COMP 250

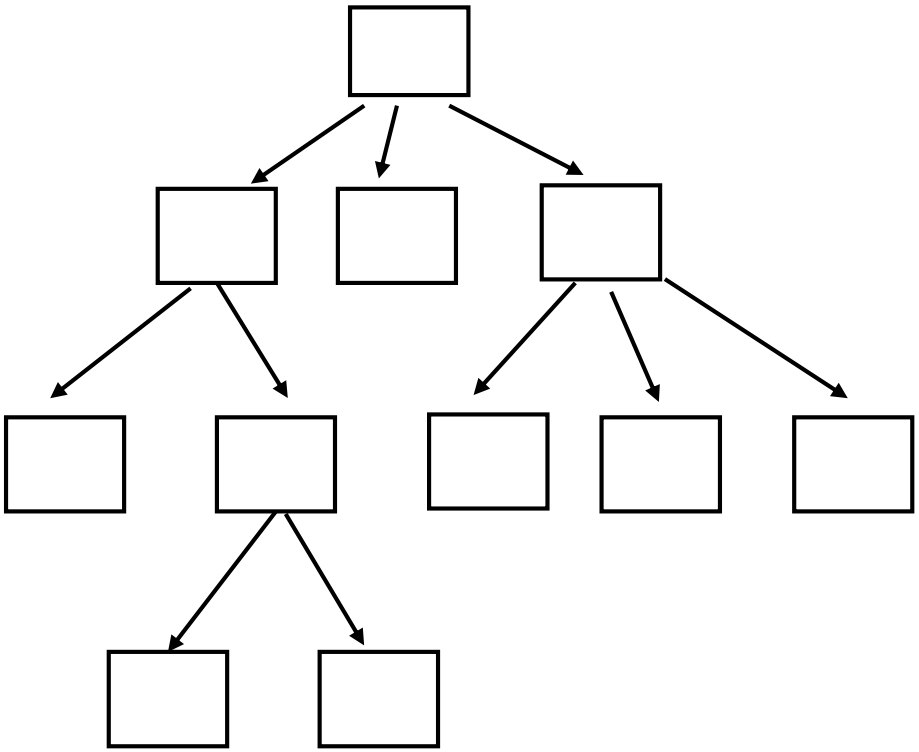
Lecture 20

tree traversal

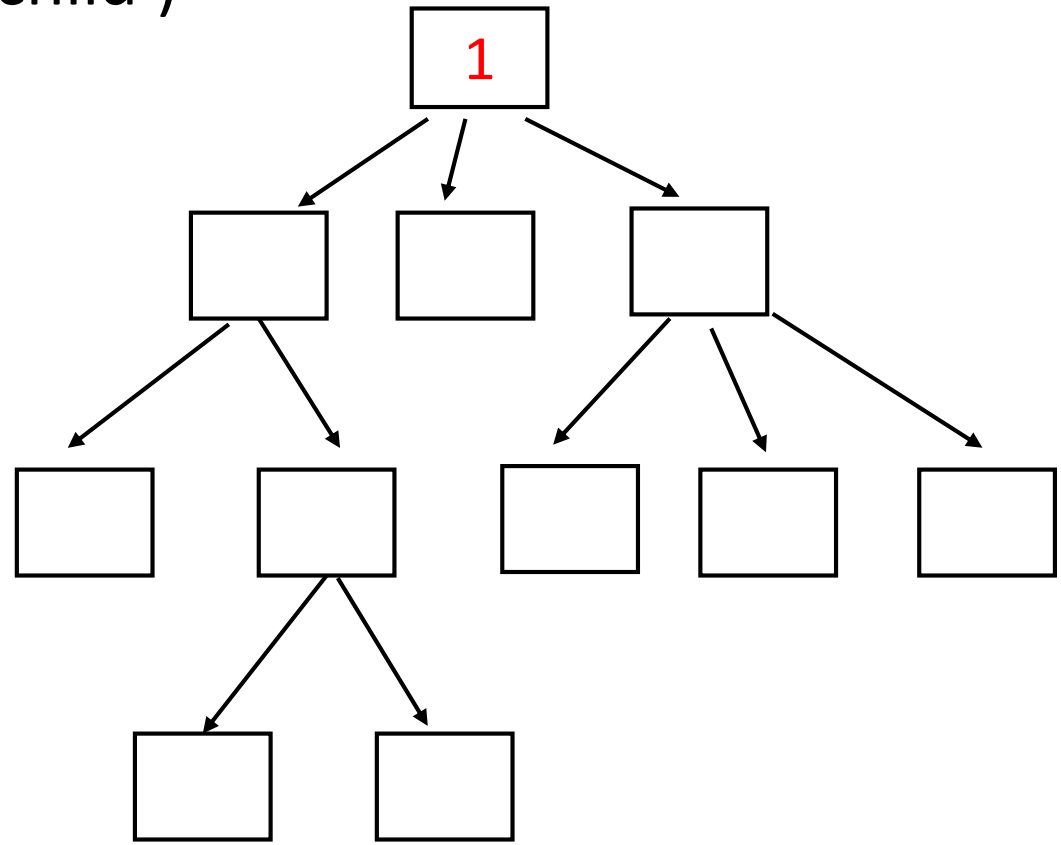
Oct. 25/26, 2017

Tree Traversal

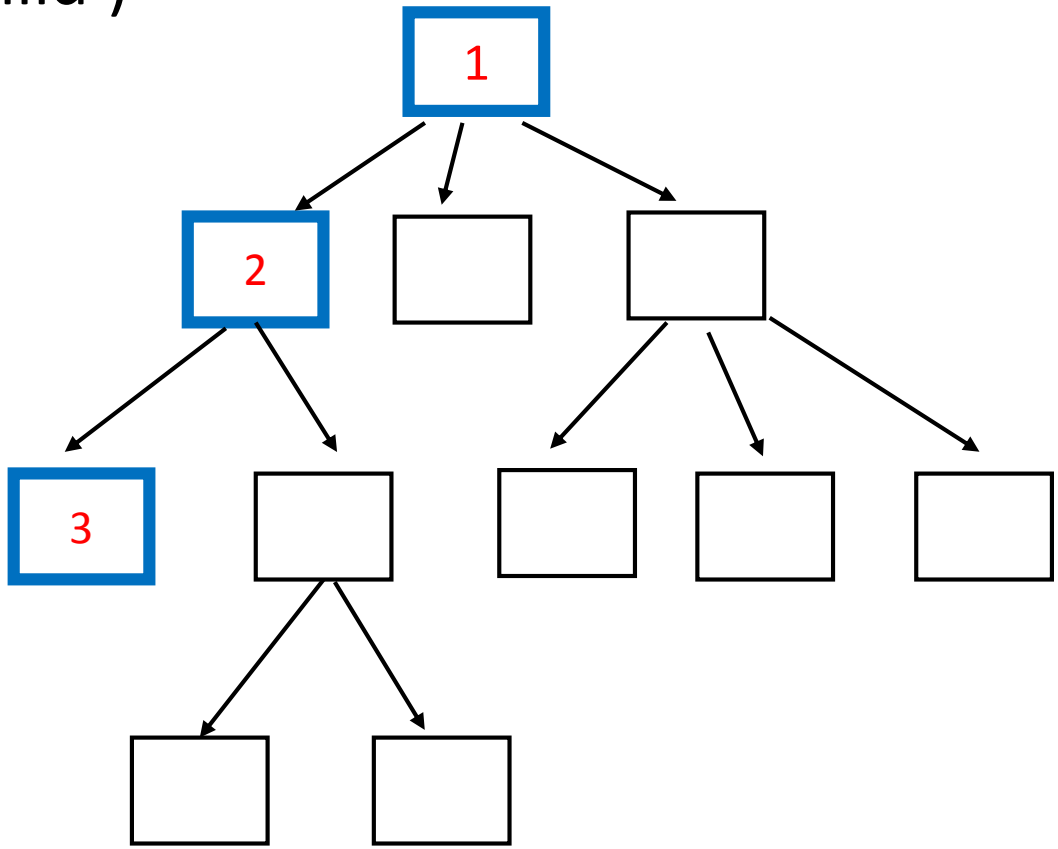
How to visit (enumerate, iterate through, traverse…) all the nodes of a tree ?



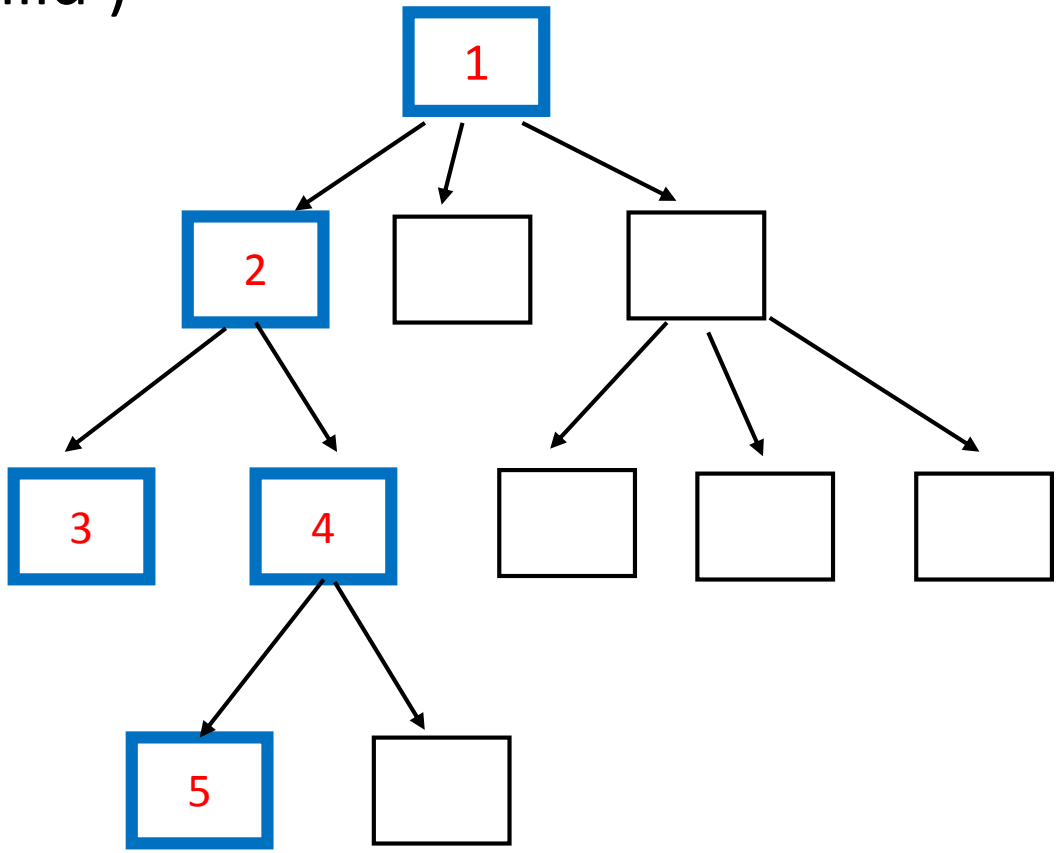
```
depthfirst (root){                    //        "preorder"
   if (root is not empty){
      visit root
      for each child of root
         depthfirst( child )
   }
}
```



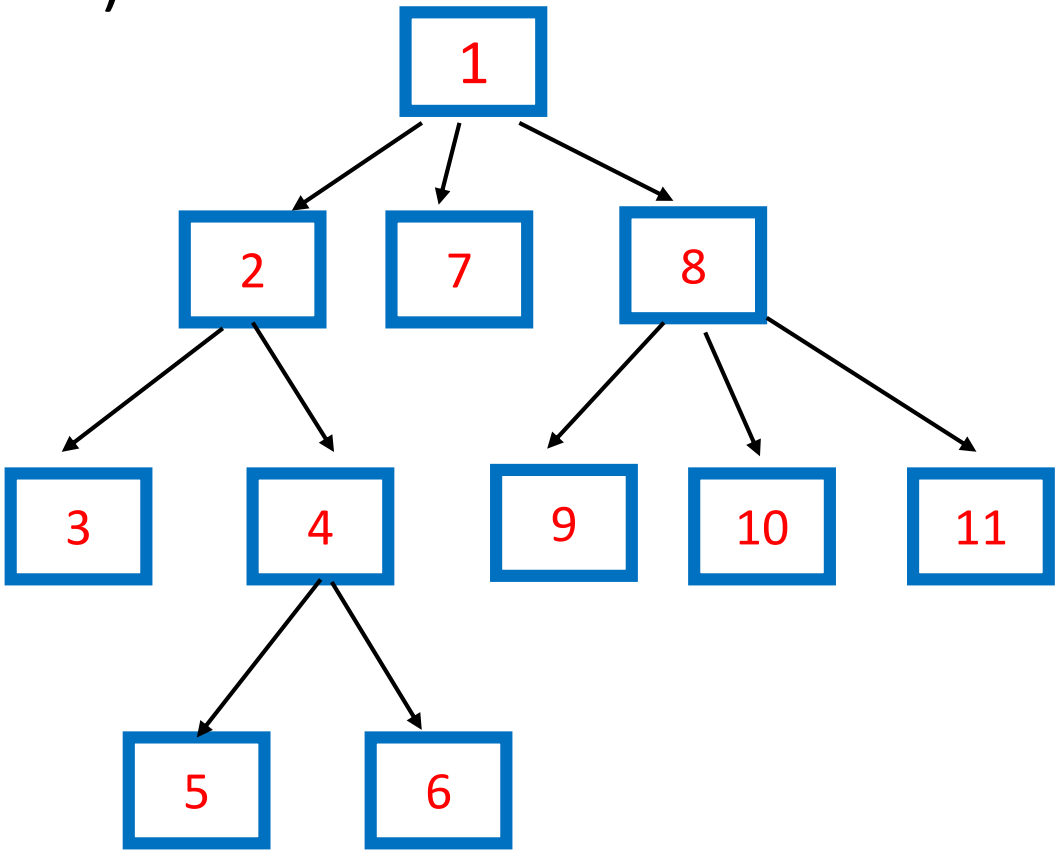
```
depthfirst (root){                    //        "preorder"
   if (root is not empty){
      visit root
      for each child of root
         depthfirst( child )
   }
}
```



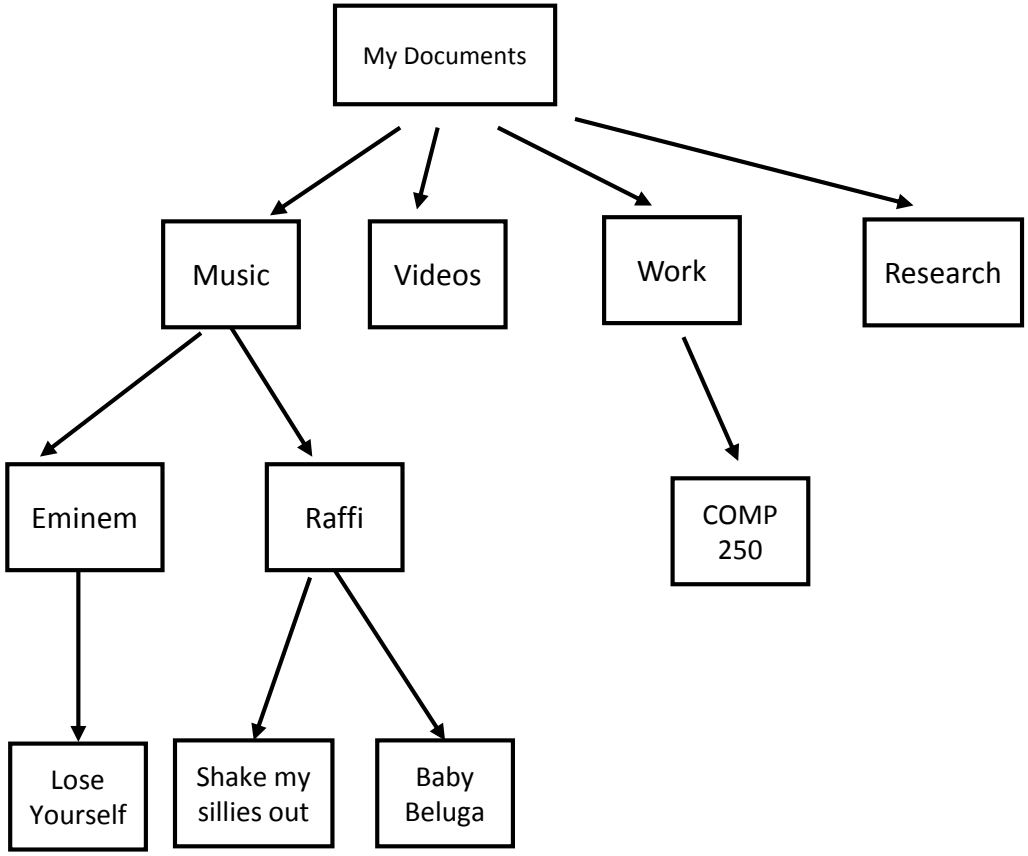
```
depthfirst (root){                    //        "preorder"
    if (root is not empty){
        visit root
        for each child of root
            depthfirst( child )
    }
}
```



```
depthfirst (root){                    //        "preorder"
   if (root is not empty){
      visit root
      for each child of root
         depthfirst( child )
   }
}
```



Preorder Traversal
e.g. Printing a directory (visit = print)

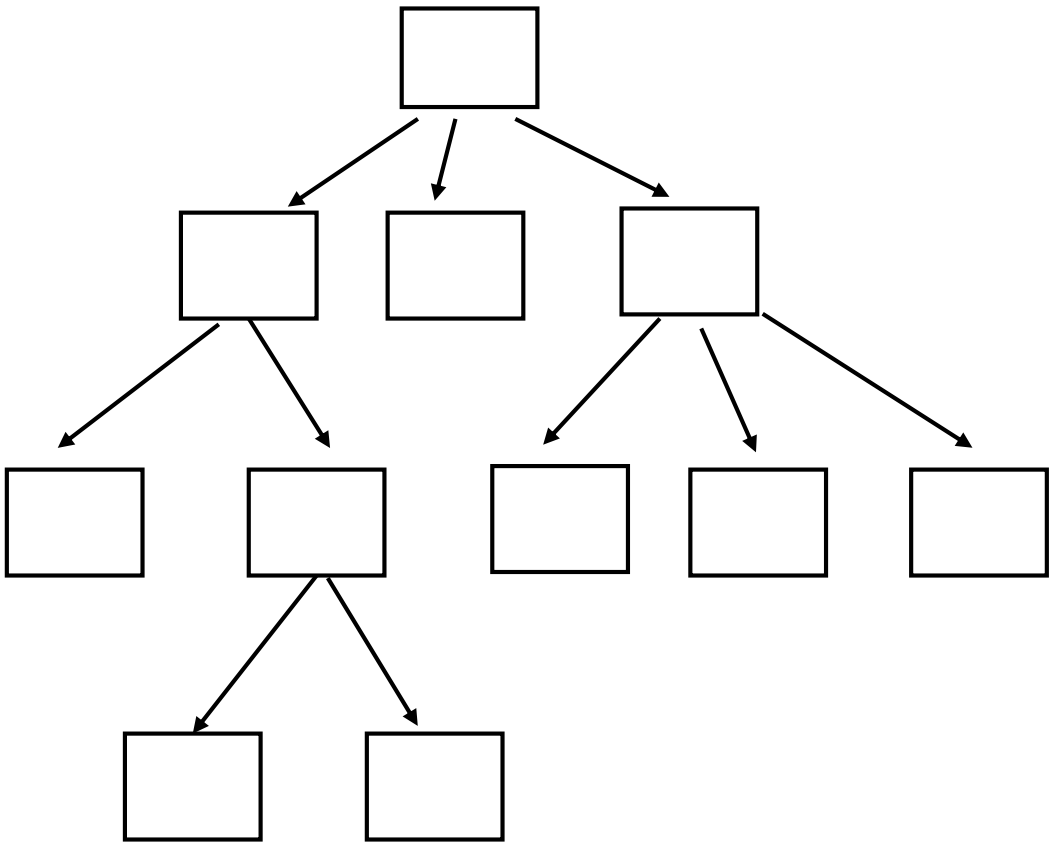


Documents (directory)
- Music (directory)
  - Eminem (directory)
    - Lose Yourself (file)
  - Raffi (directory)
    - Shake My Sillies Out (file)
    - Baby Beluga (file)
- Videos (directory)
  - : (file)
- Work (directory)
  - COMP250 (directory)
  - :
- Research (directory)
  - :

“Visit” implies that you do something at that node.

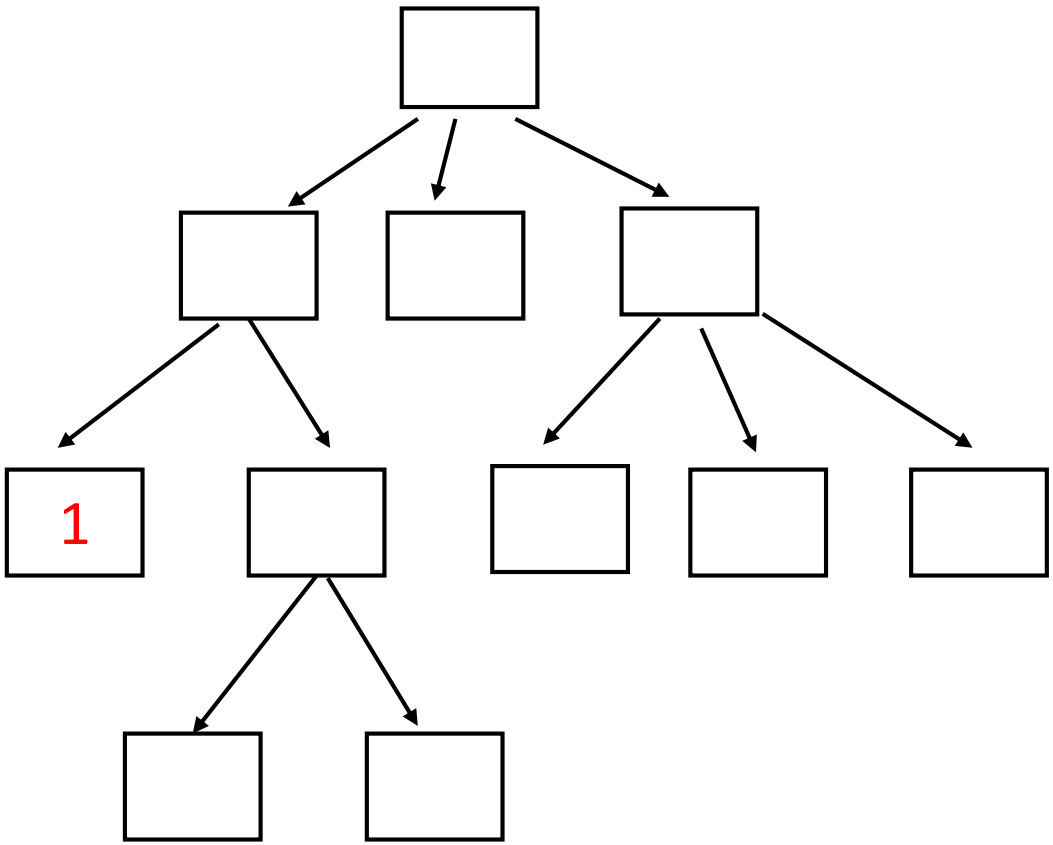
Analogy: you aren’t visiting London UK if you just fly through Heathrow.

```
depthfirst (root){                    //        "postorder"
   if (root is not empty){
      for each child of root
         depthfirst( child )
      visit root
   }
}
```

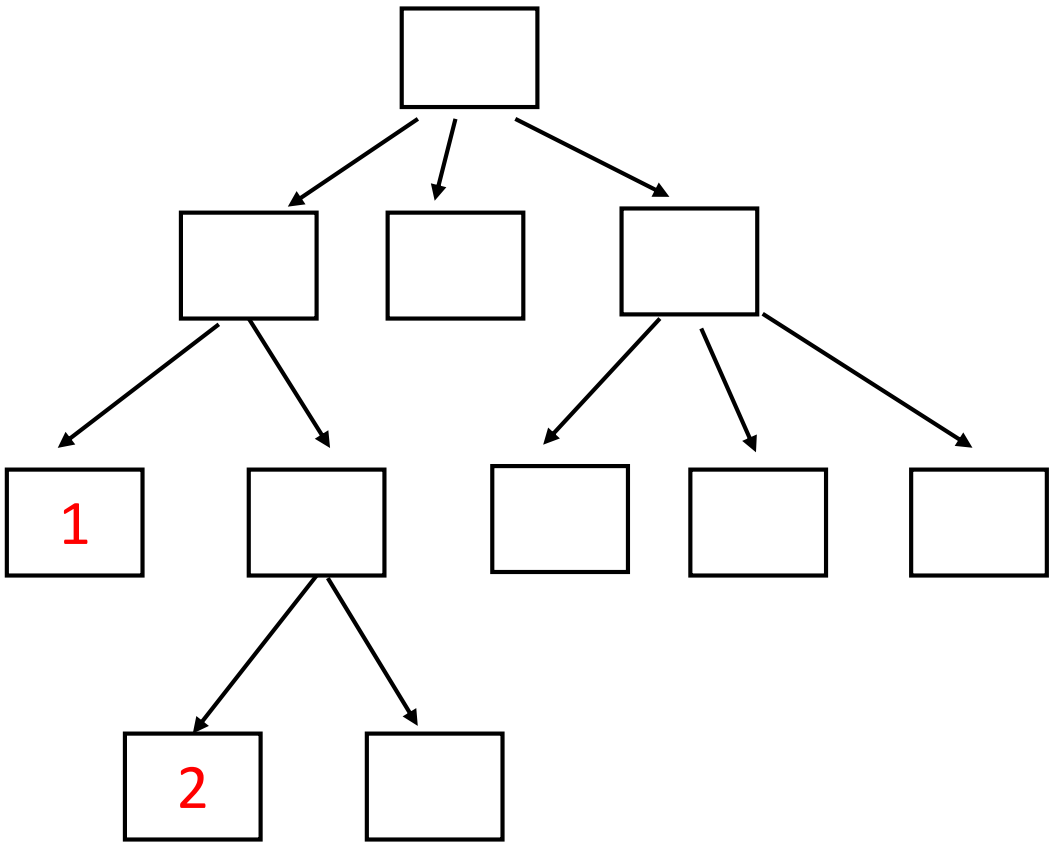


Q: Which node is visited first?

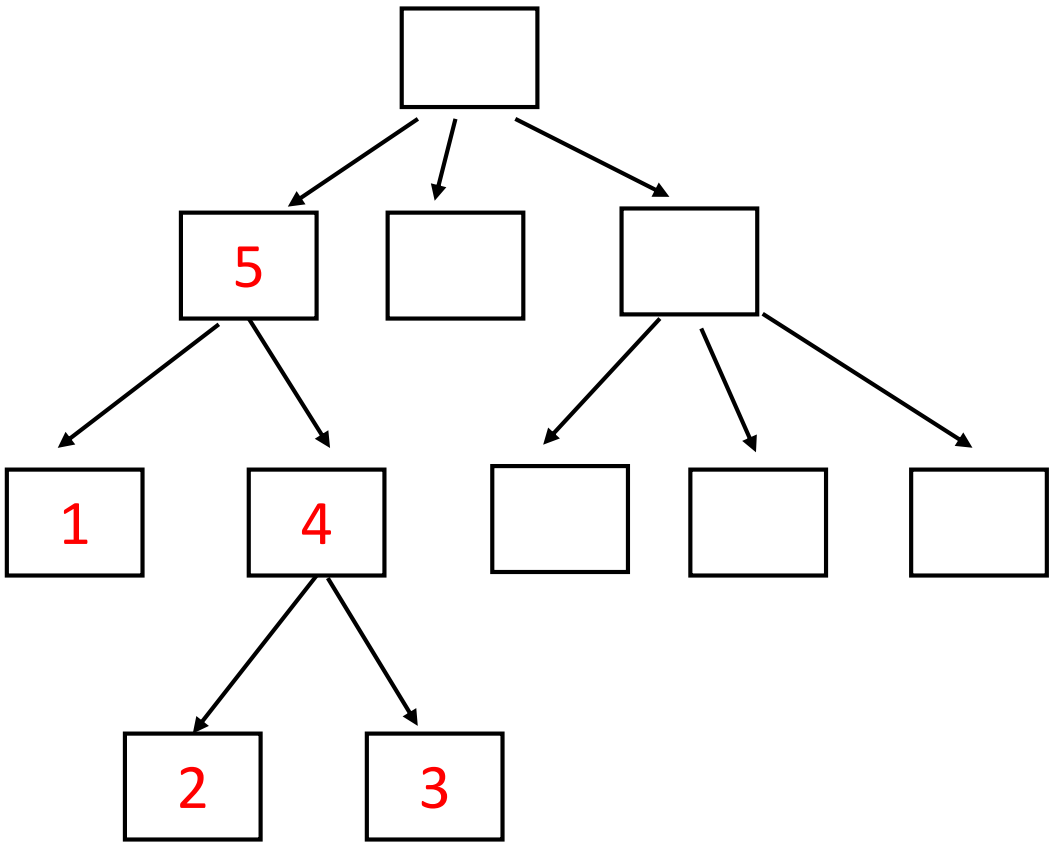
```
depthfirst (root){                    //        "postorder"
   if (root is not empty){
        for each child of root
             depthfirst( child )
       visit root
   }
}
```



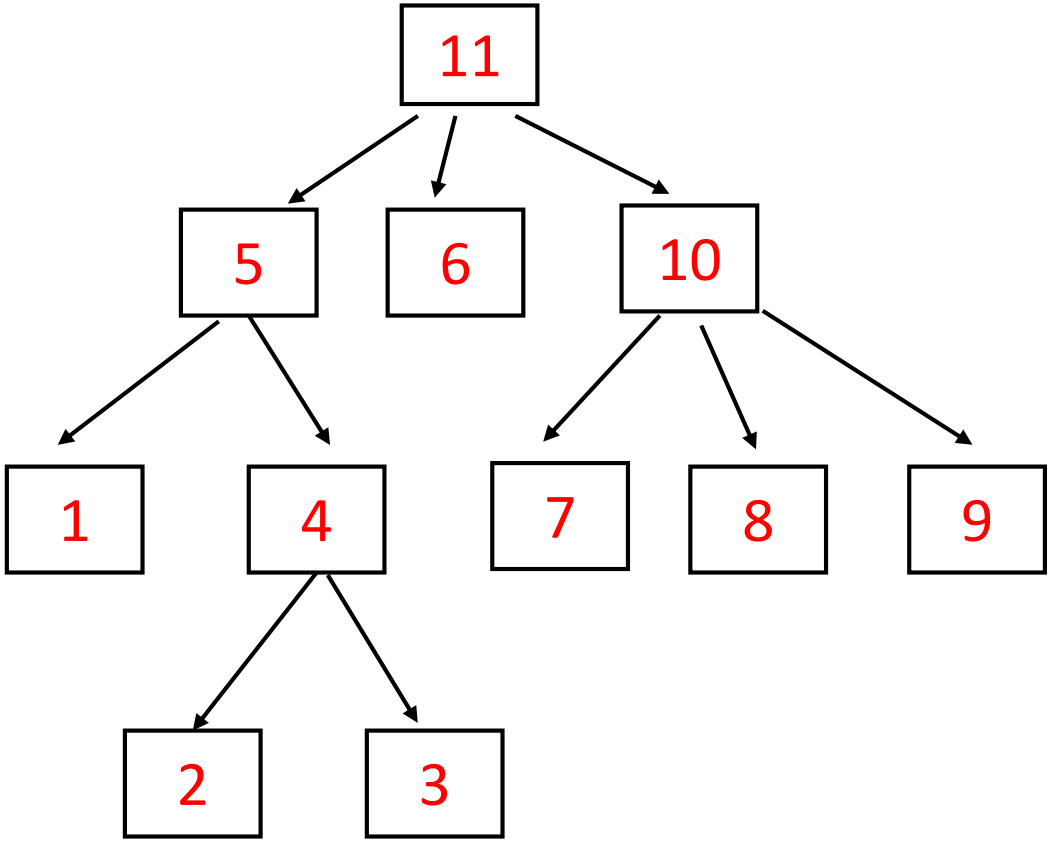
```
depthfirst (root){                    //        "postorder"
   if (root is not empty){
        for each child of root
             depthfirst( child )
       visit root
   }
}
```



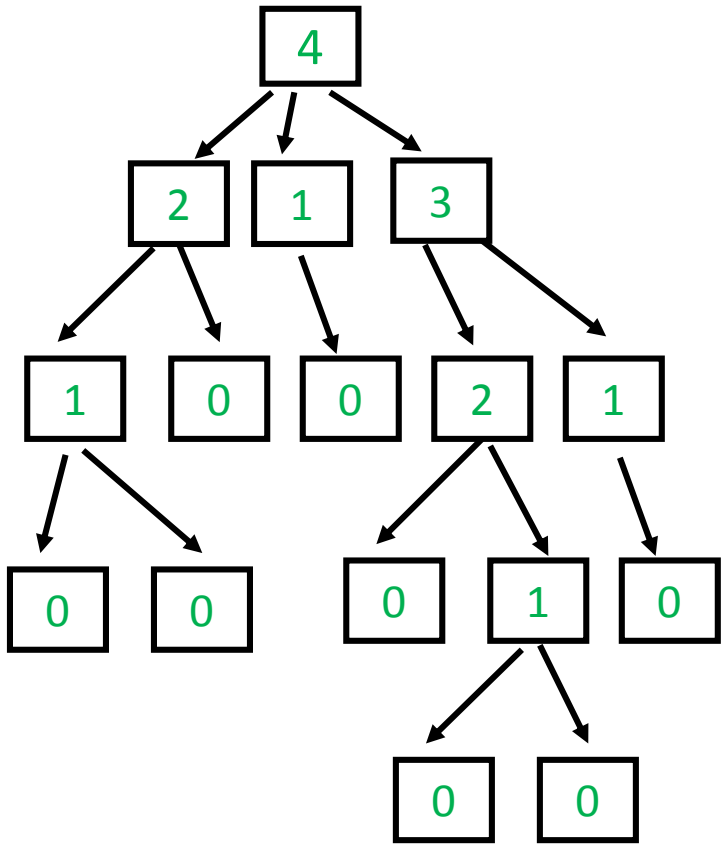
```
depthfirst (root){                    //        "postorder"
   if (root is not empty){
      for each child of root
         depthfirst( child )
      visit root
   }
}
```



```
depthfirst (root){                    //          "postorder"
   if (root is not empty){
        for each child of root
             depthfirst( child )
        visit root
   }
}
```



Example 1 postorder: recall last lecture




```
height(v){
    if (v is a leaf)
        return 0
    else{
        h = 0
        for each child w of v
            h = max(h, height(w))
        return 1 + h
    }
}
```

visit = return value of height

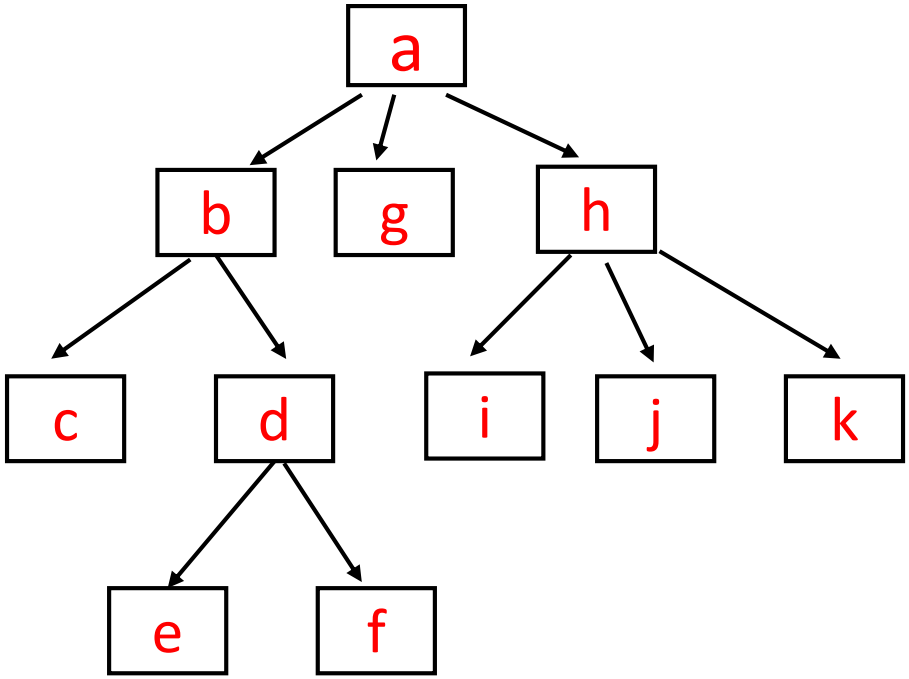
Example 2 Postorder: What is the total number of bytes in all files in a directory?

```
numBytes(root){
   if root is a leaf
      return number of bytes at root
   else {
       sum = 0
       for each child of root{
             sum += numBytes(child)
       }
       return sum
    }
}
```

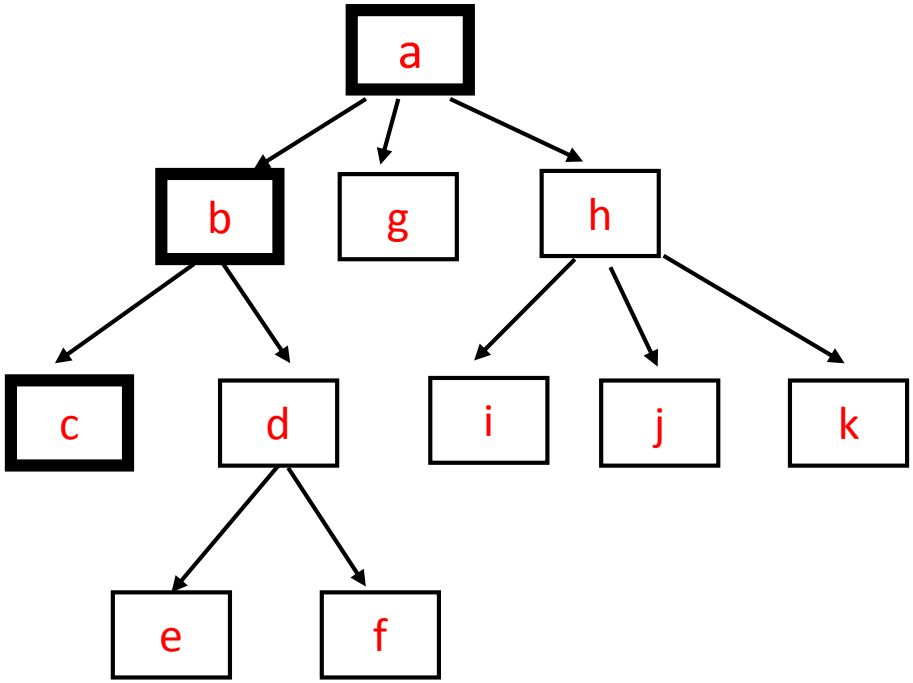
By ‘visit’ here, we mean determining the number of bytes for a node, e.g. If we were to store ‘sum’ at the node.

NOTE: Same call sequence occurs for preorder vs postorder.

Letter order corresponds to depthfirst() call order

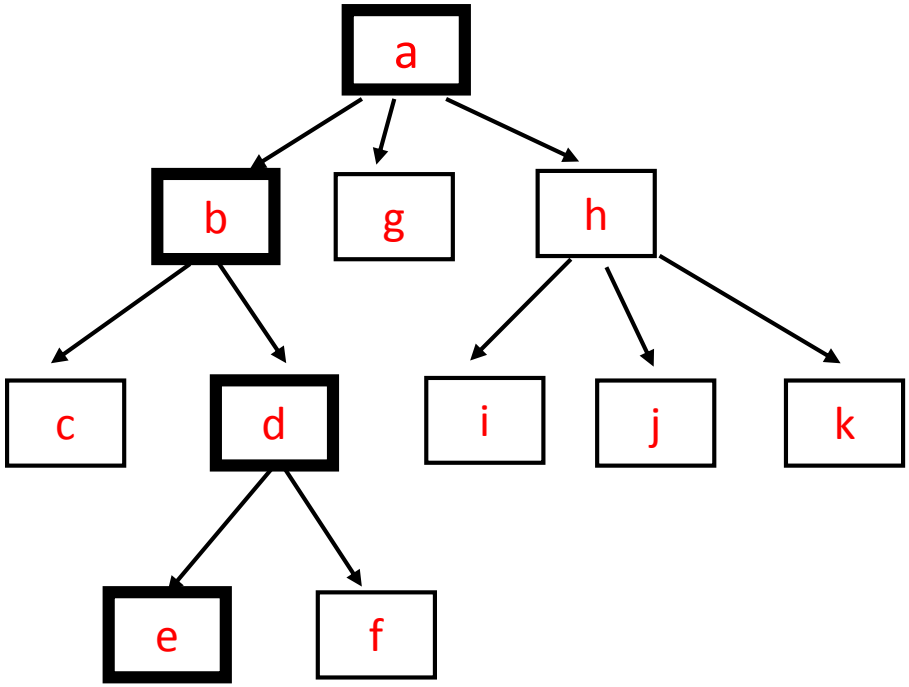


Call stack for depthfirst()



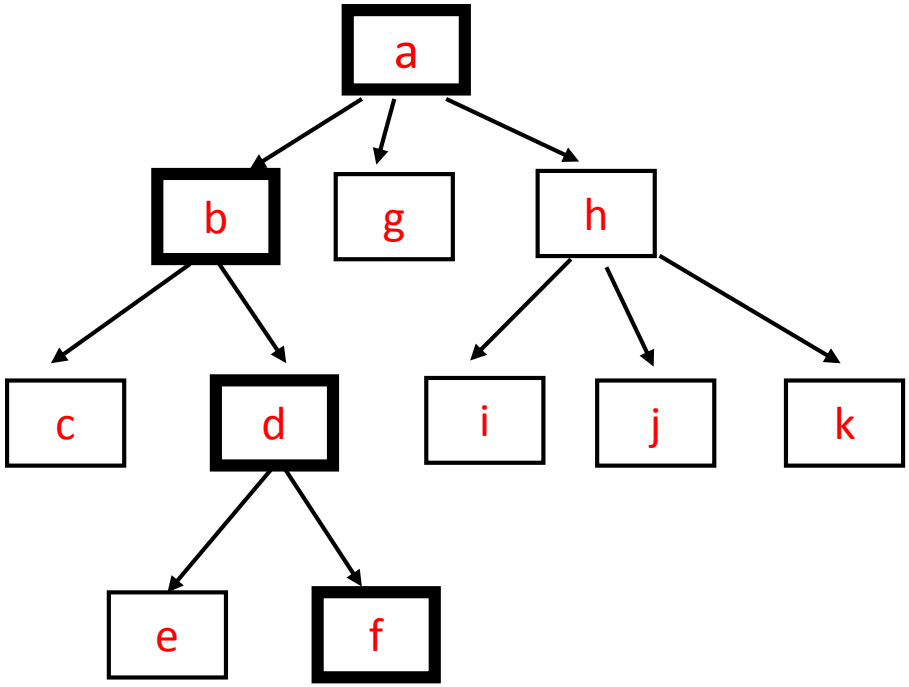
		c
	b	b
a	a	a

Call stack for depthfirst()



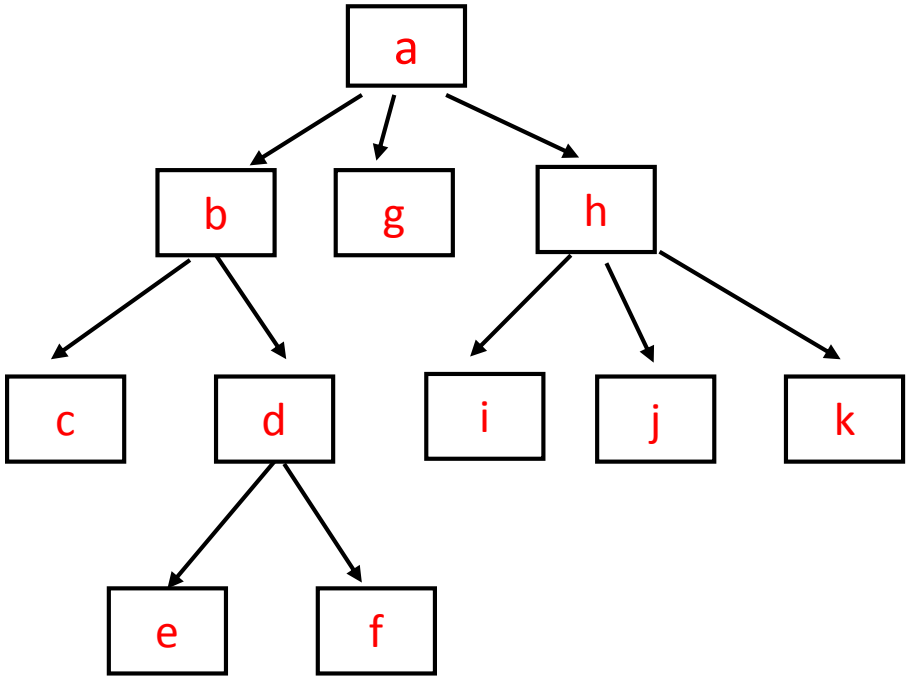
					e
		c		d	d
	b	b	b	b	b
a	a	a	a	a	a

Call stack for depthfirst()



					e		f
		c		d	d	d	d
	b	b	b	b	b	b	b
a	a	a	a	a	a	a	a

Call stack for depthfirst()



					e		f													
		c		d	d	d	d	d						i		j		k		
	b	b	b	b	b	b	b	b	b		g		h	h	h	h	h	h	h	
a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a

Tree traversal

Recursive

- depth first (pre- versus post-order)

Non-Recursive

- using a stack
- using a queue

```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)

}
```

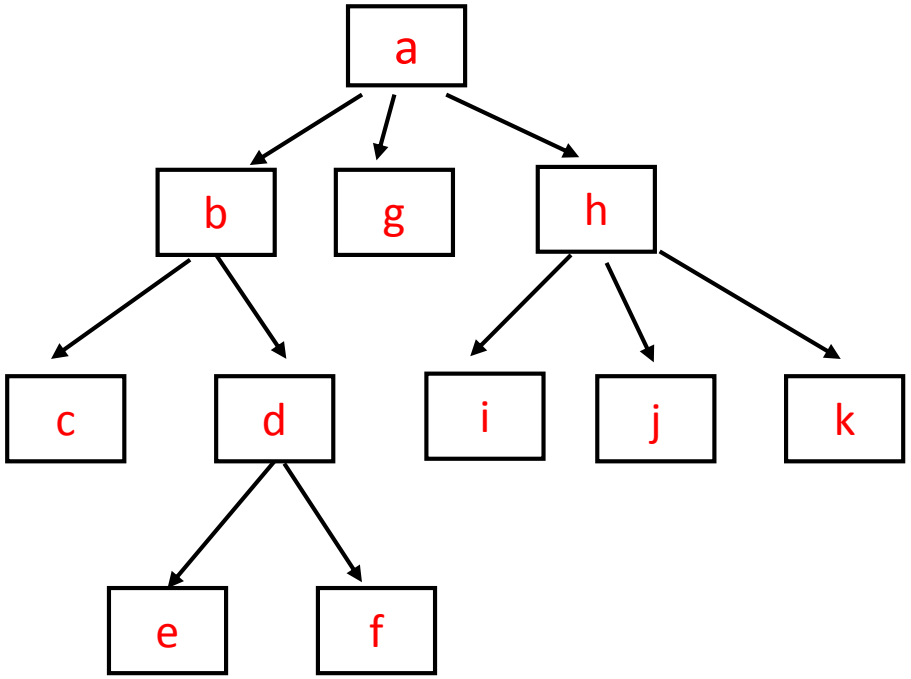
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur

    }
}
```

```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```

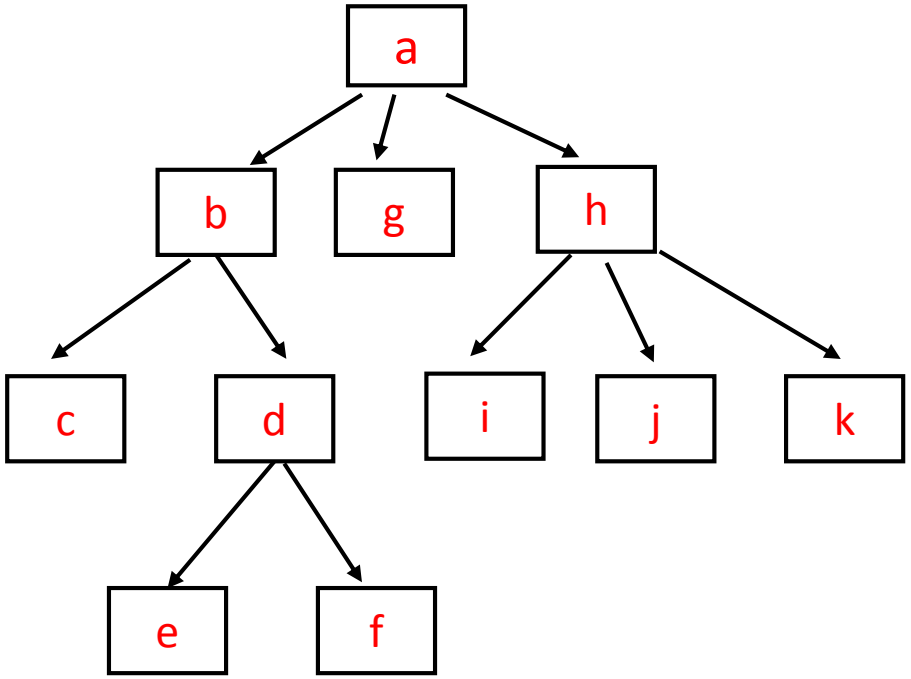
What is the order of nodes visited ?

```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```

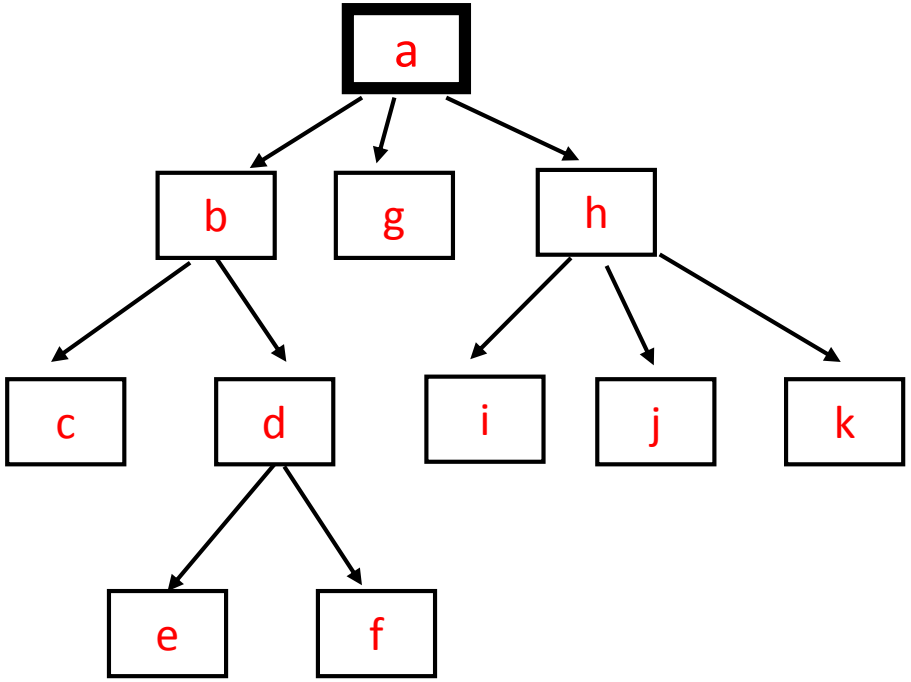


a

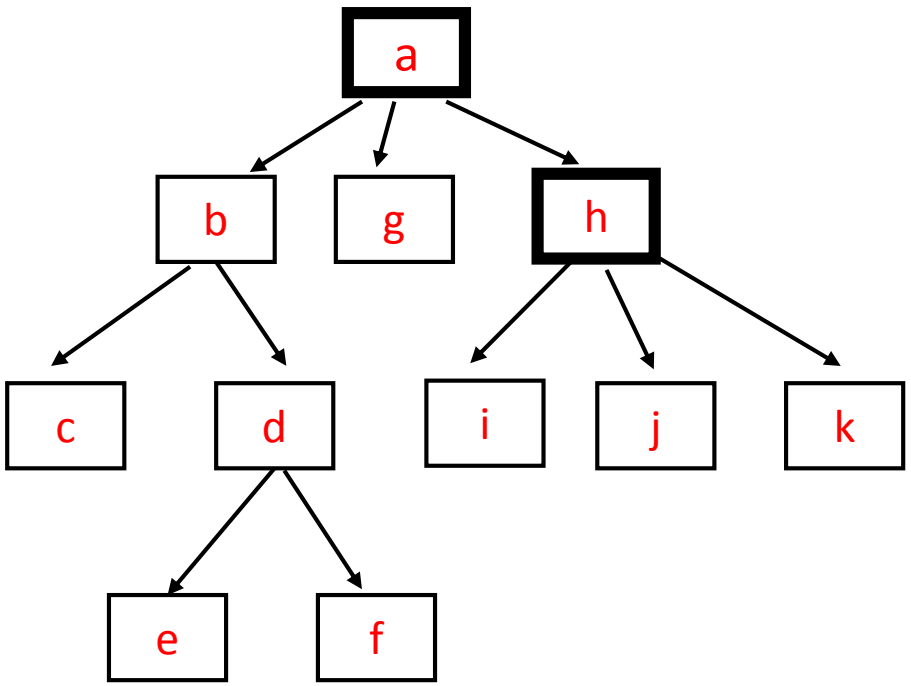
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```



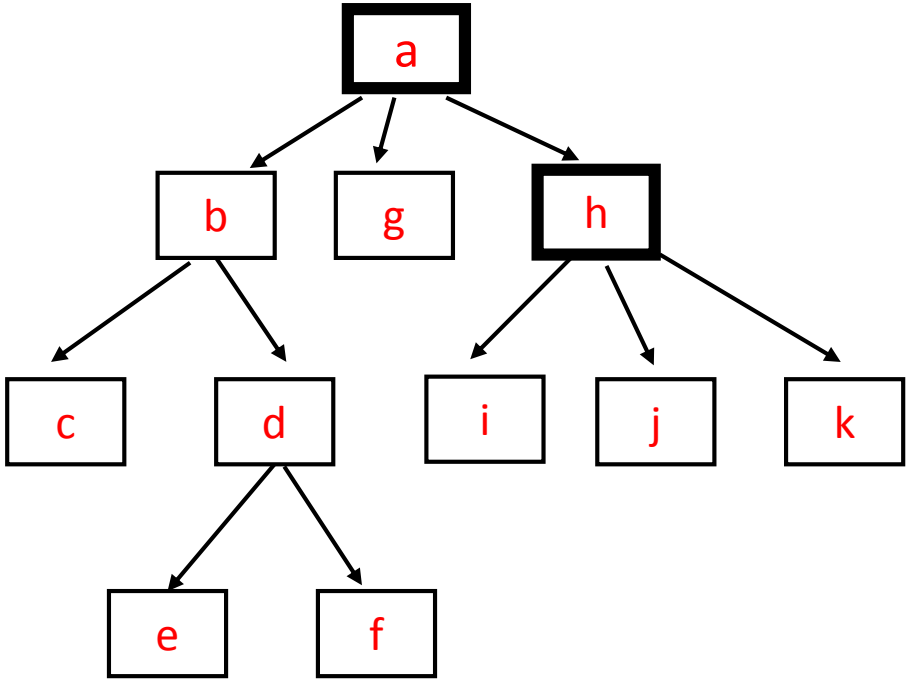
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```



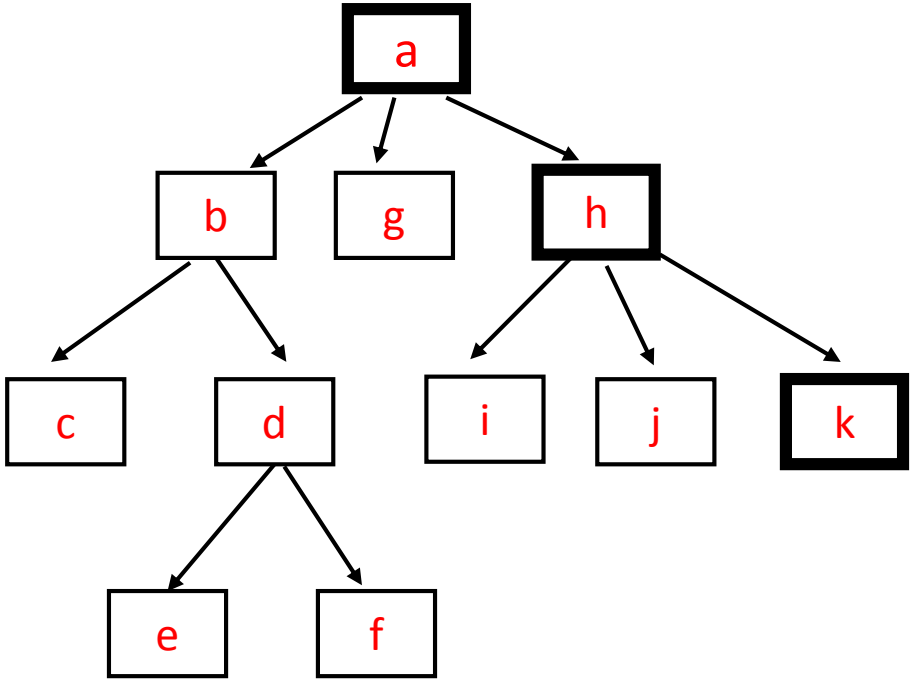
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```

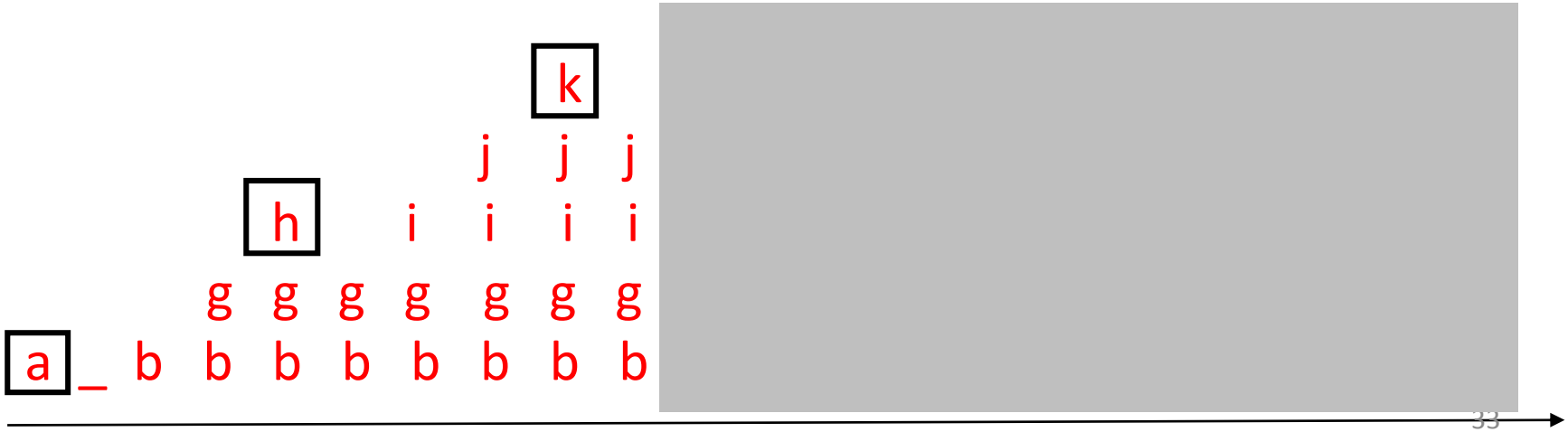


```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```

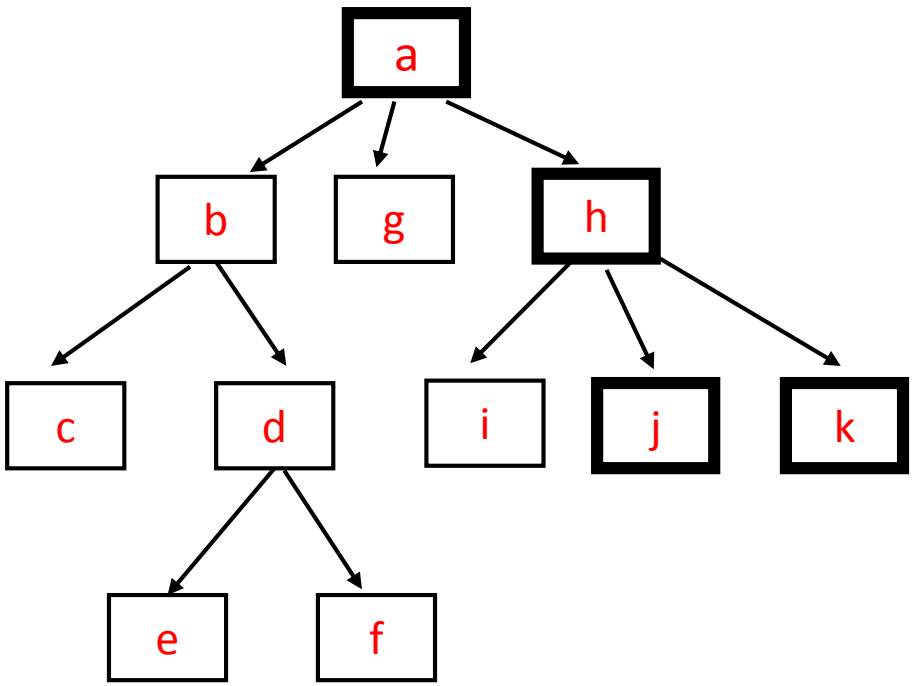


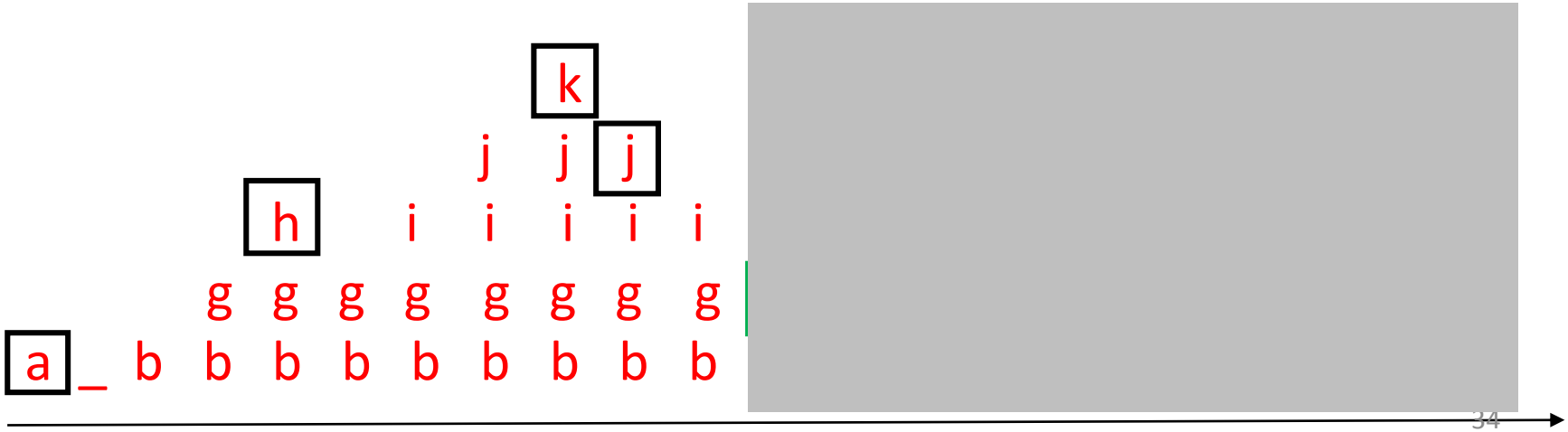
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```



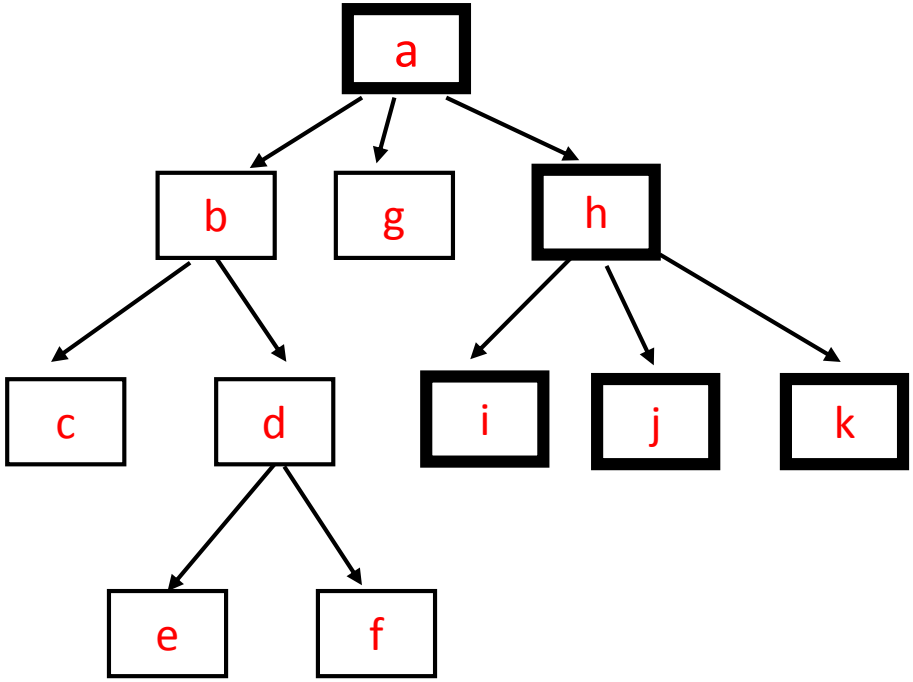


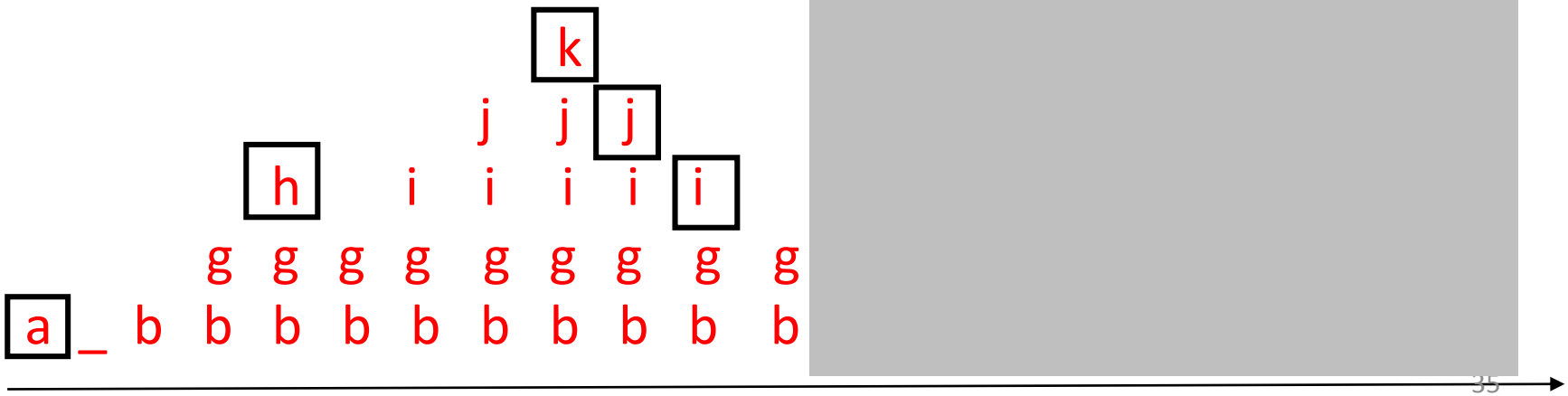
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```



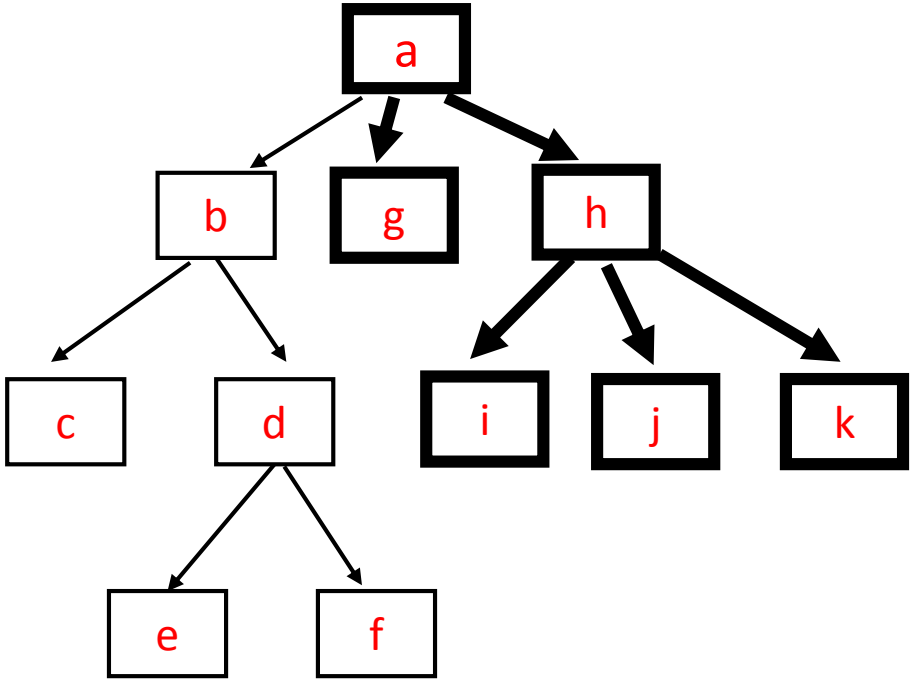


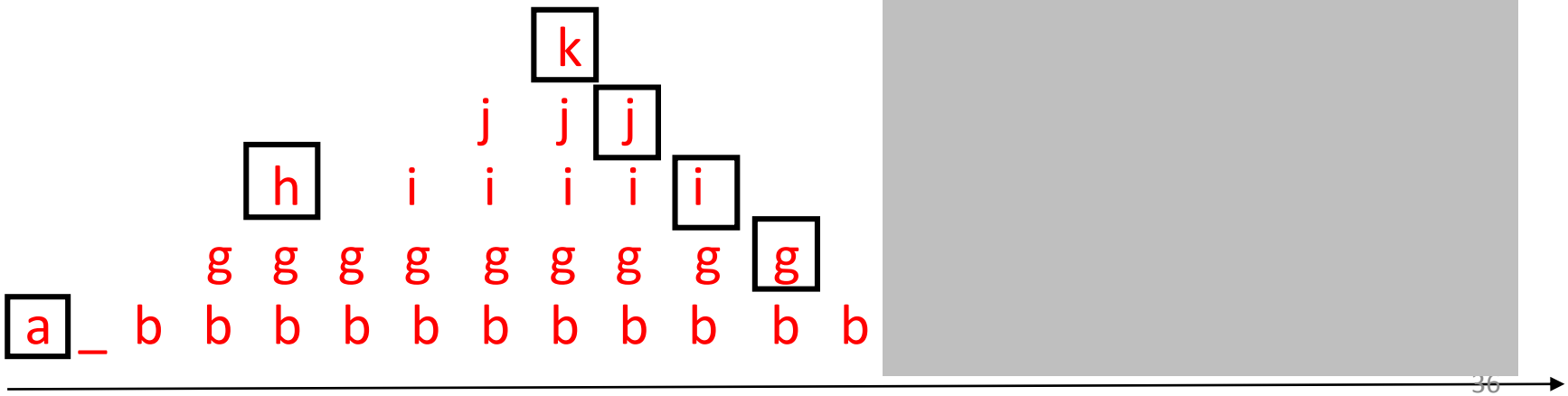
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```



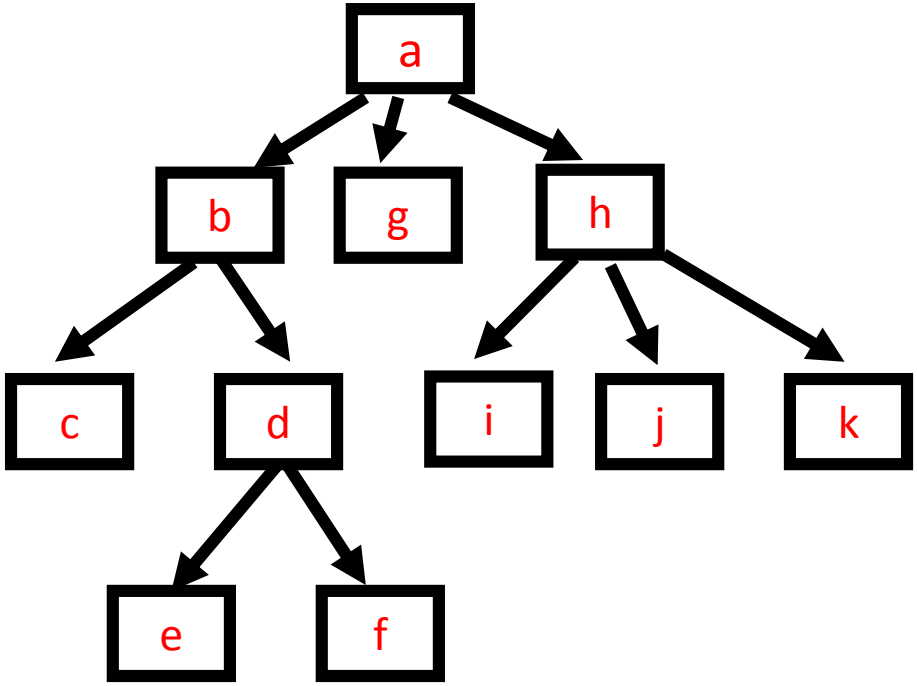


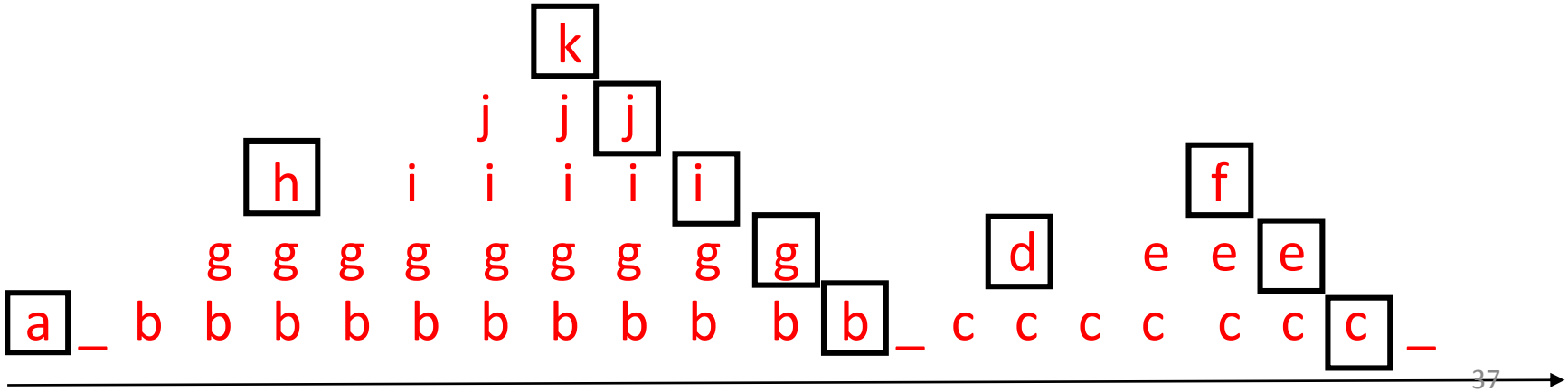
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```



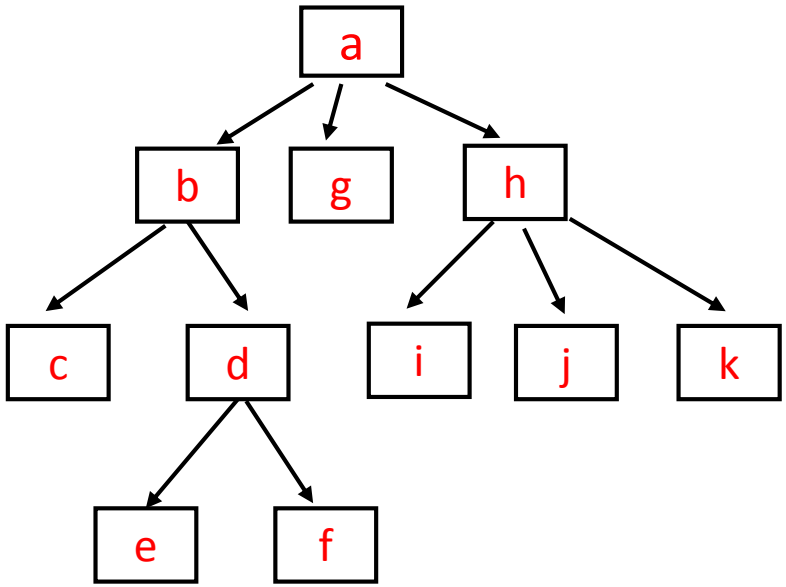


```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```





Stack based method is depth first, but visits children from right to left



recursive abcdefghijk

non-recursive (stack) ahkjigbdfec

Pre- or post order?

```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
        visit cur
    }
}
```

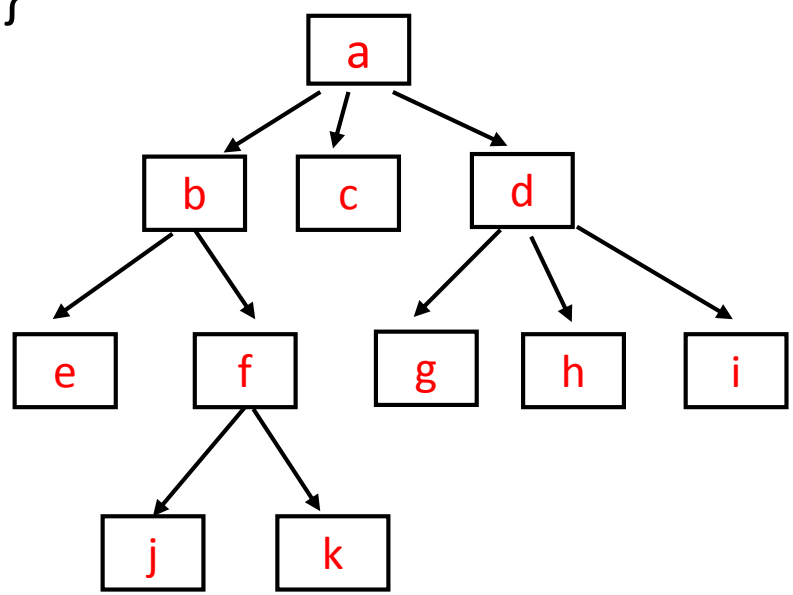
Moving the visit does not make it post order. Why not?

What if we use a queue instead?

```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```

```
treeTraversalUsingQueue(root){
    initialize empty queue  q
    q.enqueue(root)
    while q is not empty {
        cur = q.dequeue()
        visit cur
        for each child of cur
            q.enqueue(child)
    }
}
```

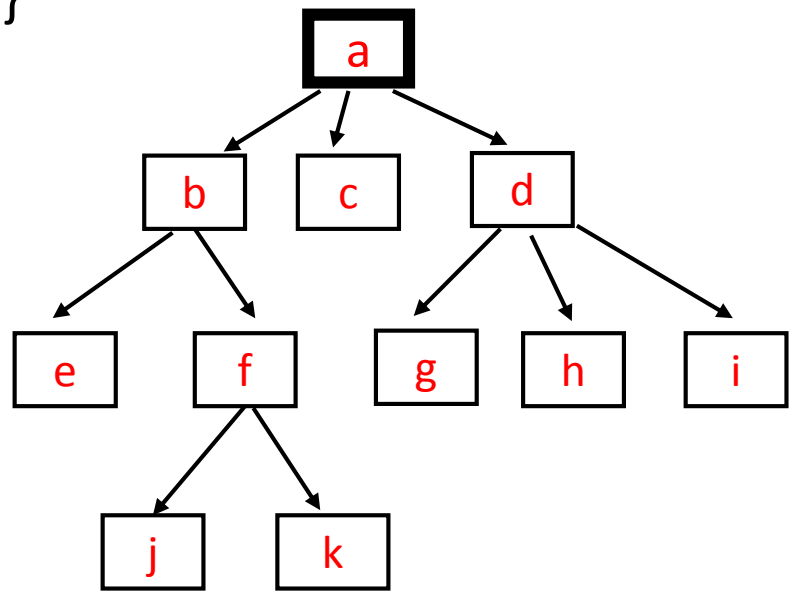
```
treeTraversalUsingQueue(root){
    initialize empty queue  q
    q.enqueue(root)
    while q is not empty {
        cur = q.dequeue()
        visit cur
        for each child of cur
            q.enqueue(child)
    }
}
```



Queue state at start of the while loop

a

```
treeTraversalUsingQueue(root){
    initialize empty queue  q
    q.enqueue(root)
    while q is not empty {
        cur = q.dequeue()
        visit cur
        for each child of cur
            q.enqueue(child)
    }
}
```

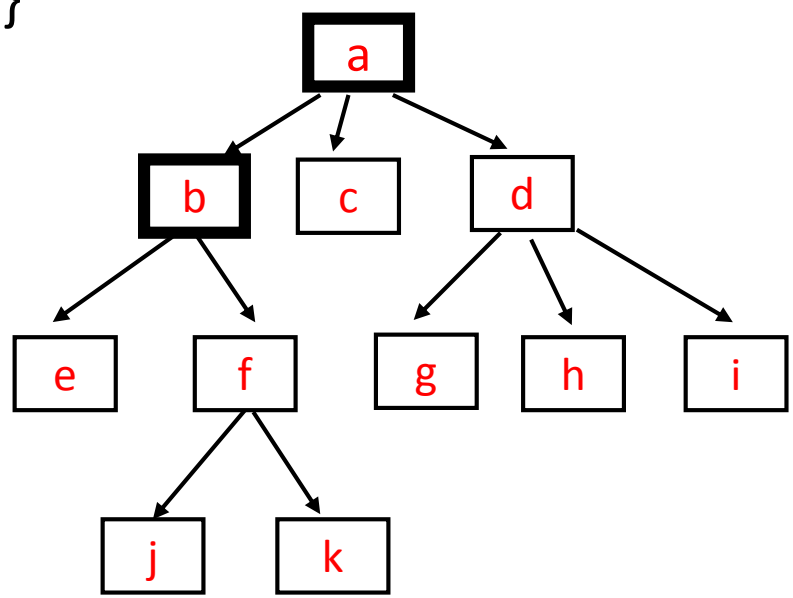


Queue state at start of the while loop

a

b c d

```
treeTraversalUsingQueue(root){
    initialize empty queue  q
    q.enqueue(root)
    while q is not empty {
        cur = q.dequeue()
        visit cur
        for each child of cur
            q.enqueue(child)
    }
}
```



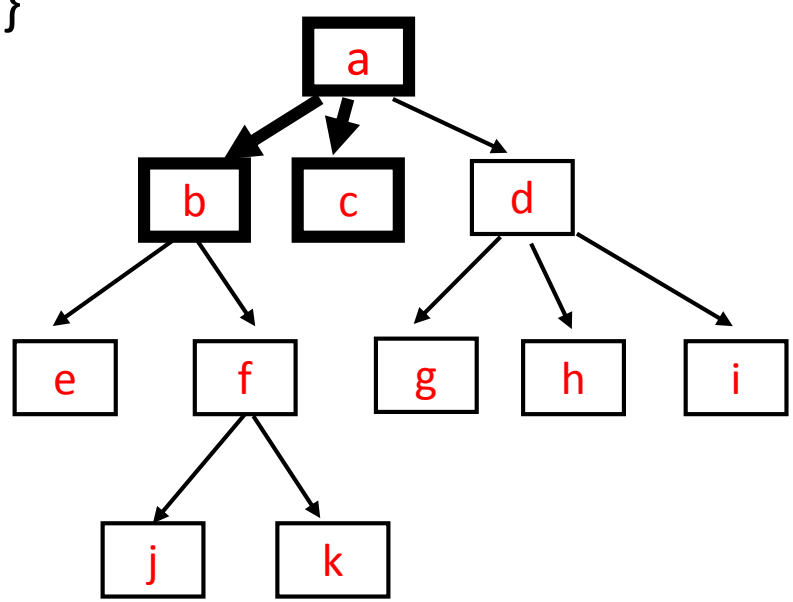
Queue state at start of the while loop

a

b c d

c d e f

```
treeTraversalUsingQueue(root){
    initialize empty queue  q
    q.enqueue(root)
    while q is not empty {
        cur = q.dequeue()
        visit cur
        for each child of cur
            q.enqueue(child)
    }
}
```



Queue state at start of the while loop

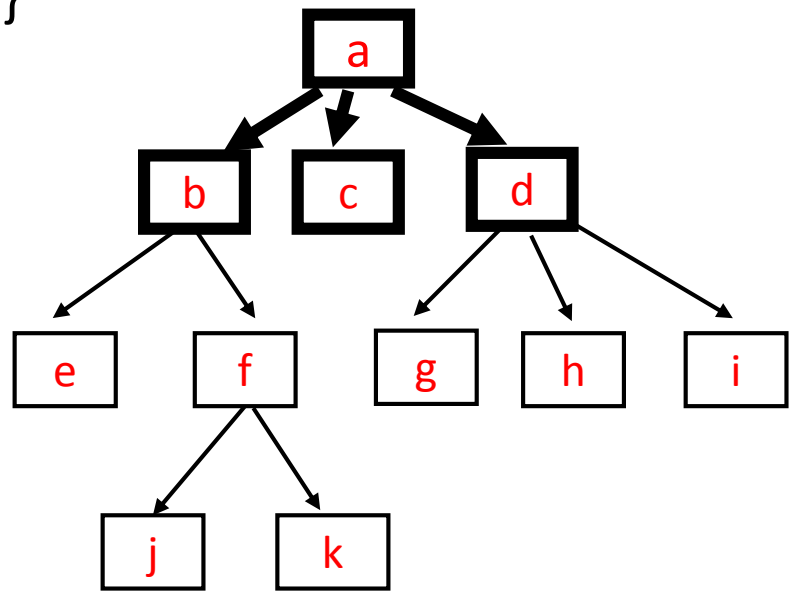
a

b c d

c d e f

d e f

```
treeTraversalUsingQueue(root){
    initialize empty queue  q
    q.enqueue(root)
    while q is not empty {
        cur = q.dequeue()
        visit cur
        for each child of cur
            q.enqueue(child)
    }
}
```



Queue state at start of the while loop

a

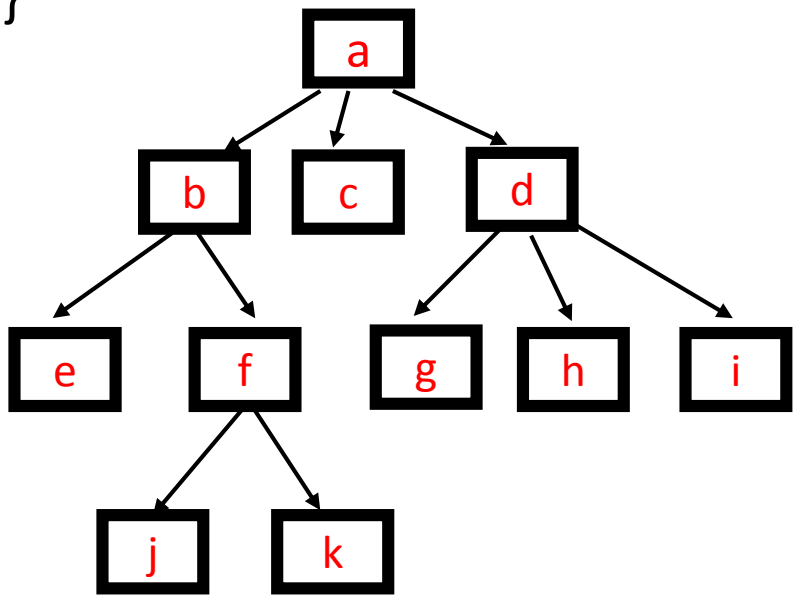
b c d

c d e f

d e f

e f g h i

```
treeTraversalUsingQueue(root){
    initialize empty queue  q
    q.enqueue(root)
    while q is not empty {
        cur = q.dequeue()
        visit cur
        for each child of cur
            q.enqueue(child)
    }
}
```



a

b c d

c d e f

d e f

e f g h i

f g h i

g h i j k

h i j k

i j k

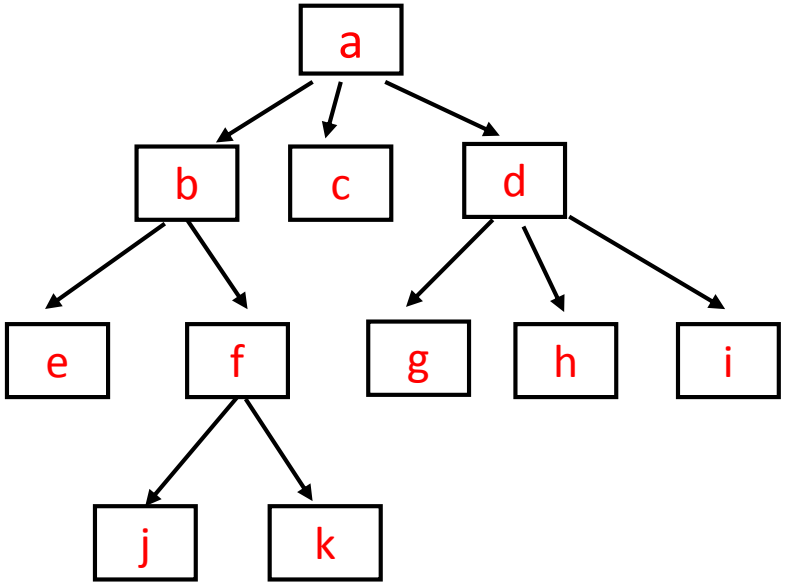
j k

k

breadth first traversal

for each level i

visit all nodes at level i

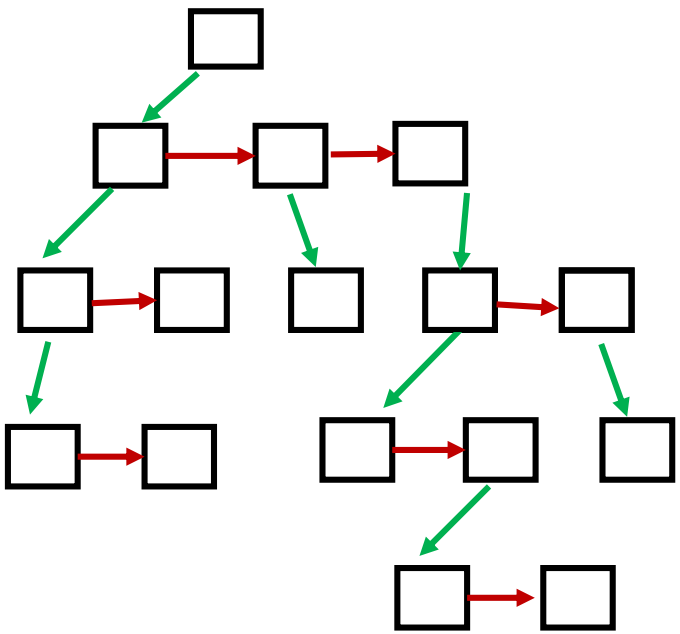


order visited: abcdefghijk

Implementation Details
Recall: 'first child, next sibling'

```
class  TreeNode<T>{
    T   element;
    TreeNode<T>   firstChild;
    TreeNode<T>   nextSibling;
      :
      :
}

class  Tree<T>{
   TreeNode<T>   root;
      :
      :
}
```



```
for each child{
    …
}
```

means:

```
child = cur.firstChild
while (child  !=  null){
    …..
    child =  child.nextsibling
}
```