COMP 250

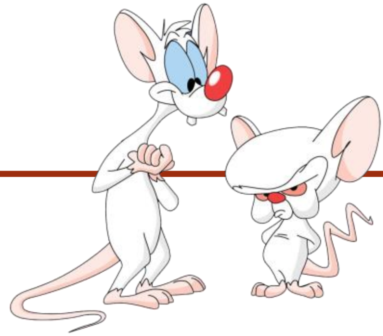
INTRODUC TER SCIENCE

Week 12-1 : Binary

Giulia Alberini, Fall 2020

Slides adapted from Michael Langer's

WHAT ARE WE GOING TO DO IN THIS VIDEO?



- Binary Search Trees

B H

BSTNode

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

```
clas
   K key;
   BSTNode<K> leftchild;
   BSTNode<K> rightchild;
       :
}
```

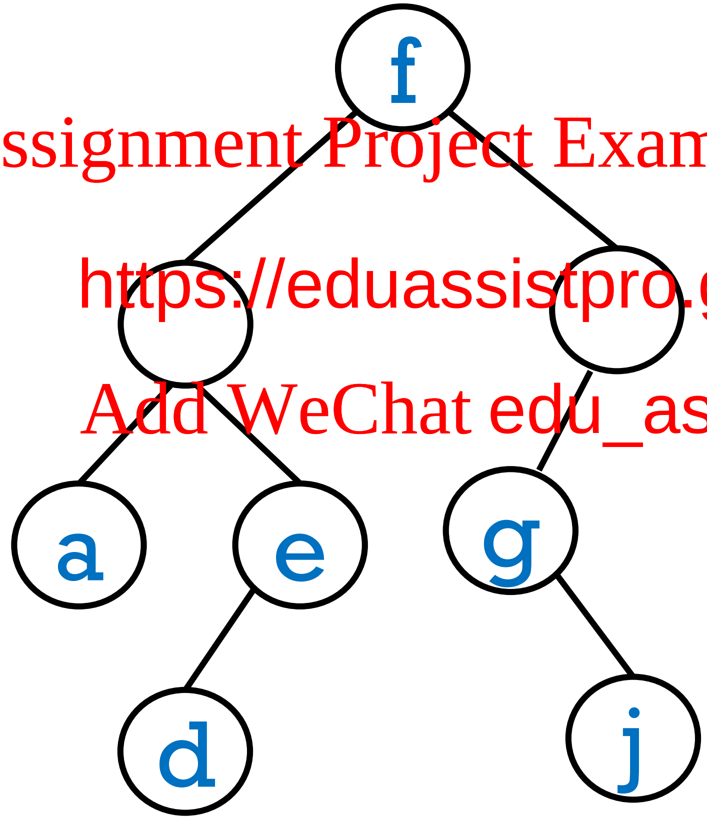
BINARY SEARCH TREE DEFINITION

- binary tree

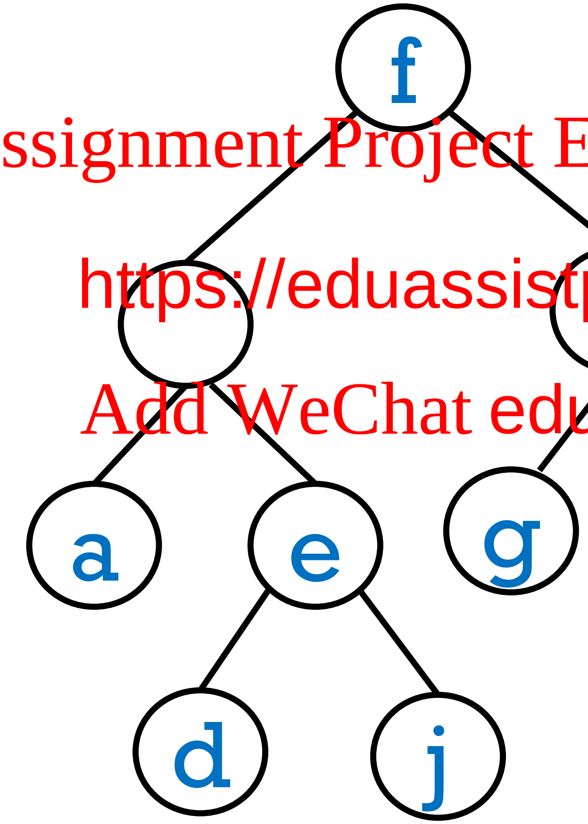
- keys are comparable, a s)

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

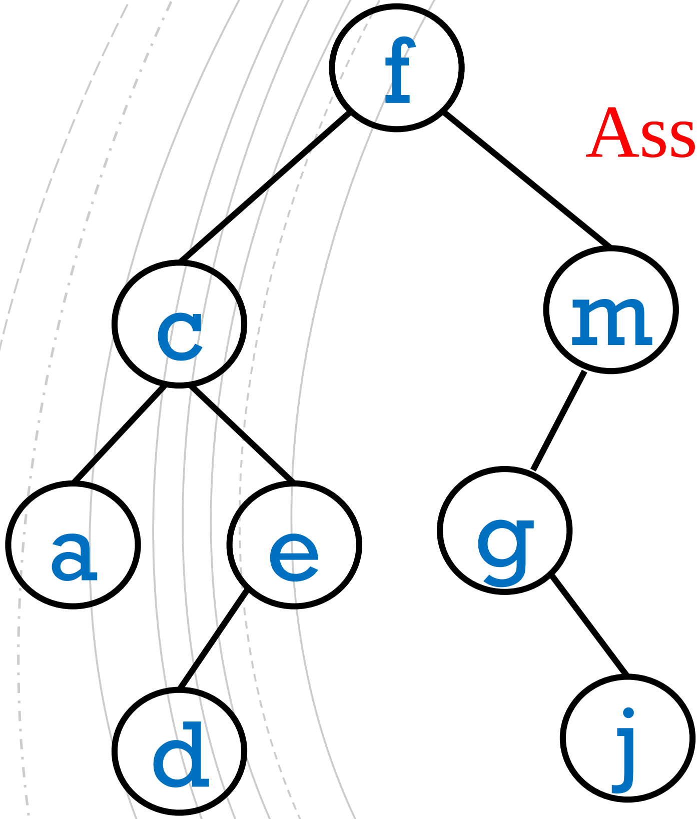
EXAMPLE



THIS IS NOT A BST. WHY NOT?



BST - TRAVERSALS



An in-order traversal on a BST visits the natural order

he key.

a c

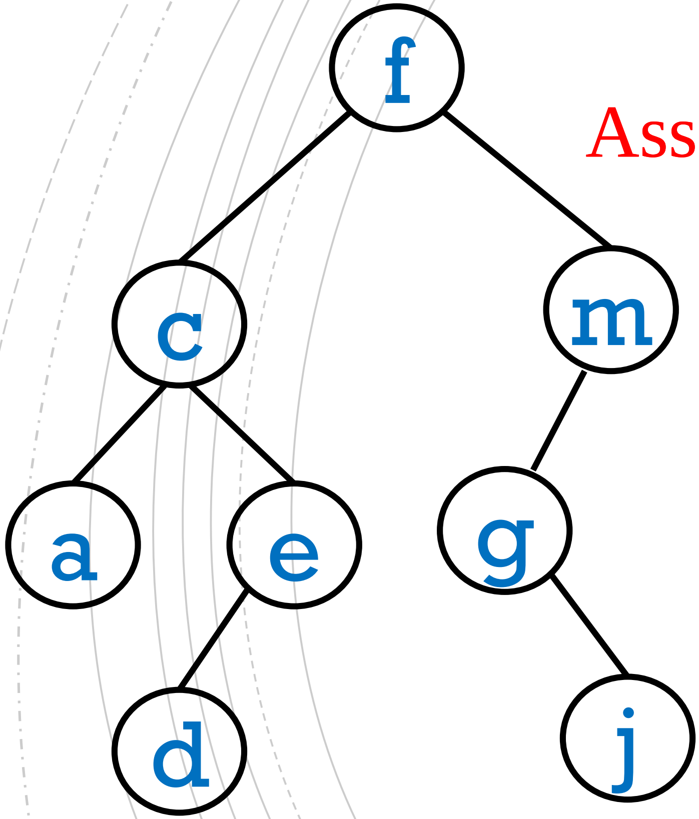
BINARY SEARCH TREE ADT

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

We can define the operations of a BST without ow they are ed. (ADT)

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

FIND()

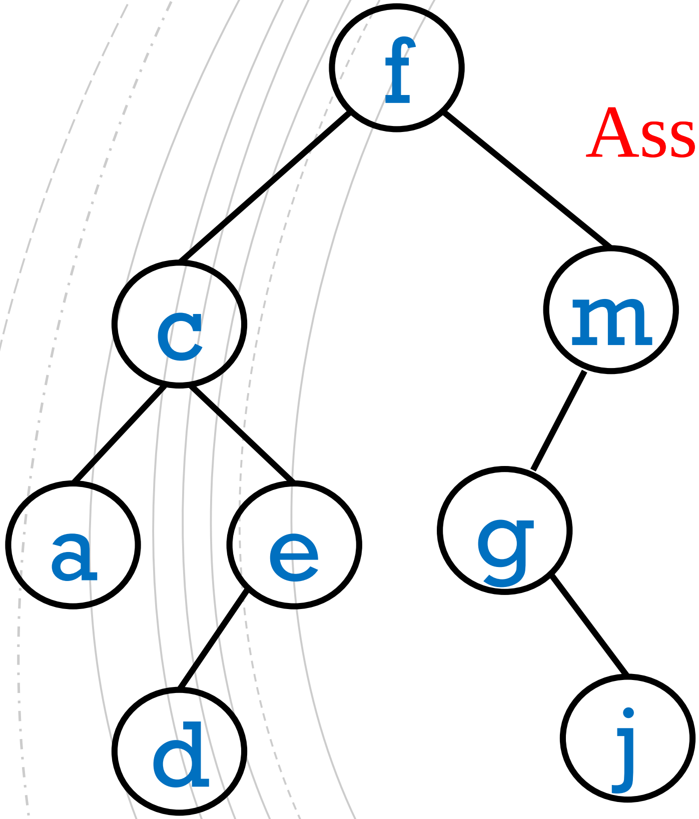


viour:

, g) returns the g node

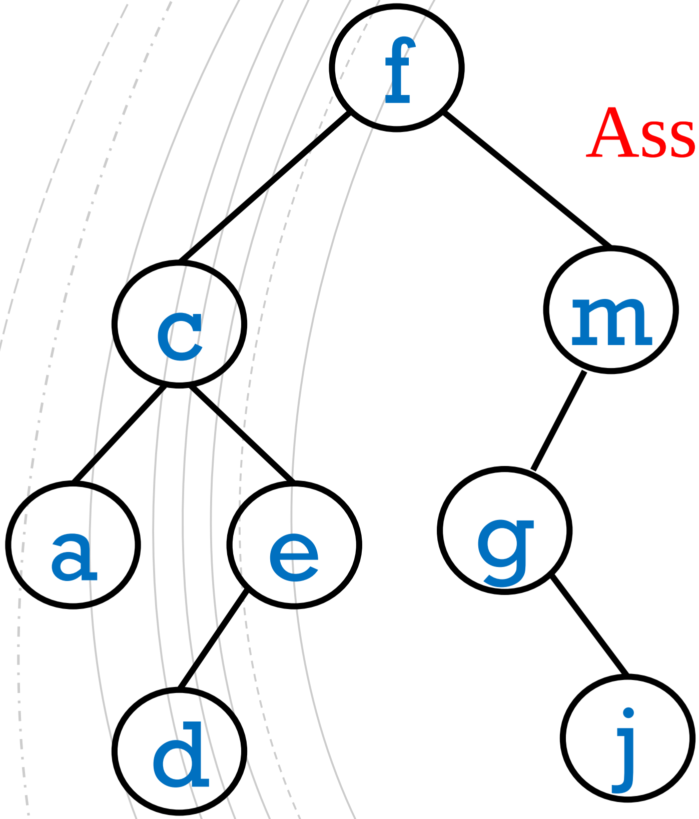
- f s) returns null.

FIND() – IMPLEMENTATION



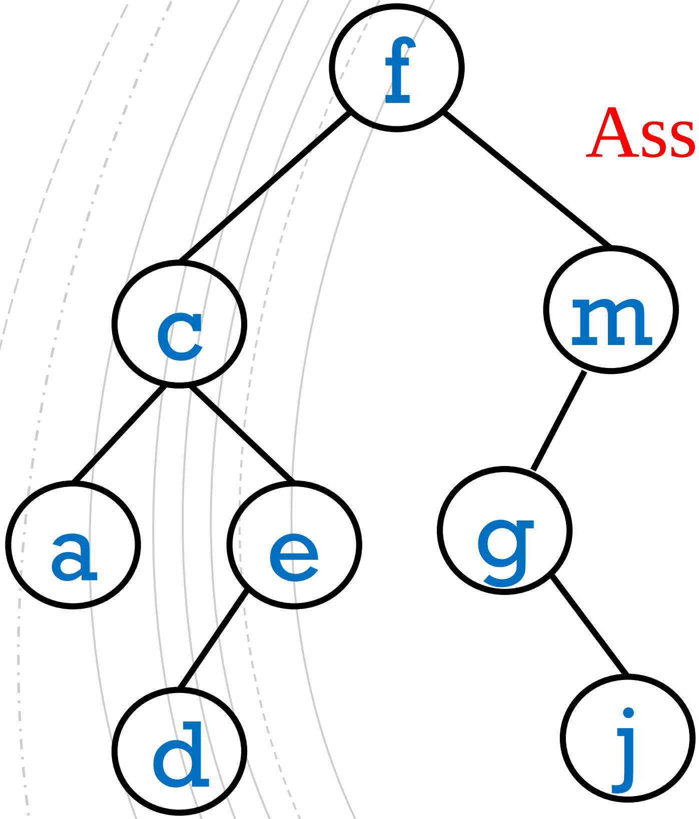
```
find(root, key){ // returns a node
    if (root == null)
                    null
                  ot.key == key))
                    oot



}
```

FIND() – IMPLEMENTATION



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

FINDMIN()



uld return the node with

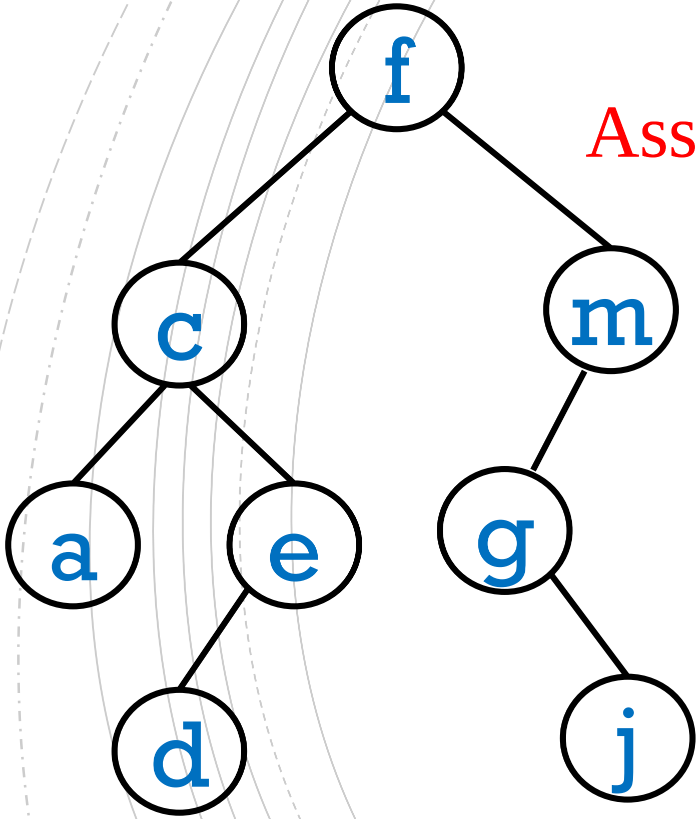
ey. So, for example given

the

eft,

- `findMin(root)` **returns … ?**

FINDMIN()



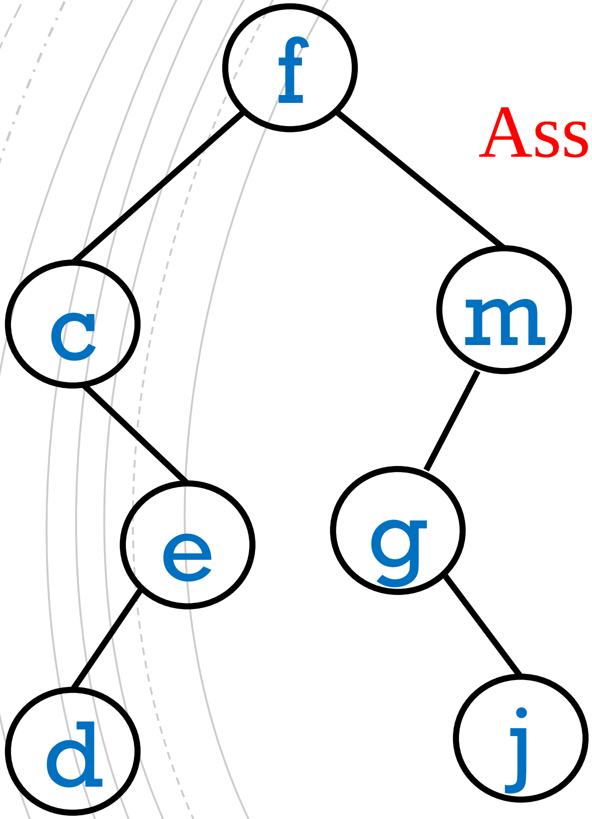
uld return the node with

ey. So, for example given

the eft,

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

FINDMIN()



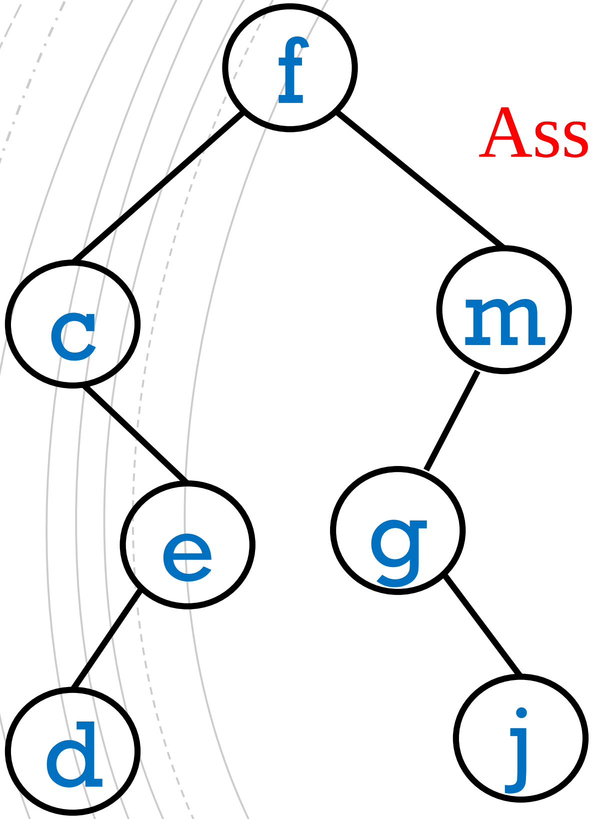
uld return the node with

ey. So, for example given

the eft,

- `findMin(root)` **returns … ?**

FINDMIN()



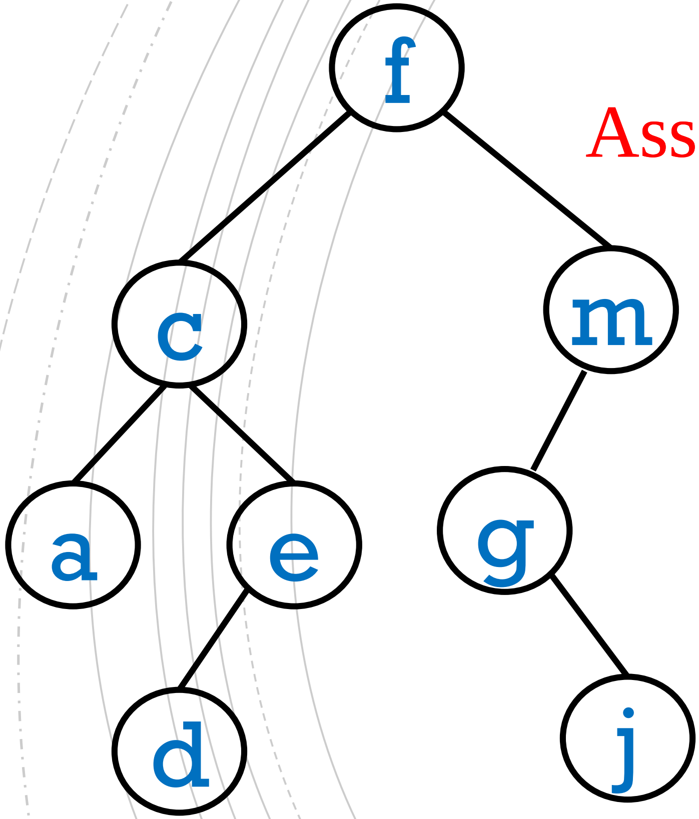
uld return the node with

ey. So, for example given

the eft,

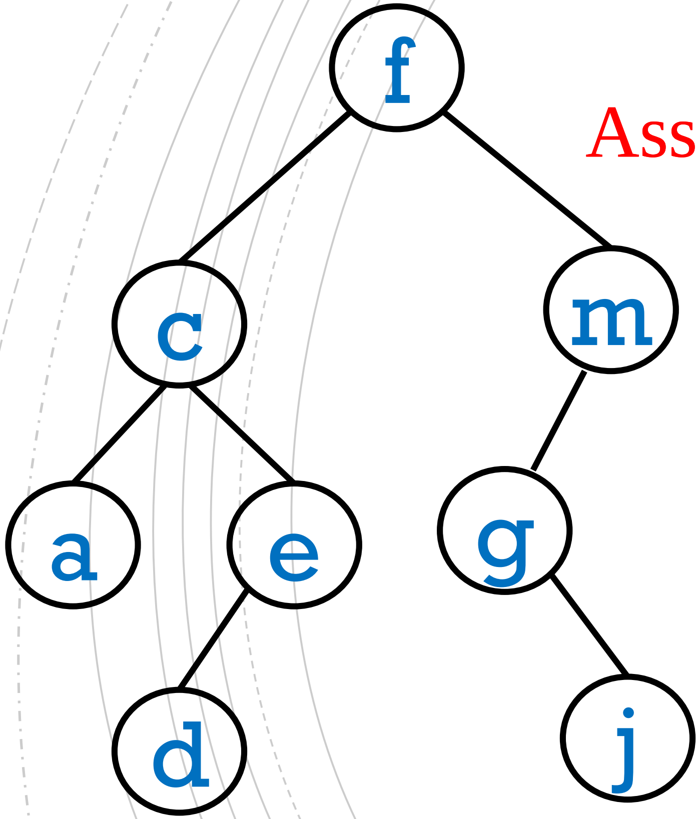
- `findMin(root)` **returns the c node**

FINDMIN() - IMPLEMENTATION



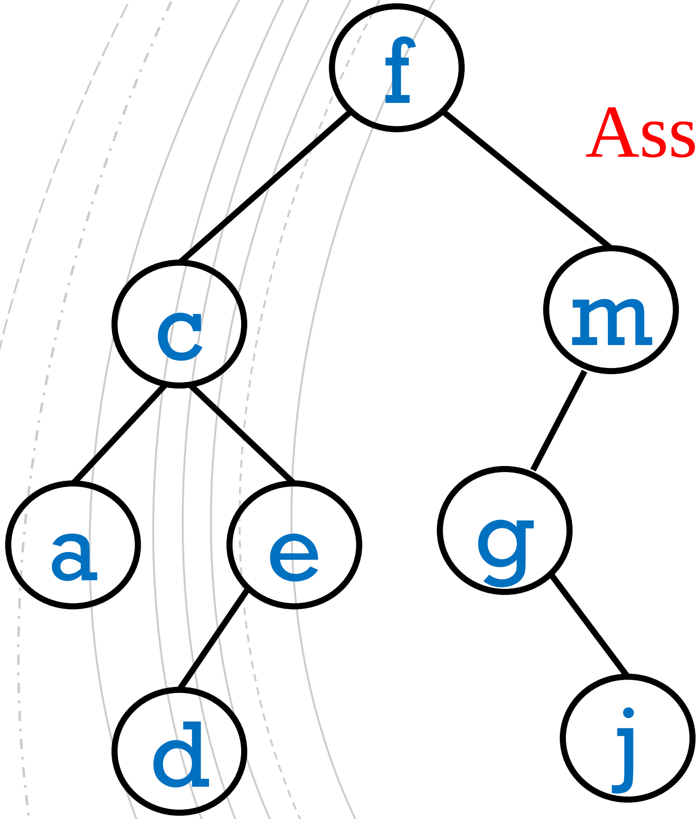
```
findMin(root){ // returns a node
        if (root == null)
                return null



}
```

FINDMIN() - IMPLEMENTATION



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

FINDMAX()



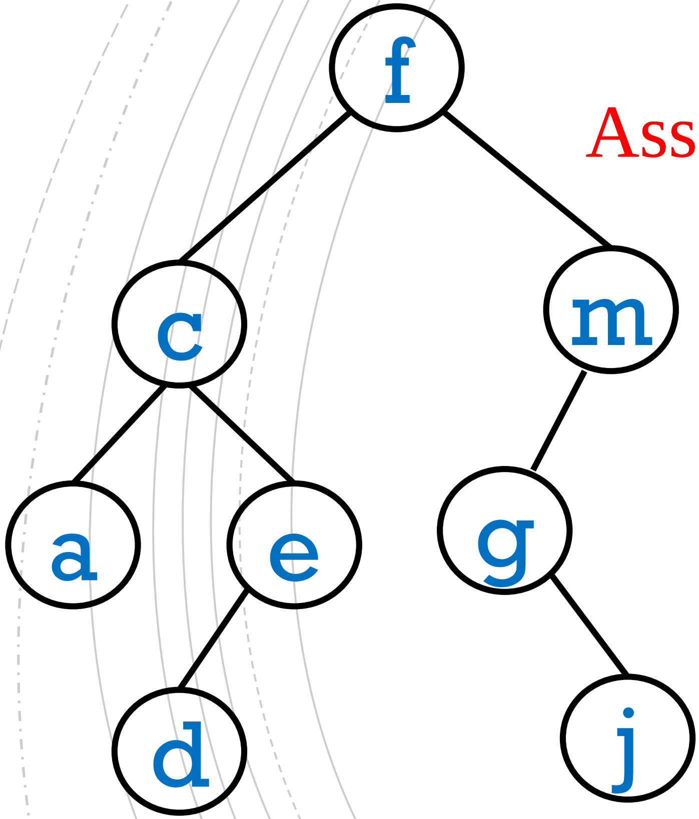
uld return the node with

ey. So, for example given

the eft,

- `findMax(root)` **returns … ?**

FINDMAX()



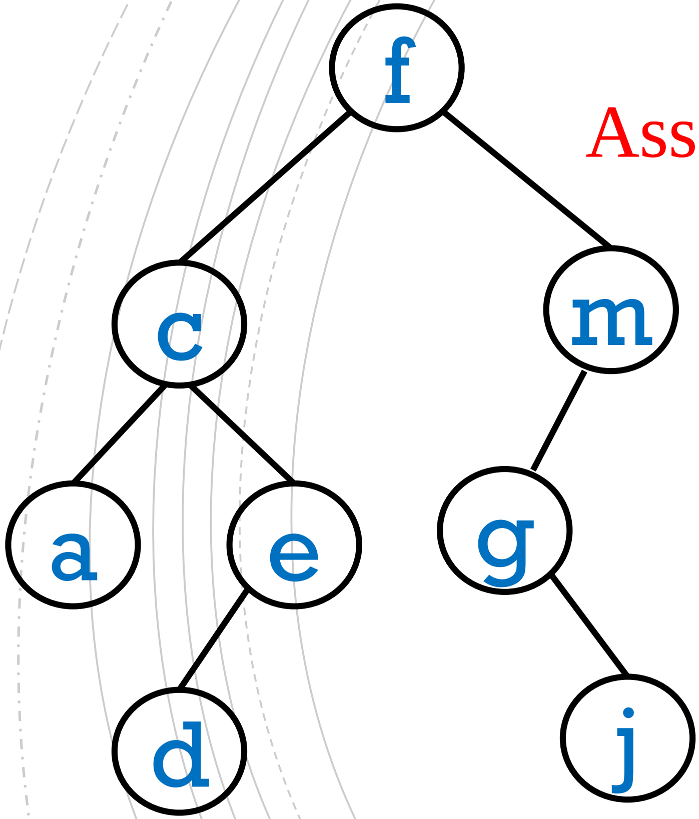
uld return the node with

ey. So, for example given

the eft,

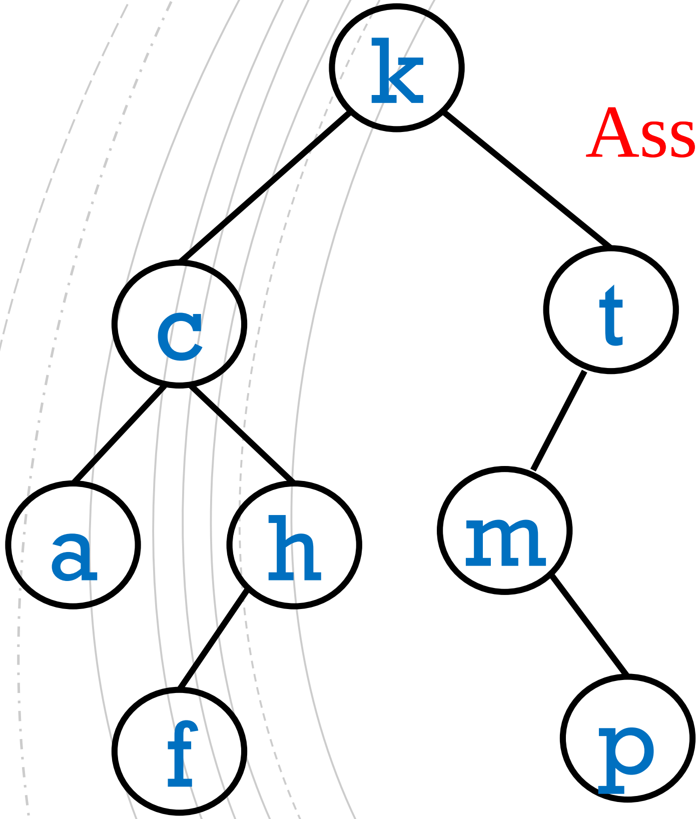
- `findMax(root)` returns the m node.

FINDMAX() – IMPLEMENTATION



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

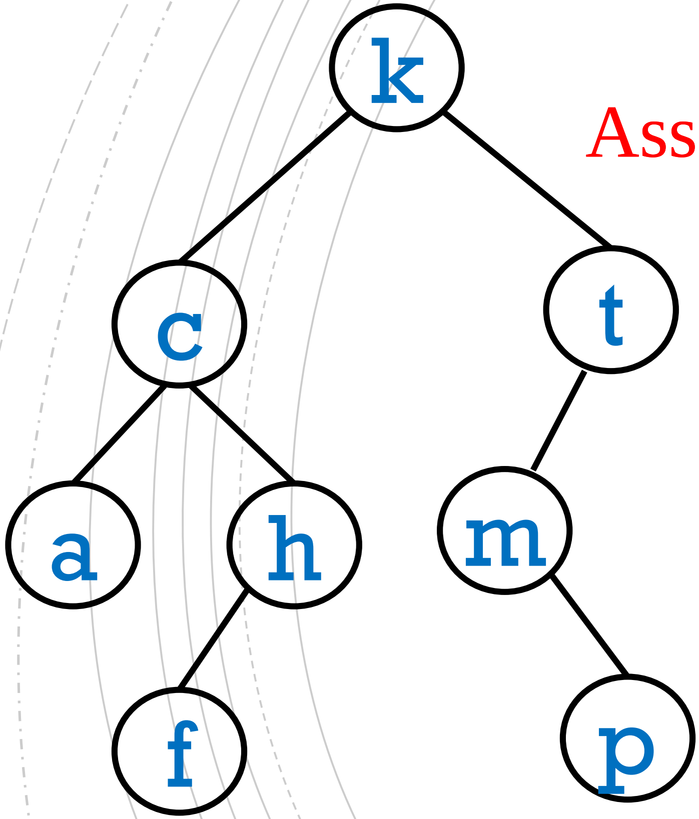
ADD()



ld add a BSTNode to the tree.

- H

ADD()

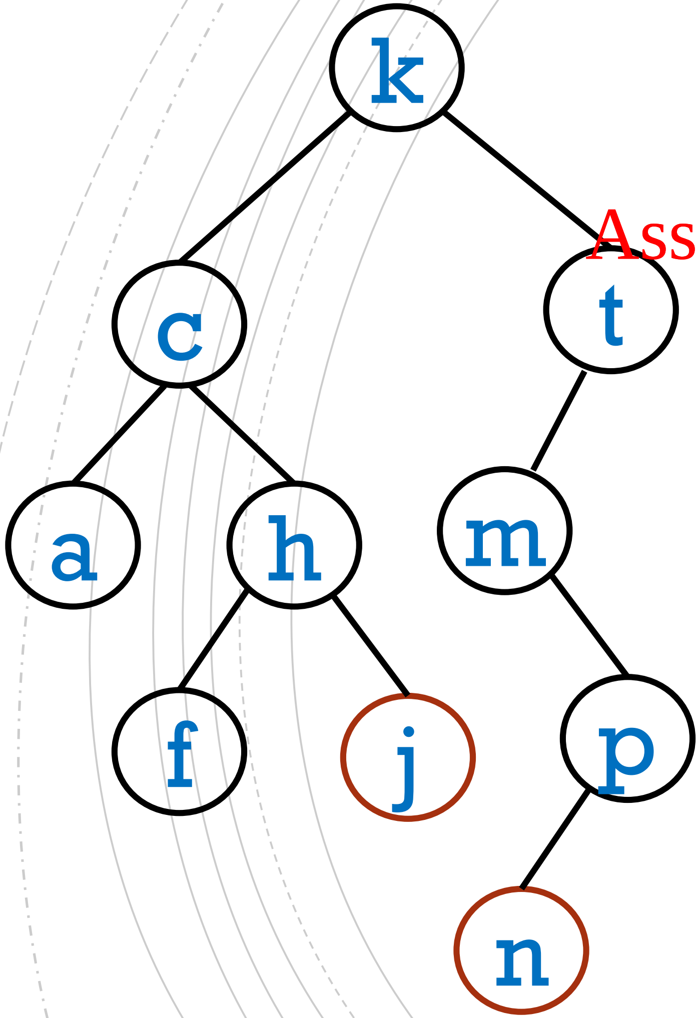


ld add a BSTNode to the tree.

- a

A new node is always a leaf.

ADD()

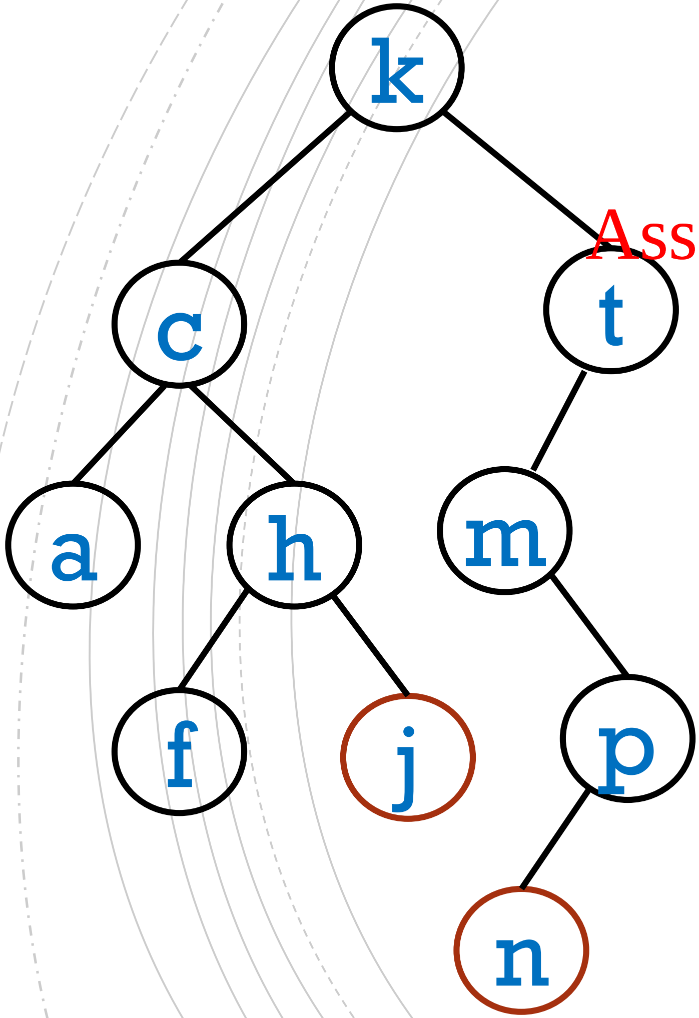


uld add a BSTNode to the

- a
- `add(n)` ?

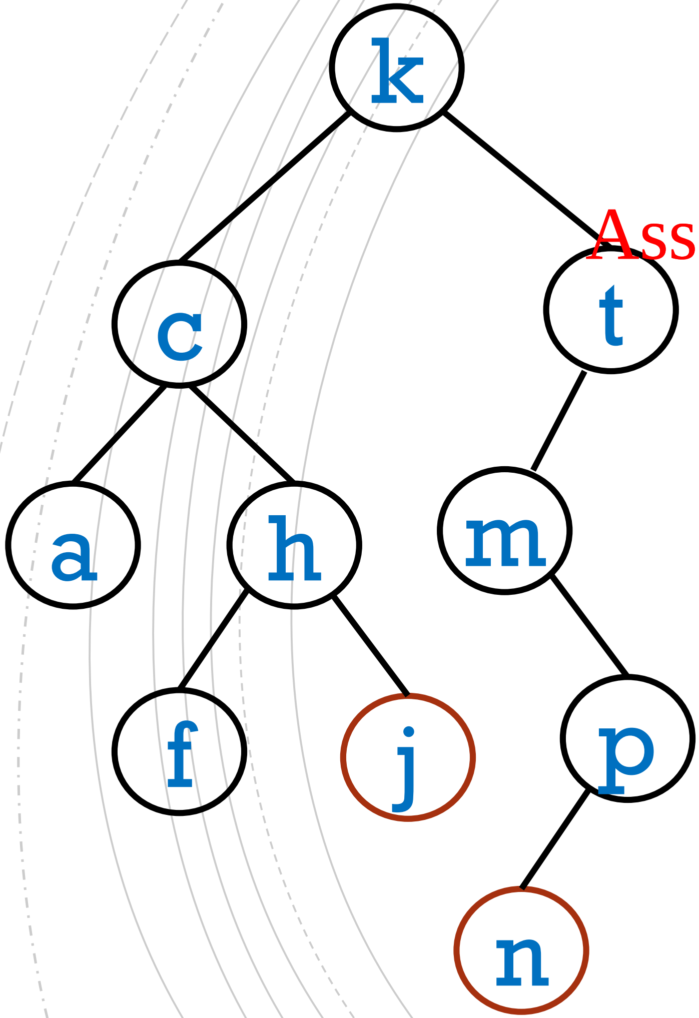
A new node is always a leaf.

ADD() - IMPLEMENTATION



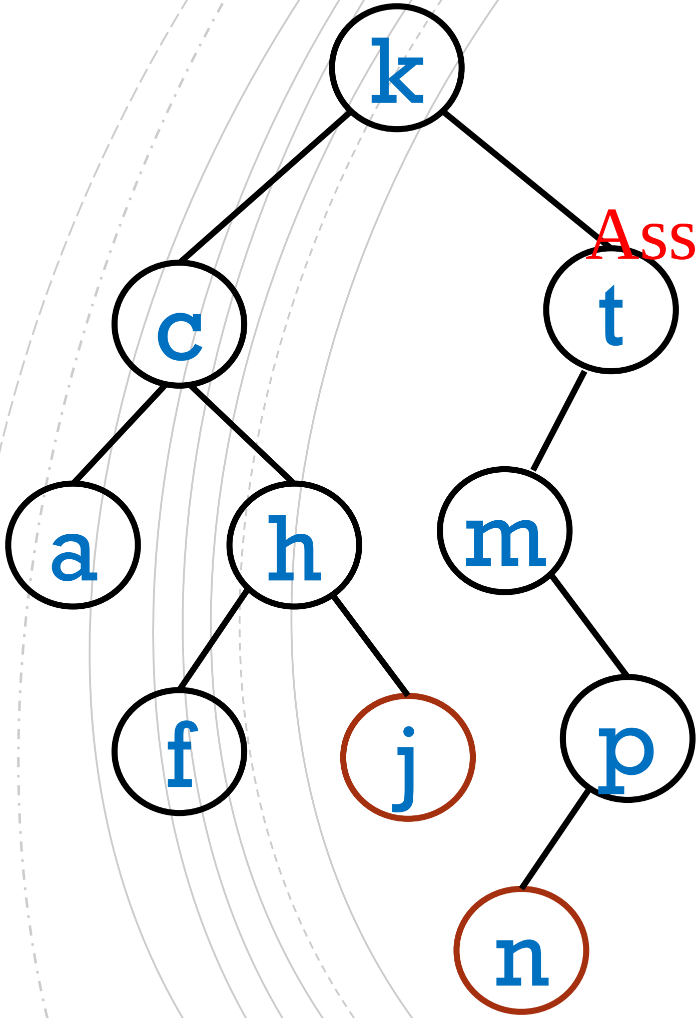
```
add(root, key){ // returns root node




    return root
}
```

ADD() - IMPLEMENTATION



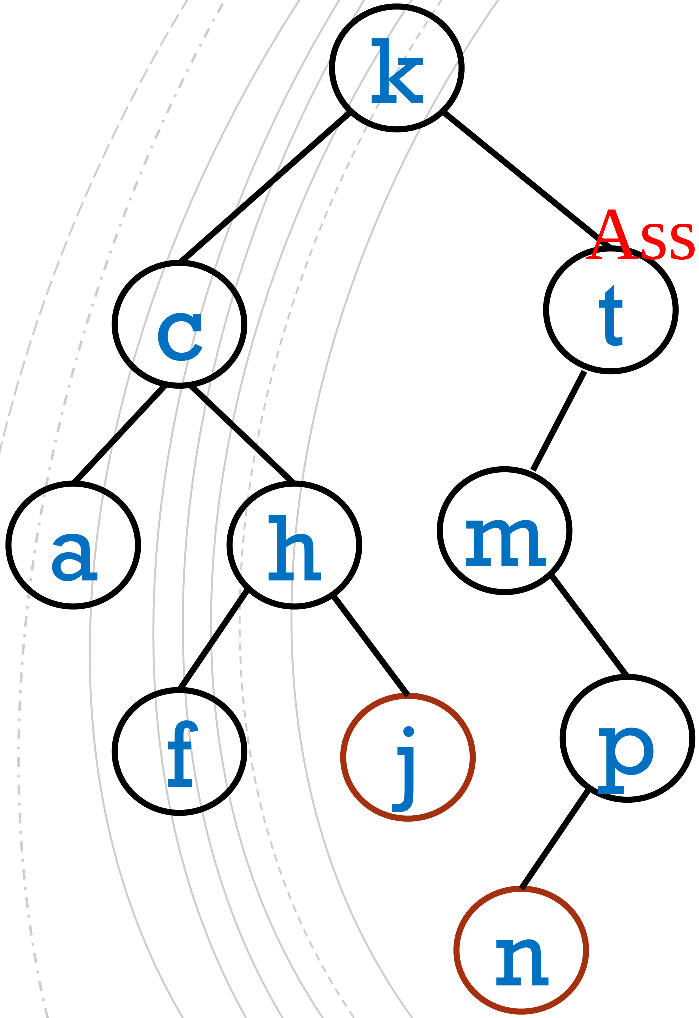
```
add(root, key){ // returns root node
                         ull)
                          BSTnode(key)




    return root
}
```

ADD() - IMPLEMENTATION



```
add(root, key){ // returns root node
                ull)
                  BSTnode(key)
   else i         root.key){
      roo          add(root.left,key)



   return root
}
```

ADD() - IMPLEMENTATION

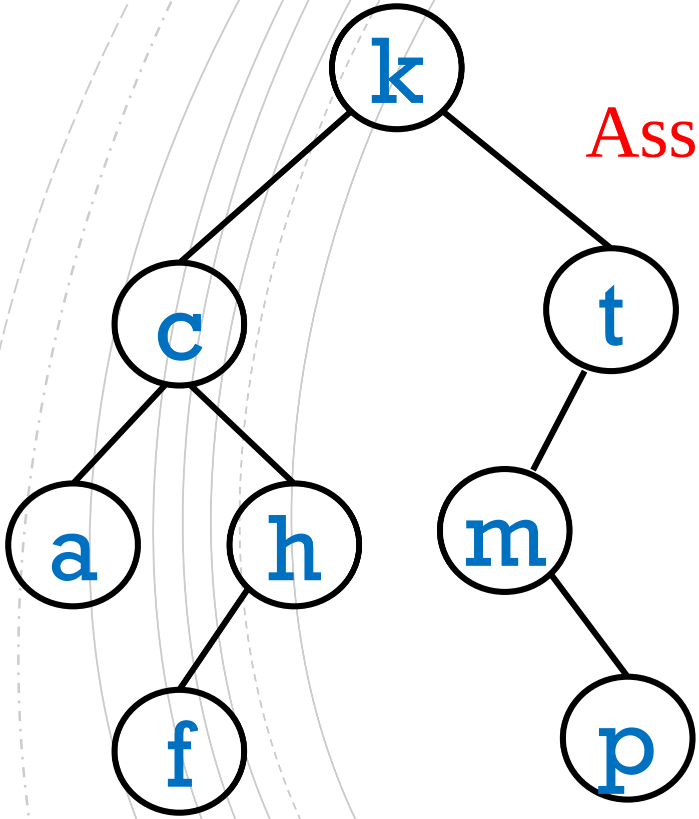


```
add(root, key){ // returns root node
                        ull)
                          BSTnode(key)
    else i              root.key){
      roo               add(root.left,key)
    else if (key > root.key){
      root.right =  add(root.right,key)
    return root
}
```

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

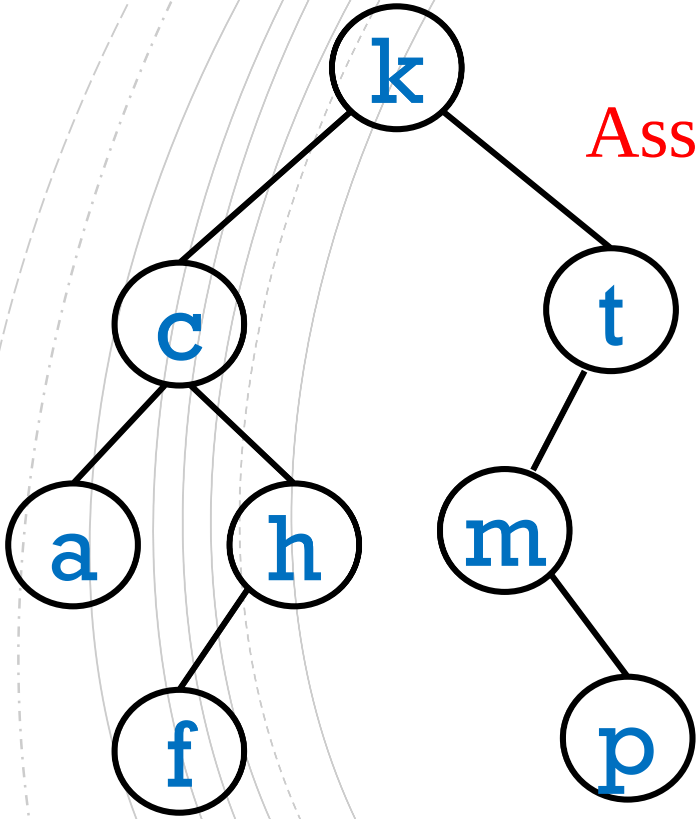
REMOVE()

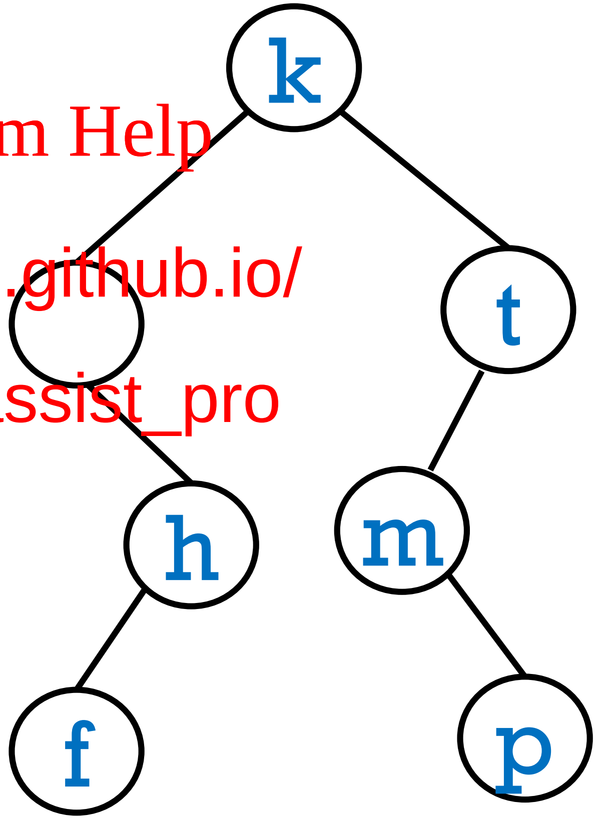
`remove(c)` →



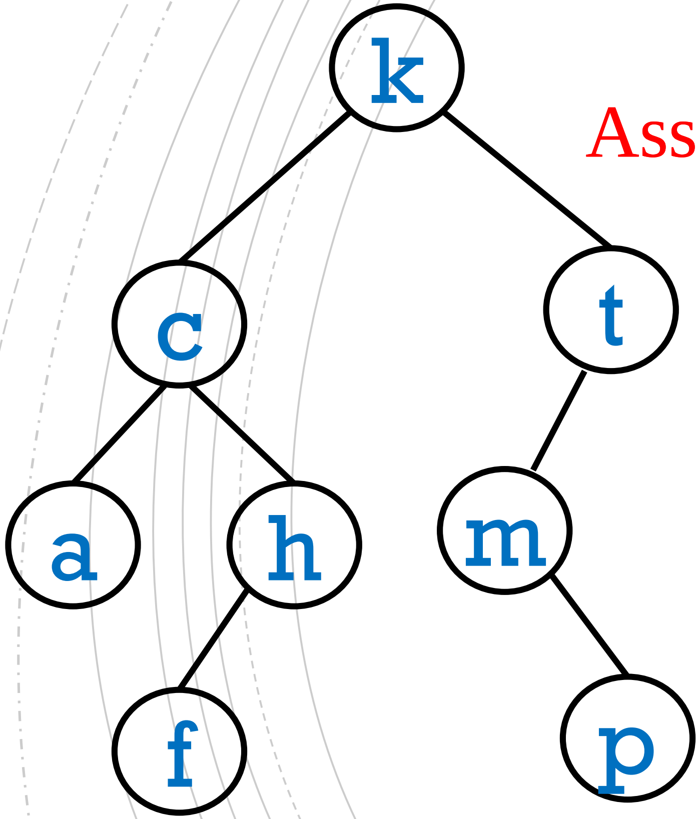
REMOVE()

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

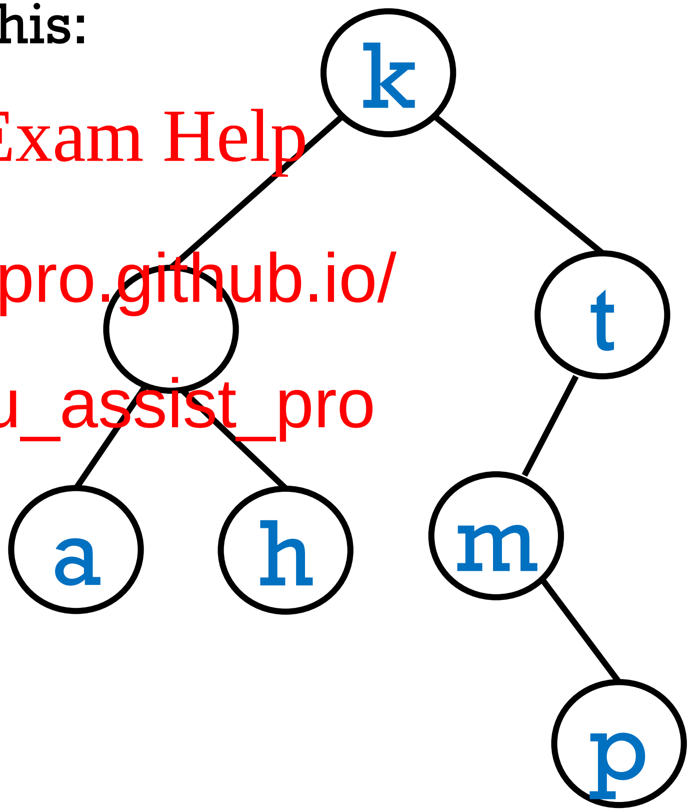




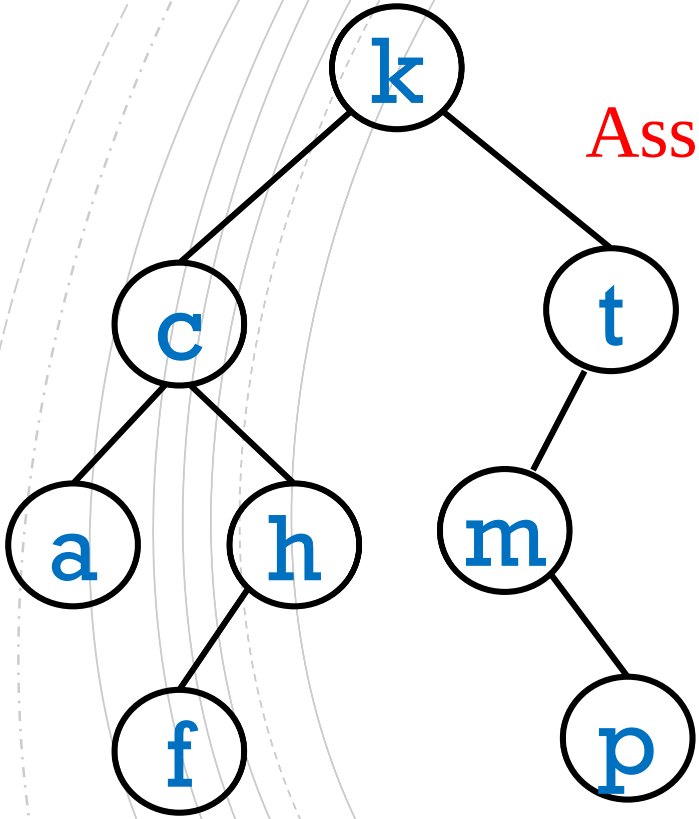
REMOVE()



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

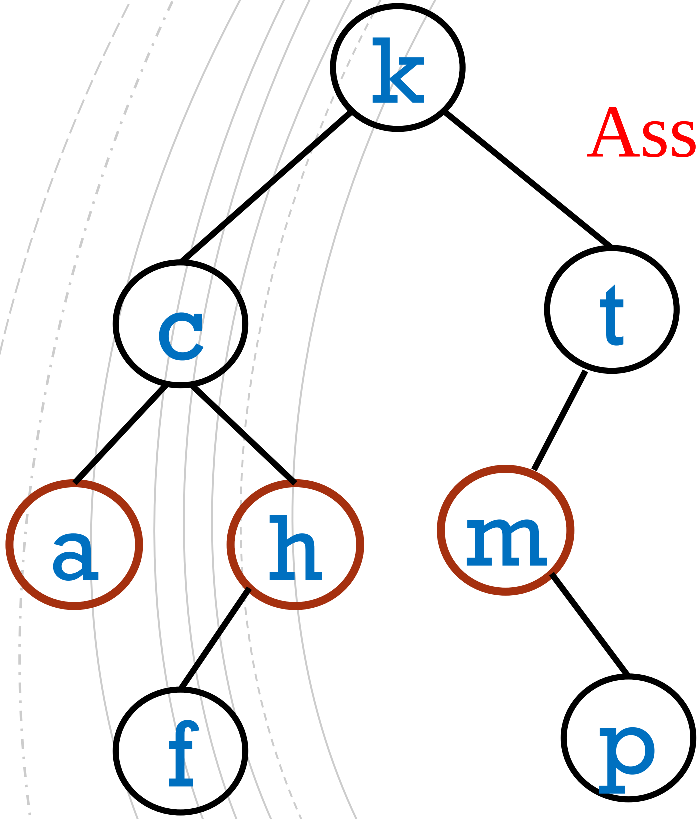


REMOVE() - IMPLEMENTATION



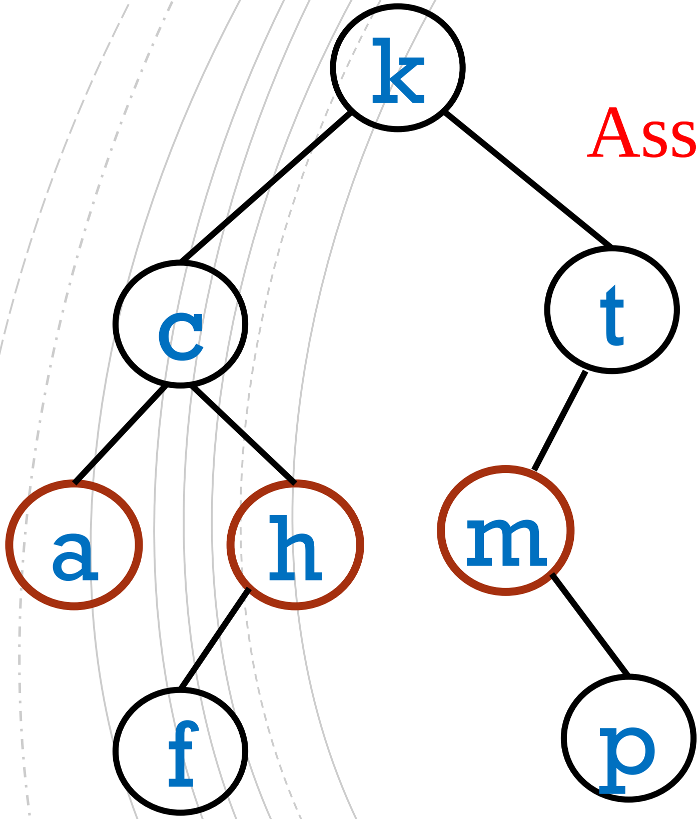
```
remove(root, key){ // returns root node
    if( root  == null )
         return null
    else if ( key < root.key )

                    root.key )

    else




    }
    return root
}
```

REMOVE() - IMPLEMENTATION



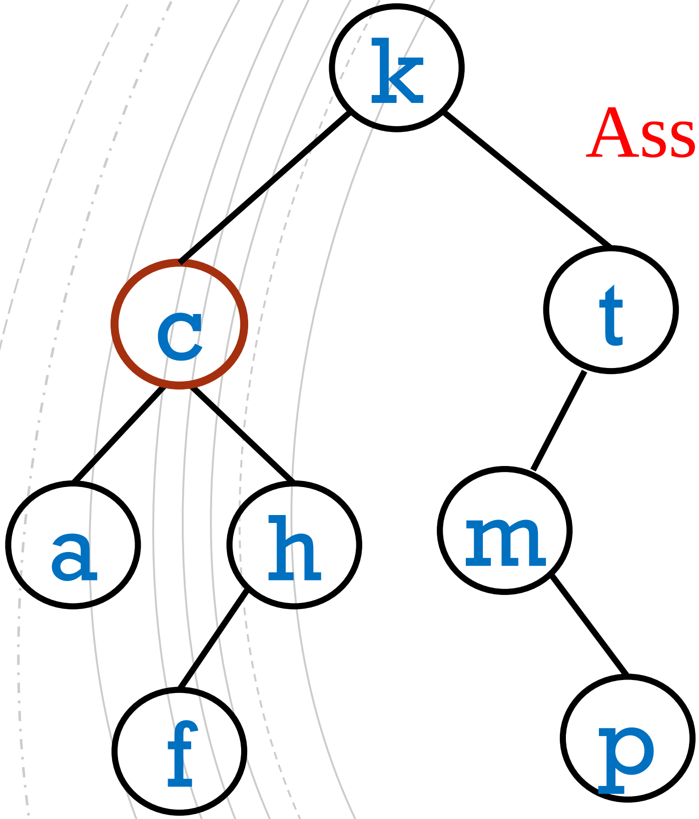
```
remove(root, key){ // returns root node
    if( root  == null )
         return null
    else if ( key < root.key )
         root.left = remove(root.left, key)
    else if ( key > root.key )
         root.right = remove(root.right, key)
    else




    }
    return root
}
```

REMOVE() - IMPLEMENTATION



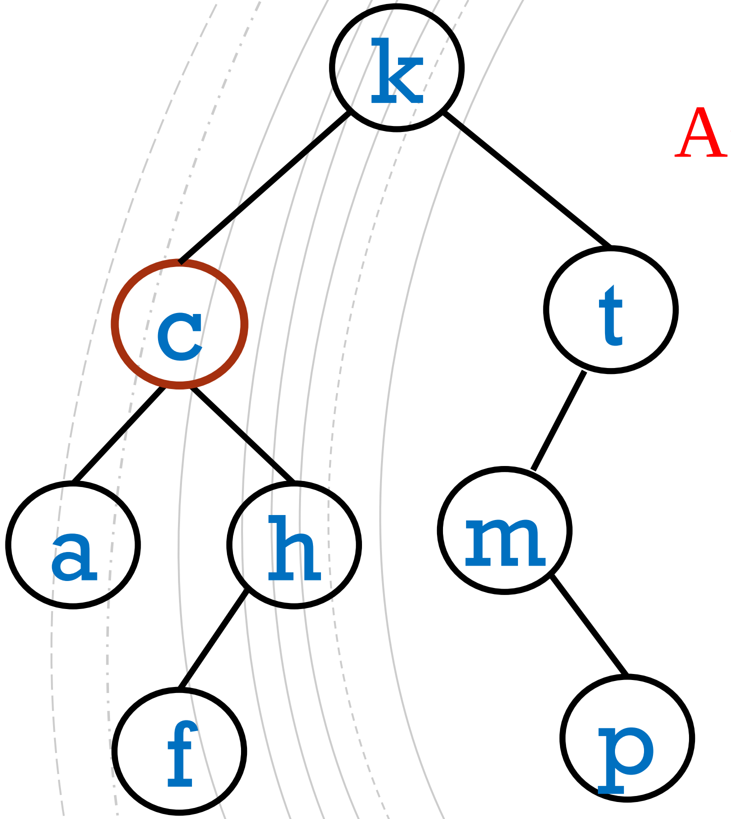
```
remove(root, key){ // returns root node
    if( root  == null )
         return null
    else if ( key < root.key )
         root.left = remove(root.left, key)
                    root.key )
                    = remove(root.right, key)
    else i          ft == null)
         r          .right
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



    }
    return root
}
```

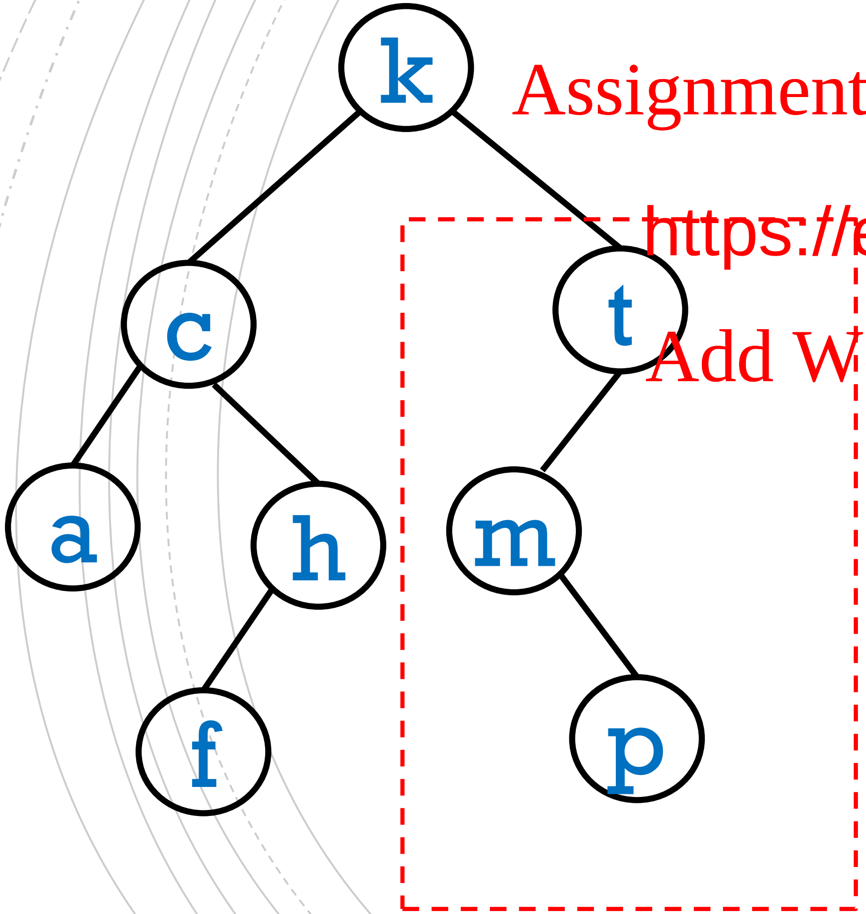
REMOVE() - IMPLEMENTATION



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

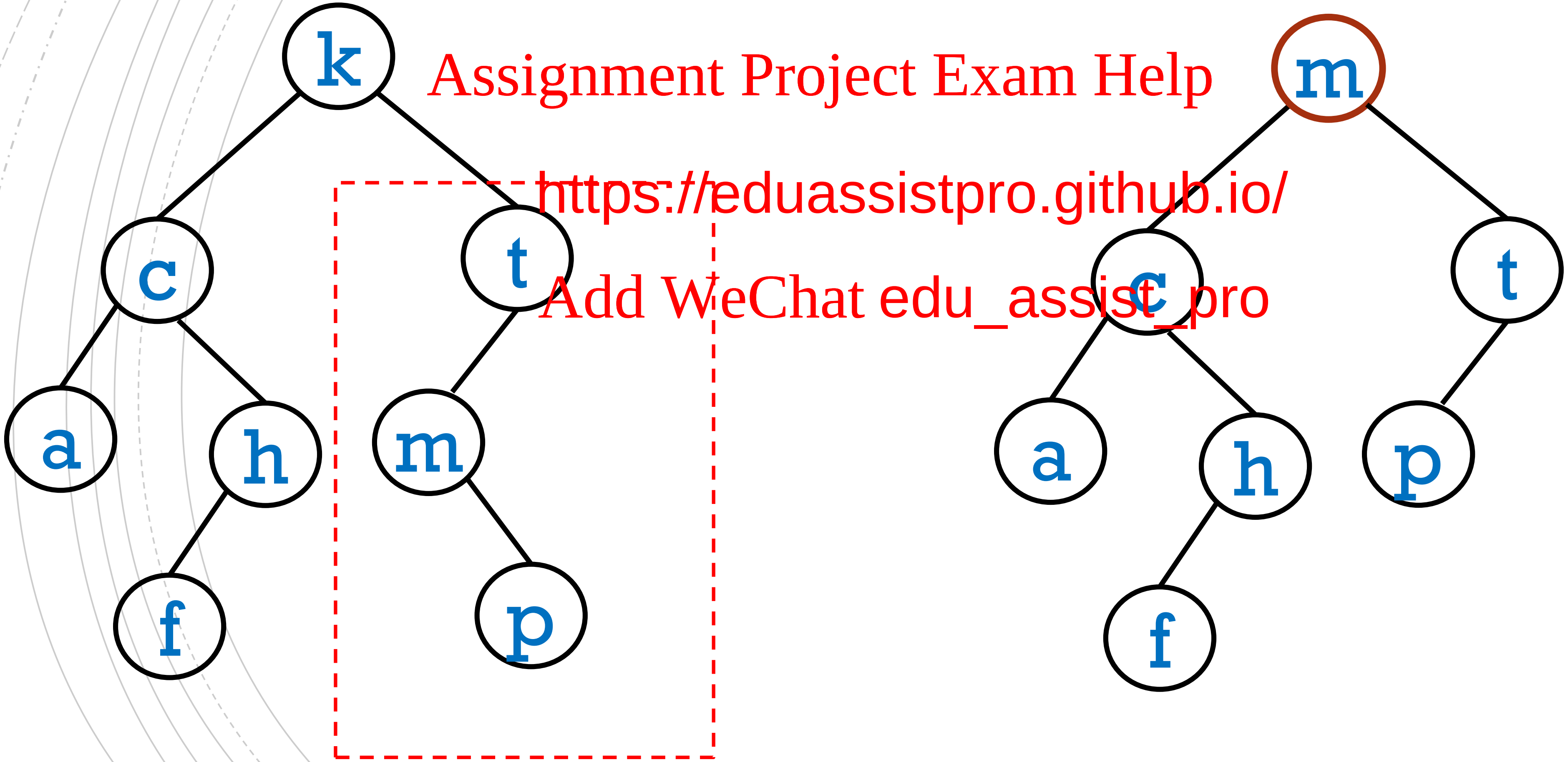
REMOVE() - EXAMPLE

`remove(k) →`



REMOVE() - EXAMPLE

`remove(k)` →

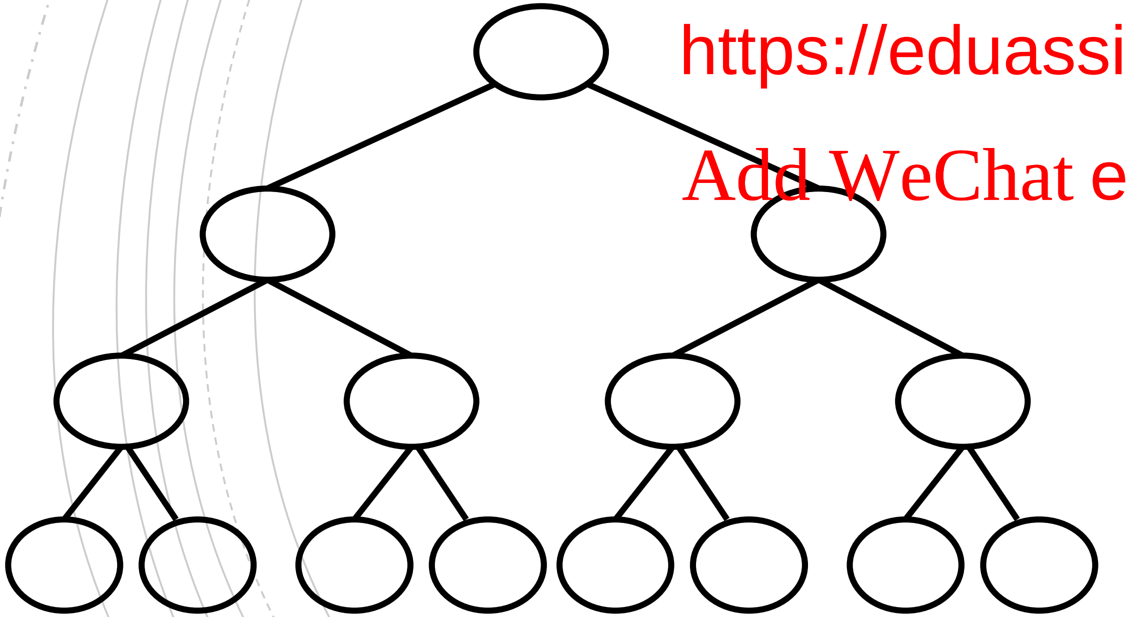


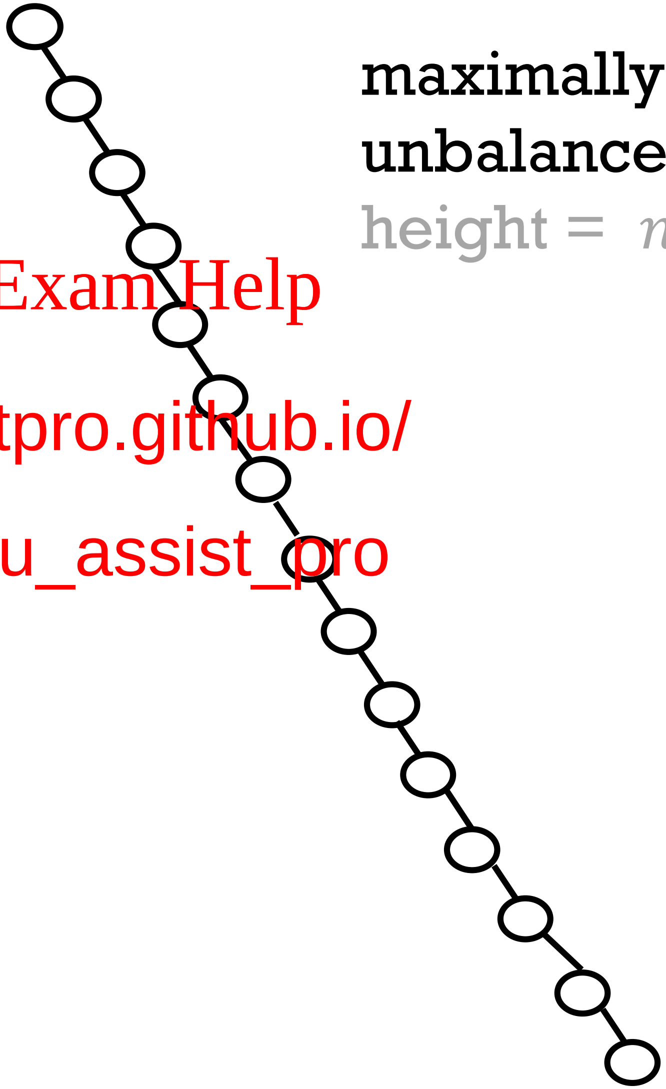
BALANCED VS UNBALANCED

balanced

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

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





maximally unbalanced

height = $n - 1$

BEST VS WORST CASE SCENARIO

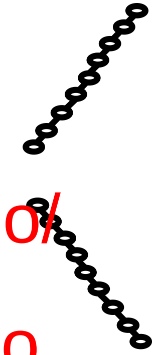
	best case	worst case
findMin()		
findMax()		
find(key)		
add(key)		
remove(key)		

BEST VS WORST CASE SCENARIO

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

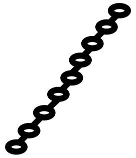
BEST VS WORST CASE SCENARIO

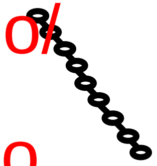
	best case	worst case
findMin()	$O(1)$	$O(n)$
findMax()		$O(n)$
find(key)		
add(key)		
remove(key)		



BEST VS WORST CASE SCENARIO

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





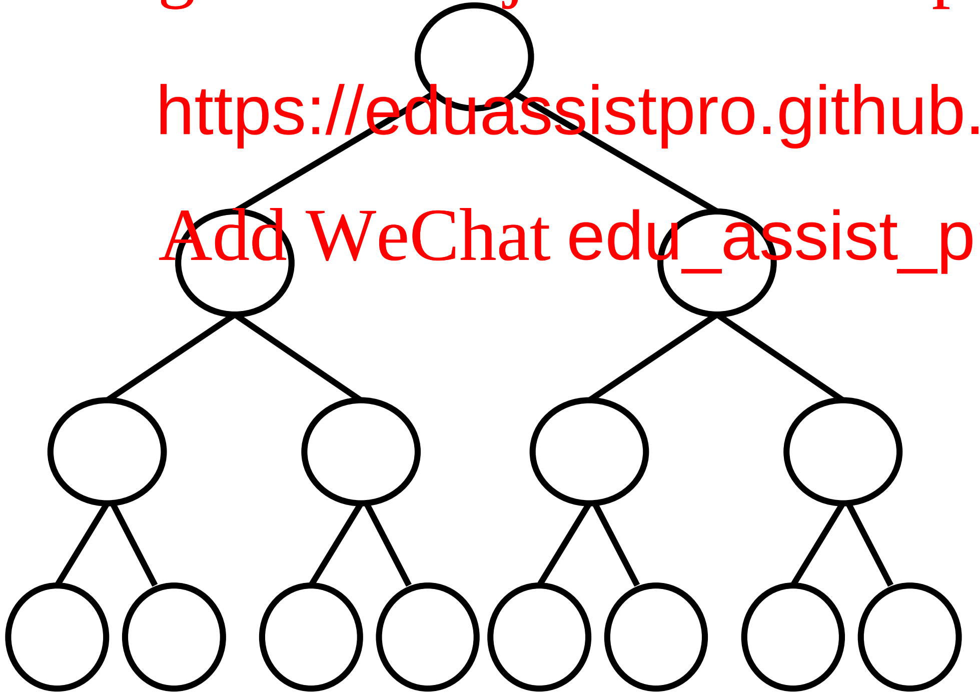
BEST VS WORST CASE SCENARIO

	best case	worst case
findMin()	$O(1)$	$O(n)$
findMax()		(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)

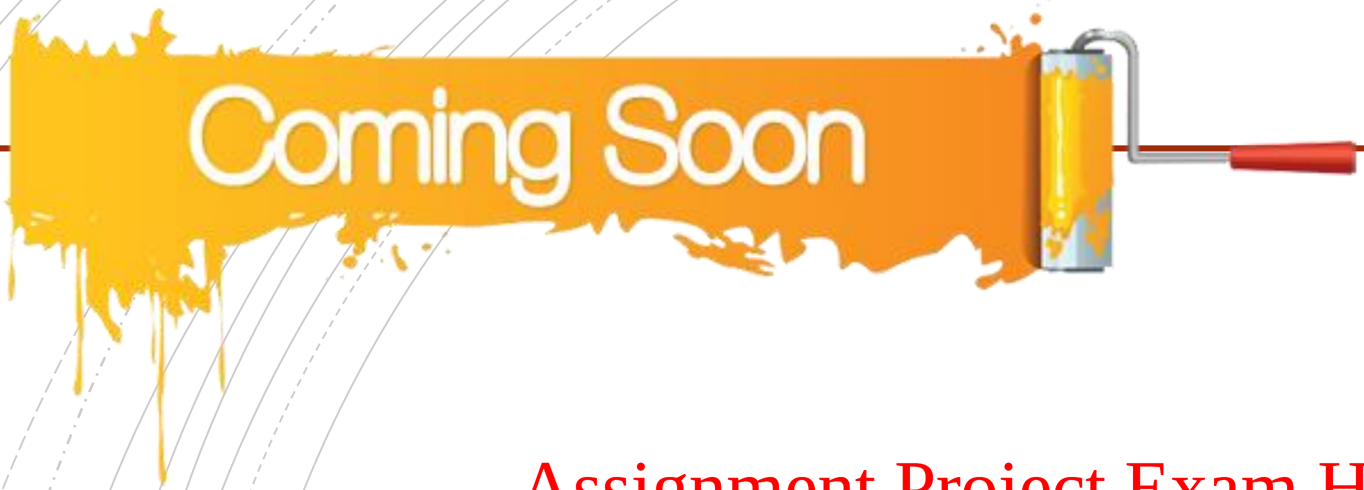
When a binary search tree is balanced, then finding a key is very similar to a binary search.



BALANCED BINARY SEARCH TREES

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

	best case	worst case
findMin()	$O(\log n)$	$O(\log n)$
findMax()		$O(\log n)$
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

- Heaps