

COMP 250

Lecture 22

binary search trees

Oct. 30, 2017

~~(binary search) tree~~

binary (search tree)

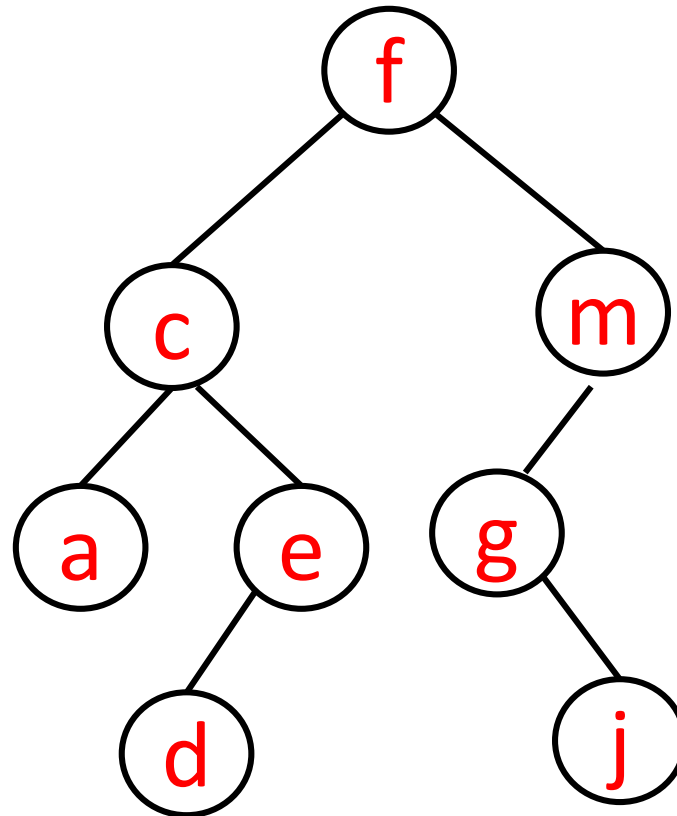
```
class BSTNode< K >{  
    K          key;  
    BSTNode< K > leftchild;  
    BSTNode< K > rightchild;  
    :  
}
```

The keys are “comparable” <, =, >
e.g. numbers, strings.

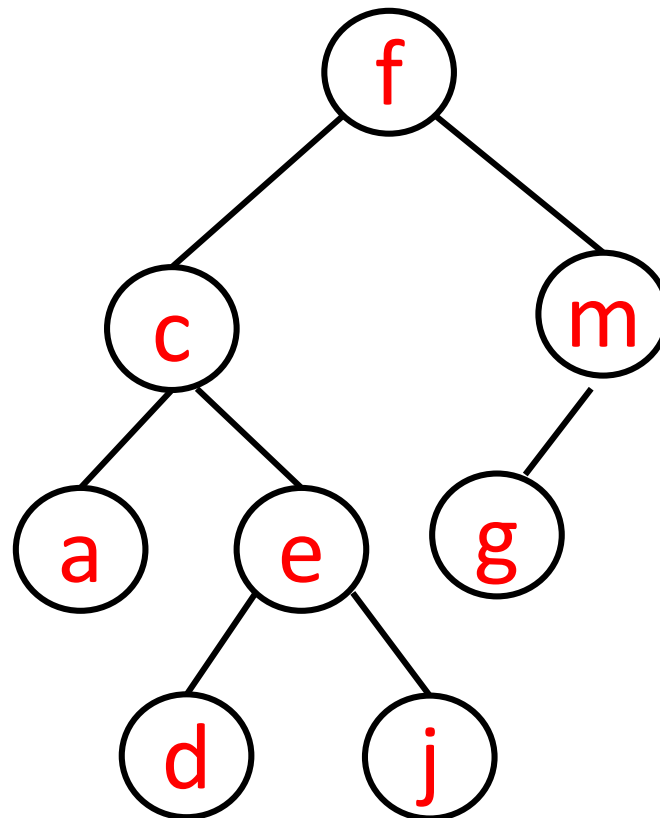
Binary Search Tree Definition

- binary tree
- keys are comparable, unique (no duplicates)
- for each node, all descendants in left subtree are less than the node, and all descendants in the node's right subtree are greater than the node
(comparison is based on node key)

Example

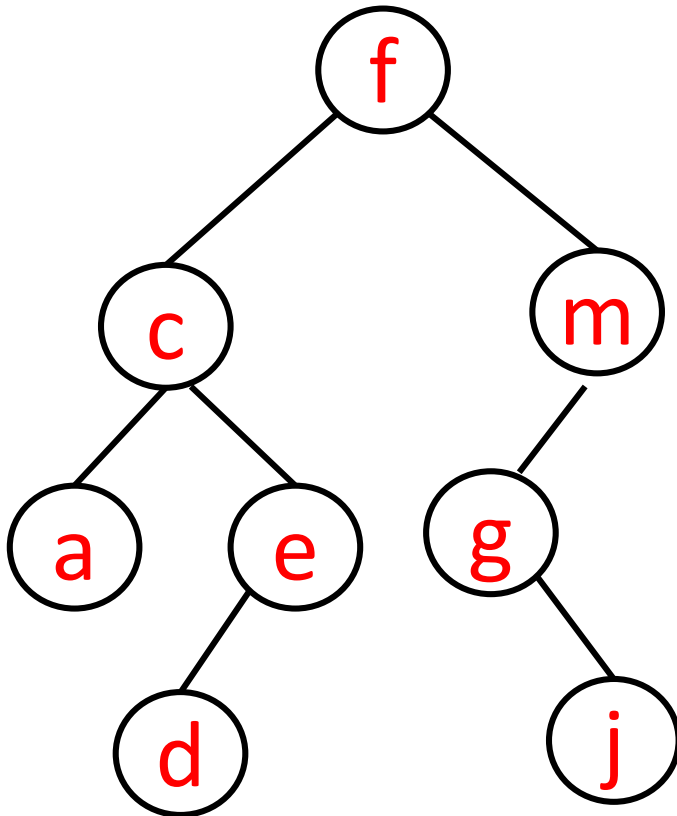


This is not a BST. Why not?



Claim: An in-order traversal on a BST visits the nodes in order.

Proof: Exercise



acdefgjm

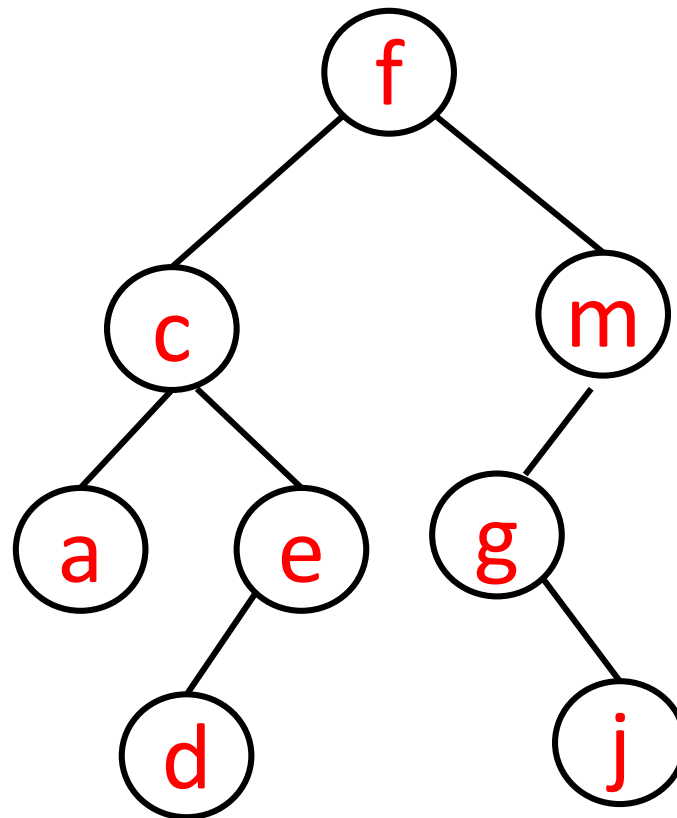
Binary Search Tree ADT

- 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(root, **g**) returns **g** node
find(root, **s**) returns null



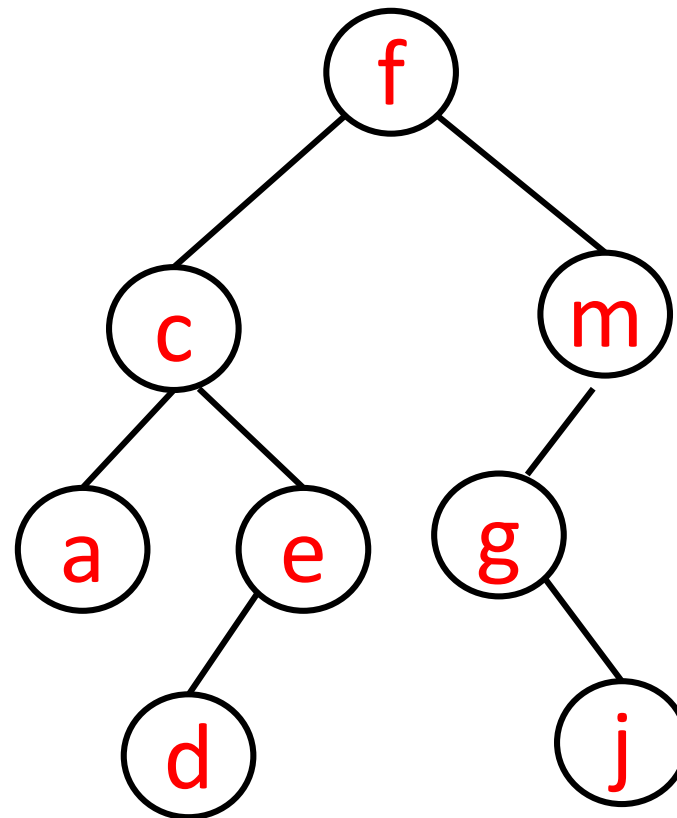
```
find(root, key){  
    if (root == null)  
        return null  
    else if (root.key == key)  
        return root
```

// returns a node

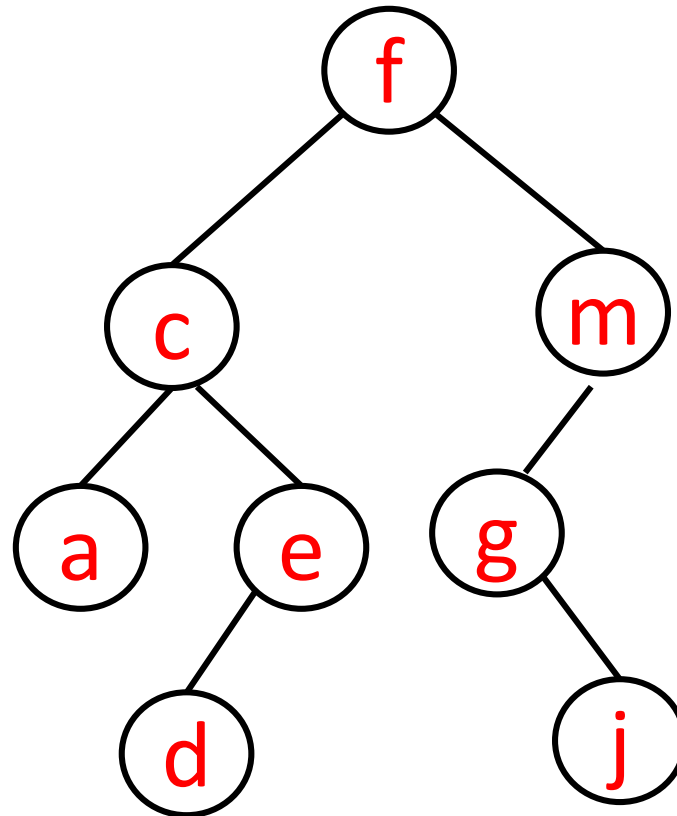
```
}
```

```
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)
}
```

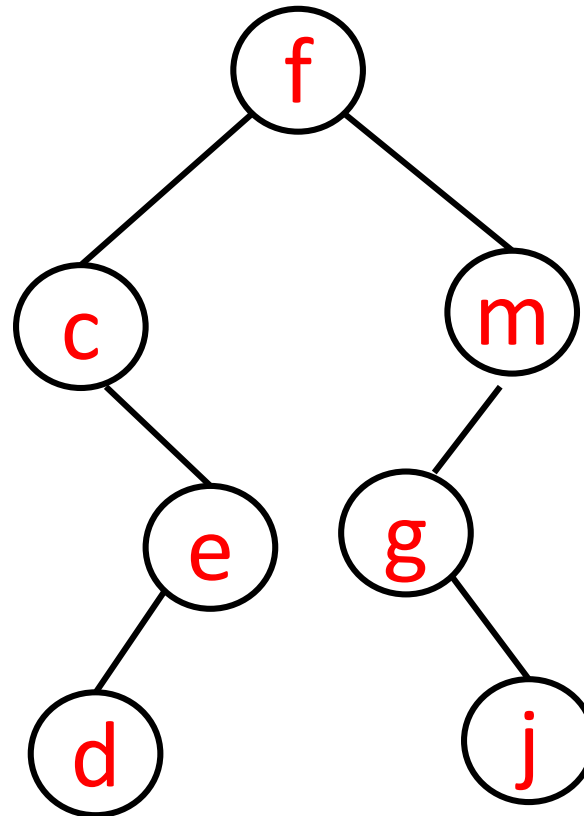
findMin() returns



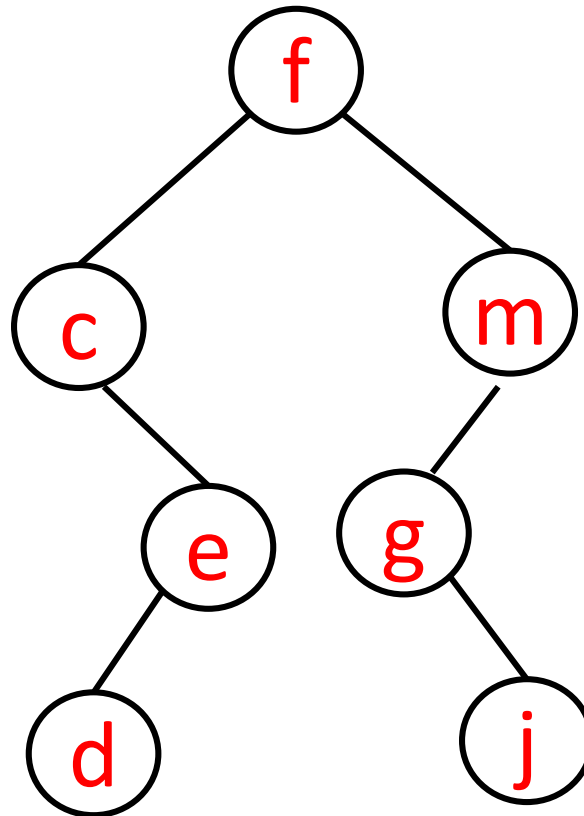
findMin() returns **a** (node)



findMin() returns



findMin() returns **c** node




```
findMin(root){  
    if (root == null)  
        return null
```

// returns a node

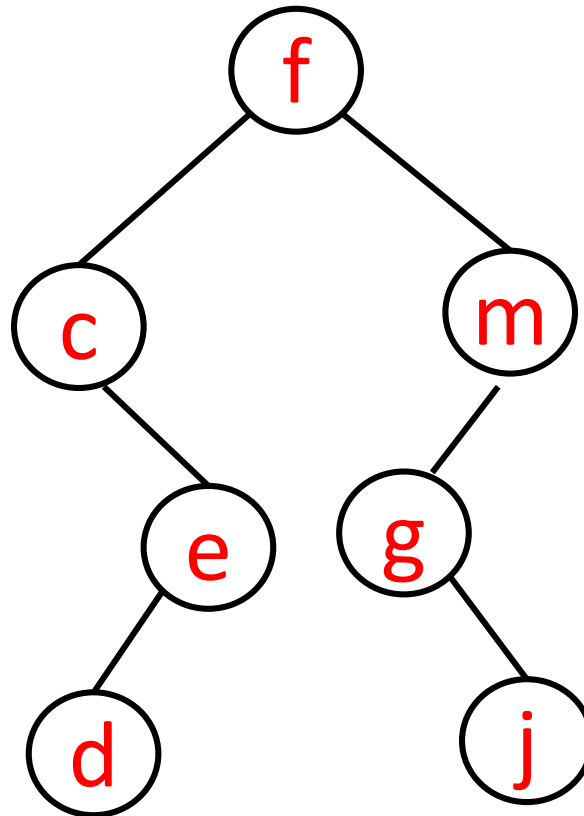
```
}
```



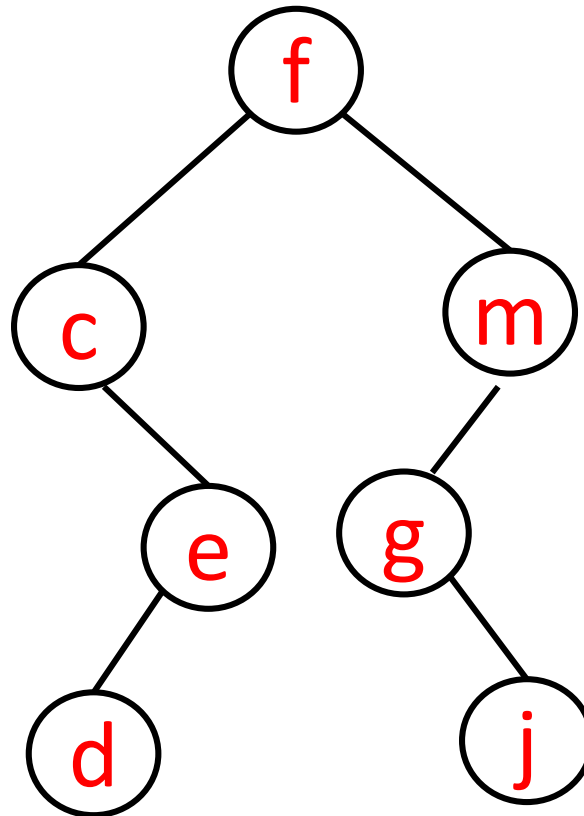
```
findMin(root){                                     // returns a node
    if (root == null)
        return null
    else if (root.left == null)
        return root
    
}
```

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

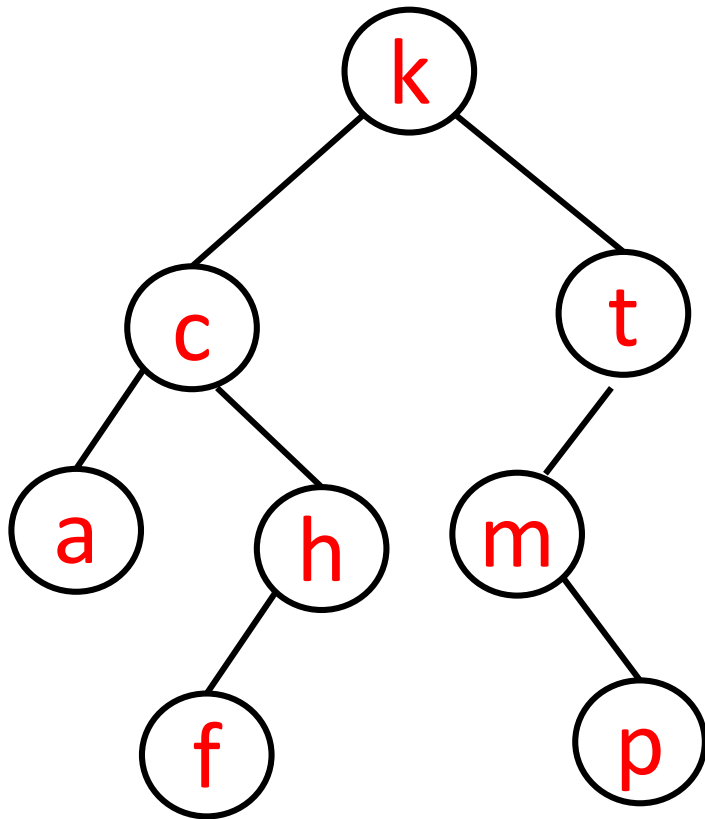
findMax() returns ?



findMax() returns **m** node



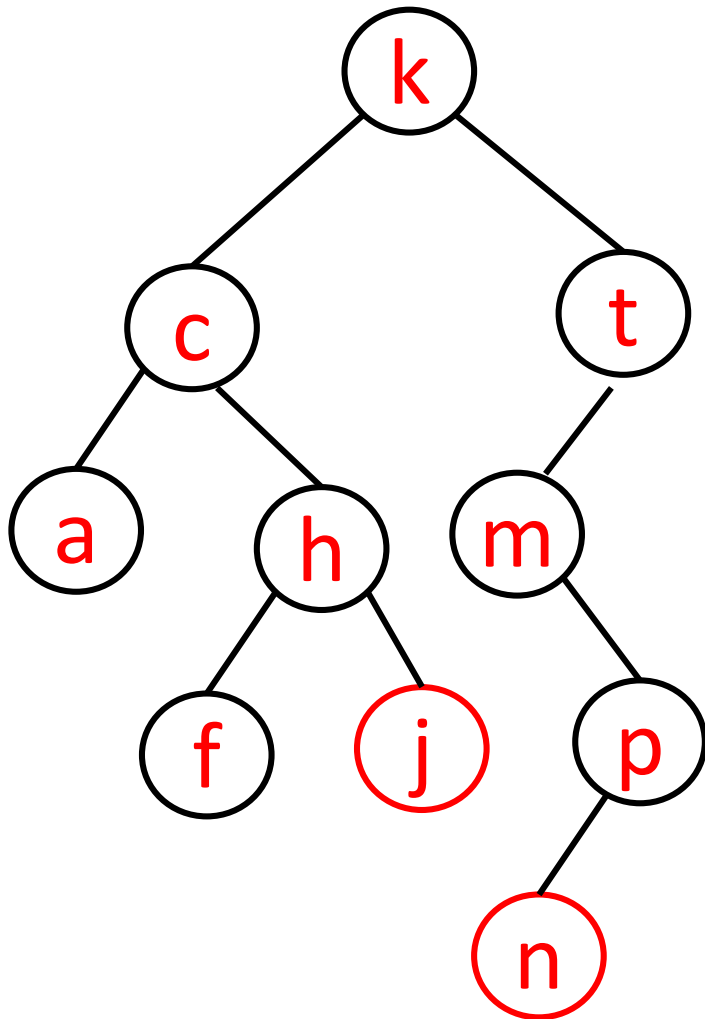
```
findMax(root){                                // returns a node
    if (root == null)
        return null
    else if (root.right == null)
        return root
    else
        return findMax (root.right)
}
```



add(**j**) ?

add(**n**) ?

A new key is
always a leaf.



add(**j**) ?

add(**n**) ?

A new key is
always a leaf.

```
add(root, key){
```


```
// returns root node
```




```
    return root
```

```
}
```



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

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

    return root
}
```

```
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
}
```

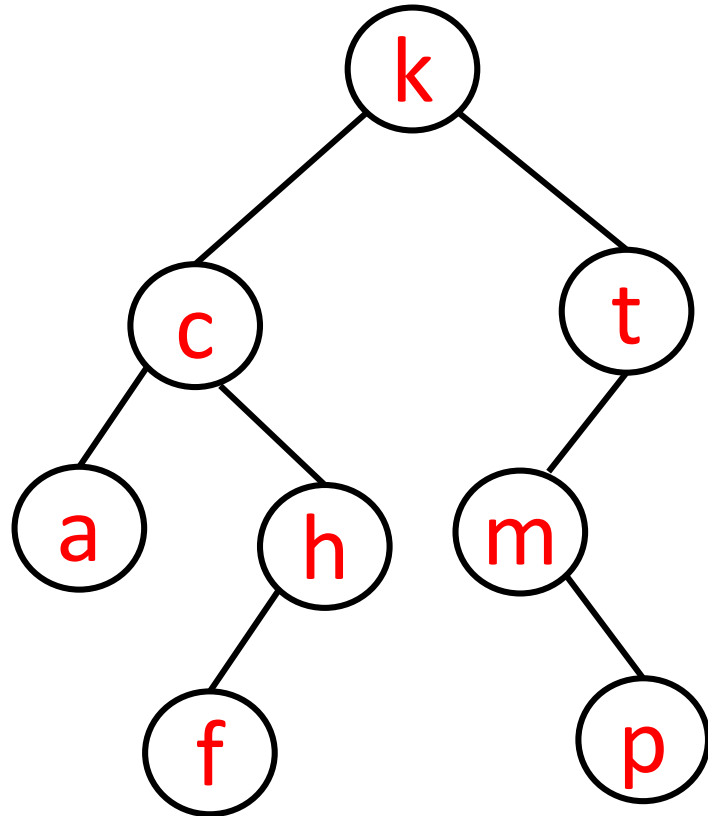
```
add(root, key){                                // returns root node
    if (root == null)
        root = new BSTnode(key)
    else if (key < root.key){
        root.left = add(root.left, key)
    }
    else if (key > root.key){
        root.right = add(root.right, key)
    }

    return root
}
```

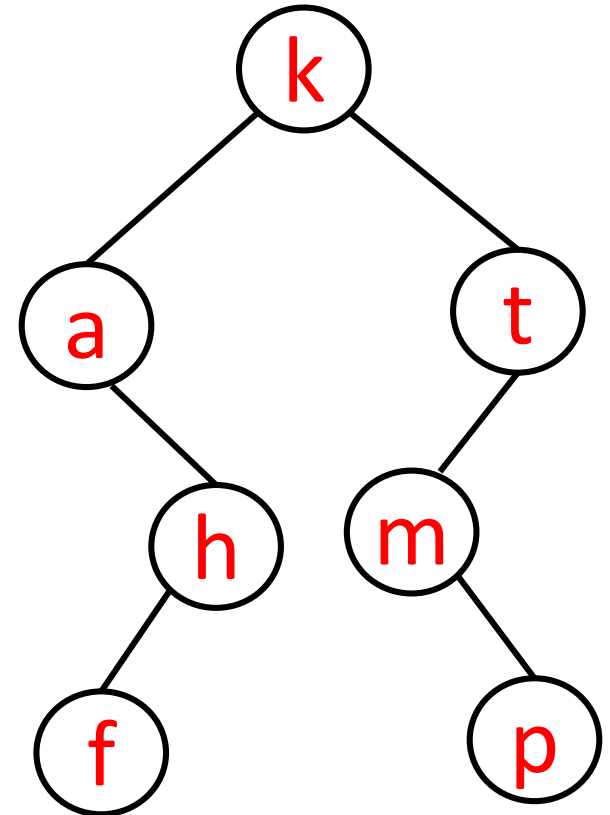
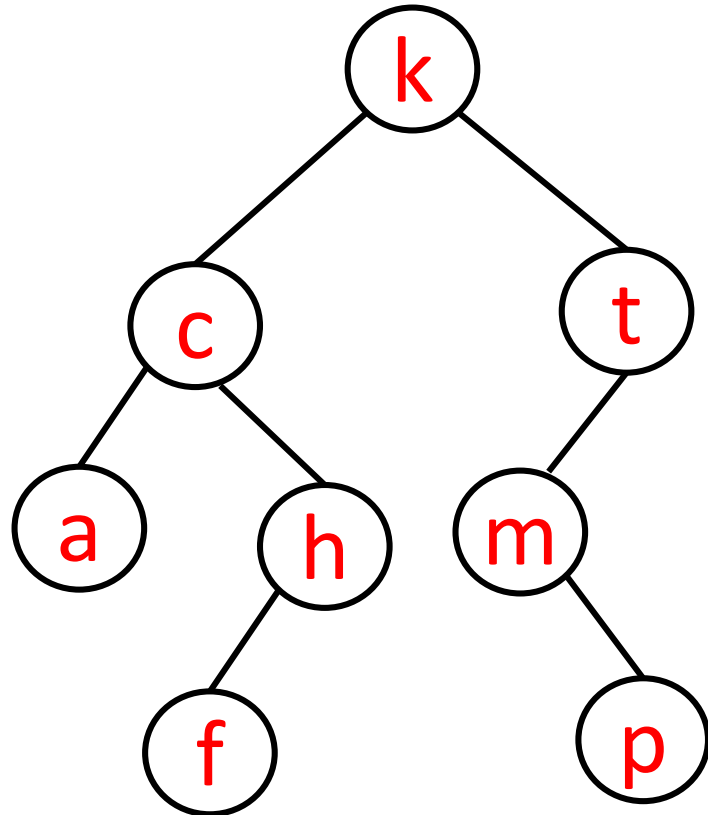
Q: What does it do if `root.key == key` ?

A: Nothing.

remove(c)

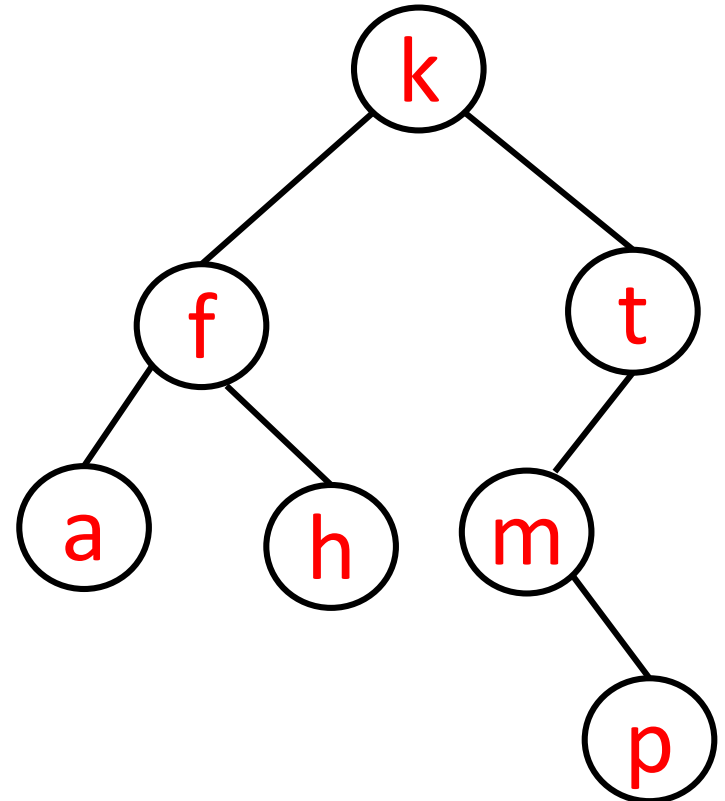
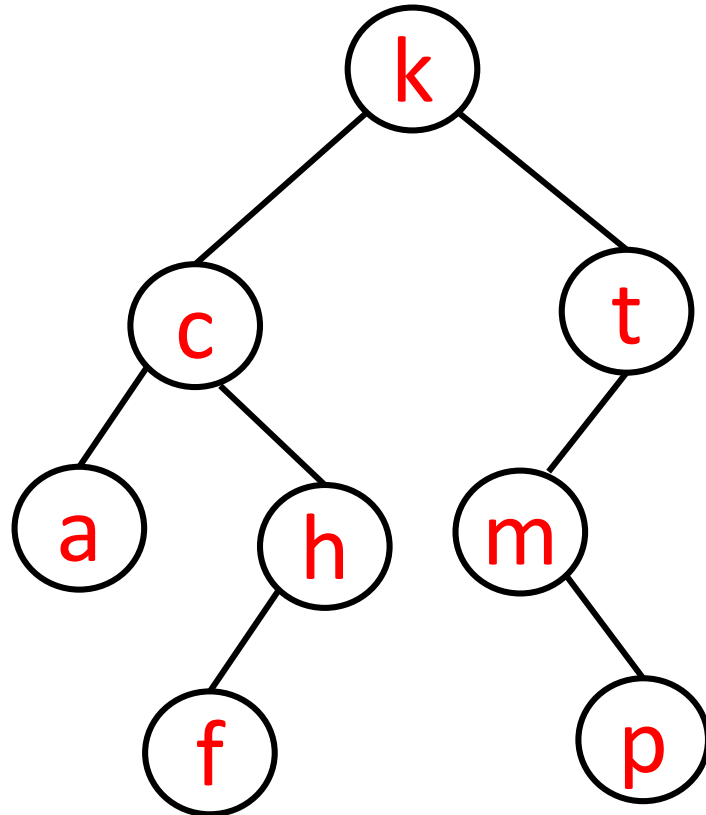


remove(c)



This is one
way to do it.

remove(c)



The algorithm I present next
does it like this.

```
remove(root, key){
```

```
    if( root == null )
```

```
        return null
```

```
    else if ( key < root.key )
```

```
    else if ( key > root.key )
```

```
    else
```

```
    }
```

```
    return root
```


```
}
```

```
// returns root node
```

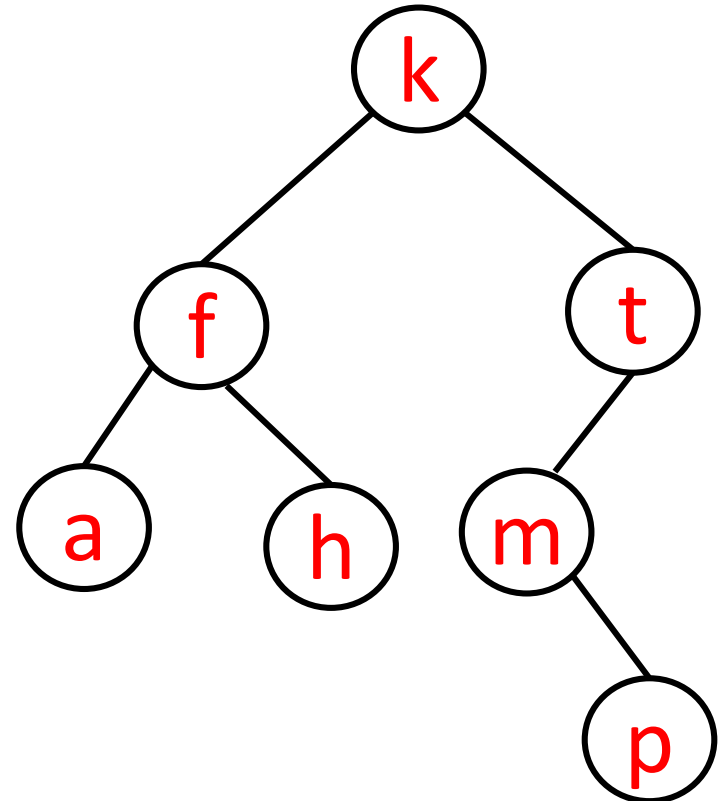
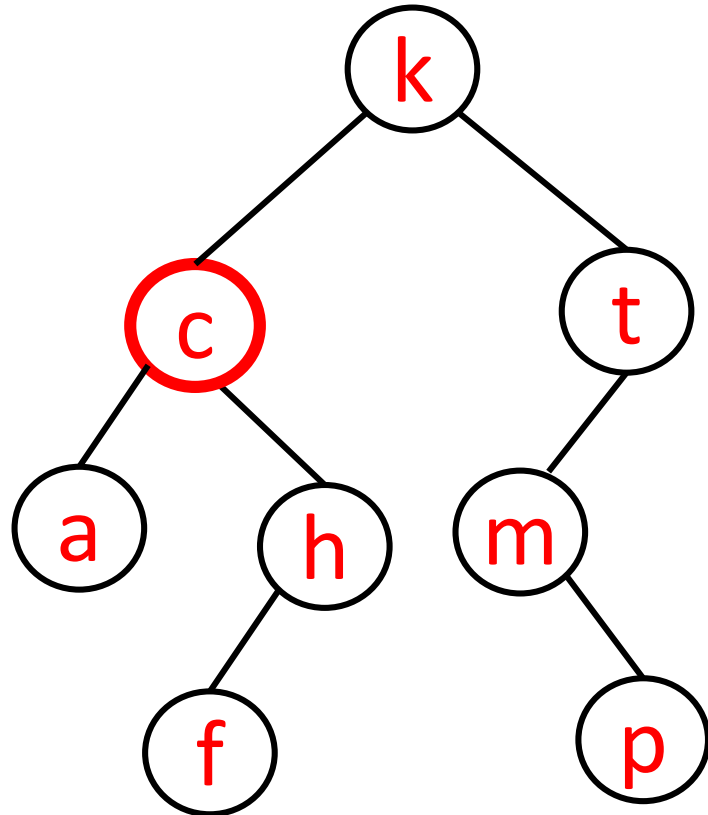


```
remove(root, key){                                     // returns root node
    if( root == null )
        return null
    else if ( key < root.key )
        root.left = remove ( root.left, key )
    else if ( key > root.key )
        root.right = remove ( root.right, key )
    else

}
return root;
}
```

```
remove(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{ 
    }
    return root;
}
```

remove(c)

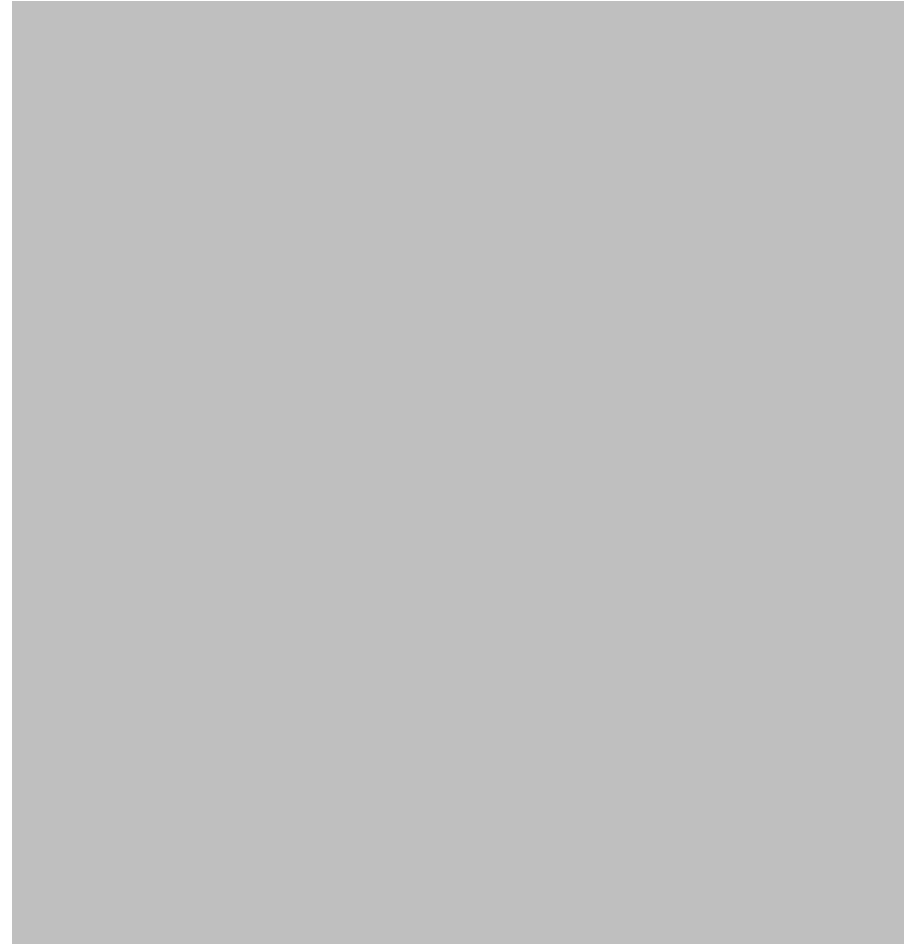
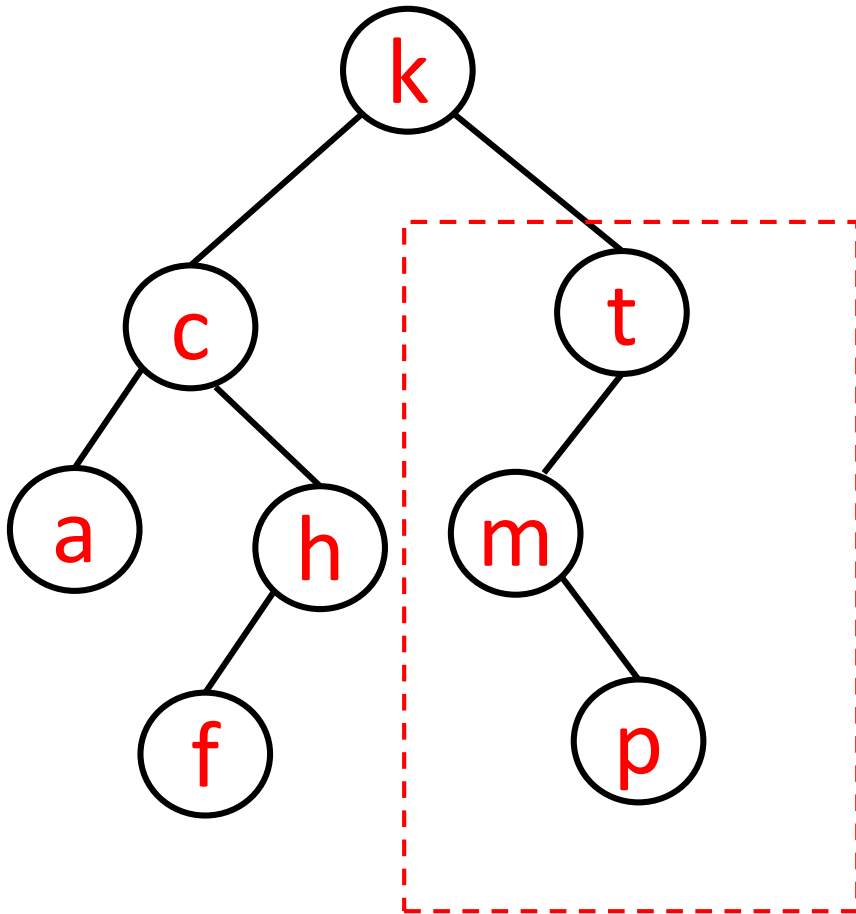


```

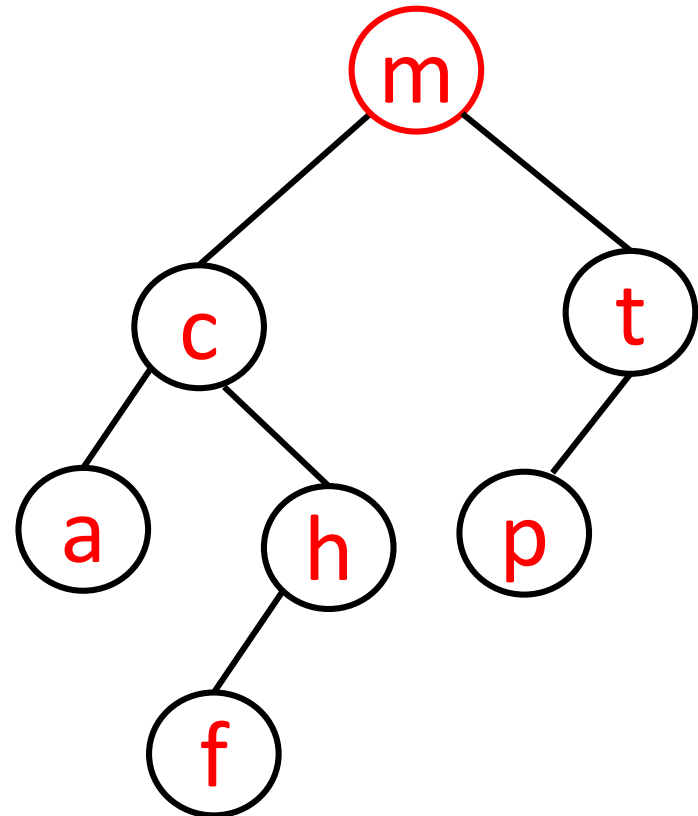
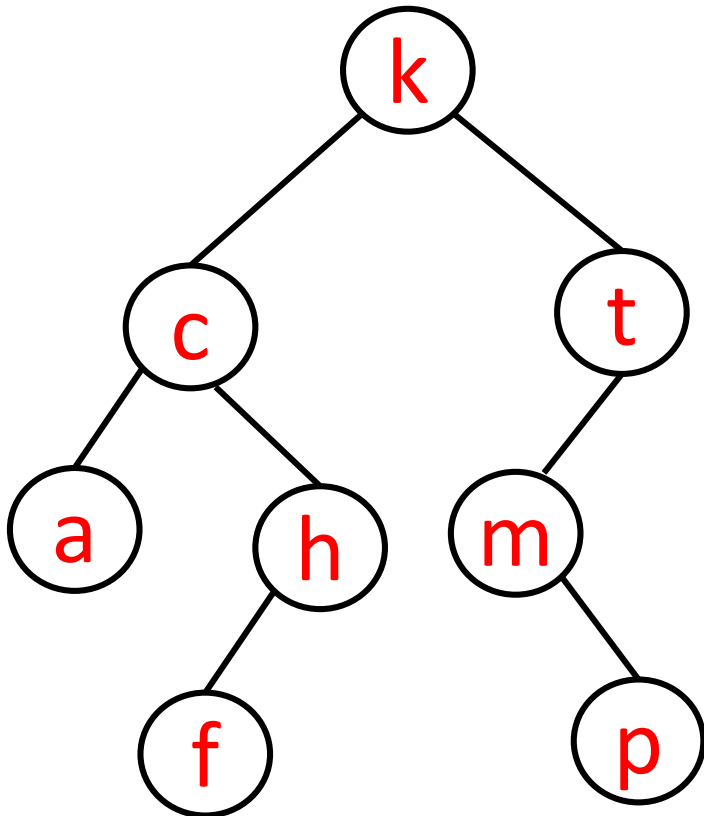
remove(root, key){                                     // returns root node
    if( root == null )
        return null
    else if ( key < root.key )
        root.left = remove ( root.left, key )
    else if ( key > root.key )
        root.right = remove ( root.right, key )
    else if root.left == null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



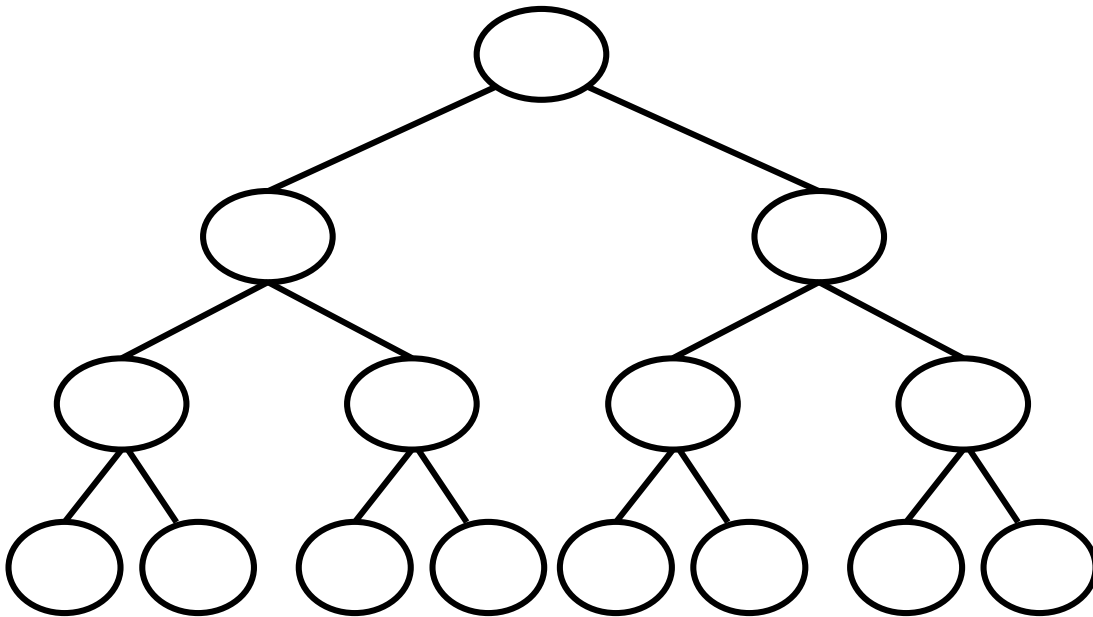
remove(**k**)



balanced

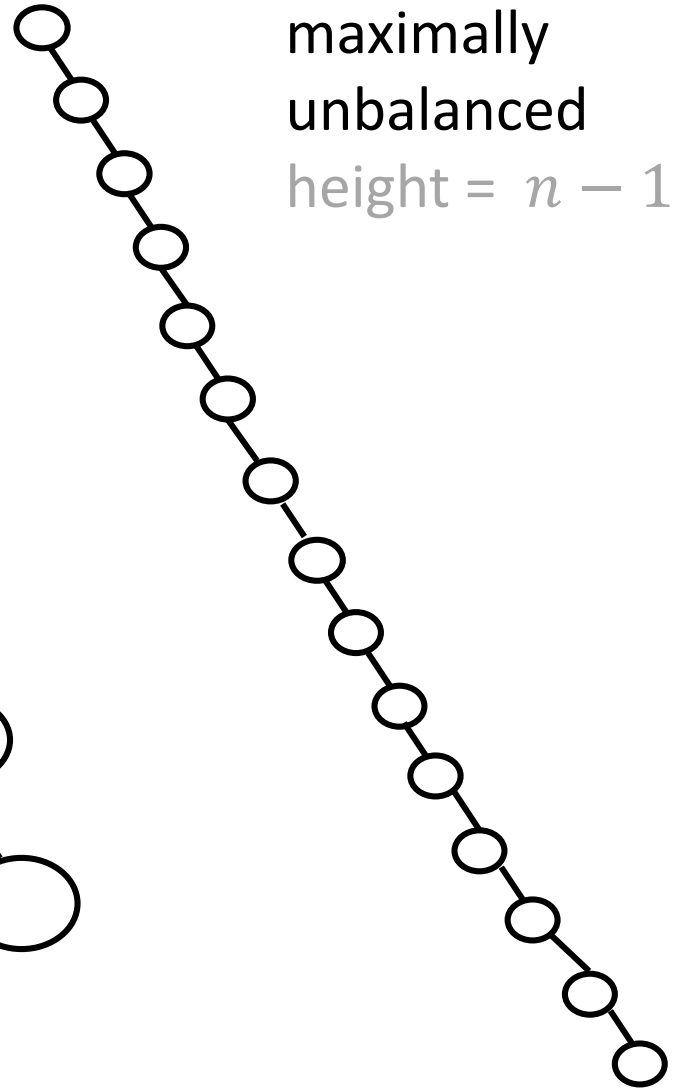
$$\text{height} = \log(n + 1) - 1$$

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



maximally
unbalanced

$$\text{height} = n - 1$$



best vs worst case ?

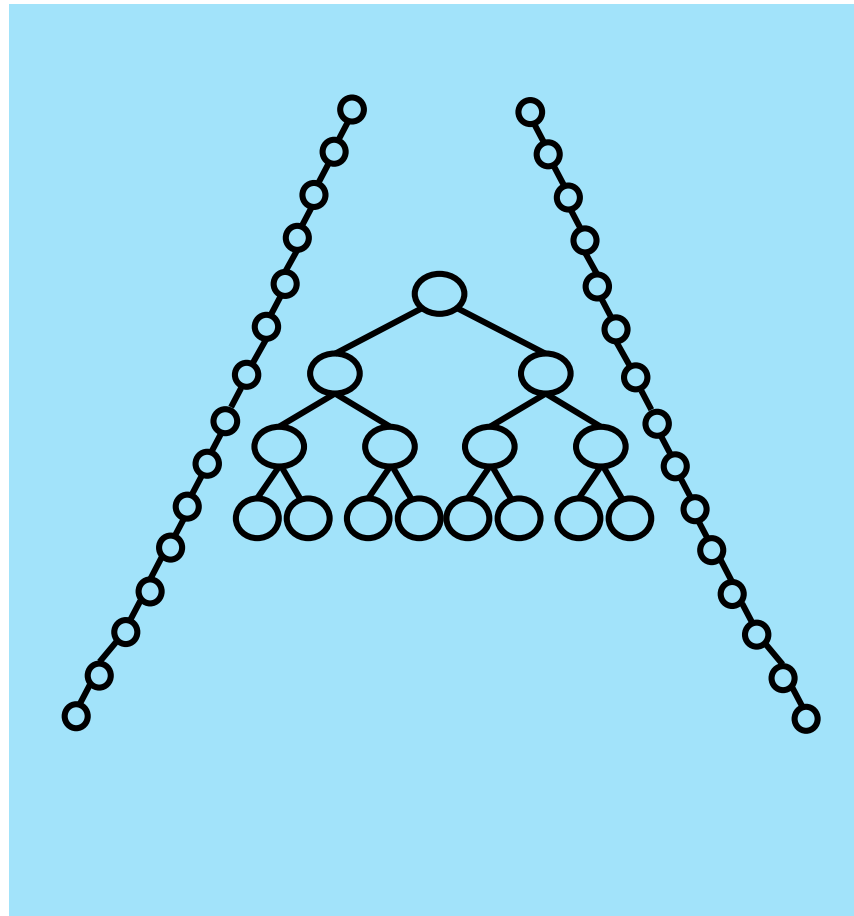
findMin()

findMax()

find(key)

add(key)

remove(key)



Binary Search Tree

best case

worst case

findMin()

findMax()

find(key)

add(key)

remove(key)



Binary Search Tree

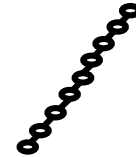
best case

worst case

findMin()

$\Theta(1)$

$\Theta(n)$



findMax()

find(key)

add(key)

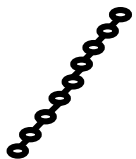
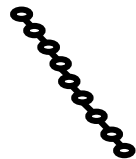
remove(key)



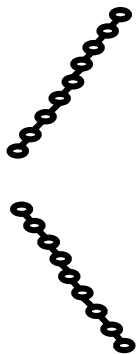
Binary Search Tree

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

Binary Search Tree

	best case	worst case	
findMin()	$\Theta(1)$	$\Theta(n)$	
findMax()	$\Theta(1)$	$\Theta(n)$	
find(key)	$\Theta(1)$	$\Theta(n)$	Could be zigzag
add(key)			
remove(key)			

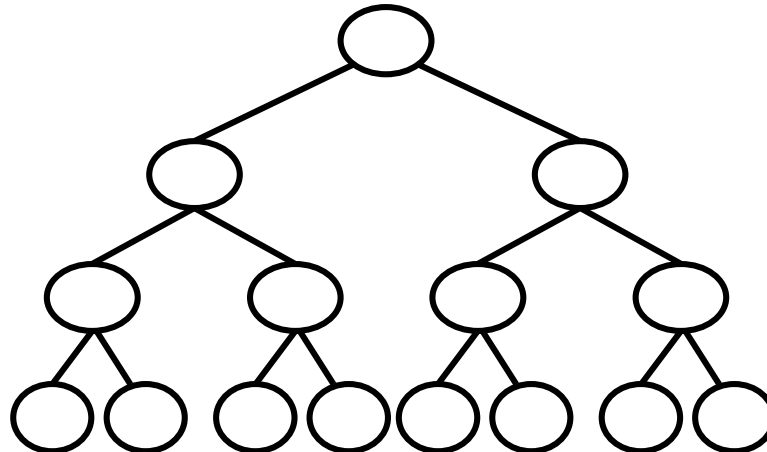
Binary Search Tree

	best case	worst case	
findMin()	$\Theta(1)$	$\Theta(n)$	 } Could be zigzag
findMax()	$\Theta(1)$	$\Theta(n)$	
find(key)	$\Theta(1)$	$\Theta(n)$	
add(key)	$\Theta(1)$	$\Theta(n)$	
remove(key)	$\Theta(1)$	$\Theta(n)$	

~~(binary search) tree~~

binary (search tree)

But wait....

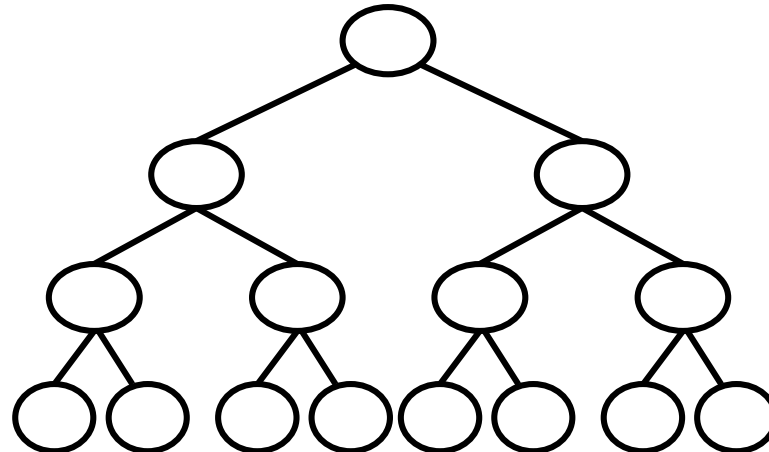


~~(binary search) tree~~

binary (search tree)

But wait....

when a binary search tree is balanced, then `find(element)` is very similar to a binary search algorithm that checks for a key match.



Balanced Binary Search Trees

(COMP 251: AVL trees, red-black trees)

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