COMP 250

Lecture 25

binary search trees

Nov. 7, 2018

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

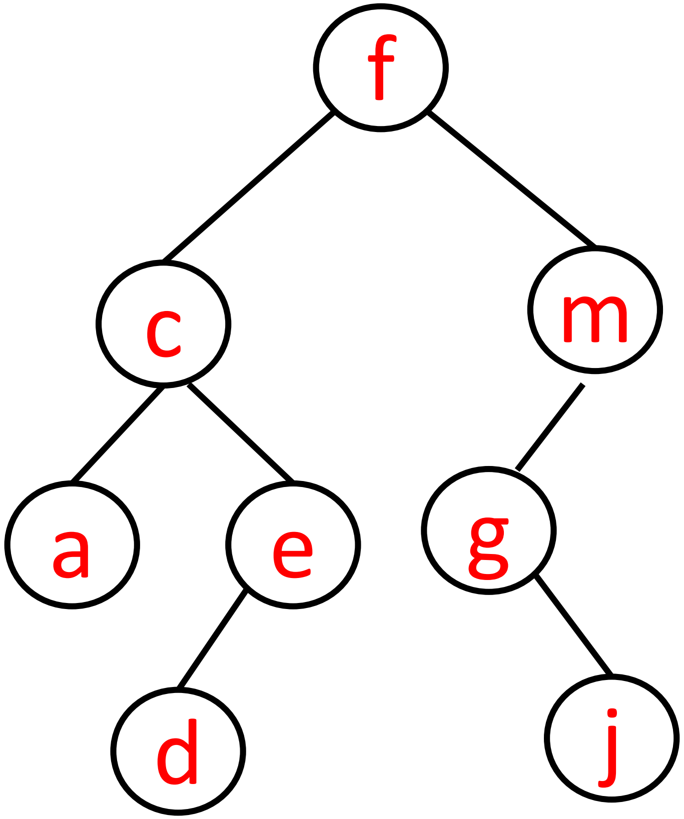
The keys are “comparable” <, =, >
e.g. numbers, strings.

Binary Search Tree Definition

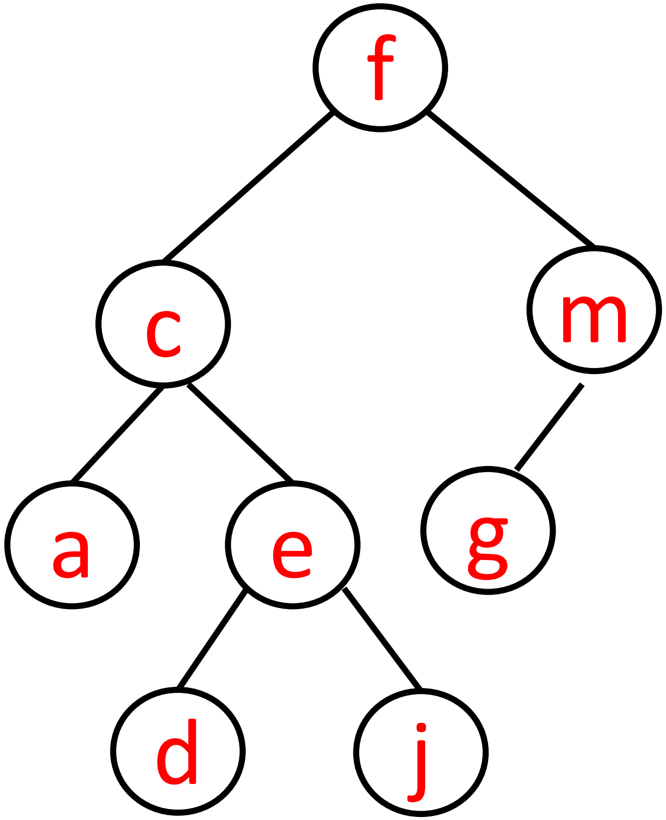
- binary tree
- keys are comparable, unique (no duplicates)
- for each node, all descendents in left subtree are less than the node, and all descendents in the node's right subtree are greater than the node

  (comparison is based on node key)

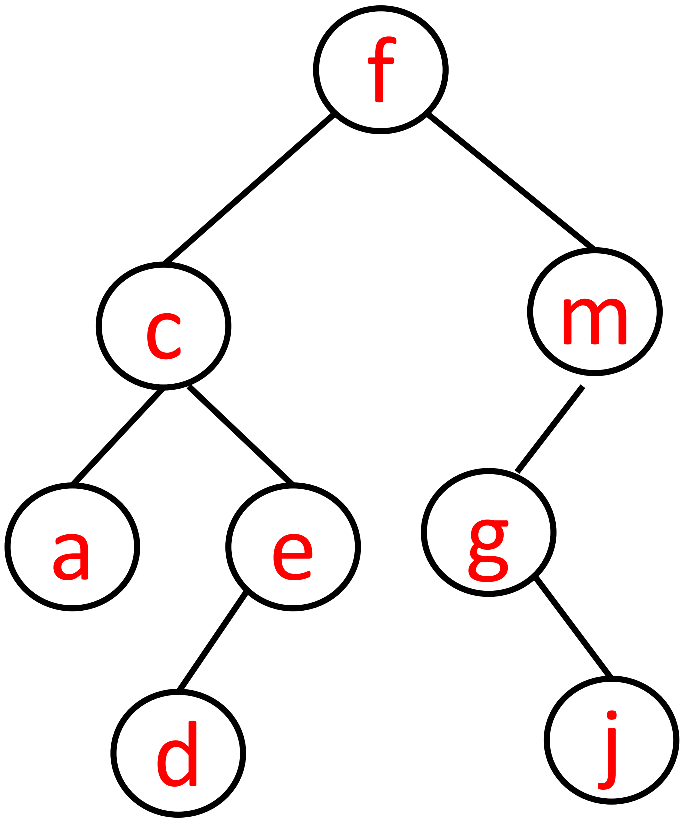
Example



This is not a BST. Why not?



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



acdefgjm

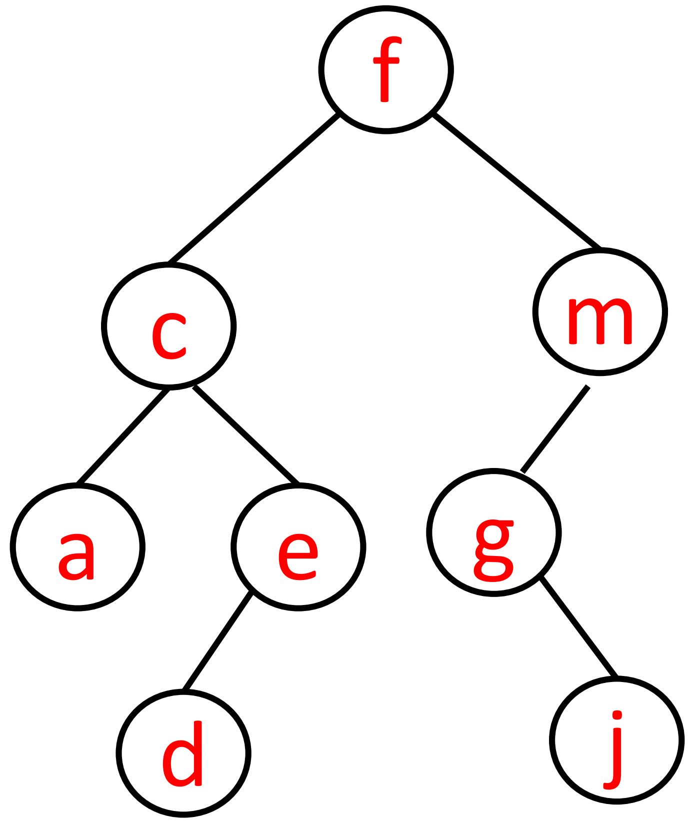
Binary Search Tree ADT

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

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

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

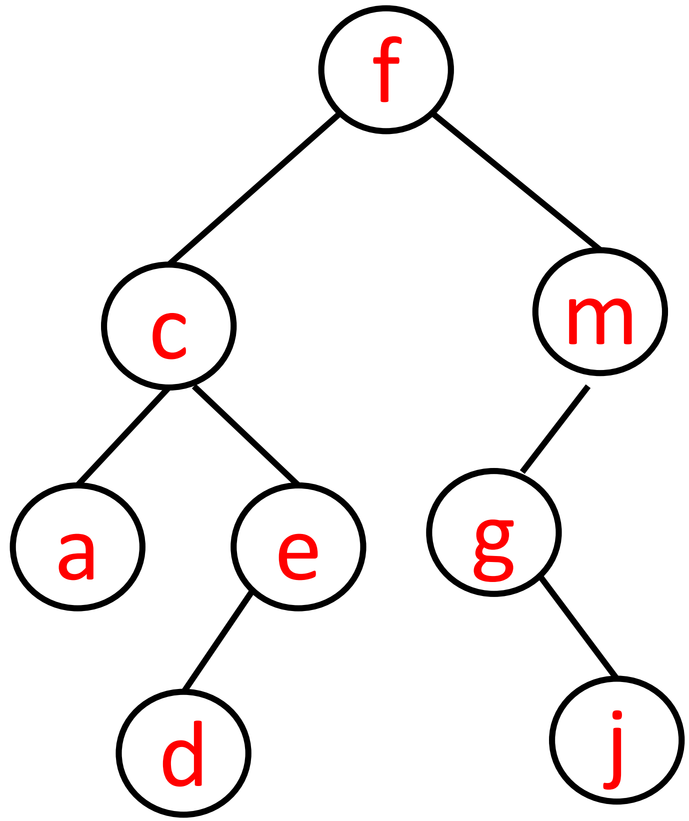
find(root, g) returns g node
find(root, s) returns null



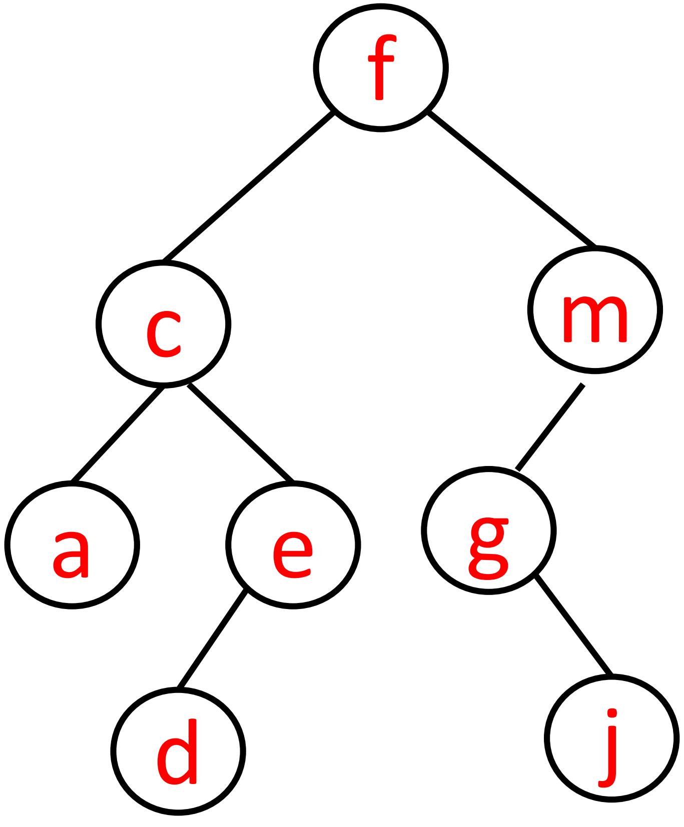
```
find(root, key){                    // returns a node
  if (root == null)
     return null
  else if (root.key == key))
     return root
}
```

```
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)
}
```

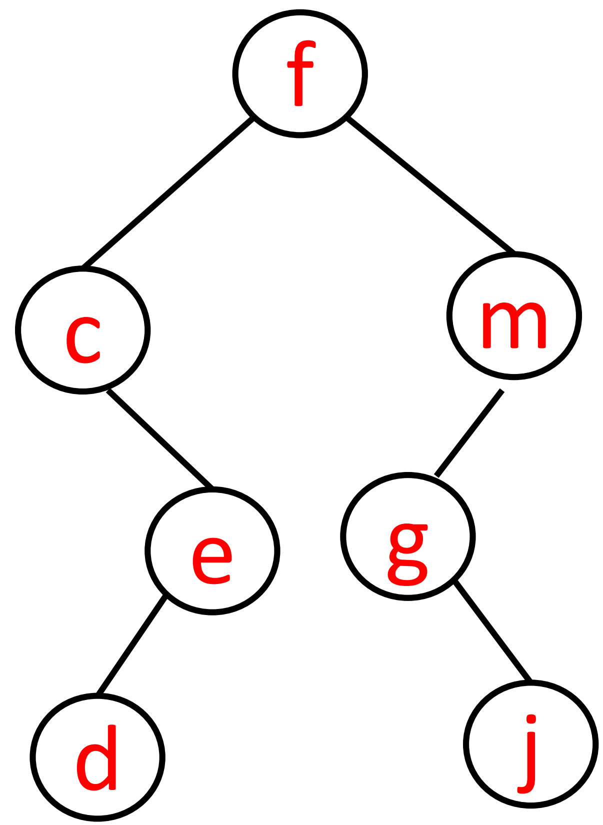
findMin() returns



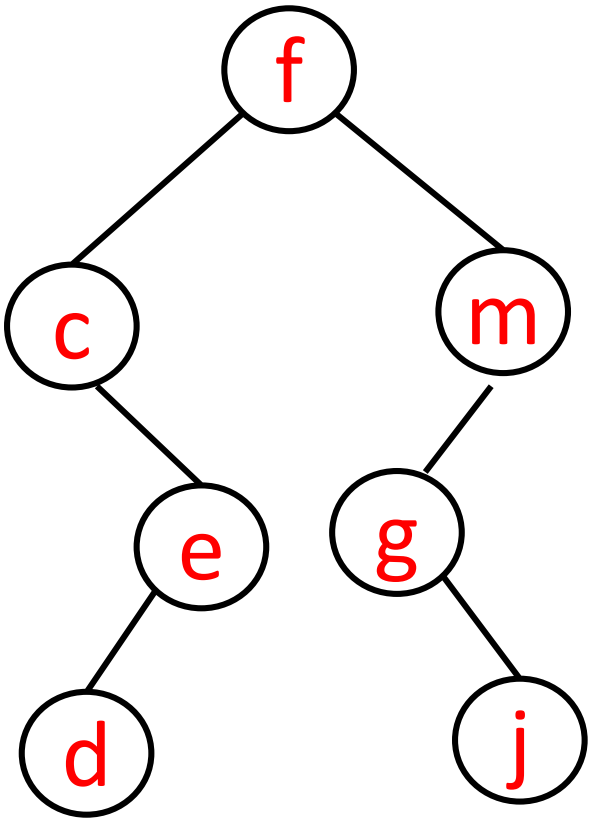
findMin() returns a (node)



findMin() returns



findMin() returns c node

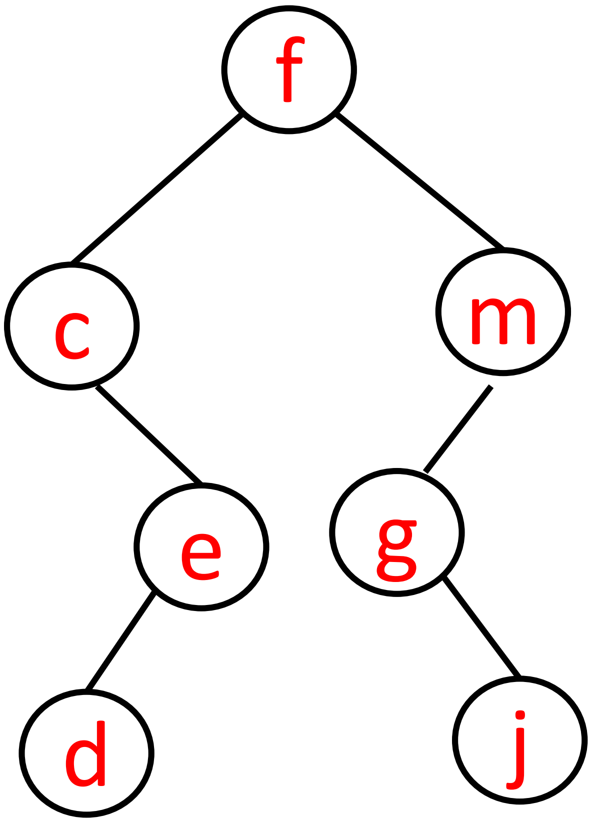


```
findMin(root){                         // returns a node
    if (root == null)
        return null

}
```

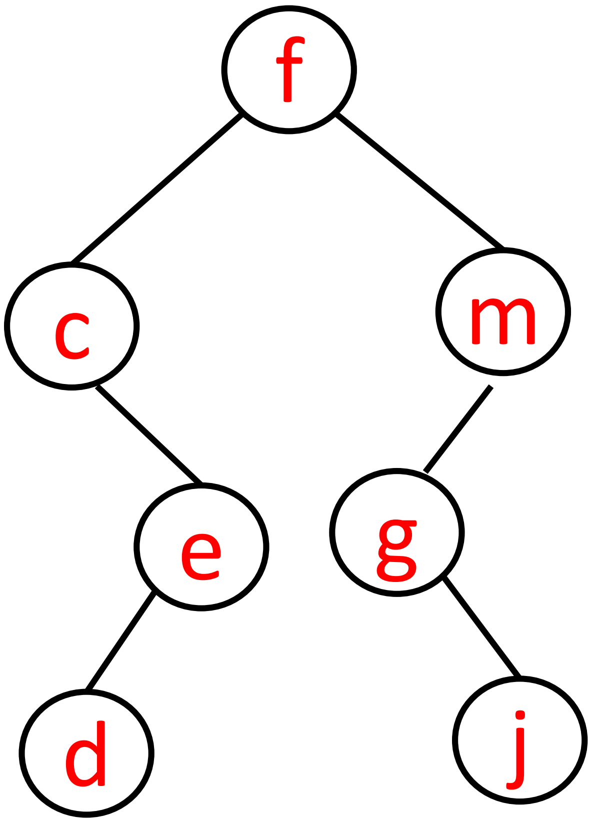
```
findMin(root){                    // returns a node
    if (root == null)
        return null
    else if (root.left == null)
        return root

}
```

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

findMax() returns ?

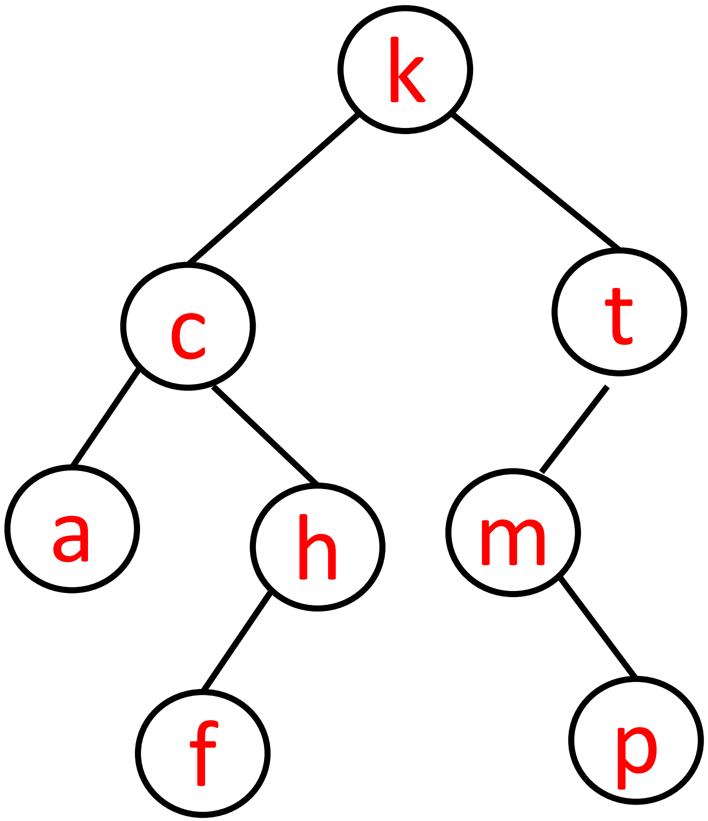


findMax() returns m node



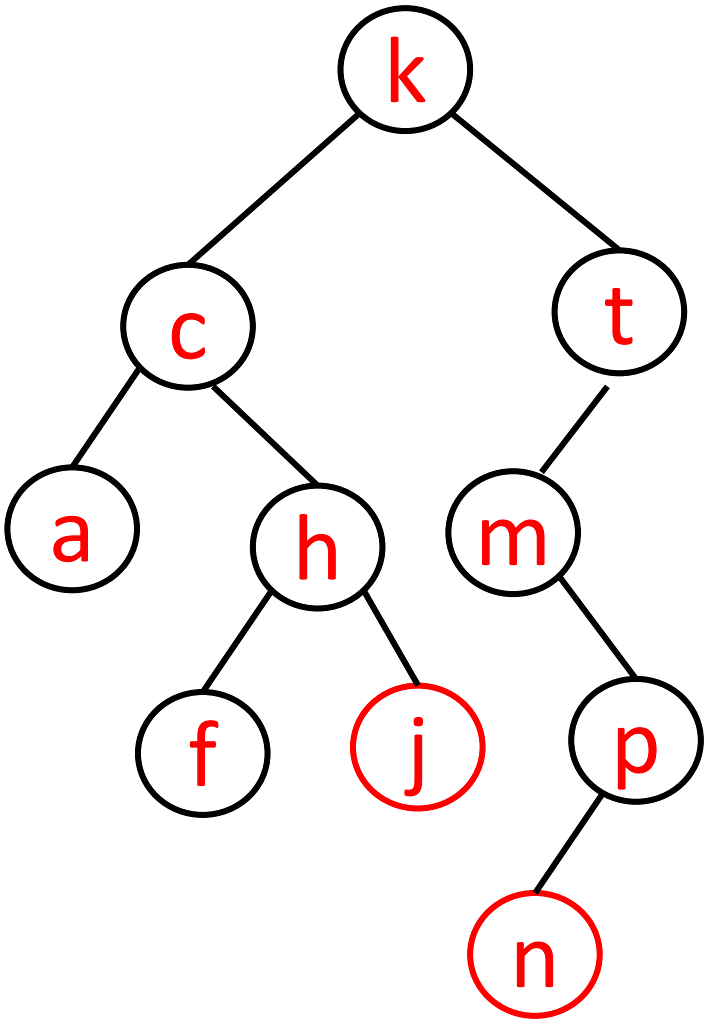
```
findMax(root){                    // returns a node
    if (root == null)
        return null

}
```

```
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
}
```

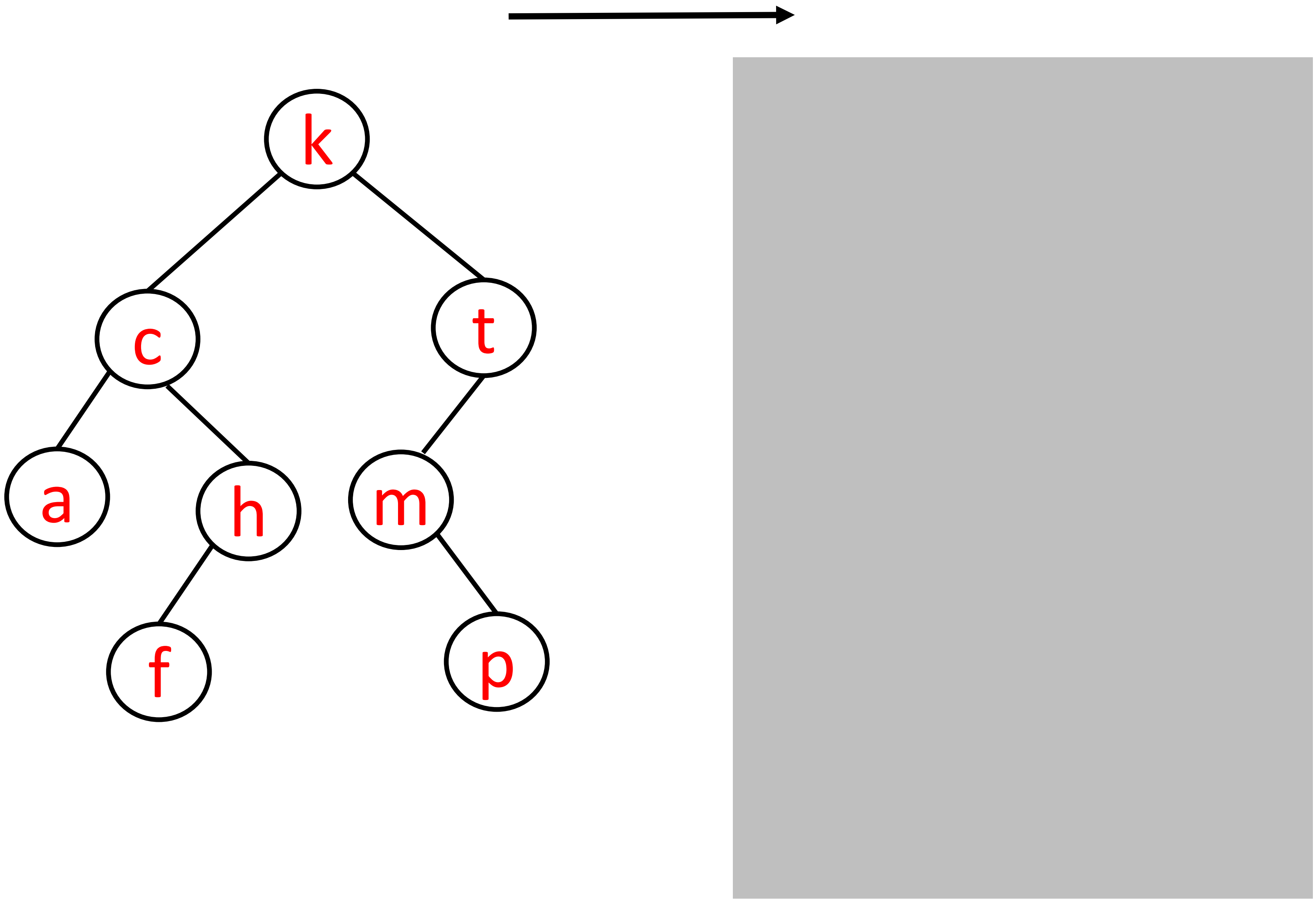
```
add(root, key){                    // returns root node
  if (root == null)
        root =  new BSTnode(key)
  else if (key < root.key){

   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
}
```

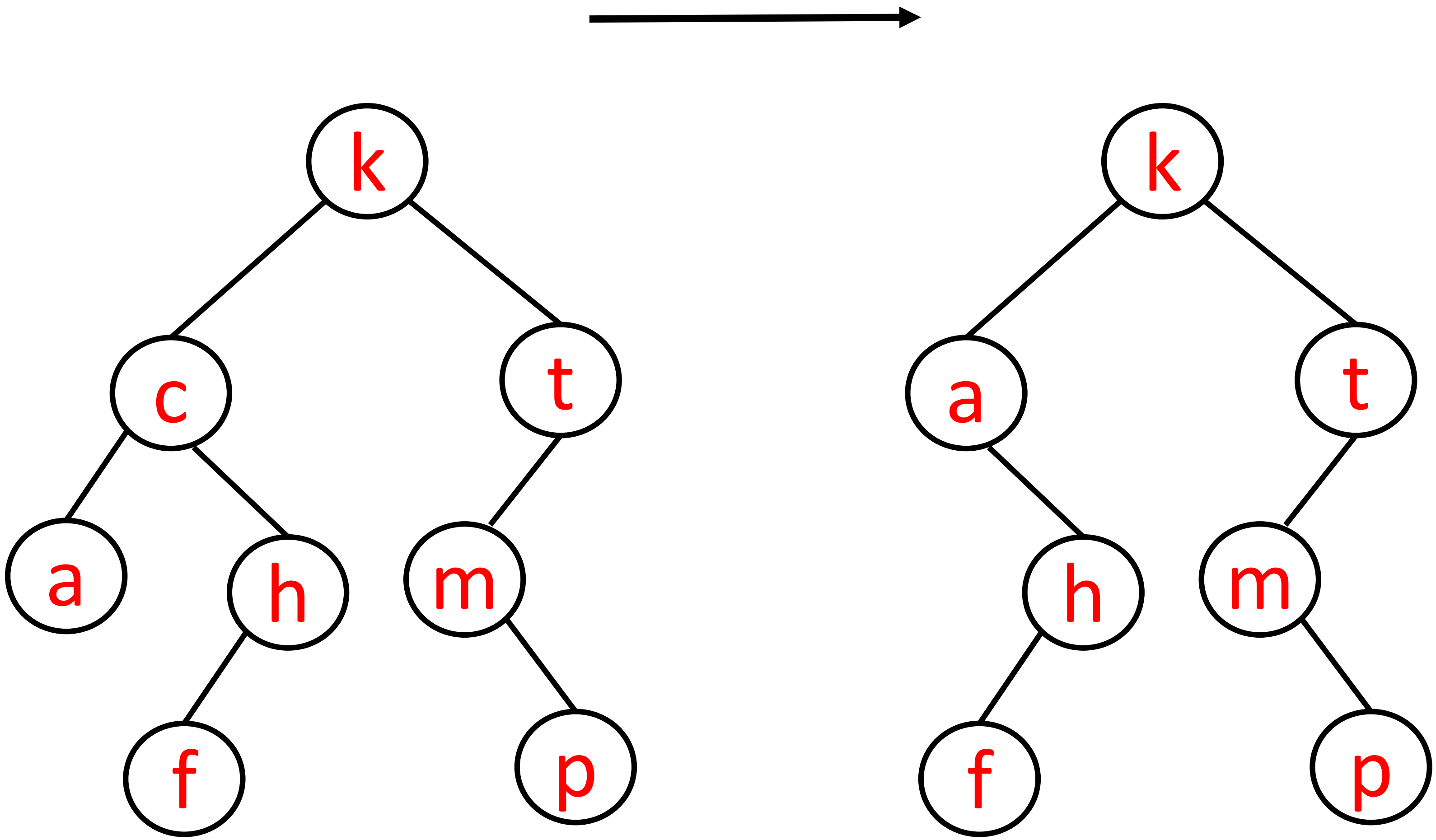
```
add(root, key){                    // returns root node
   if (root == null)
           root =  new BSTnode(key)
   else if (key < root.key){
           root.left    =  add(root.left, key)
   else if (key > root.key){
           root.right  =  add(root.right, key)

    return root
 }
```

It does nothing if root.key == key.

remove(c)

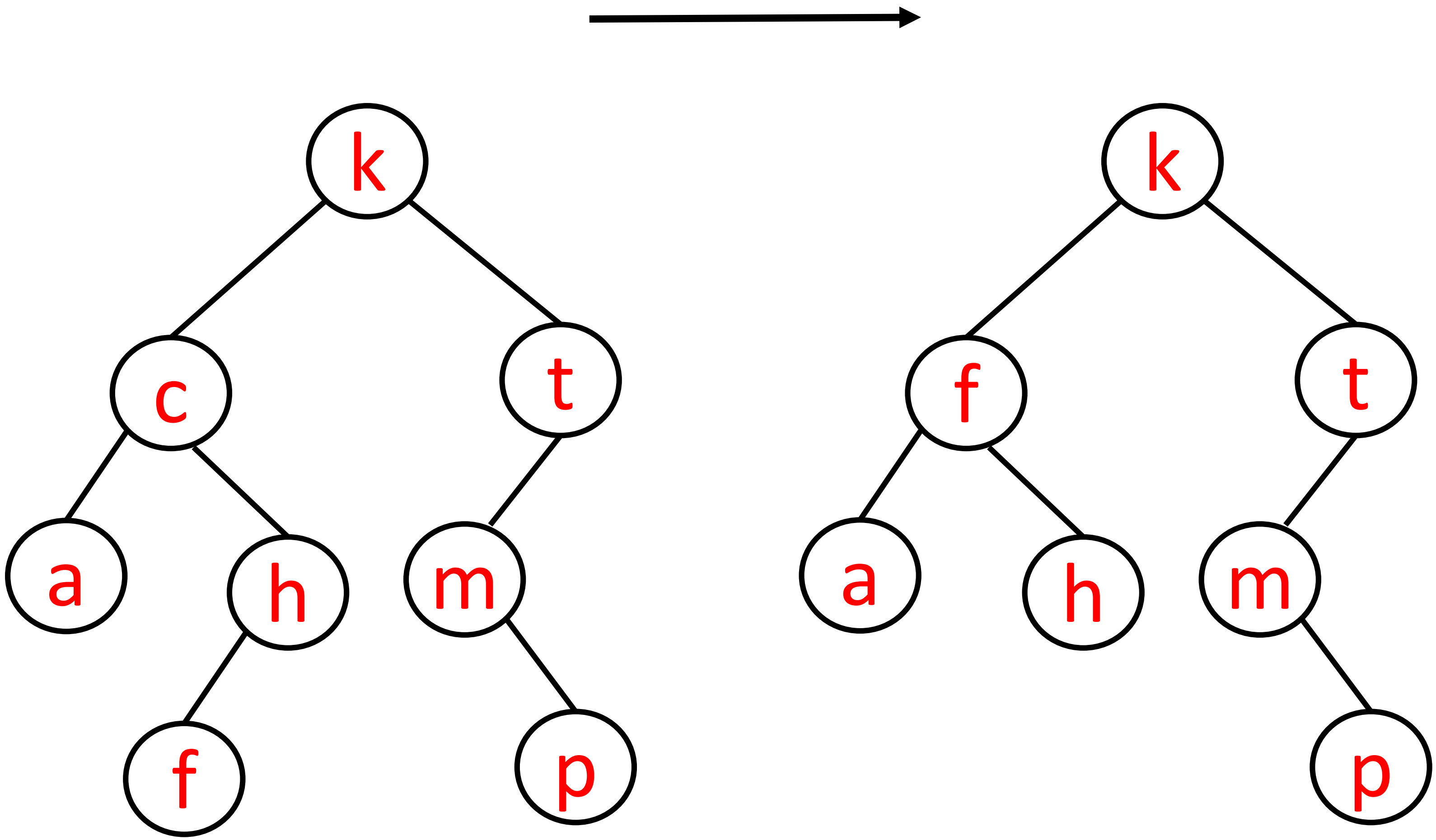


remove(c)



This is one
way to do it.

remove(c)



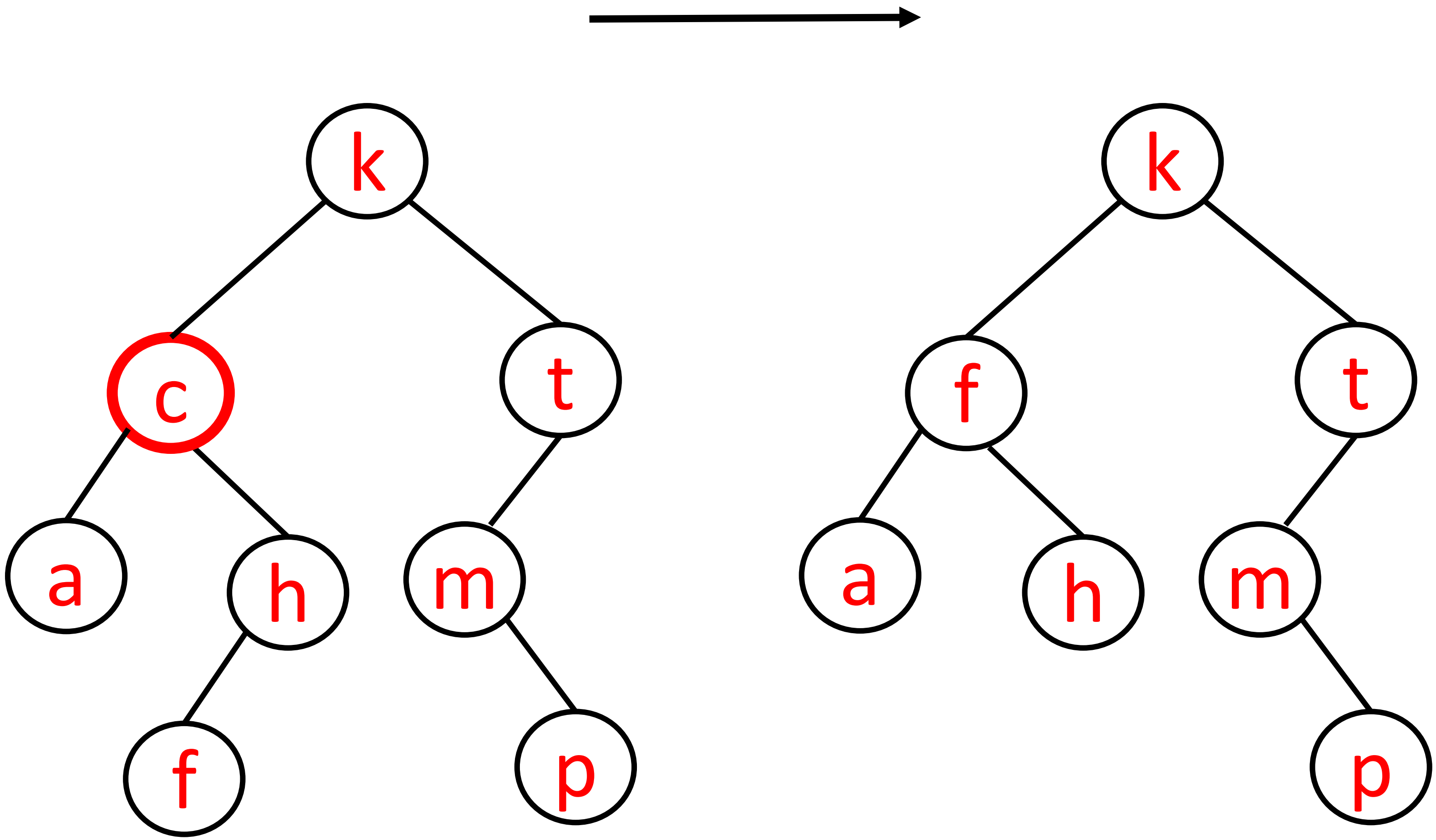
The algorithm I present next does it like this.

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

   else  if ( key > root.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
   }
   return root;
}
```

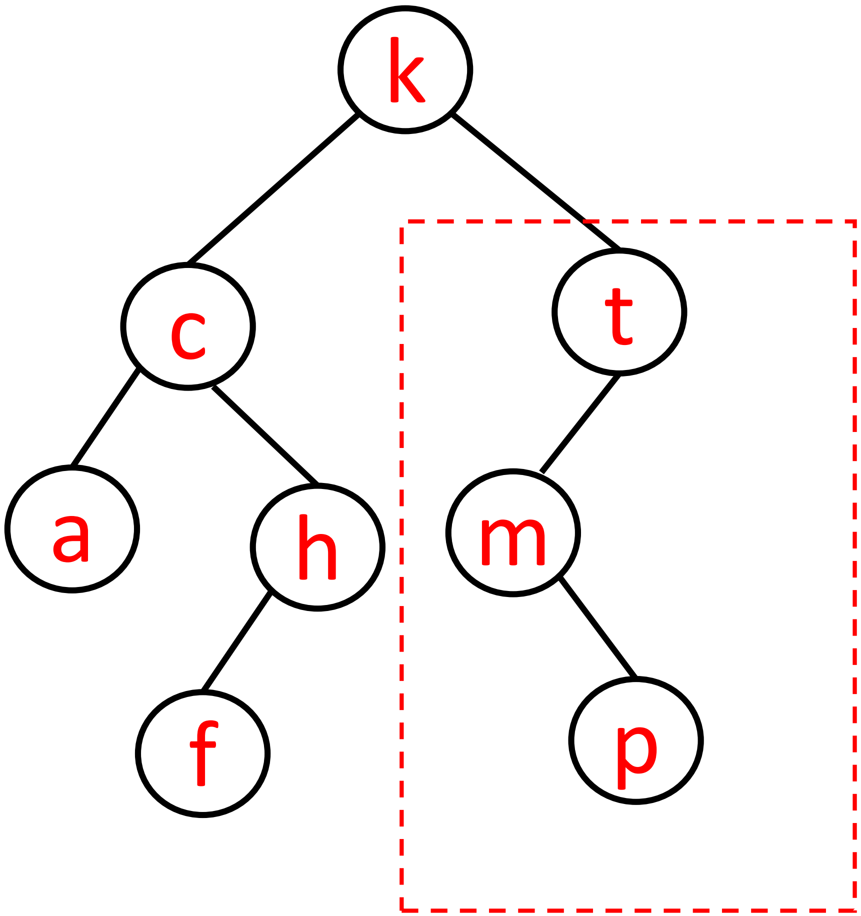
```
remove(root, key){                              // returns root node
   if( root  == null )
      return null
   else  if ( key < root.key )
      root.left =  remove ( root.left, key )
   else  if ( key > root.key )
      root.right = remove ( root.right, key)
   else  if root.left == null
    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



   }
   return root;
}
```

remove(c)

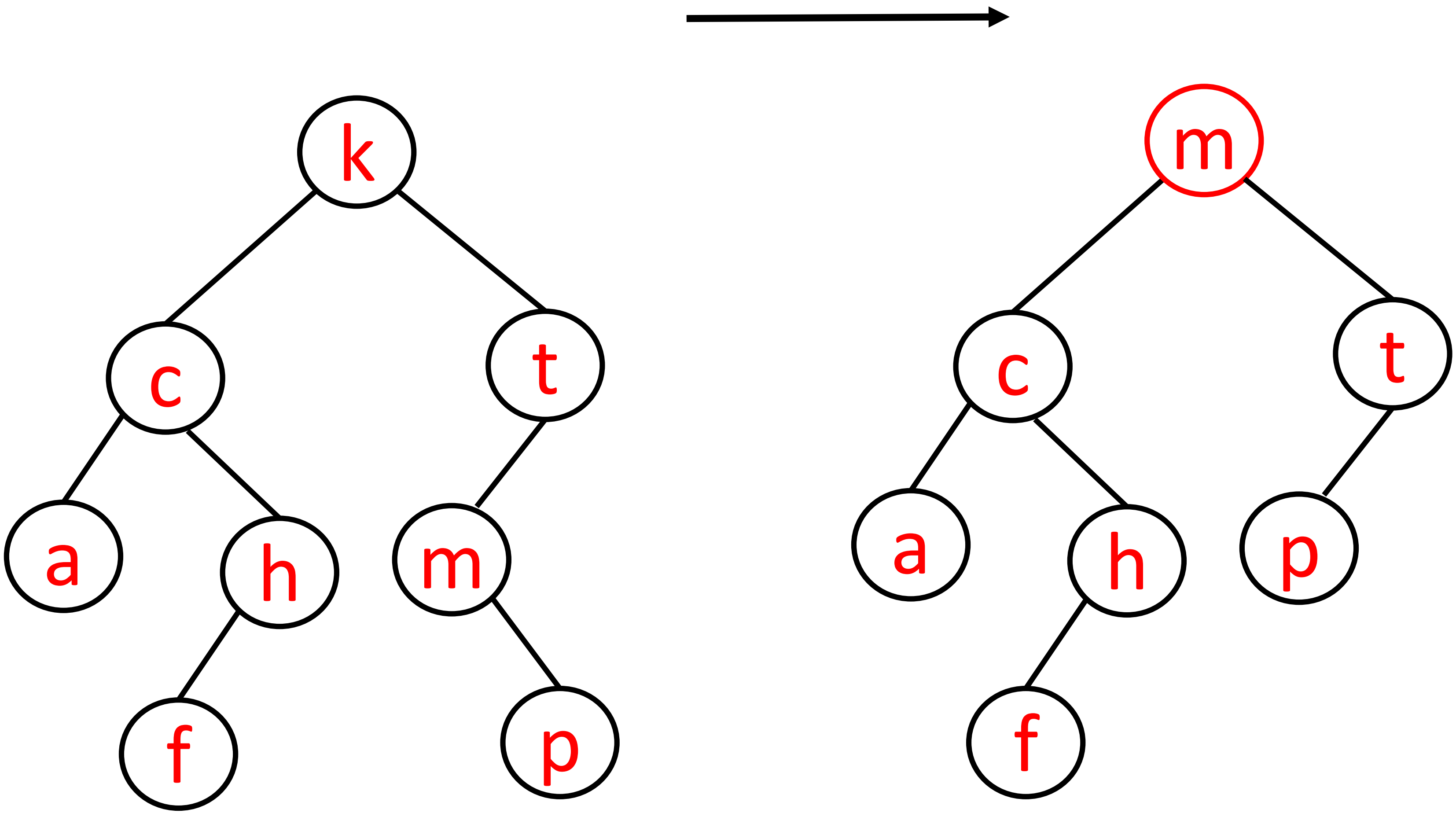


```
remove(root, key){                          // returns root node
   if( root  == null )
      return null
   else  if ( key < root.key )
      root.left =  remove ( root.left, key )
   else  if ( key > root.key )
      root.right = remove ( root.right, key)
   else  if root.left == null
      root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

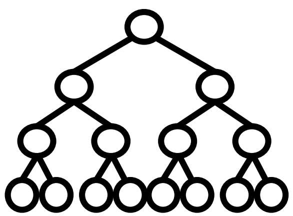
remove(k)

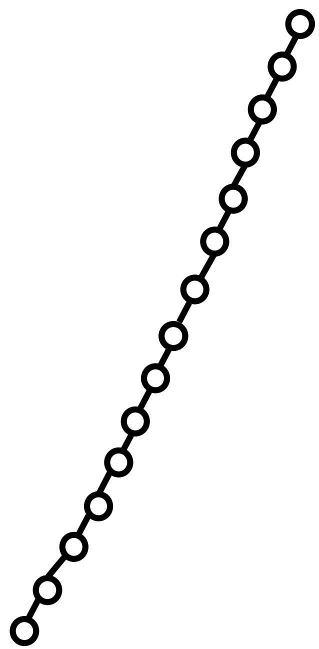


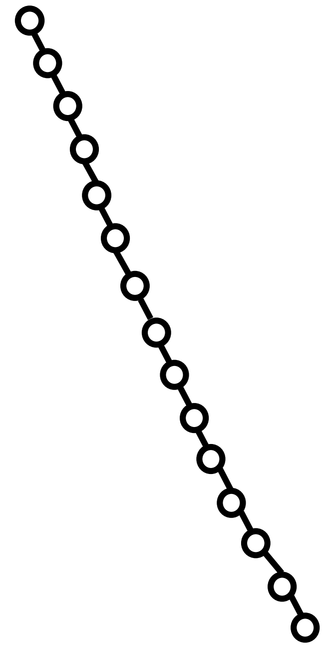
remove(k)



best vs worst case ?



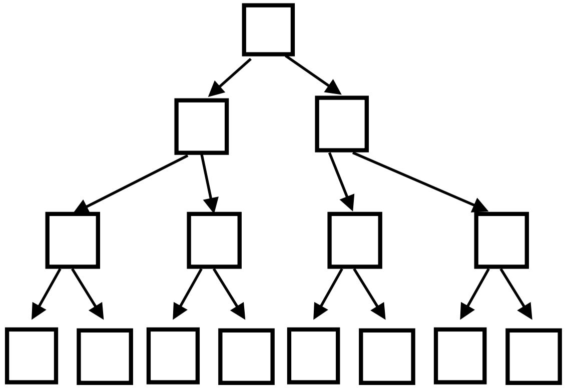




Recall from last lecture

Maximum number of nodes in a binary tree of height h:

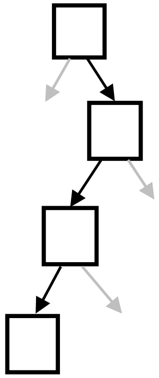
$$n = 2^{h+1} - 1$$



$$log_2(n+1) = h+1$$

Minimum number of nodes in a binary tree of height h:

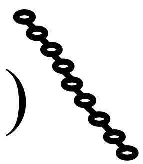
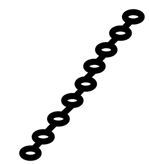
$$n = h + 1$$



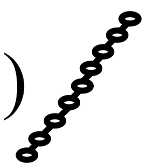
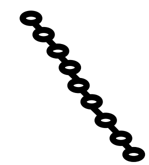
Binary Search Tree

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

Binary Search Tree

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

Binary Search Tree

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

Binary Search Tree

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

Binary Search Tree

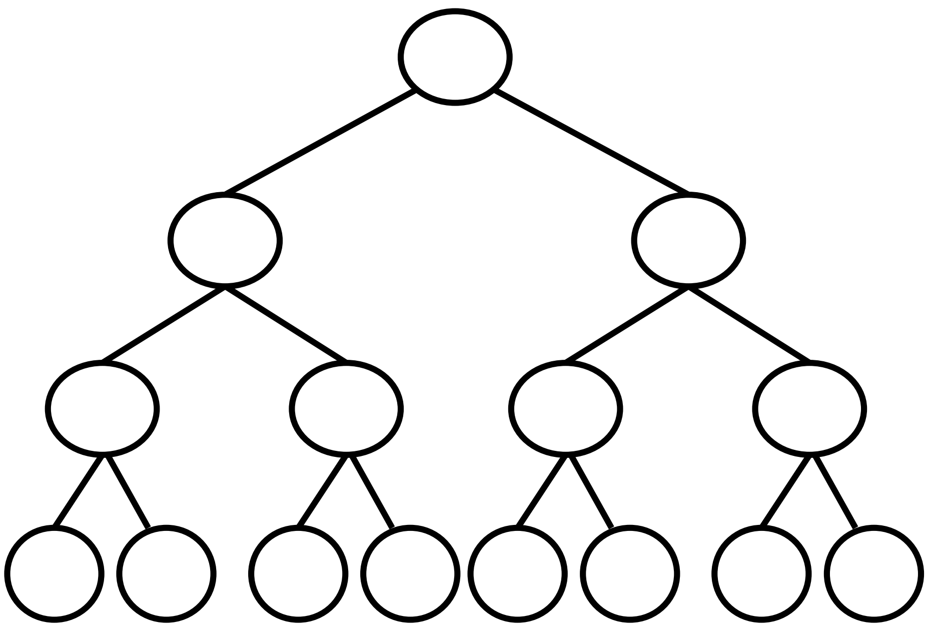
	best case	worst case
findMin()	$O(1)$	$O(n)$
findMax()	$O(1)$	$O(n)$
find(key)	$O(1)$	$O(n)$
add(key)	$O(1)$	$O(n)$
remove(key)	$O(1)$	$O(n)$

Could be zigzag

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

From slide 40, for a binary tree with all levels full:

$$h = log_2(n+1) - 1$$



Balanced Binary Search Trees (covered COMP 251)

	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)	$O(\log n)$	$O(\log n)$