

COMP 250

Lecture 20

binary search trees

Oct. 26, 2016

# Assignment 2 Q1 grading

**Evaluation**

**Rubrics**  
No Rubric Selected. [\[Associate Rubric\]](#) [\[Create Rubric\]](#)

**Score**  
 / 60  
Grade Item: A2Q1

**Student View Preview**  
- / 60

**Feedback**

Rich text editor toolbar: Bold, Italic, Underline, Paragraph, Bulleted List, Numbered List, Indent, Outdent, Undo, Redo, Link, Unlink, Image, Table, Source Code, Full Screen, Help.

Buttons: Add a File, Record Audio

Student name

Oct 24, 2016 11:19 AM - A2Q1submi... (1.46


*Please only email Antoine (TA) if necessary and only after you run tester code and you understand my solution.*


Comment file



My Home > Fall 2016 - COMP-250... ▾

Content | Discussions | Assignments | Lecture Recordings

 **McGill** | **MY COURSES**

Search Topics 

Overview  
Bookmarks  
Course Schedule

**Table of Contents** 3

Here are files I prefer not to put on the public web page. 3


Add a module...


Table of Contents ▾


Import Course ▾ Bulk Edit Related Tools ▾

≡ Here are files I prefer not to put on the public web page.


Upload / Create ▾ Add Existing Activities ▾

≡  **public web page** ▾

≡  **a1solution** ▾

≡  **A2Q1\_solution** ▾

Add a module..



Tester + SOLUTION

~~(binary search) tree~~

binary (search tree)

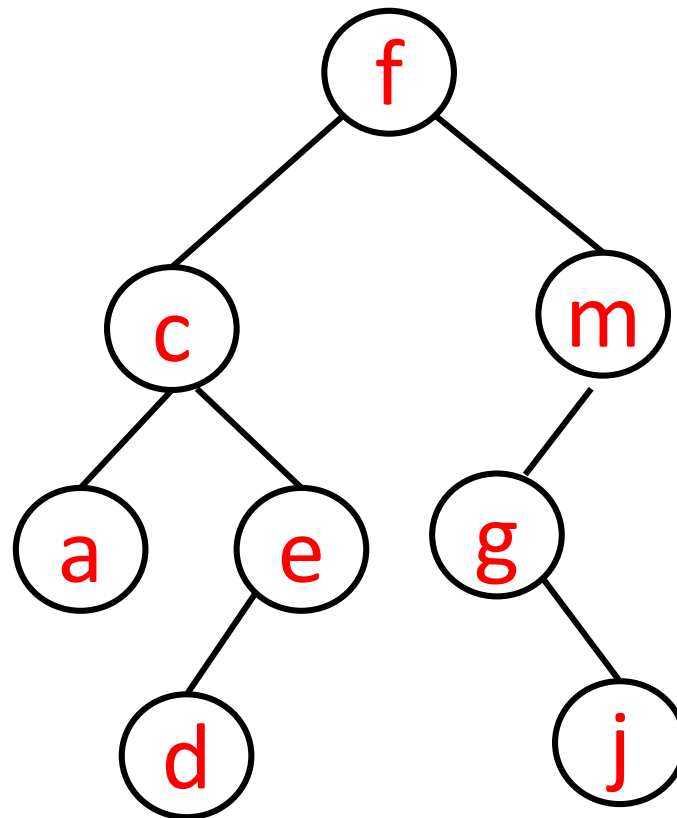
```
class BSTNode< K >{  
    K          key;  
    BSTNode< K > leftchild;  
    BSTNode< K > rightchild;  
    :  
}
```

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

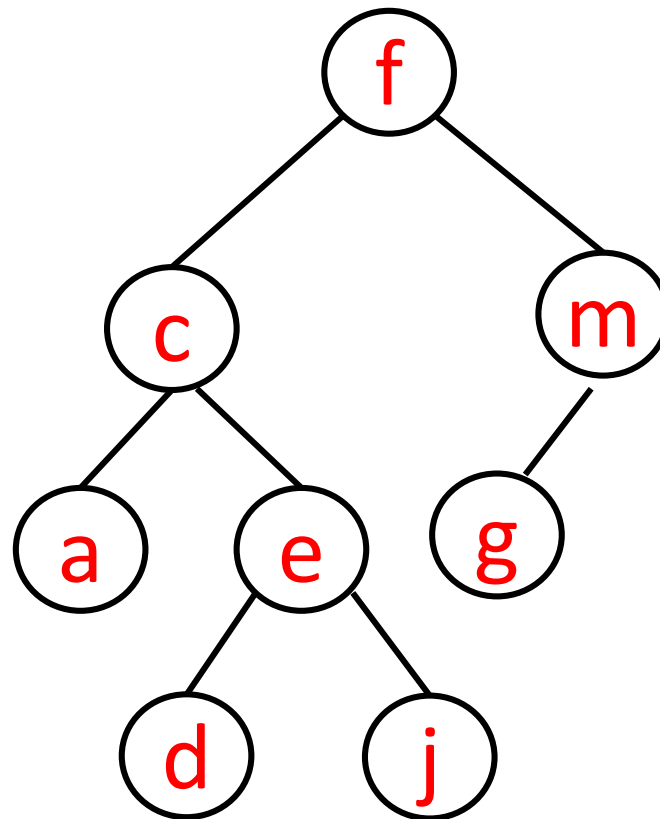
# Binary Search Tree Definition

- binary tree
- keys are comparable, unique
- 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



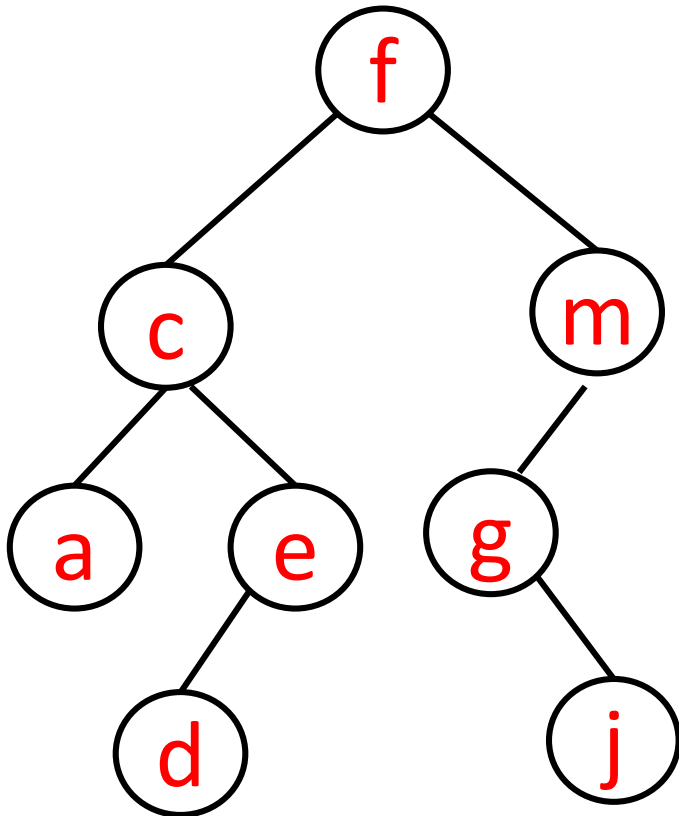
This is not a BST. Why not?





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

Proof: Exercise

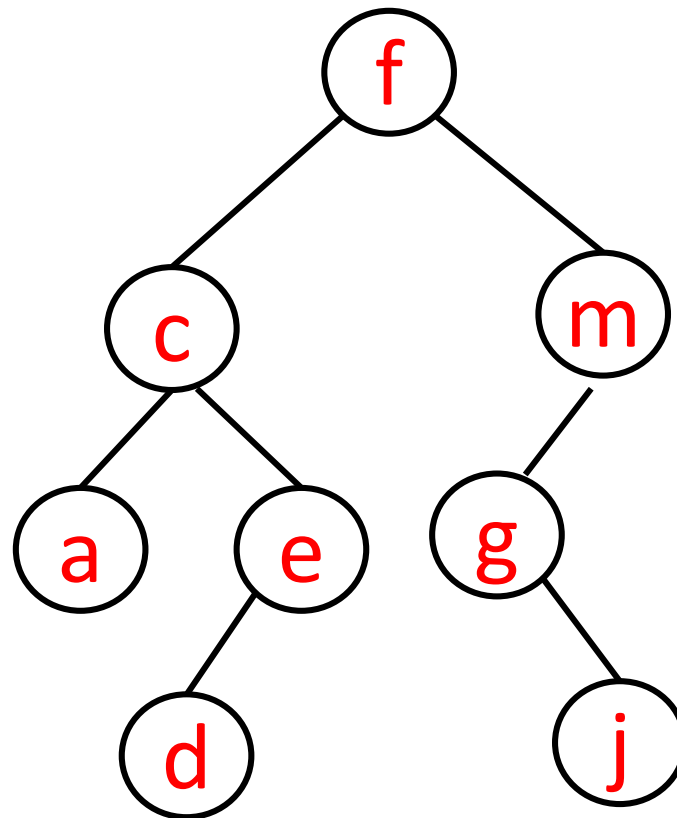


acdefgjm

# Binary Search Tree ADT

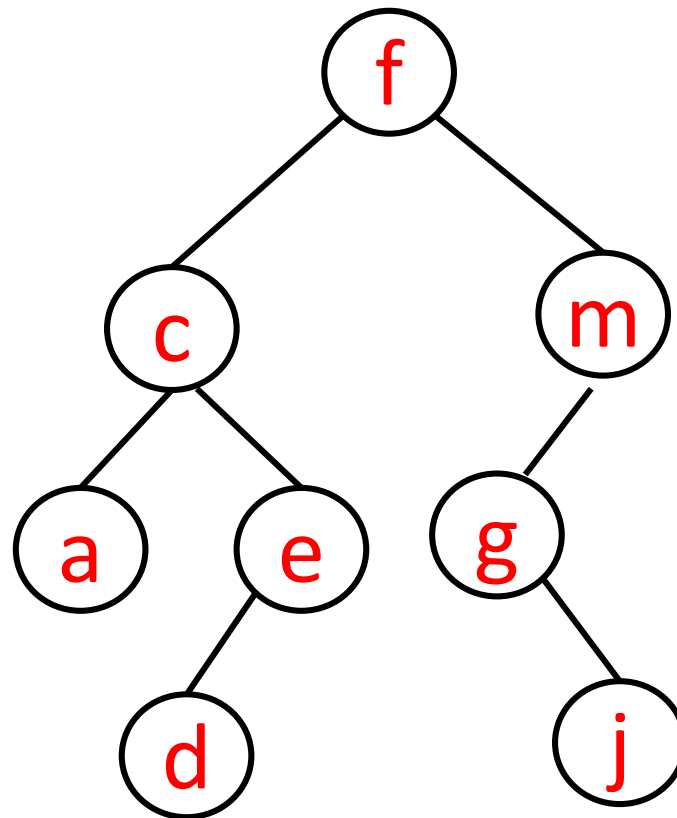
- find( key )
- findMin()
- findMax()
- add(key)
- remove(key)

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

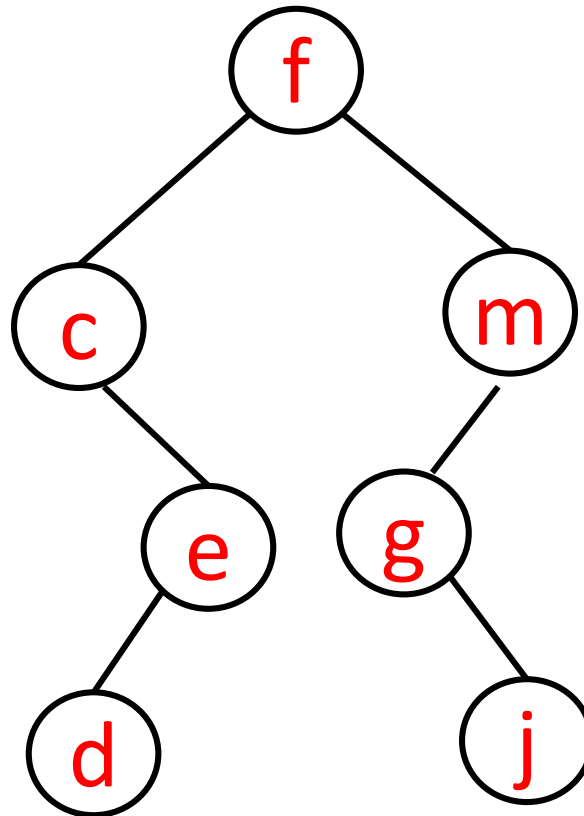


```
find(root,key){           // returns a node
    if (root == null)
        return null
    else if (root.key == key)
        return root
    else if (key < root.key)
        return find(root.left, key)
    else
        return find(root.right, key)
}
```

findMin() returns **a** node

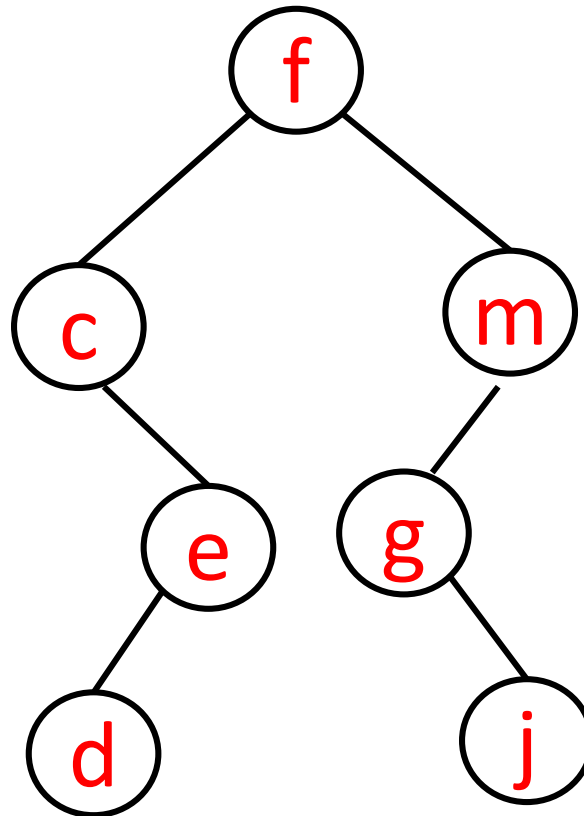


findMin() returns **c** node



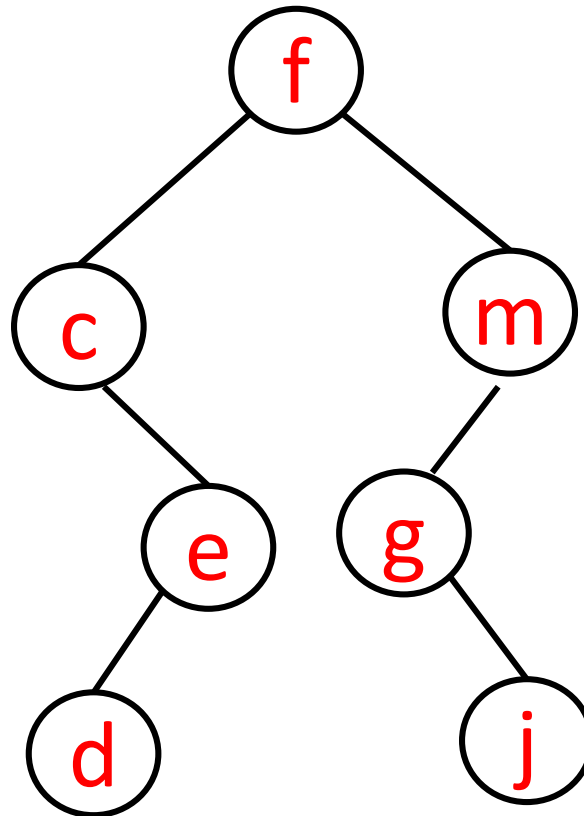
```
findMin(root){  
    if (root == null)  
        return null  
    else if (root.left == null)  
        return root  
    else  
        return findMin( root.left )  
}
```

findMax() returns ?

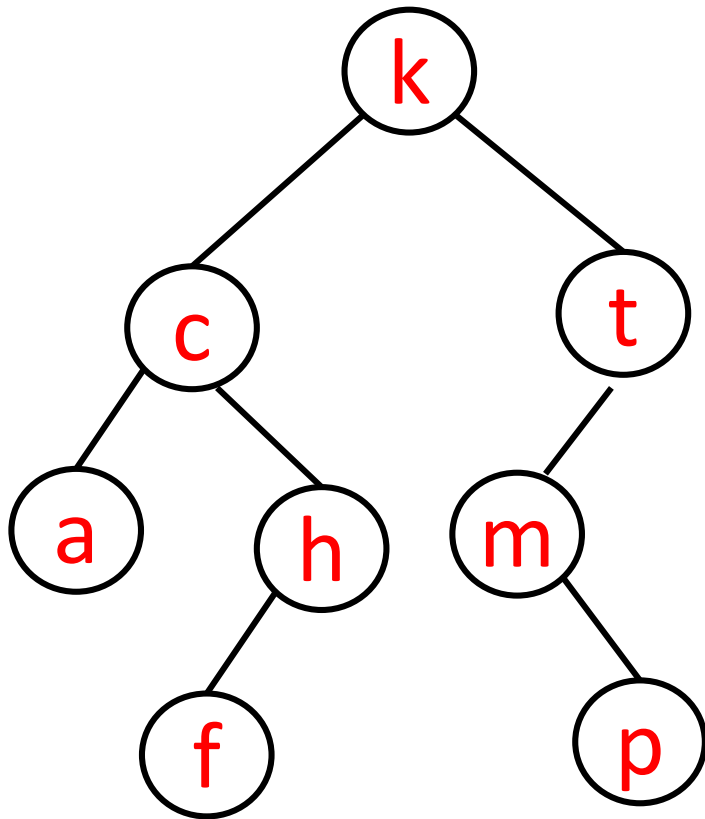




findMax() returns **m** node



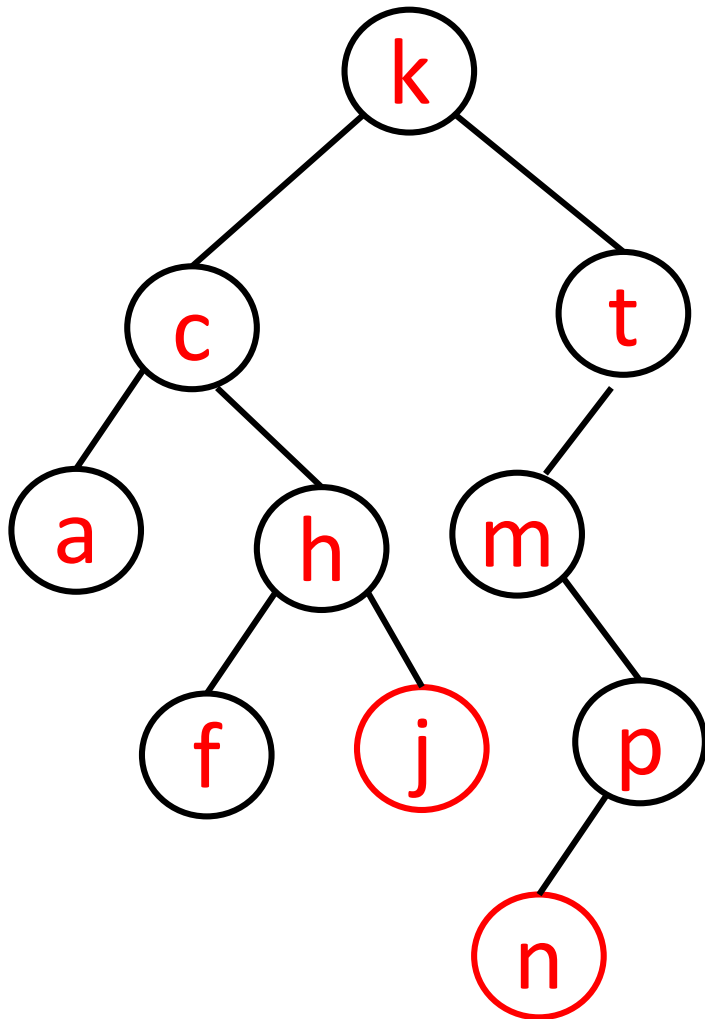
```
findMaximum(root){  
    if (root == null)  
        return null  
    else if (root.right == null)  
        return root  
    else  
        return findMaximum(root.right)  
}
```



add( **j** ) ?

add( **n** ) ?

A new key is  
always a leaf.



add( **j** ) ?

add( **n** ) ?

A new key is  
always a leaf.

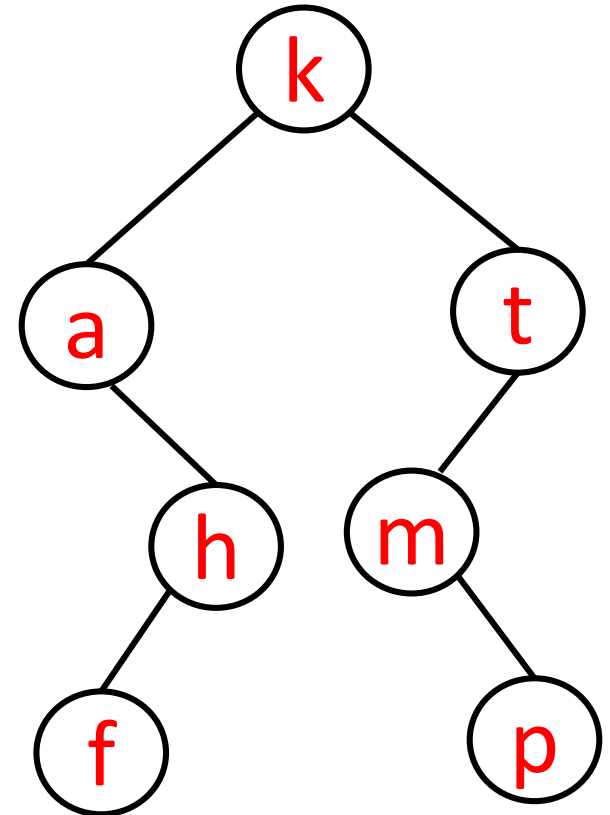
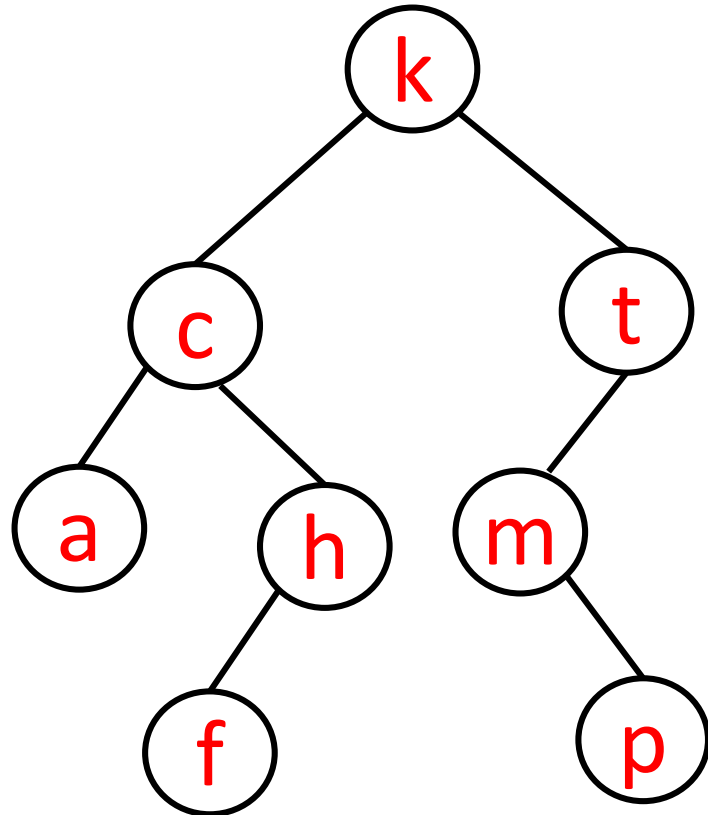
```
add(root, key){  
    if (root == null)  
        root = new BSTnode(key)  
    else if (key < root.key){  
        root.left = add(root.left, key)  
    else if (key > root.key){  
        root.right = add(root.right, key)  
  
    return root  
}
```

Does this handle base case properly ?

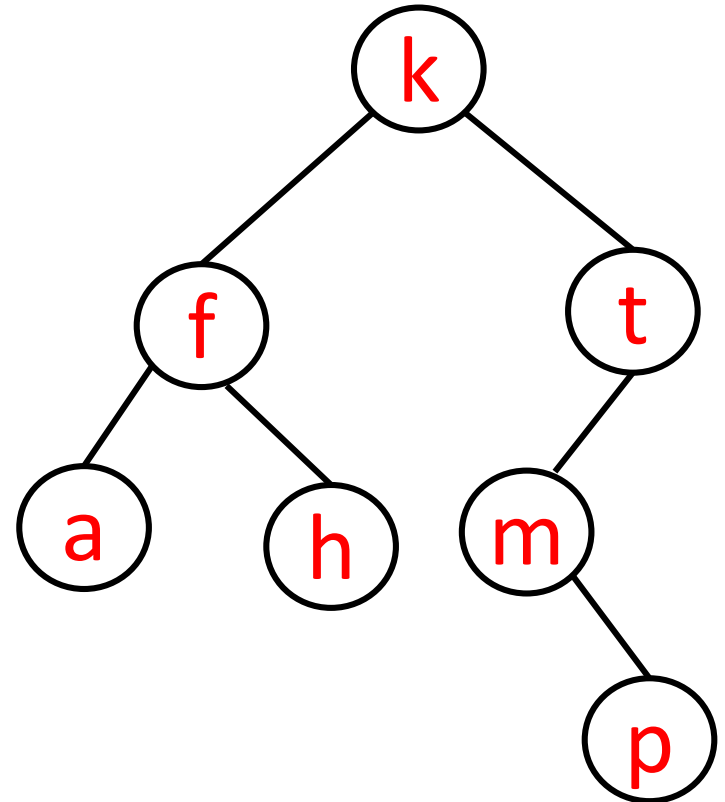
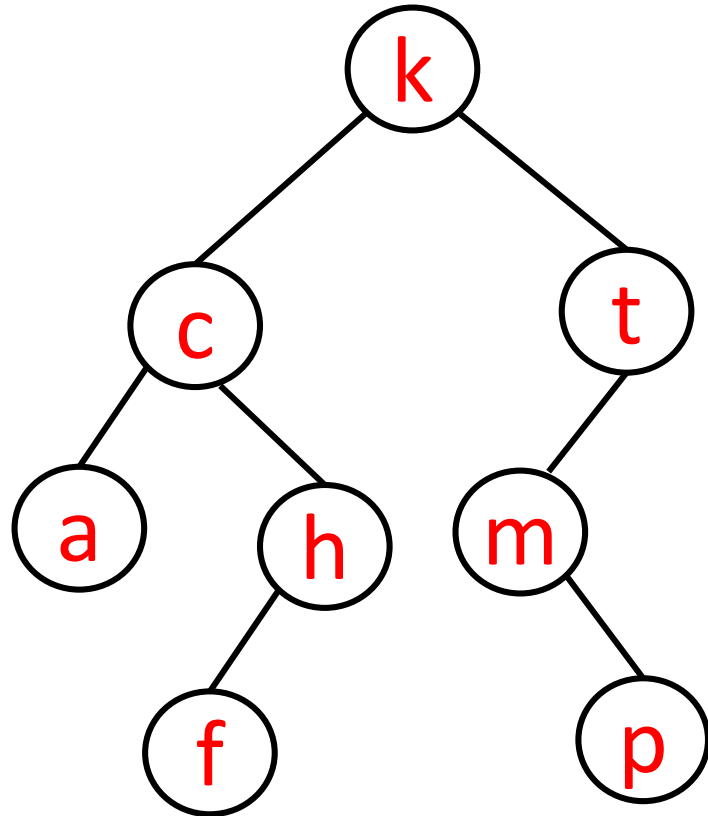
The add calls are a bit weird, no ?

What if root.key == key ?

remove( c )



remove( c )



The algorithm I present  
does it slightly differently  
(next slide)

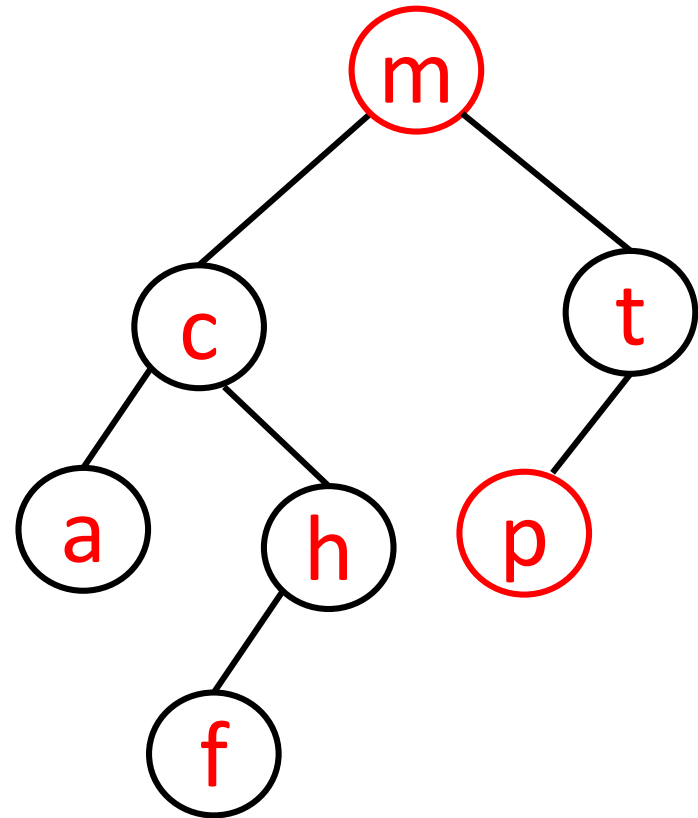
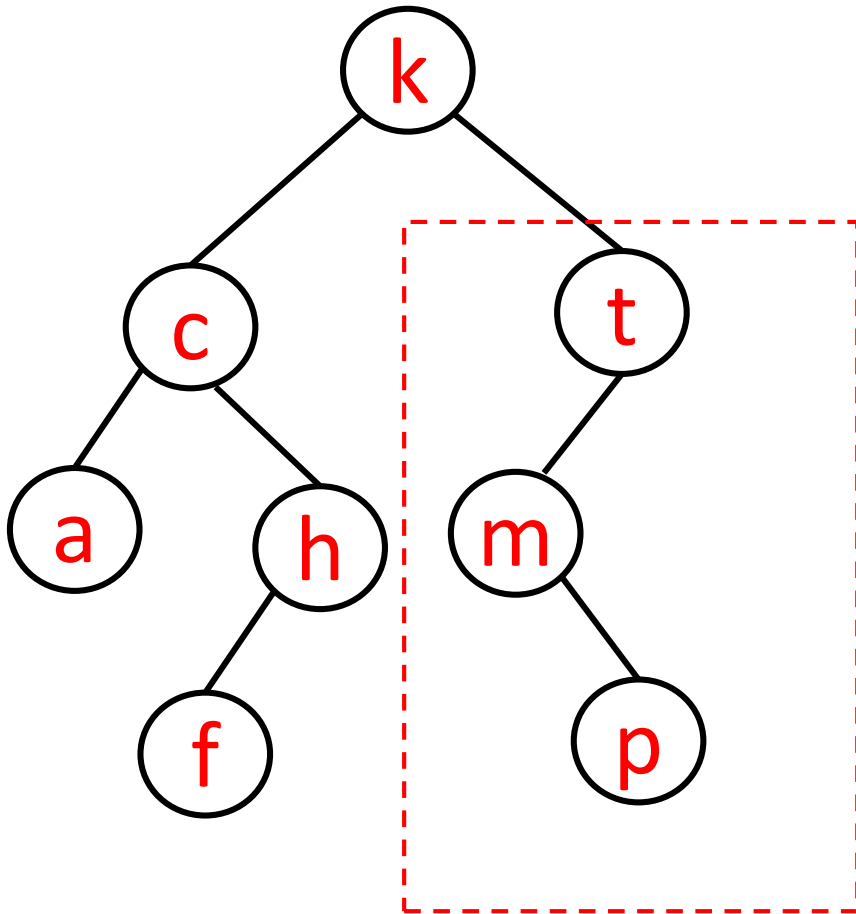
```
remove(root, key){  
    if( root == null )  
        return null  
    else if ( key < root.key )  
        root.left = remove ( root.left, key )  
    else if ( key > root.key )  
        root.right = remove ( root.right, key )  
    else if root.left == null  
        root = root.right  
    else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



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



Next lecture:

Binary Search Trees vs. Binary Search (with array list)

Best and worst case

Big O, big Omega, big Theta, and “Limits”

Some discussion of Assignment 3