COMP 250

Lecture 26

binary search trees

March. 14, 2022

A binary search tree is a particular kind of binary tree.

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

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

Binary Search Tree Definition

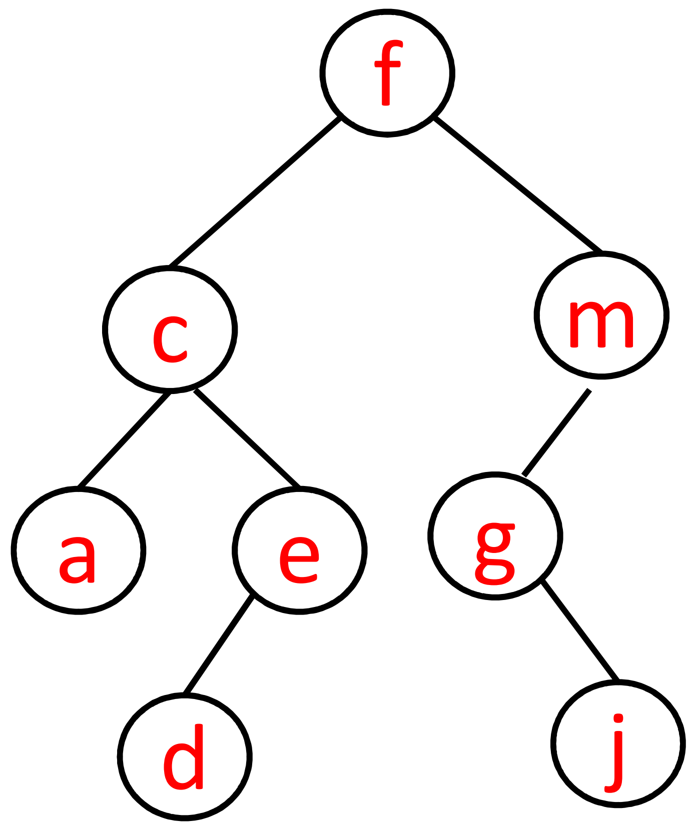
- binary tree

- Each node has an element called a “key”.

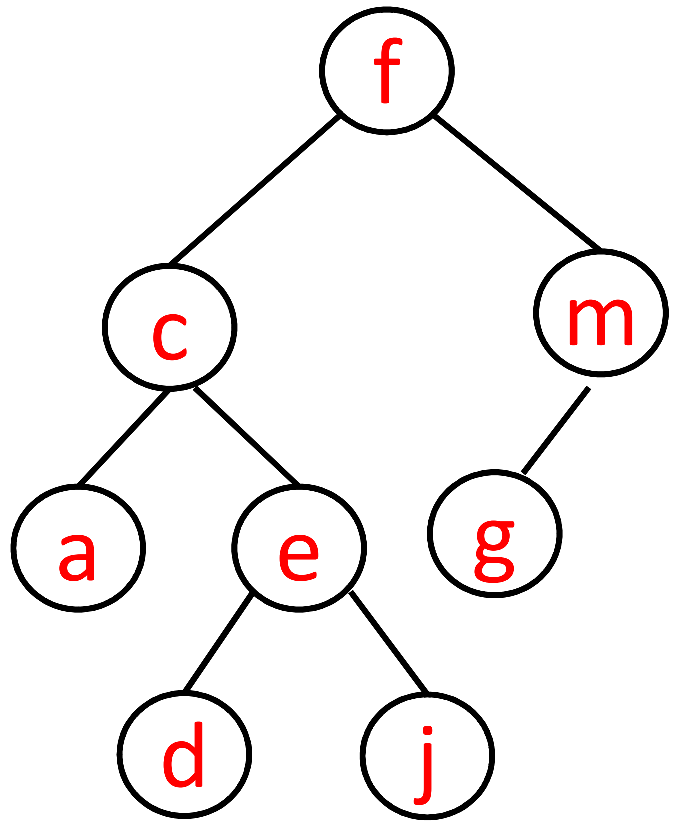
  Keys are comparable & unique (no duplicates).

- For each node, the key in all descendents in left subtree are less than the node key, and the keys in all descendents in the right subtree are greater than the node key.

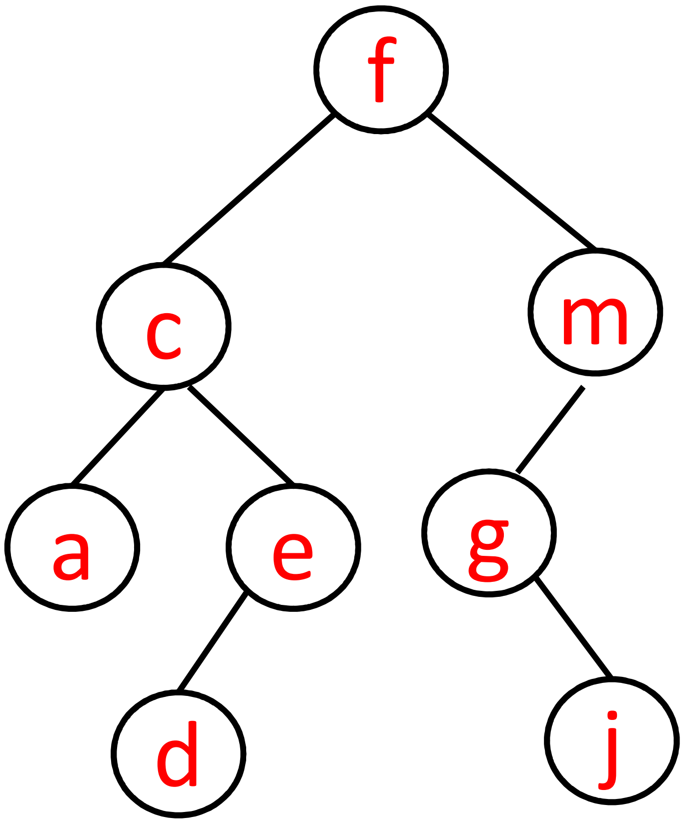
Example of a binary search tree



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

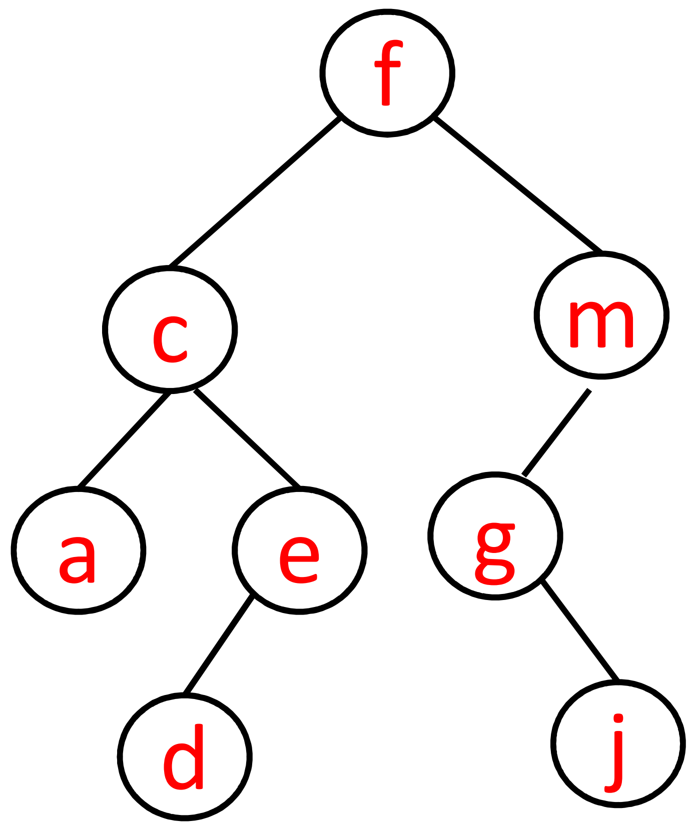
Binary Search Tree Operations

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

We will use recursive helper methods.

These helper methods take the root as a parameter.

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



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

}
```

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

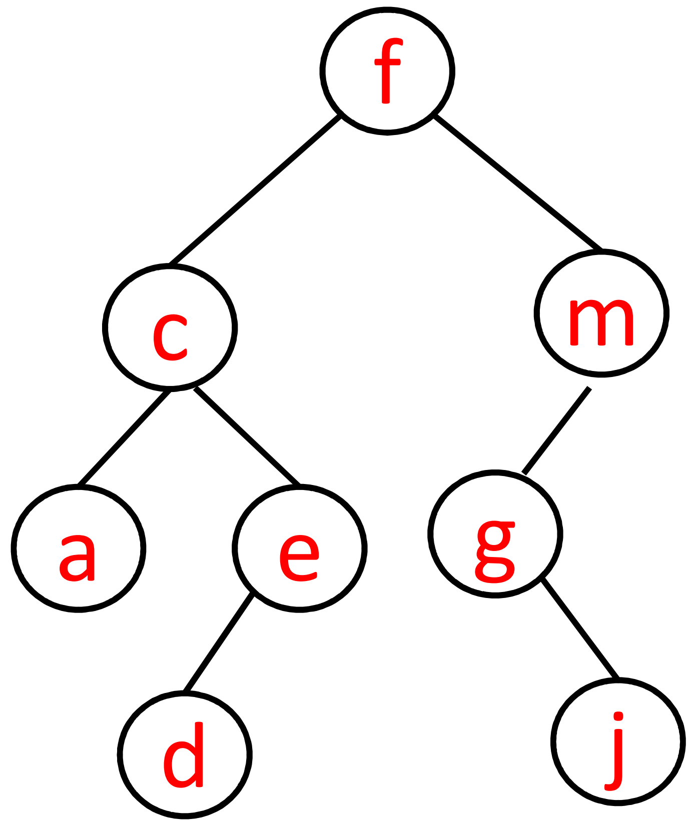
Time Complexity

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

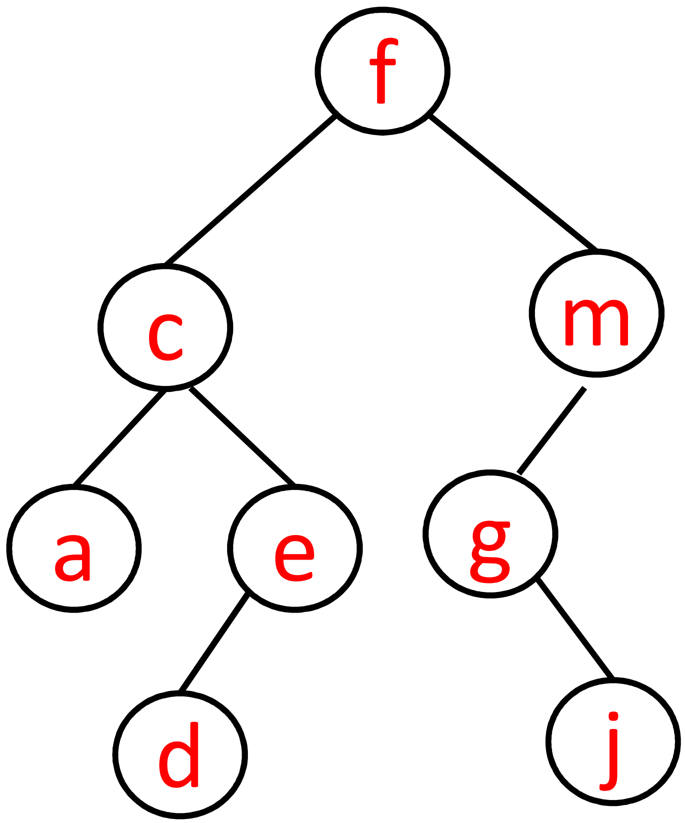
Time Complexity

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

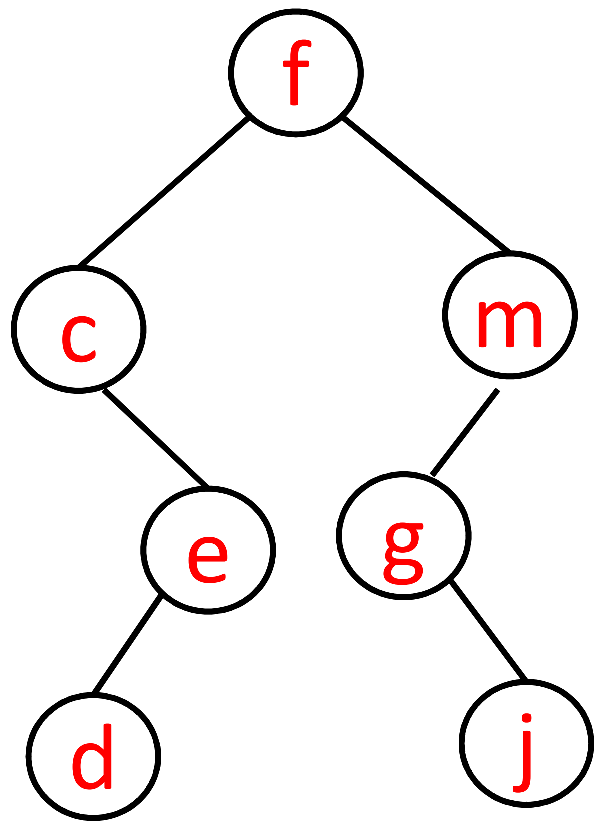
findMin() returns



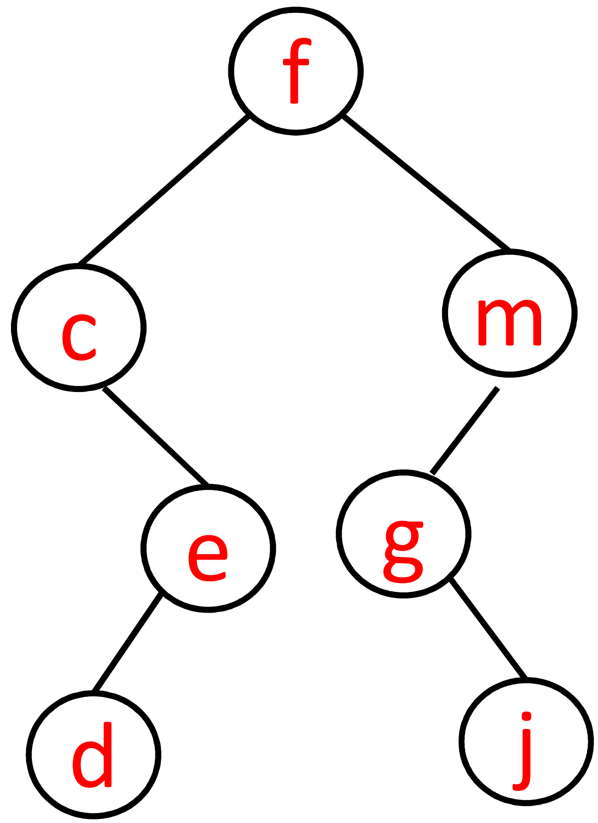
findMin() returns a node



findMin() returns



findMin() returns c node



pass in the root as parameter

```
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
    else

}
```

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

Time Complexity

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

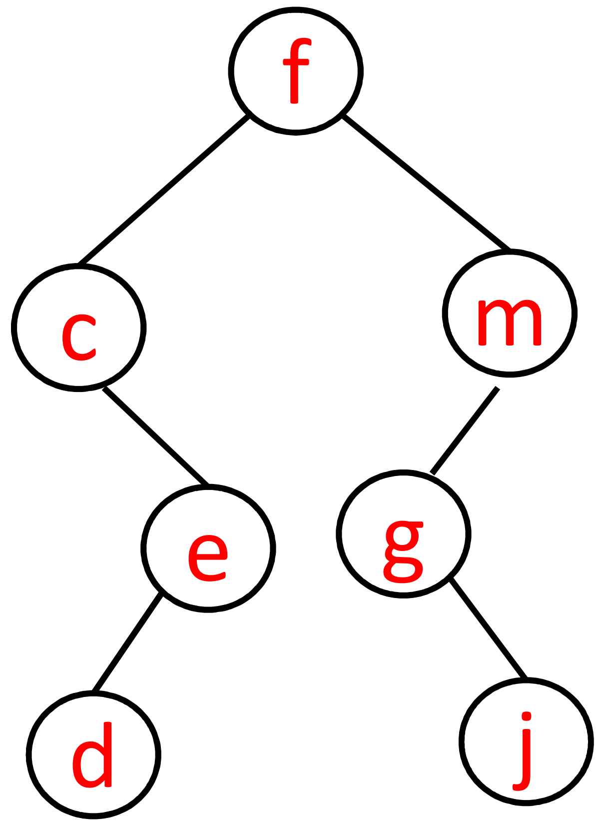
Time Complexity

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

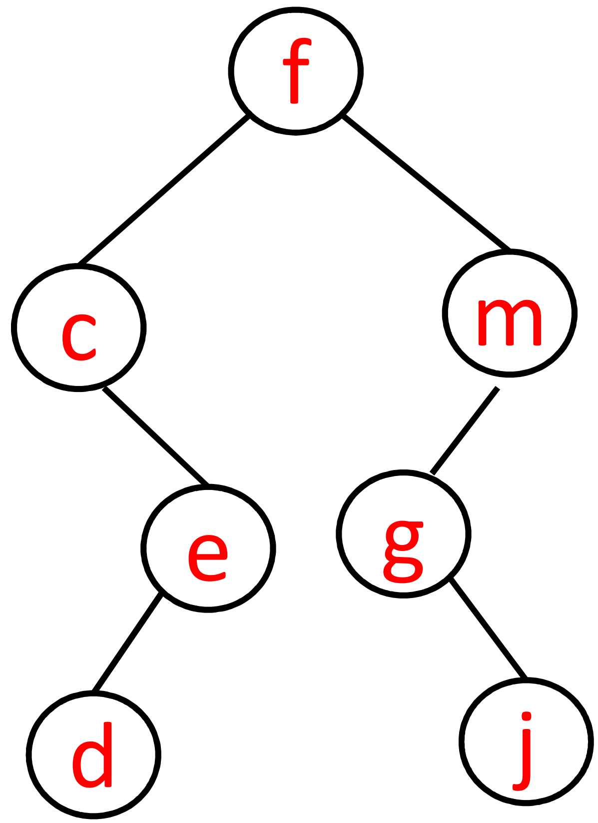
Time Complexity

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

findMax() returns ?



findMax() returns node m



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

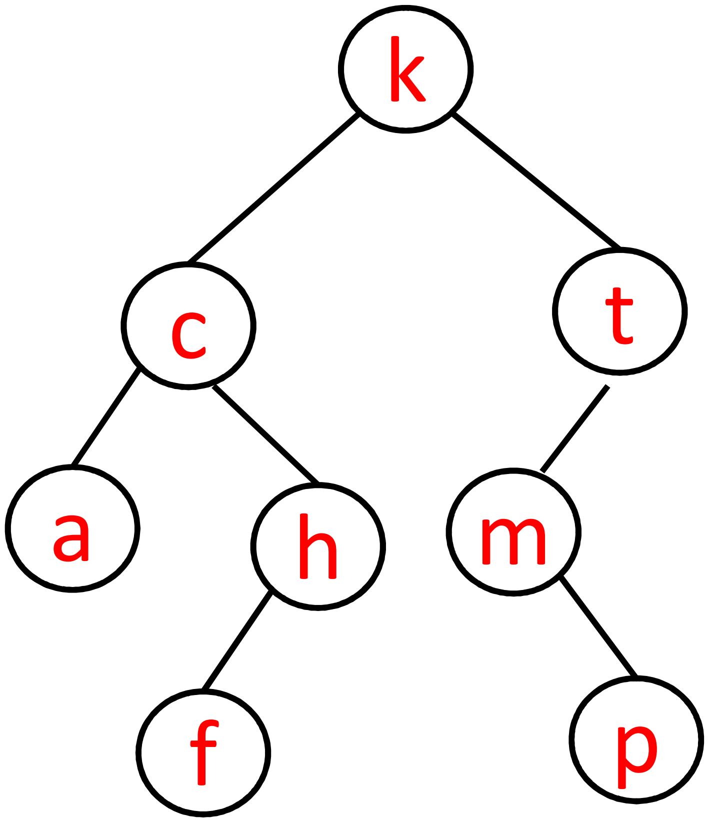
```
findMax(root){                    // returns a node
   if (root == null)
        return null
   else if (root.right == null))
        return root
   else
        return findMax (root.right)
 }
```

Time Complexity

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

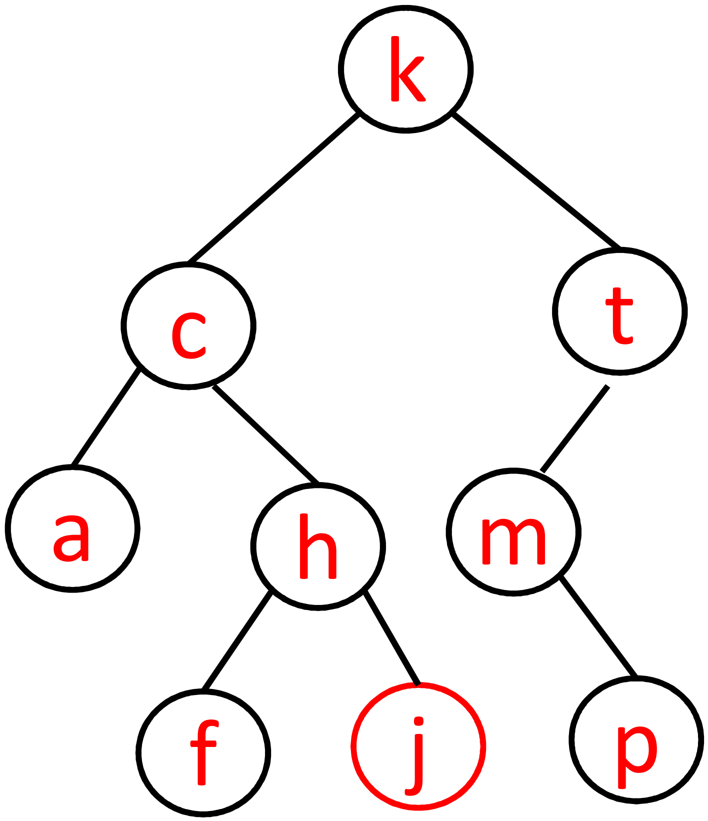
Time Complexity

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

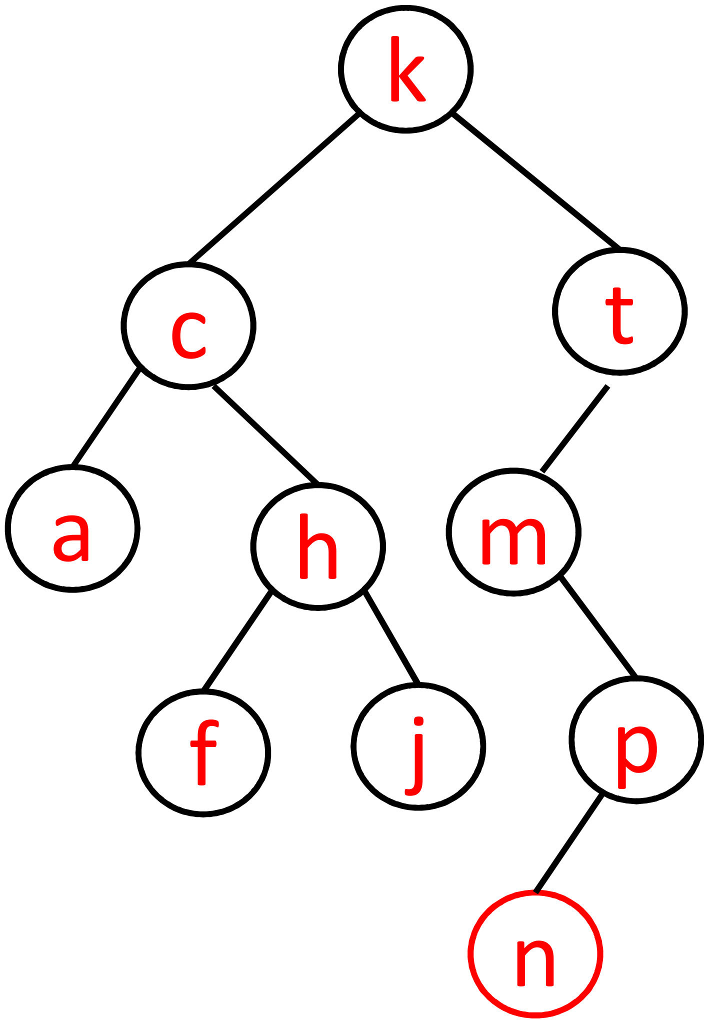


add(j)   ?

A new key is put at a new leaf.



add(n) ?


k
c
t
a
h
m
f
j
p
n

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

}
```

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

}
```

// assuming no duplicates allowed

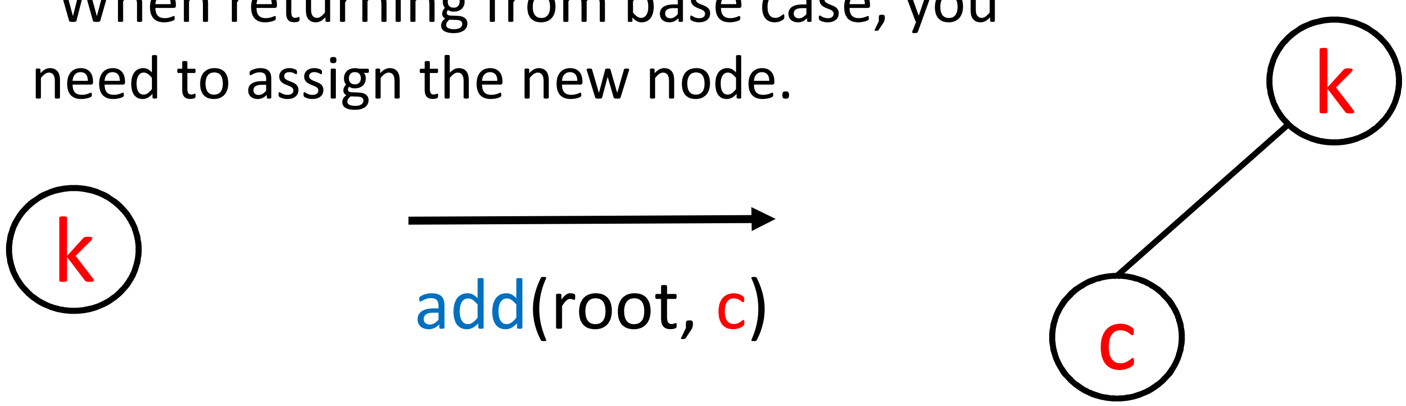
```
add(root, key){                    // returns root node
   if (root == null)
            root =  new BSTnode(key)
   else if (key < root.key){
            root.left    =  add(root.left, key)

}
```

```
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)
   //   If  root.key == key ,  then do nothing.
    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)
   //   If  root.key == key ,  then do nothing.
    return root
 }
```

Q: Why is it necessary to assign root.left ?

A: When returning from base case, you need to assign the new node.



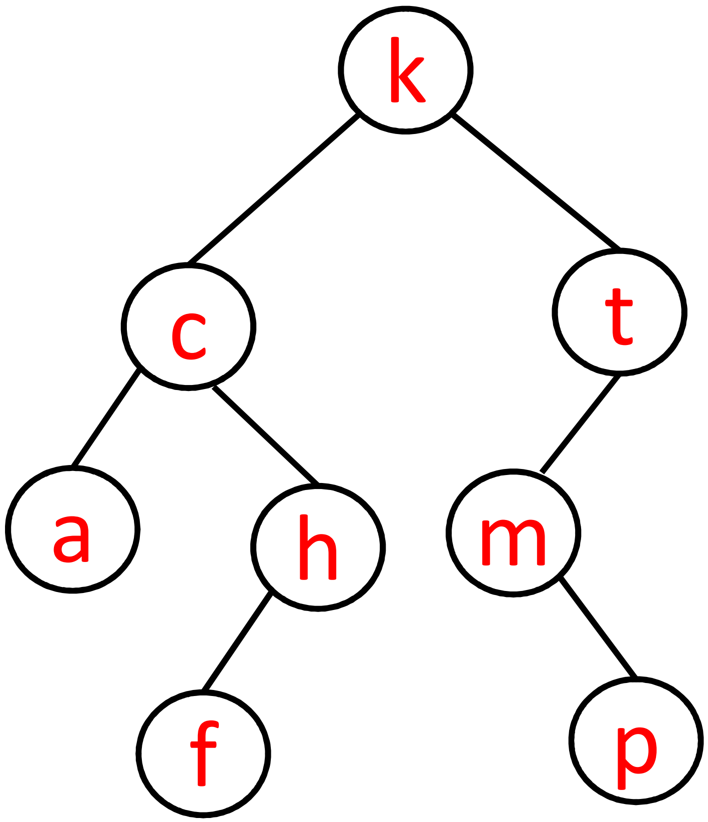
Time Complexity

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

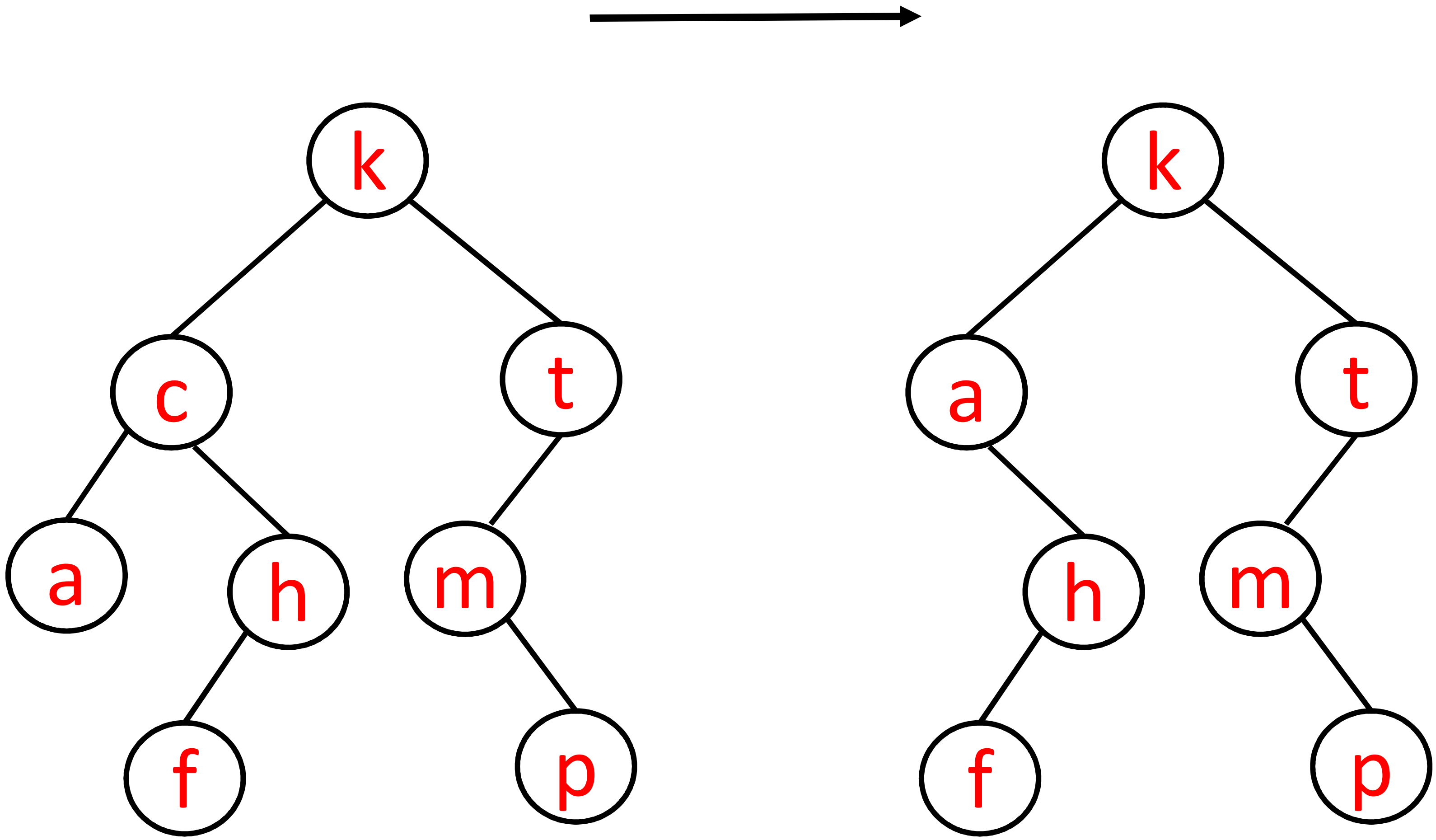
Time Complexity

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

remove(c)

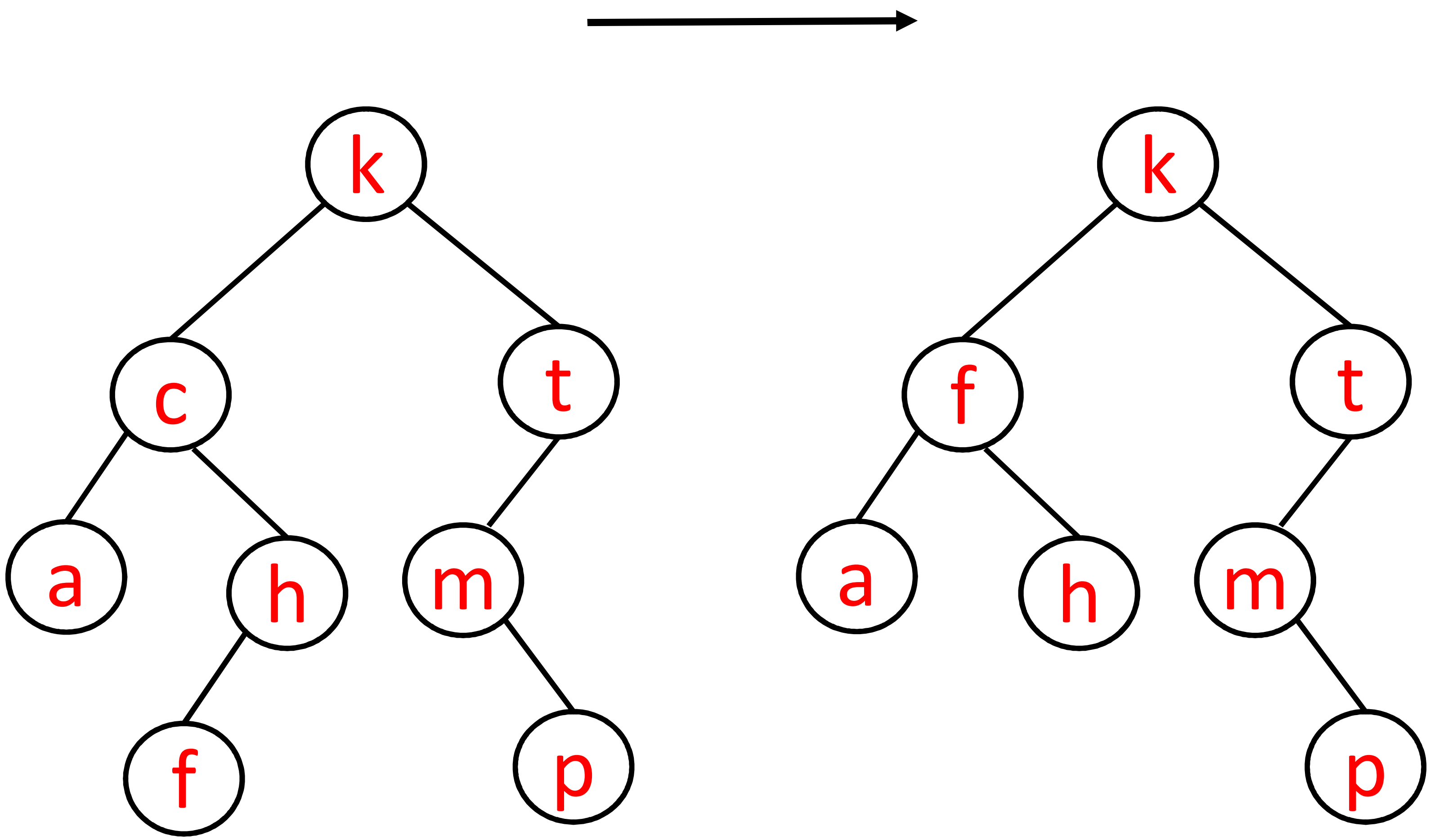


remove(c)



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

   return root
}
```

```
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   //   key == root.key

        What are the cases to consider?

   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   //   key == root.key
```

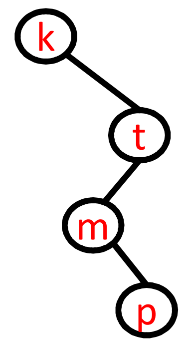
Three cases are shown at right.

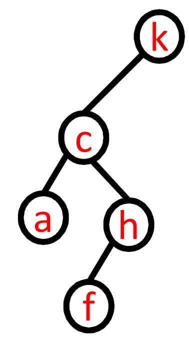
- left child is null
- right child is null
- neither child is null

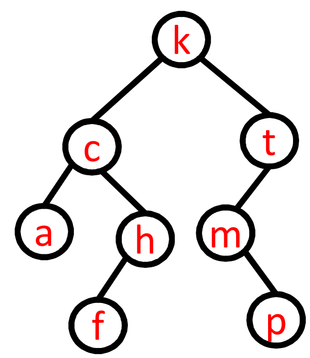
```
    return root;
}
```

Example:

remove(k)



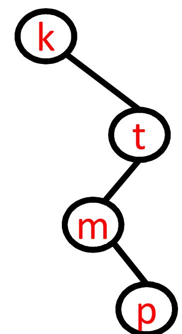




```
remove(root, key){                              // returns root node
   if( root  == null )
      return null
   else  if ( key < root.key )
      root.left =  remove ( root.left, key )
   else  if ( key > root.key )
      root.right = remove ( root.right, key)
   else    //   key == root.key
  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

    //   Note above that if root.right is also null,   then root
         will become null,   e.g. if we are removing a leaf.

   return root;
}
```

Example:

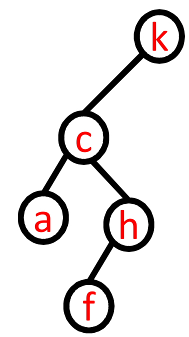
remove(k)



```
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    //   key == root.key
      if root.left == null
          root  =  root.right
      else  if root.right == null  //  and root.left is not null
          root  = root.left
      else{   //   neither left nor right child is null

      }
   return root;
}
```

Example:

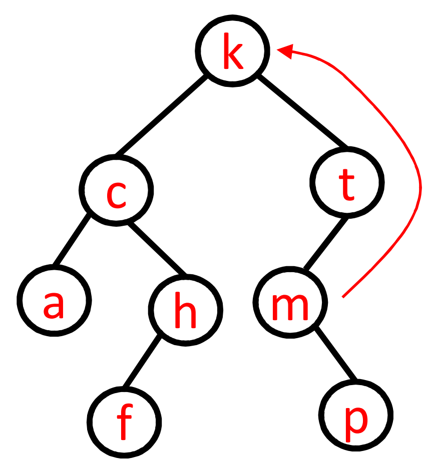
remove(k)



```
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  //   key == root.key
      if root.left == null
         root  =  root.right
      else  if root.right == null
         root  = root.left
      else  {   //   neither left nor right child is null
         root.key    =  findMin( root.right).key

      }
      return root;
}
```

Example:

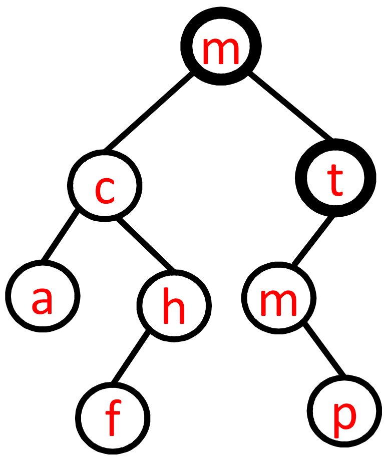
remove(k)



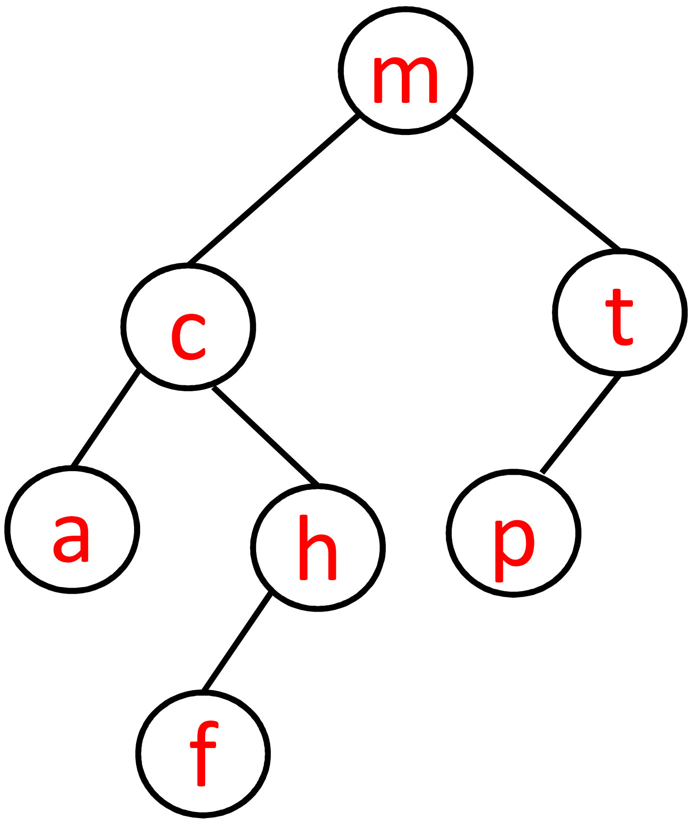
```
remove(root, key){                    // returns root node
  if( root  == null )
     return null
  else  if ( key < root.key )
     root.left =  remove ( root.left, key )
  else  if ( key > root.key )
     root.right = remove ( root.right, key)
  else  //   key == root.key
     if root.left == null
        root  =  root.right
     else  if root.right == null
        root  = root.left
     else  {   //   neither left nor right child is null
         root.key    =  findMin( root.right).key
         root.right  =  remove( root.right,  root.key )
     }
     return root;
}
```

Example:

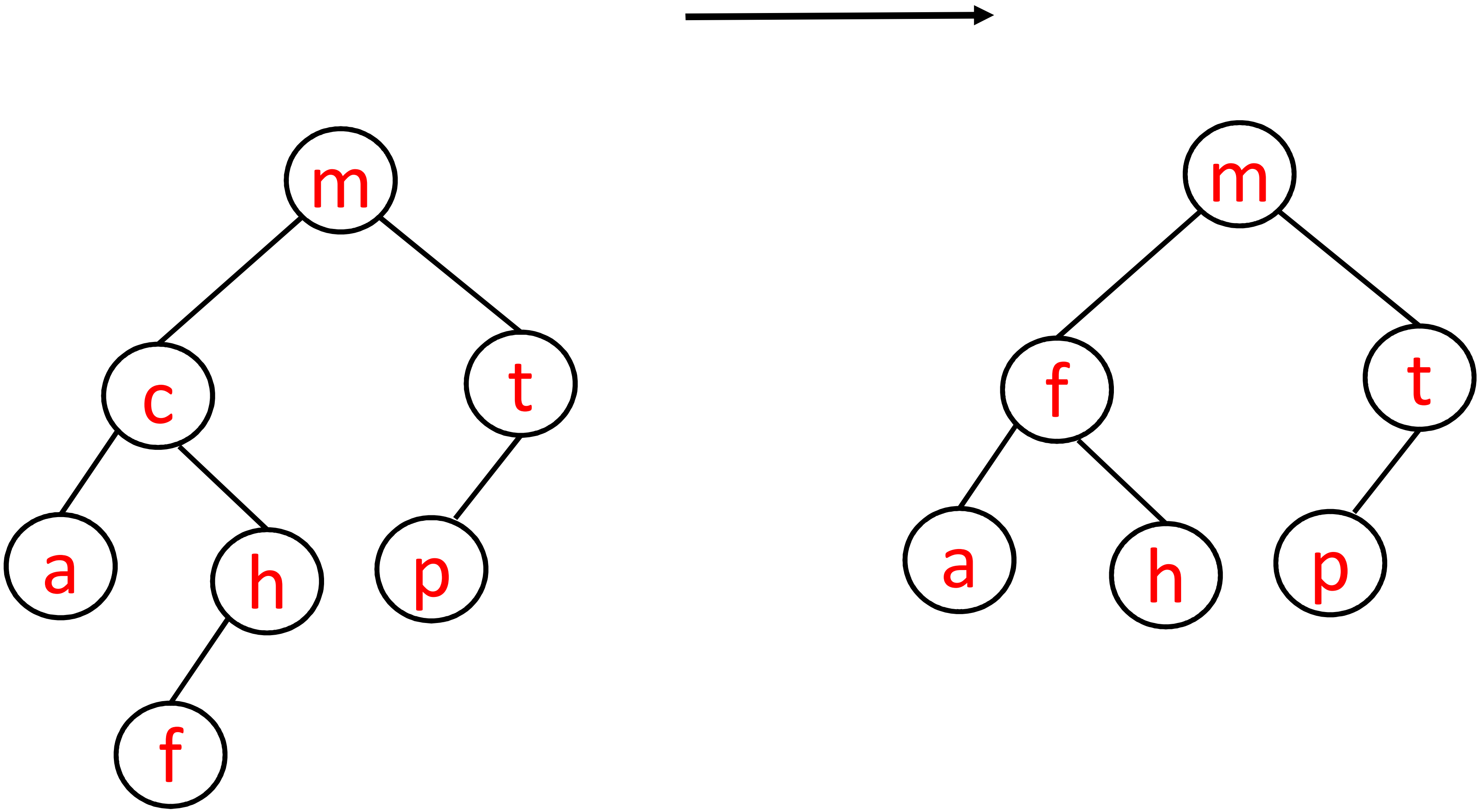
remove(k)



remove(c)



remove(c)



Time Complexity

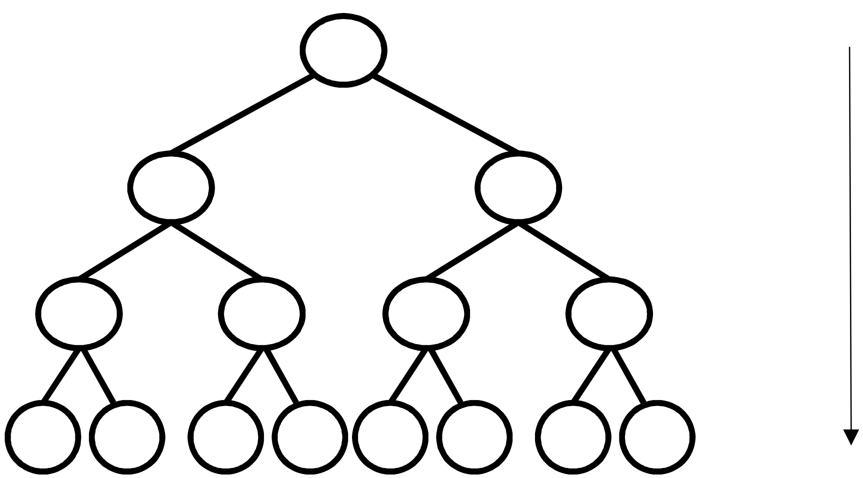
	best case	worst case
find(key)	O(1)	O(n)
findMin()	O(1)	O(n)
findMax()	O(1)	O(n)
add(key)	O(1)	O(n)
remove(key)		

Time Complexity

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

ASIDE: Balanced Binary Search Trees

When a binary search tree is *balanced*, then finding a key is very similar to a binary search. In COMP 251, you will learn algorithms for maintaining balanced binary search trees.



From last lecture, for a binary tree with all levels full:

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

ASIDE: Balanced Binary Search 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)	$O(\log n)$	$O(\log n)$