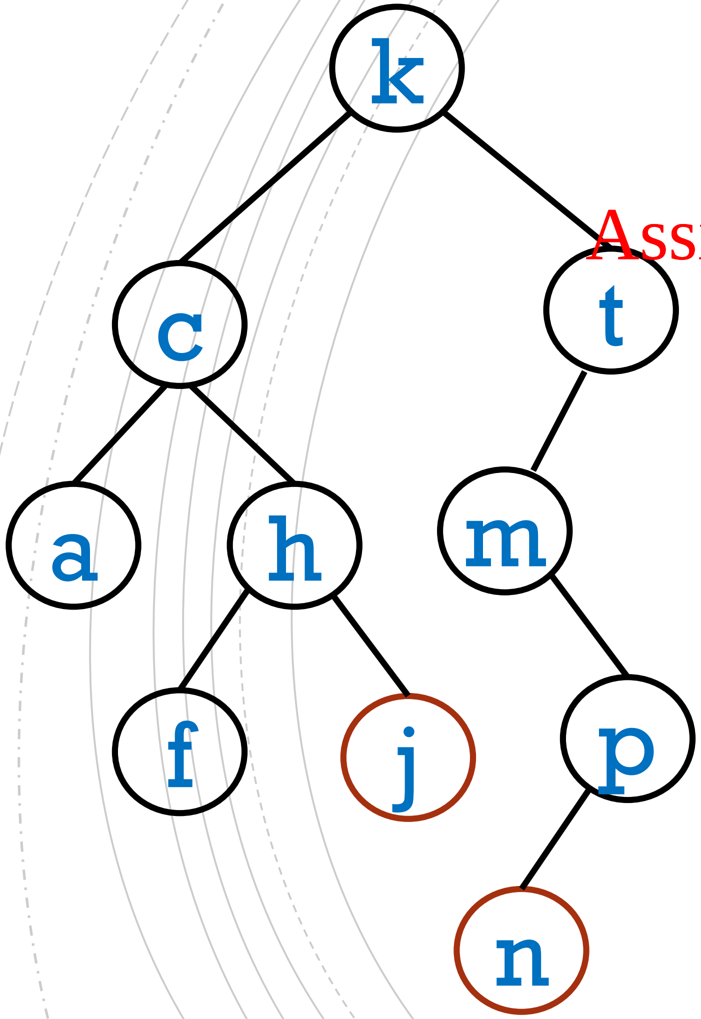
```
add(root, key){ // returns root node
    if (root == null)
        root = new BSTnode(key)




    return root
}
```

ADD() - IMPLEMENTATION



```
add(root, key){ // returns root node
    if (root == null)
        root = new BSTnode(key)
    else if (key < root.key){
        root.left =  add(root.left,key)



    return root
}
```

ADD() - IMPLEMENTATION

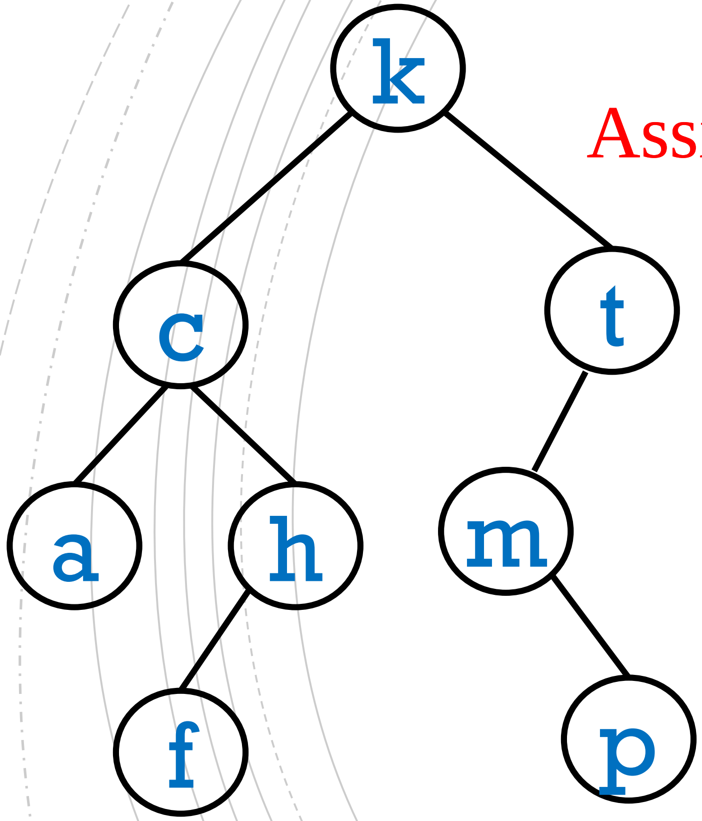


```
add(root, key){ // returns root node
    if (root == null)
        root = new BSTnode(key)
    else if (key < root.key){
        root.left =  add(root.left,key)
    else if (key > root.key){
        root.right =  add(root.right,key)
    return root
}
```

Q: What happens if root.key == key?
A: Nothing!

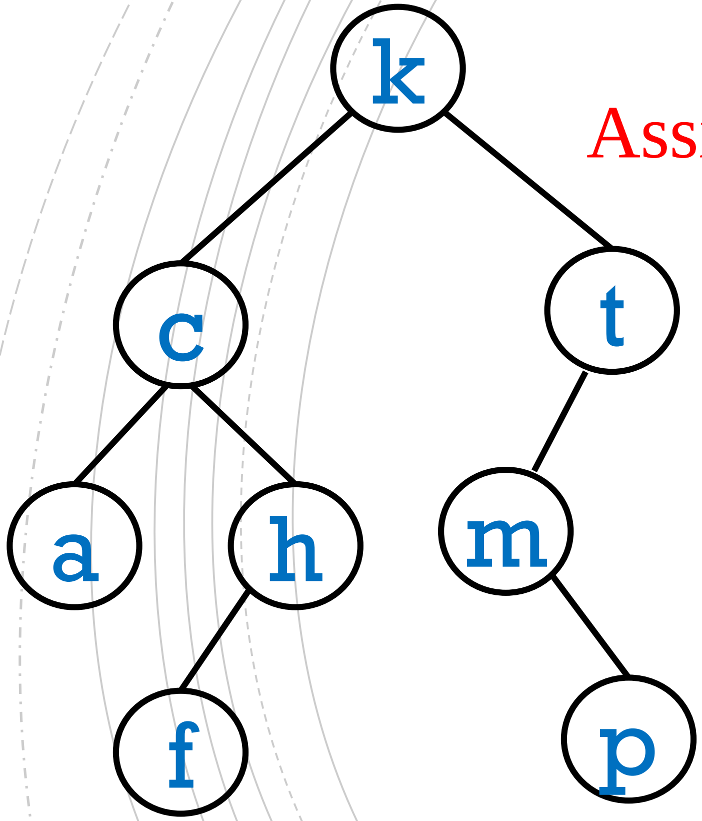
REMOVE()

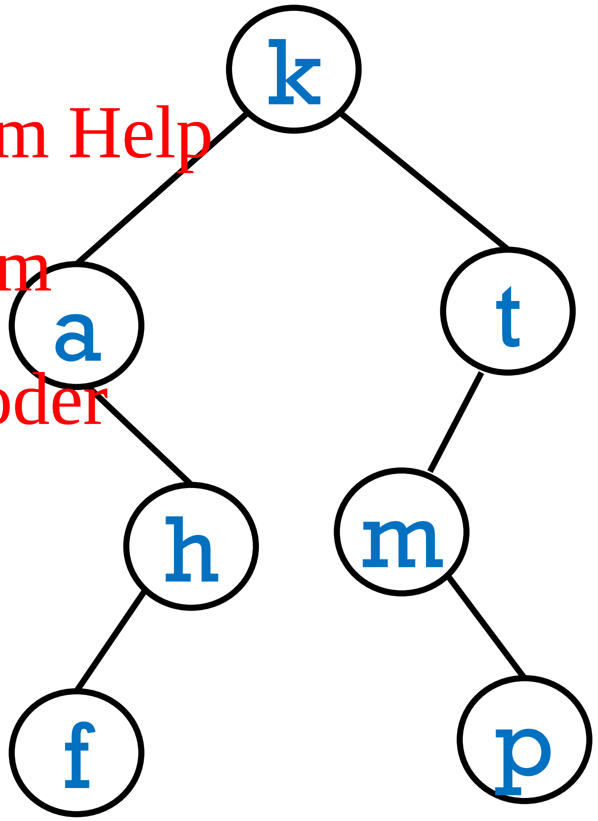
`remove(c) →`



REMOVE()

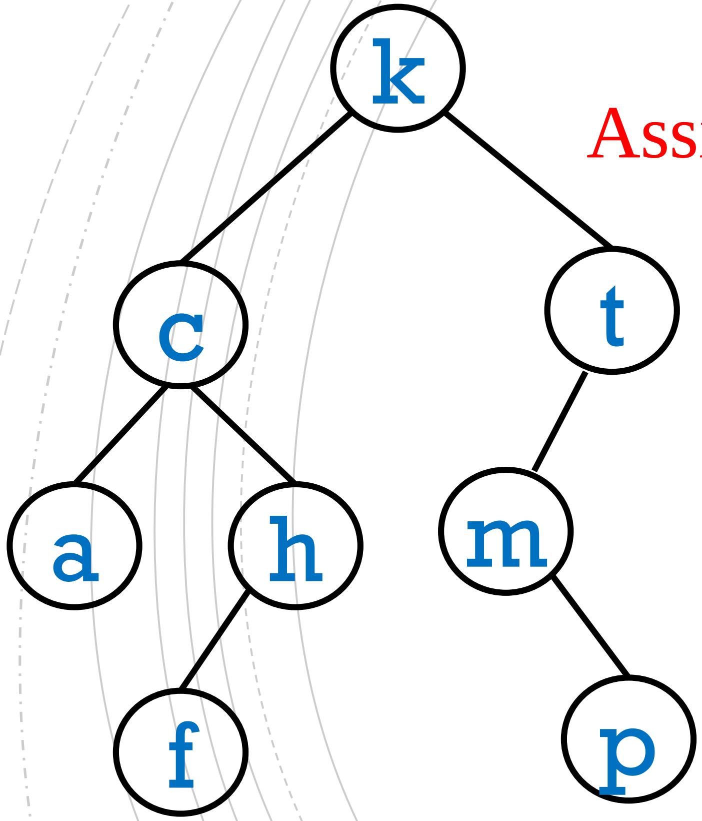
`remove(c)` → this is one way to do it

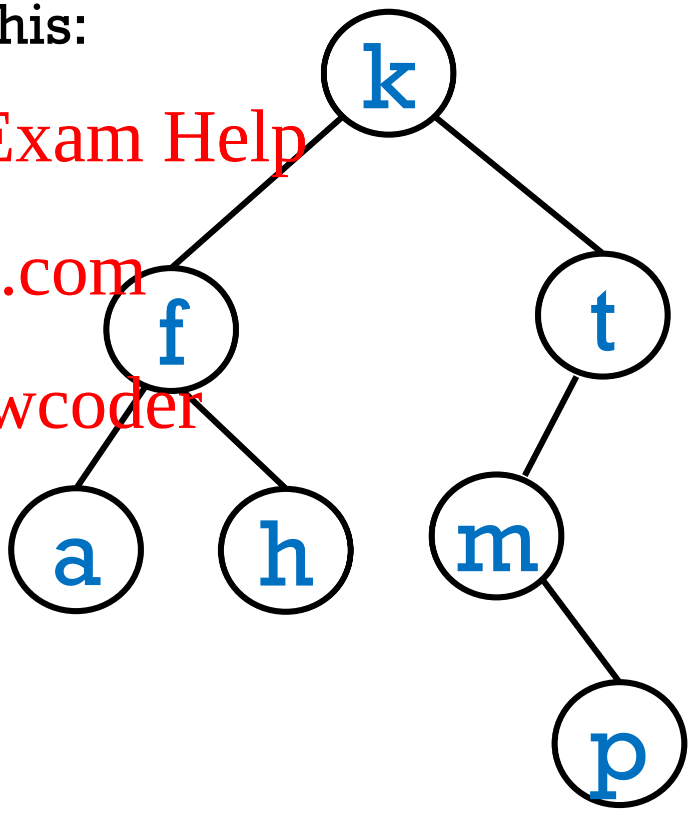




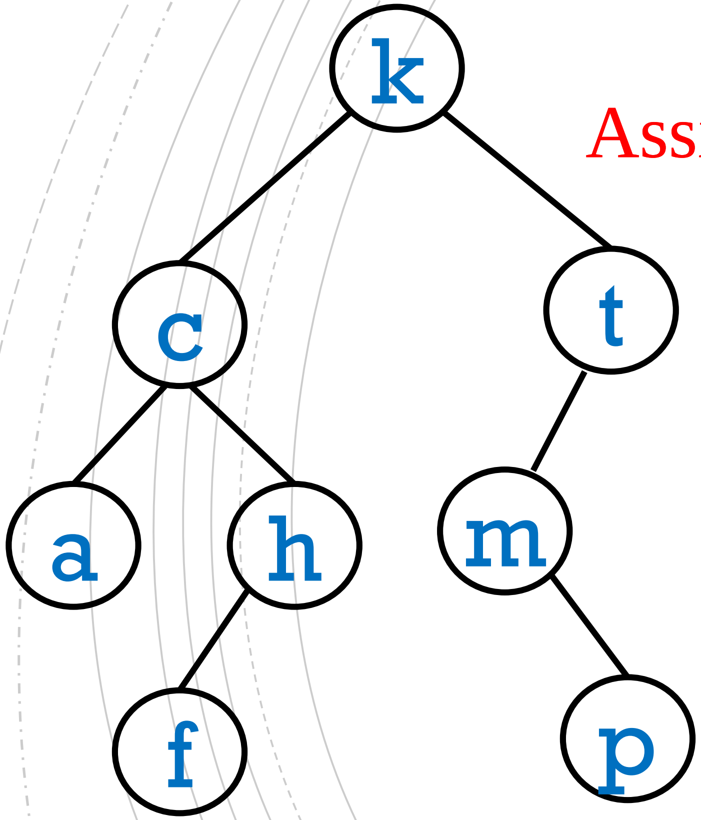
REMOVE()

`remove(c)` → the following algorithm does this:



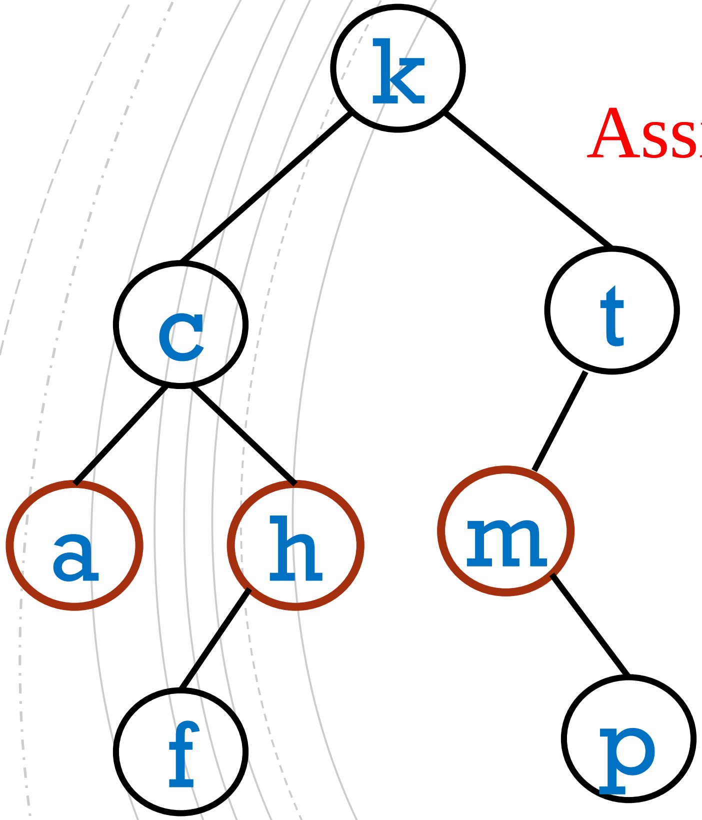


REMOVE() - IMPLEMENTATION



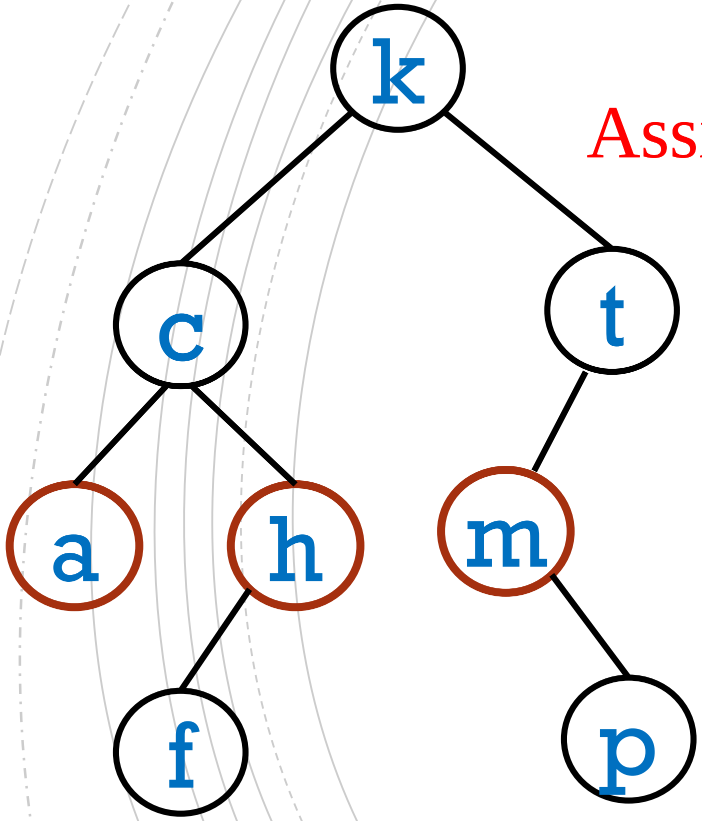
```
remove(root, key){ // returns root node
    if( root  == null )
         return null
    else if ( key < root.key )

    else if ( key > root.key )

    else



    }
    return root
}
```

REMOVE() - IMPLEMENTATION



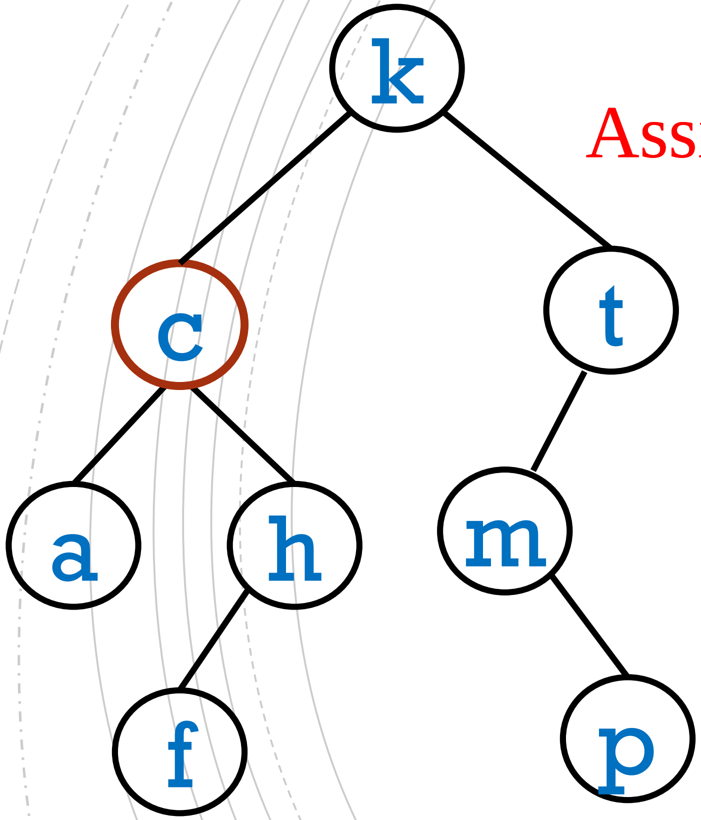
```
remove(root, key){ // returns root node
    if( root  == null )
         return null
    else if ( key < root.key )
         root.left = remove(root.left, key)
    else if ( key > root.key )
         root.right = remove(root.right, key)
    else





    }
    return root
}
```

REMOVE() - IMPLEMENTATION



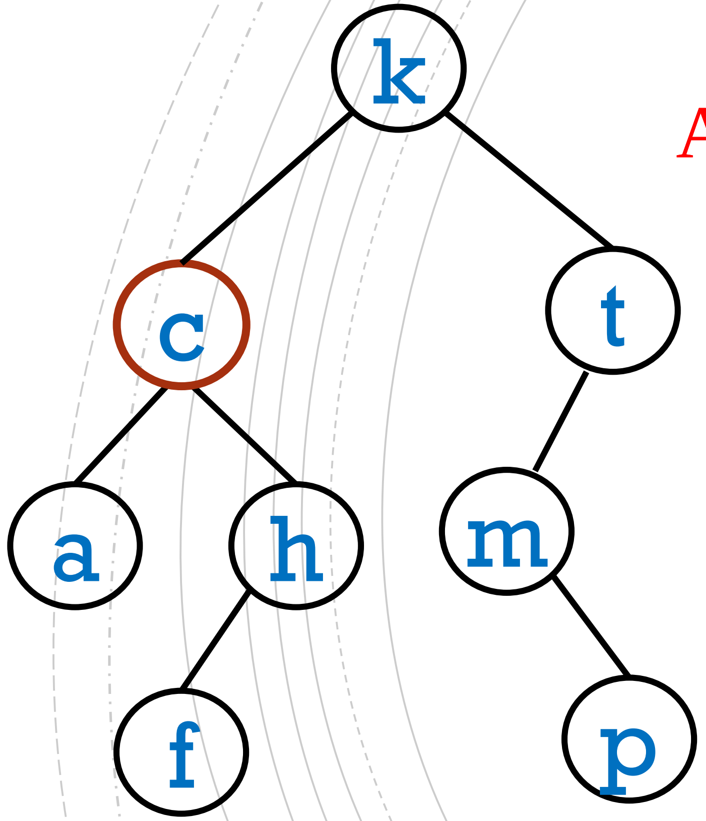
```
remove(root, key){ // returns root node
    if( root  == null )
        return null
    else if ( key < root.key )
        root.left = remove(root.left, key)
    else if ( key > root.key )
        root.right = remove(root.right, key)
    else if (root.left == null)
        root = root.right
    else if (root.right == null)
        root = root.left



    }
    return root
}
```

REMOVE() - IMPLEMENTATION



```
remove(root, key){ // returns root node
    if( root  == null )
        return null
    else if ( key < root.key )
        root.left = remove(root.left, key)
    else if ( key > root.key )
        root.right = remove(root.right, key)
    else if (root.left == null)
        root = root.right
    else if (root.right == null)
        root = root.left




    }
    return root
}
```

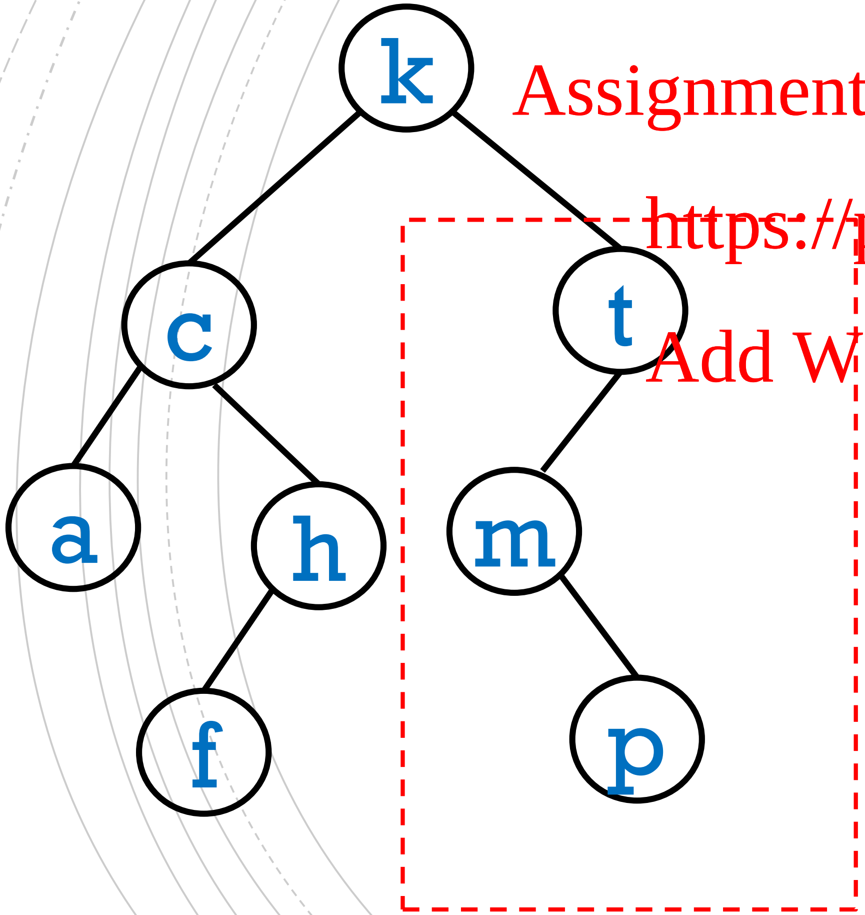
REMOVE() - IMPLEMENTATION



```
remove(root, key){ // returns root node
    if( root  == null )
        return null
    else if ( key < root.key )
        root.left = remove(root.left, key)
    else if ( key > root.key )
        root.right = remove(root.right, key)
    else if (root.left == null)
        root = root.right
    else if (root.right == null)
        root = root.left
    else {
        root.key = findMin(root.right).key
        root.right = remove(root.right, root.key)
    }
    return root
}
```

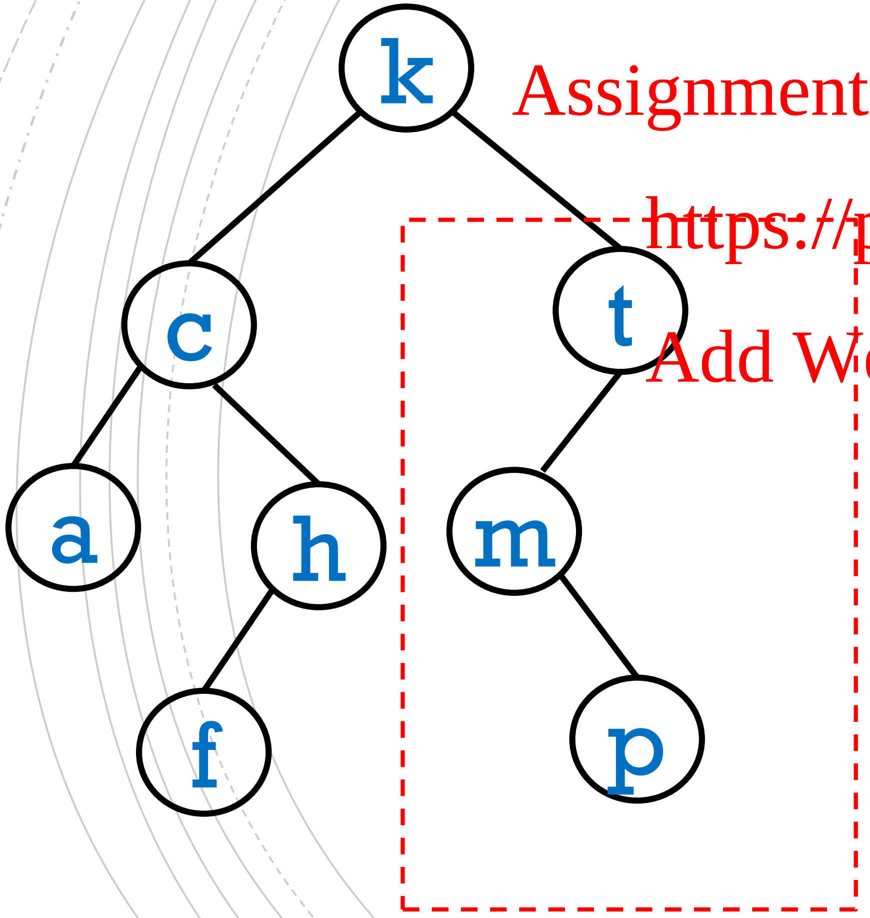
REMOVE() - EXAMPLE

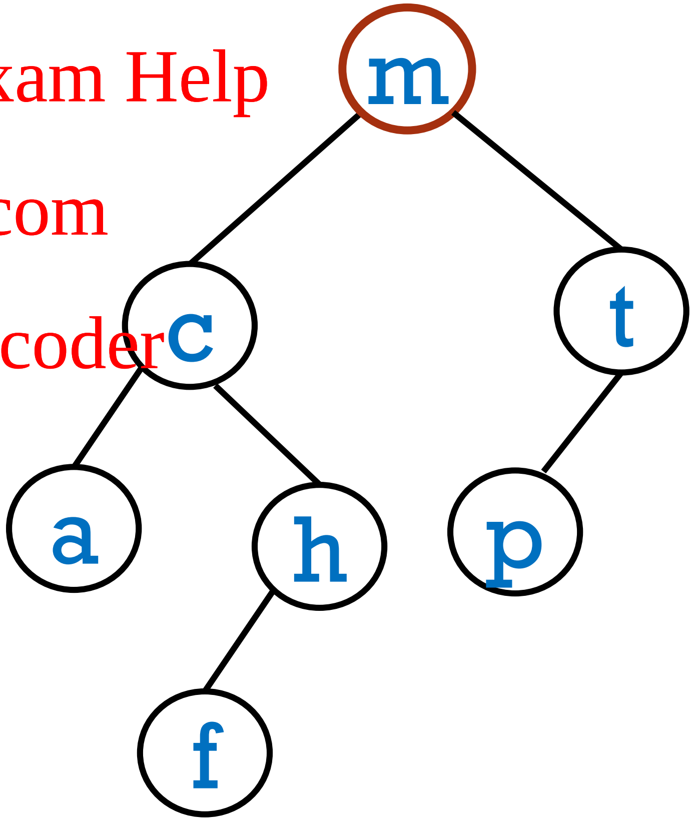
`remove(k)` →



REMOVE() - EXAMPLE

`remove(k)` →





BALANCED VS UNBALANCED

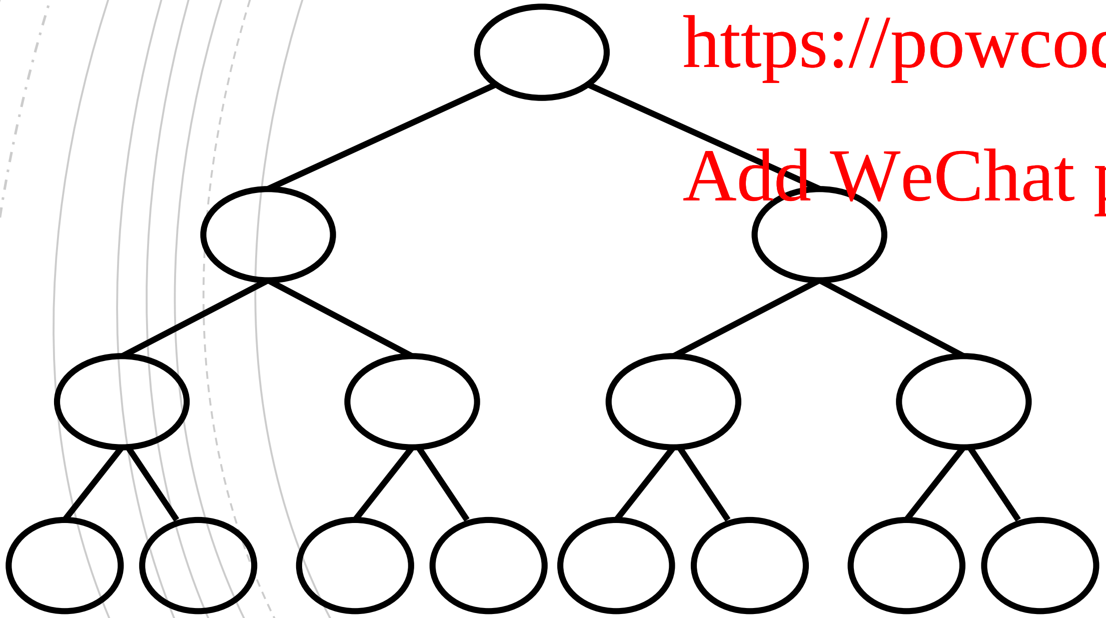
balanced

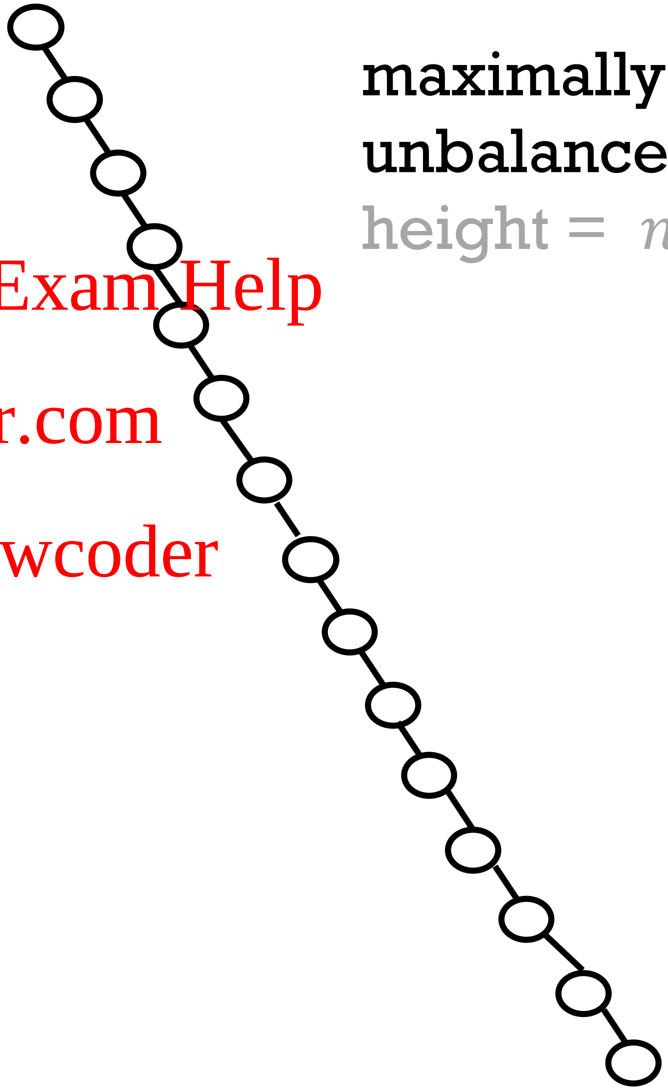
height = $\log(n+1) - 1$

$n = 2^{h+1} - 1$

maximally unbalanced

height = $n - 1$



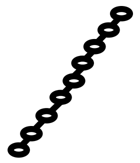


BEST VS WORST CASE SCENARIO

	best case	worst case
findMin()		
findMax()		
find(key)		
add(key)		
remove(key)		

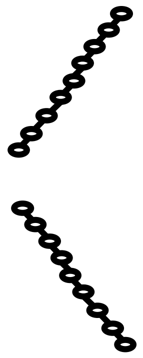
BEST VS WORST CASE SCENARIO

	best case	worst case
findMin()	$O(1)$	$O(n)$
findMax()		
find(key)		
add(key)		
remove(key)		

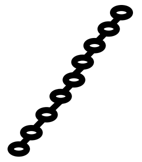
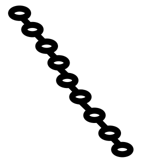


BEST VS WORST CASE SCENARIO

	best case	worst case
findMin()	$O(1)$	$O(n)$
findMax()	$O(1)$	$O(n)$
find(key)		
add(key)		
remove(key)		

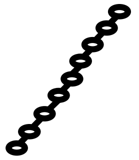


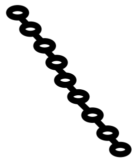
BEST VS WORST CASE SCENARIO

	best case	worst case	
findMin()	$O(1)$	$O(n)$	
findMax()	$O(1)$	$O(n)$	
find(key)	$O(1)$	$O(n)$	could be zigzag
add(key)			
remove(key)			

BEST VS WORST CASE SCENARIO

	best case	worst case
findMin()	$O(1)$	$O(n)$
findMax()	$O(1)$	$O(n)$
find(key)	$O(1)$	$O(n)$
add(key)	$O(1)$	$O(n)$
remove(key)	$O(1)$	$O(n)$

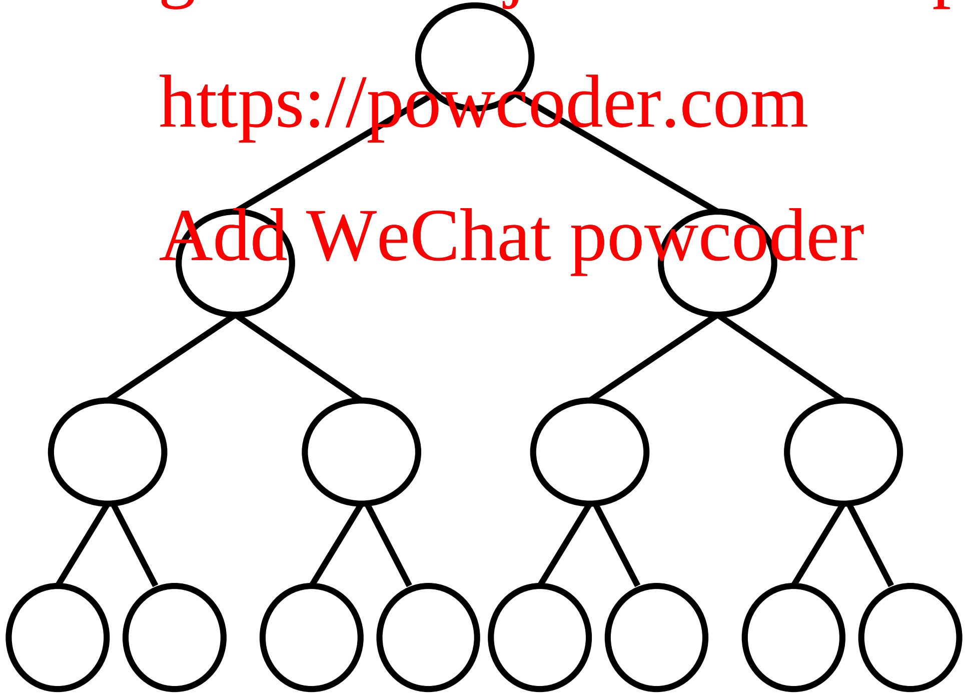




Could be zigzag

BINARY SEARCH (TREES)

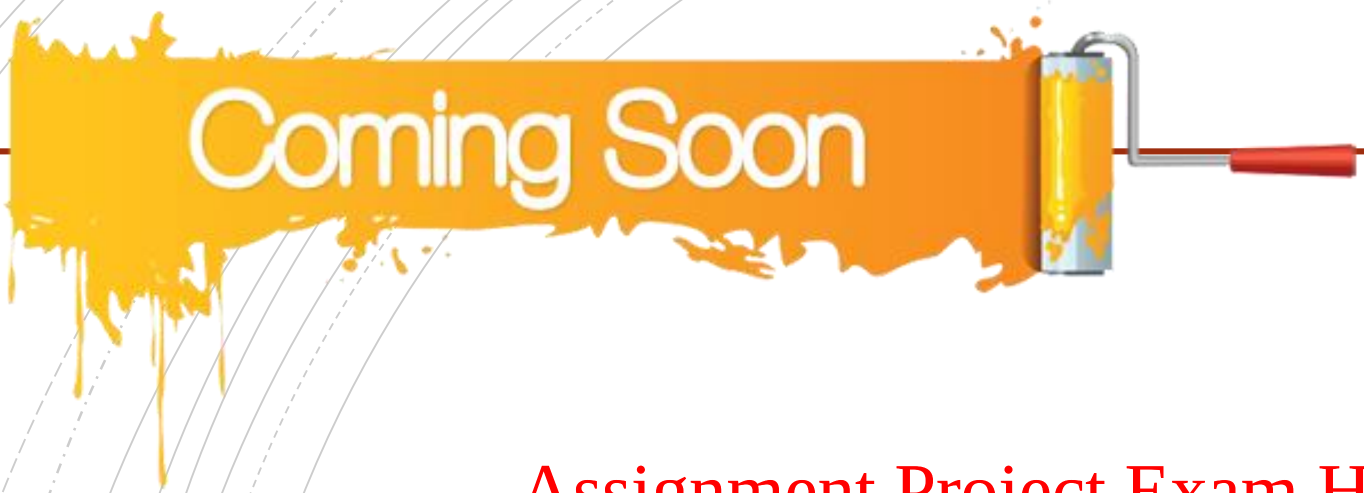
When a binary search tree is balanced, then finding a key is very similar to a binary search.



BALANCED BINARY SEARCH TREES

(COMP 251: AVL TREES, RED-BLACK TREES)

	best case	worst case
findMin()	$O(\log n)$	$O(\log n)$
findMax()	$O(\log n)$	$O(\log n)$
find(key)	$O(1)$	$O(\log n)$
add(key)	$O(\log n)$	$O(\log n)$
remove(key)	$O(\log n)$	$O(\log n)$



In the next videos:

- Heaps