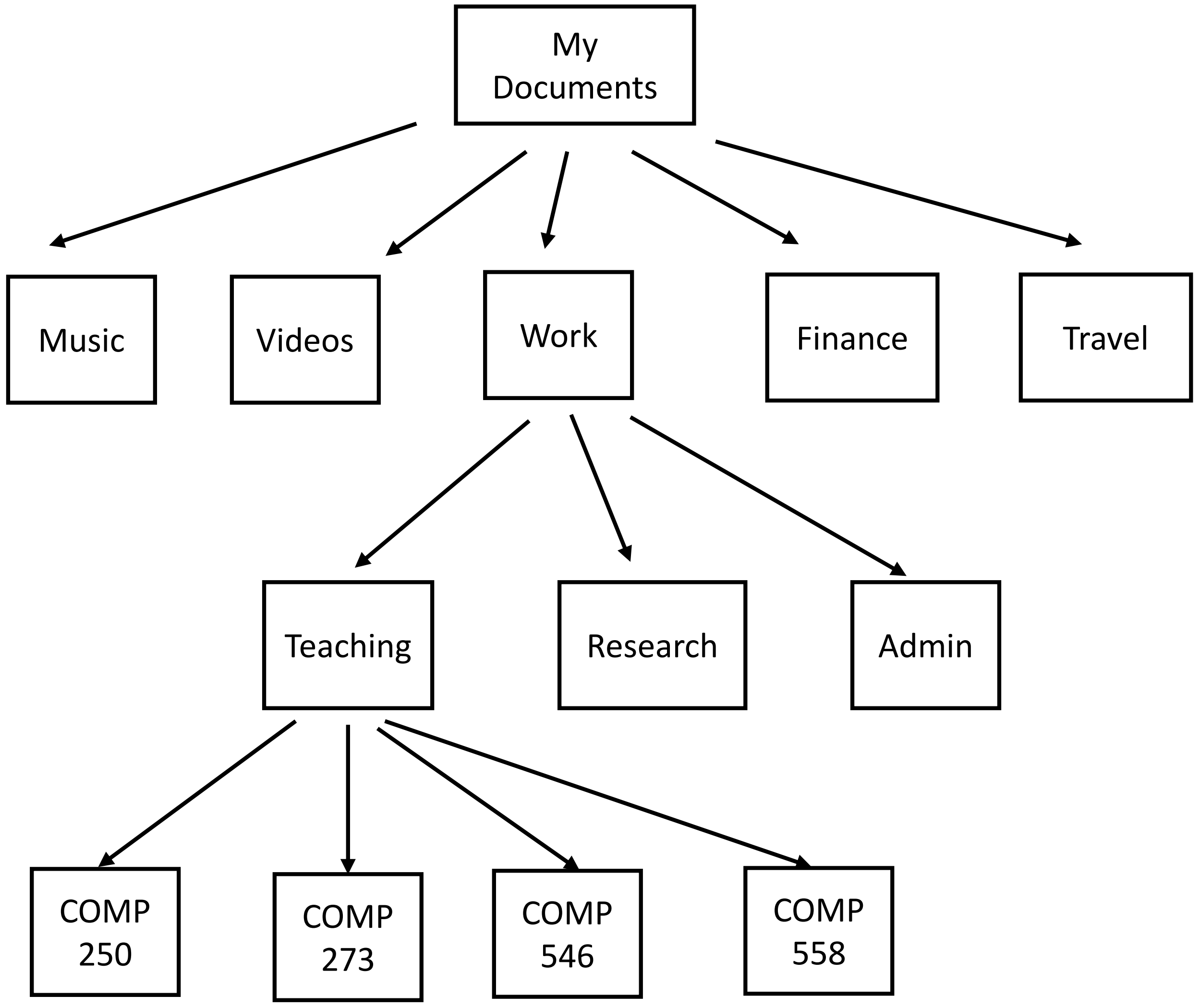
COMP 250

Lecture 23

tree traversal

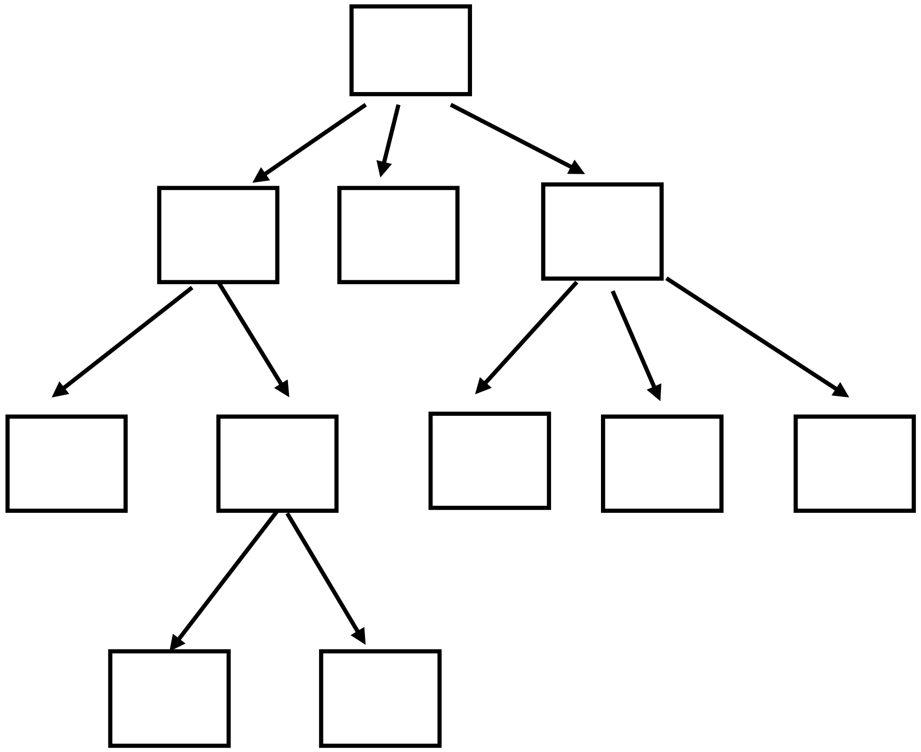
Nov. 2, 2018

Tree: Example

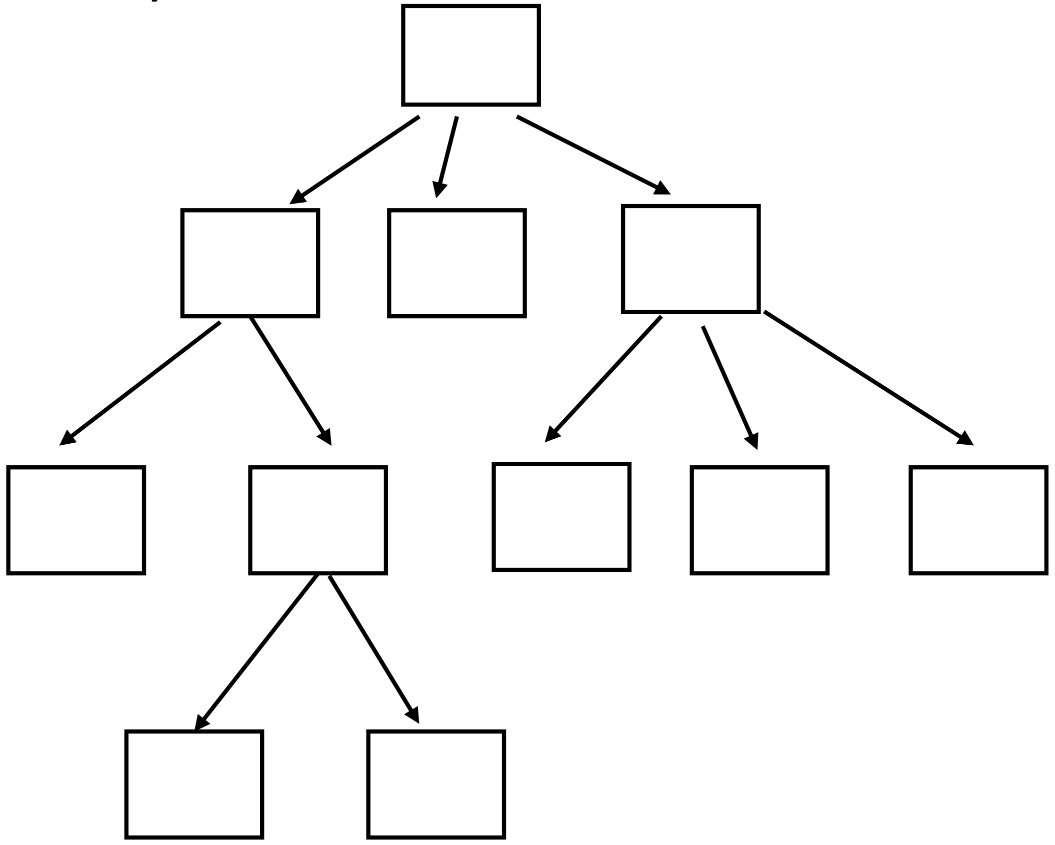


Tree Traversal

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

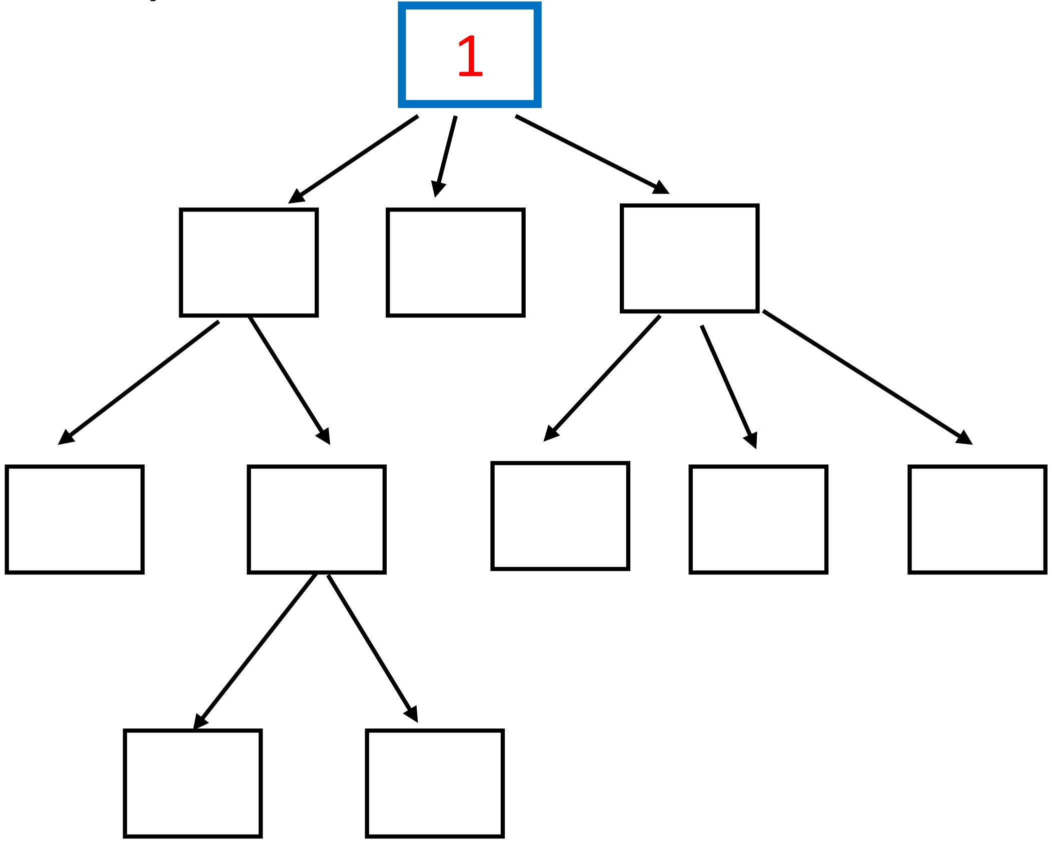


```
depthfirst (root){
    if (root is not empty){
        visit root
        for each child of root
            depthfirst( child )
    }
}
```

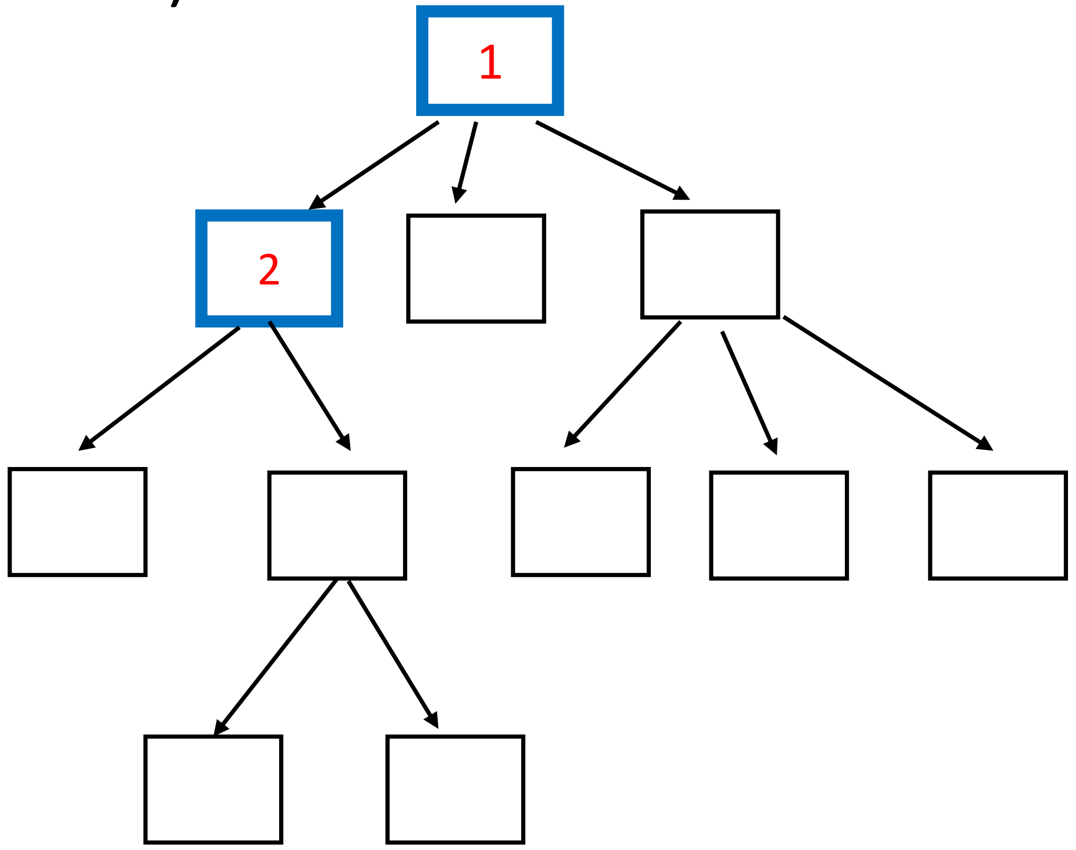


```
depthfirst (root){
    if (root is not empty){
        visit root
        for each child of root
            depthfirst( child )
    }
}
```

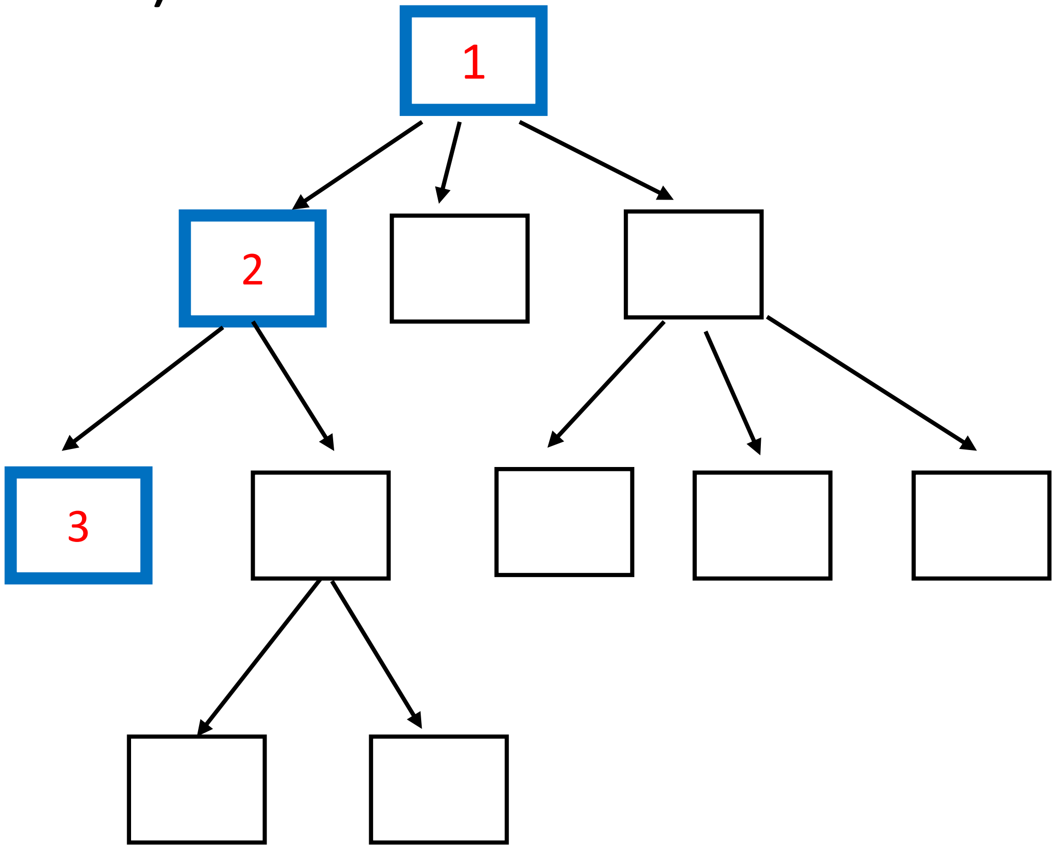
“preorder” traversal: visit the root before the children



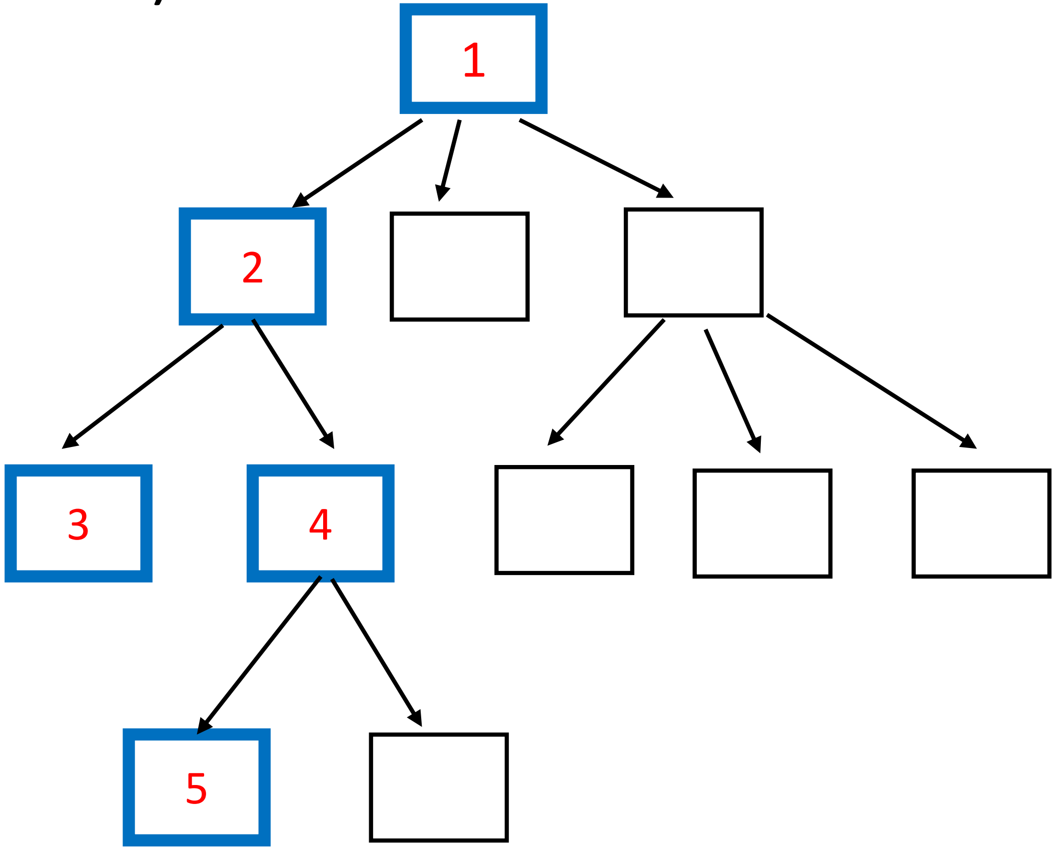
```
depthfirst (root){
    if (root is not empty){
        visit root
        for each child of root
            depthfirst( child )
    }
}
```



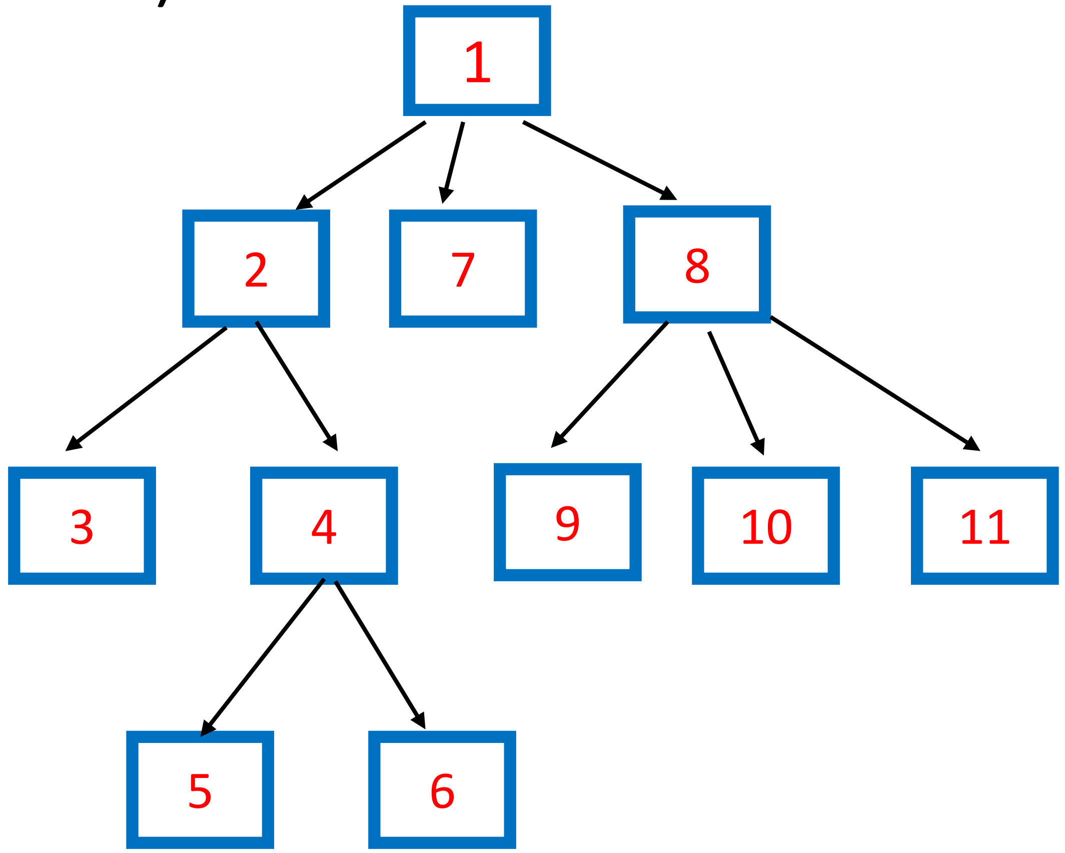
```
depthfirst (root){
    if (root is not empty){
        visit root
        for each child of root
            depthfirst( child )
    }
}
```



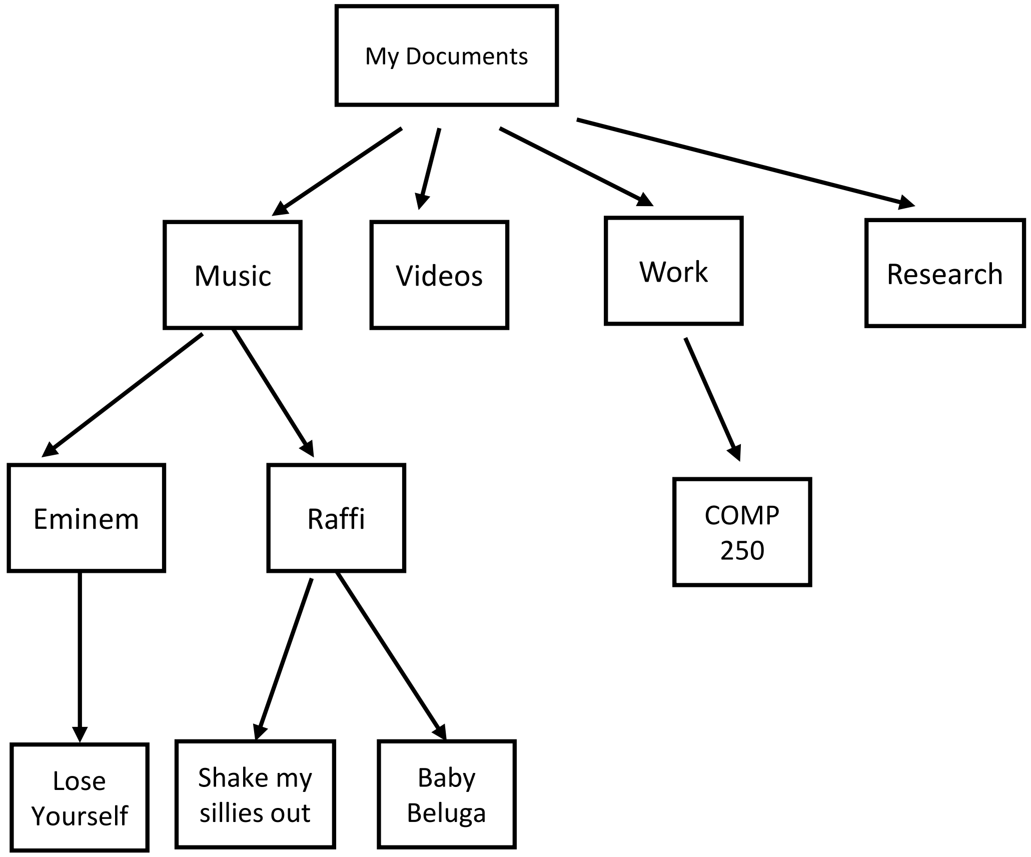
```
depthfirst (root){
    if (root is not empty){
        visit root
        for each child of root
            depthfirst( child )
    }
}
```



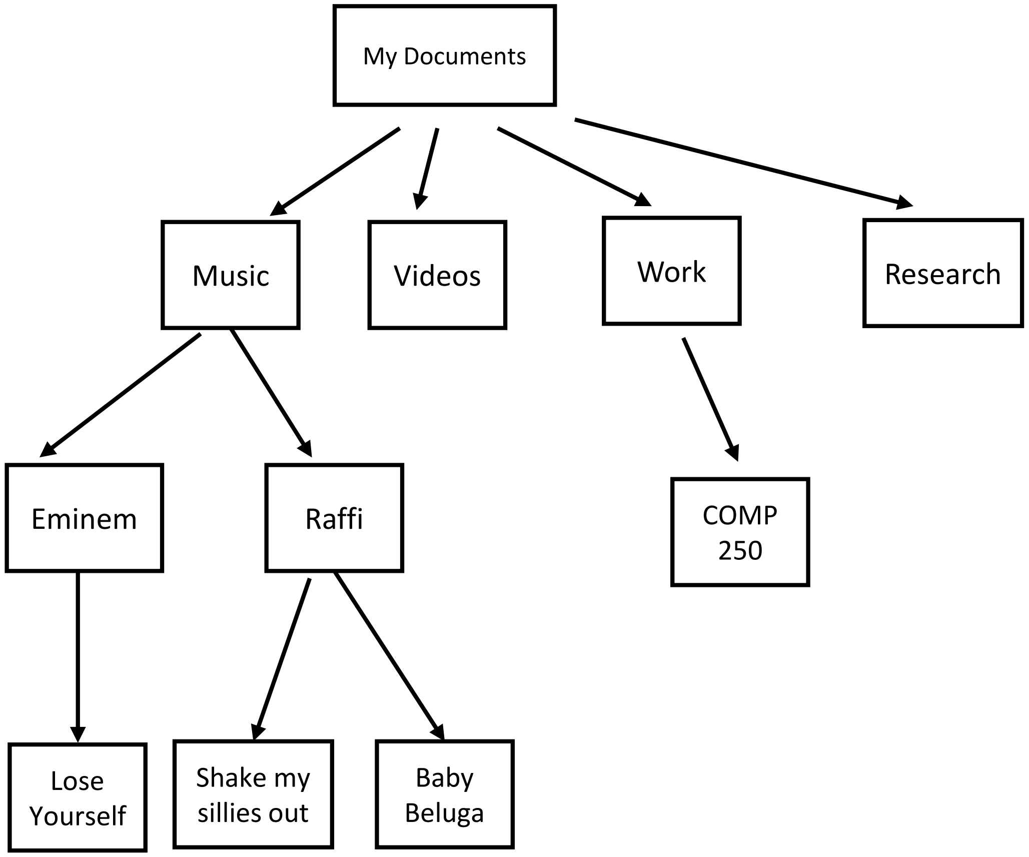
```
depthfirst (root){
    if (root is not empty){
        visit root
        for each child of root
            depthfirst( child )
    }
}
```



Example of Preorder Traversal: printing a hierarchical file system (visit = print directory or file name)



Example of Preorder Traversal: printing a hierarchical file system (visit = print directory or file name)



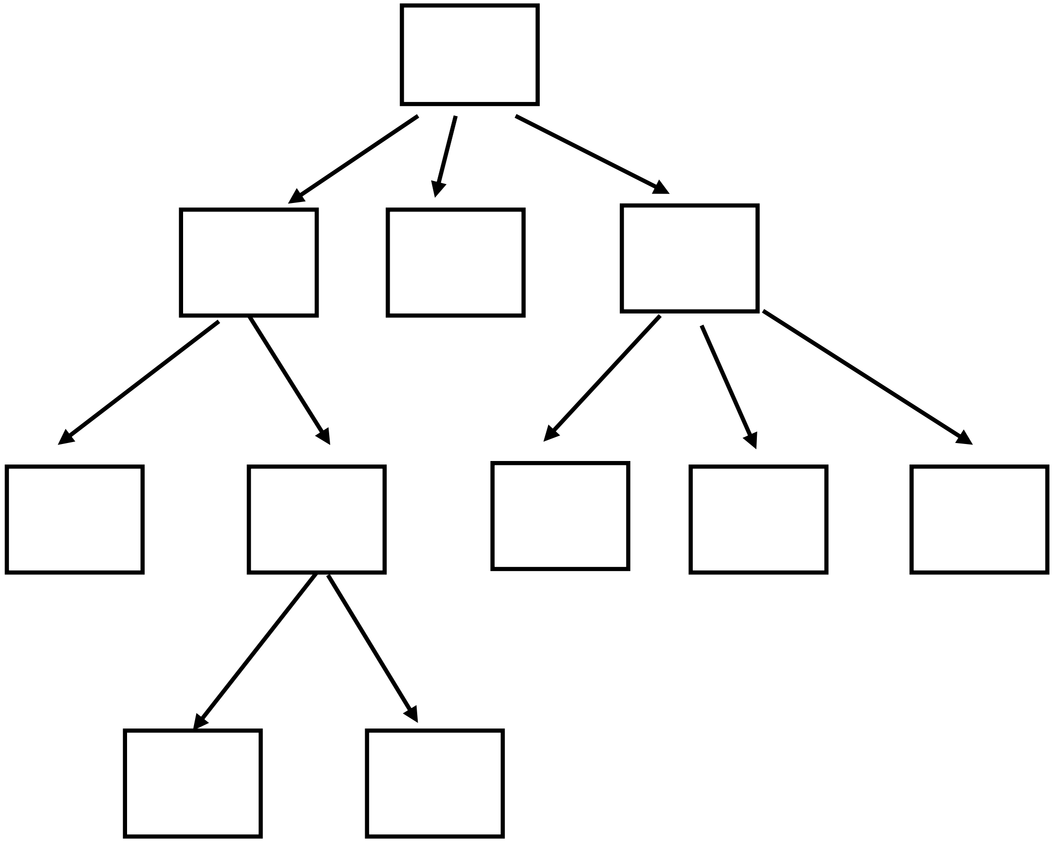
```
Documents                      (directory)
     Music                     (directory)
          Eminem               (directory)
               Lose Yourself        (file)
          Raffi                (directory)
               Shake My Sillies Out (file)
               Baby Beluga          (file)
     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.

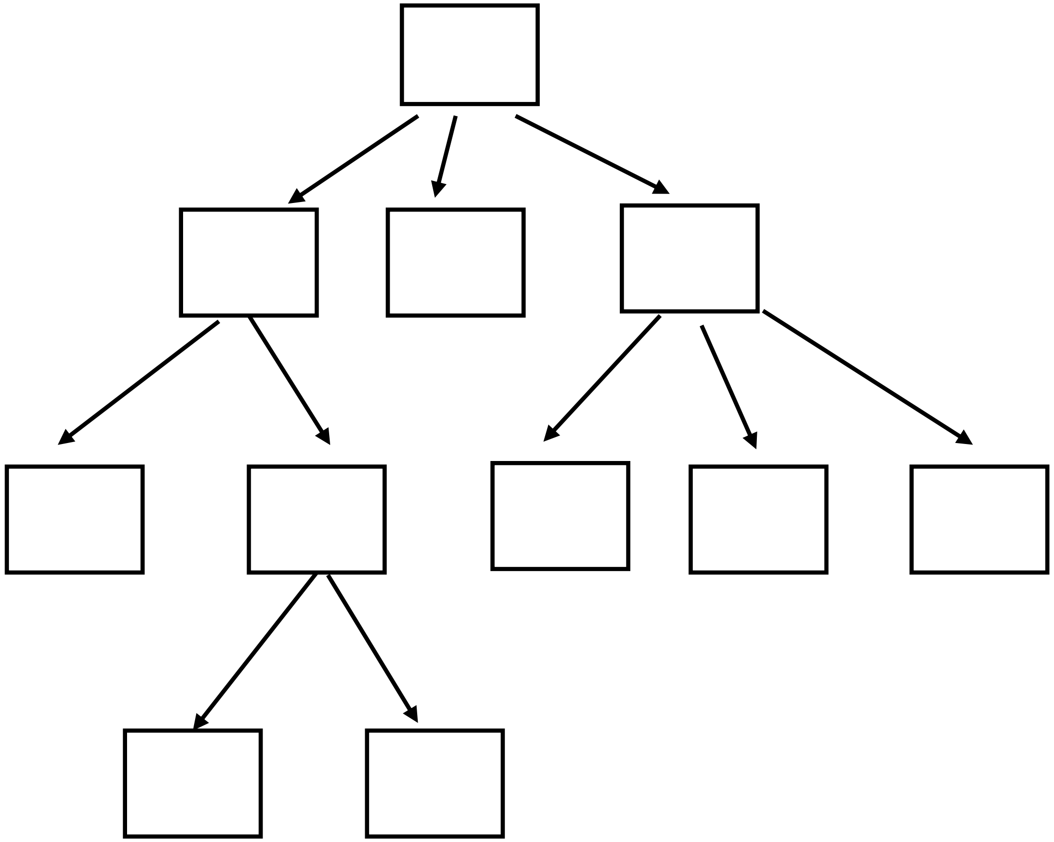
```
depthfirst (root){
    if (root is not empty){
        for each child of root
            depthfirst( child )
        visit root
    }
}
```

“postorder” traversal: visit the root after the children



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

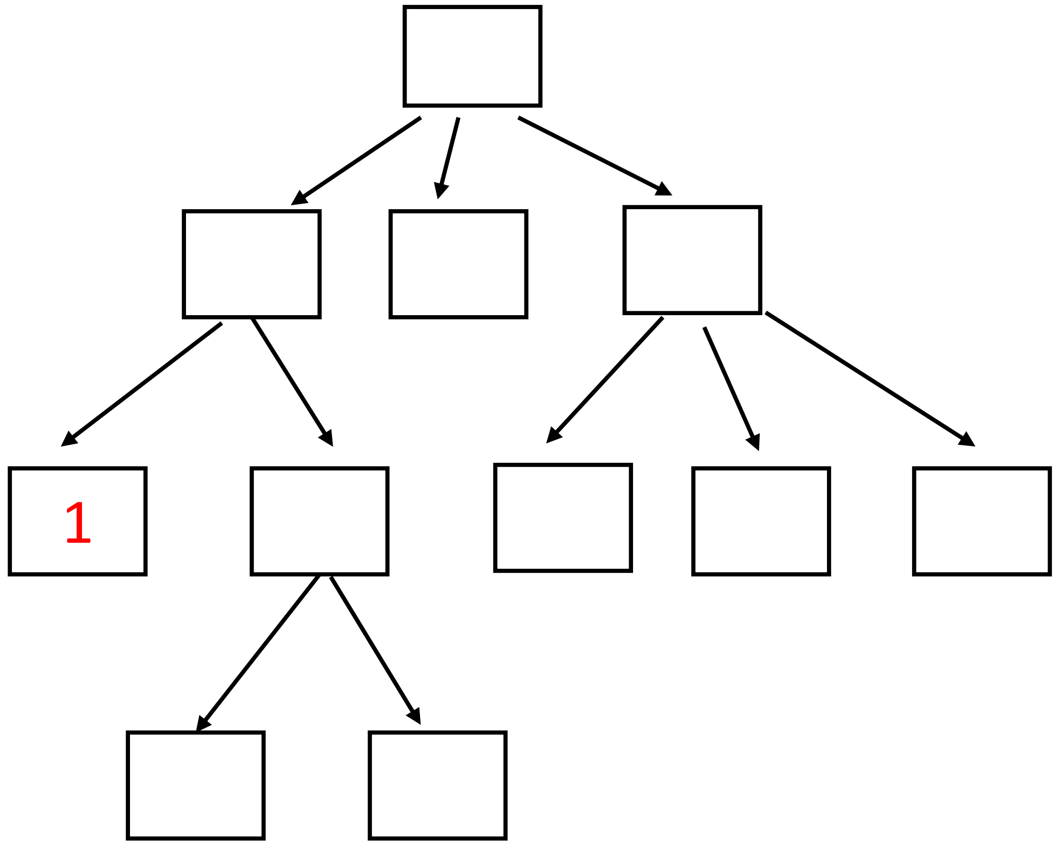
"postorder" traversal: visit the root after the children



Q: Which node is visited first?

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

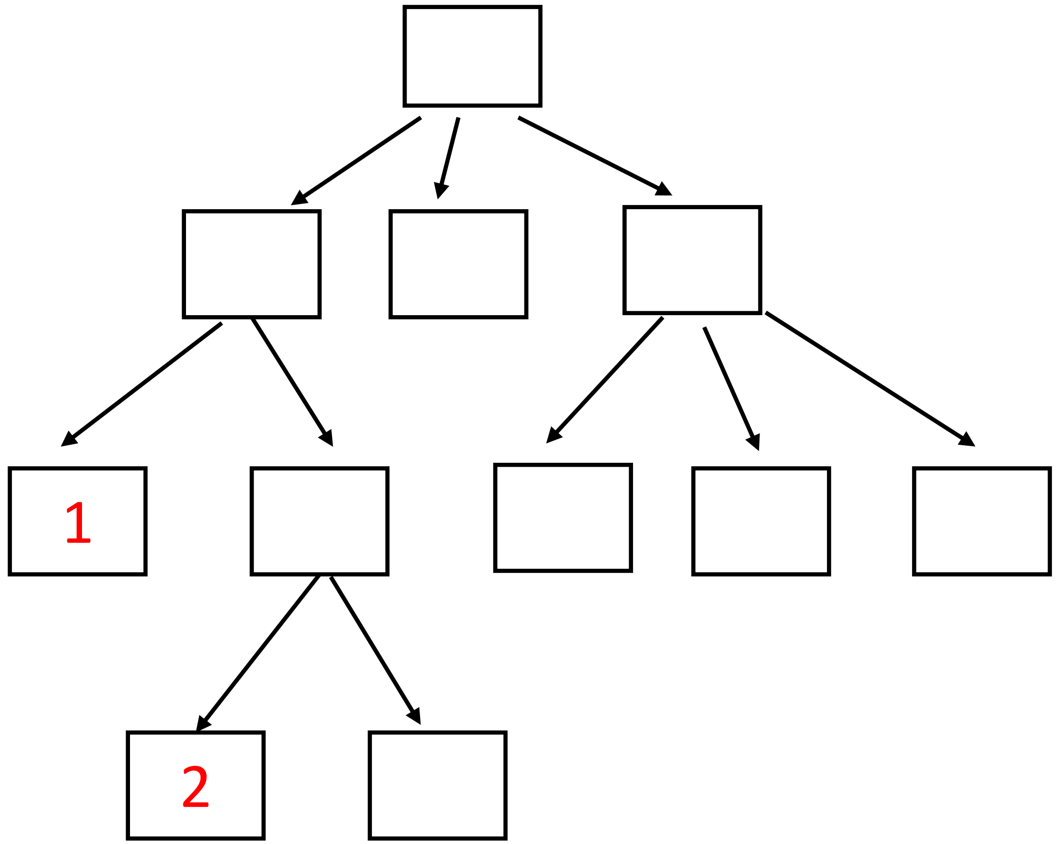
“postorder” traversal: visit the root after the children



Q: Which node is visited second?

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

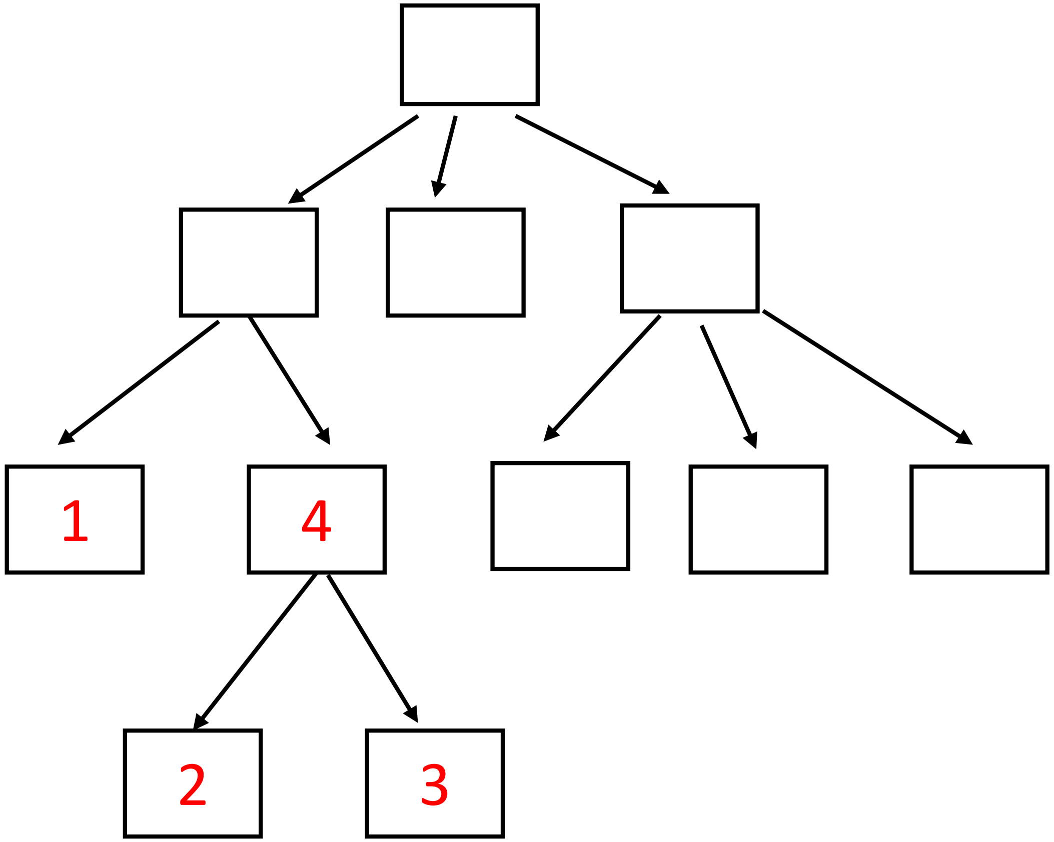
“postorder” traversal: visit the root after the children



Q: Which node is visited 3rd and 4th ?

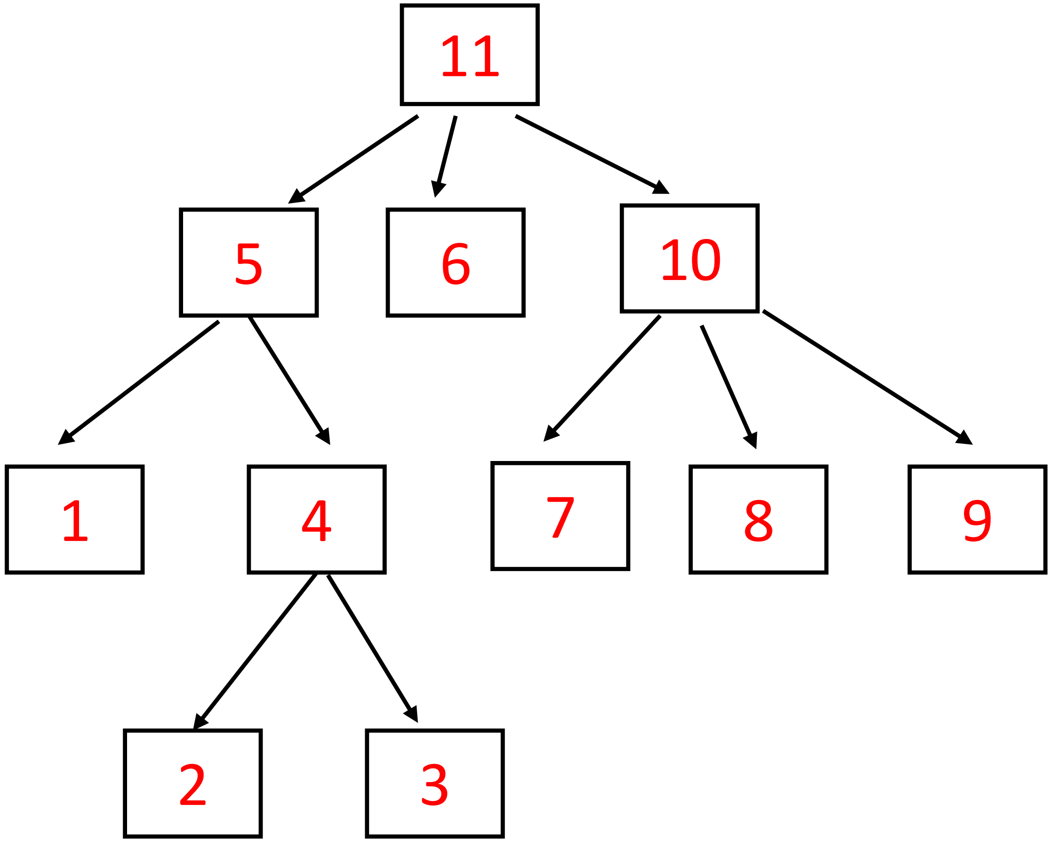
```
depthfirst (root){
    if (root is not empty){
        for each child of root
            depthfirst( child )
        visit root
    }
}
```

"postorder" traversal: visit the root after the children

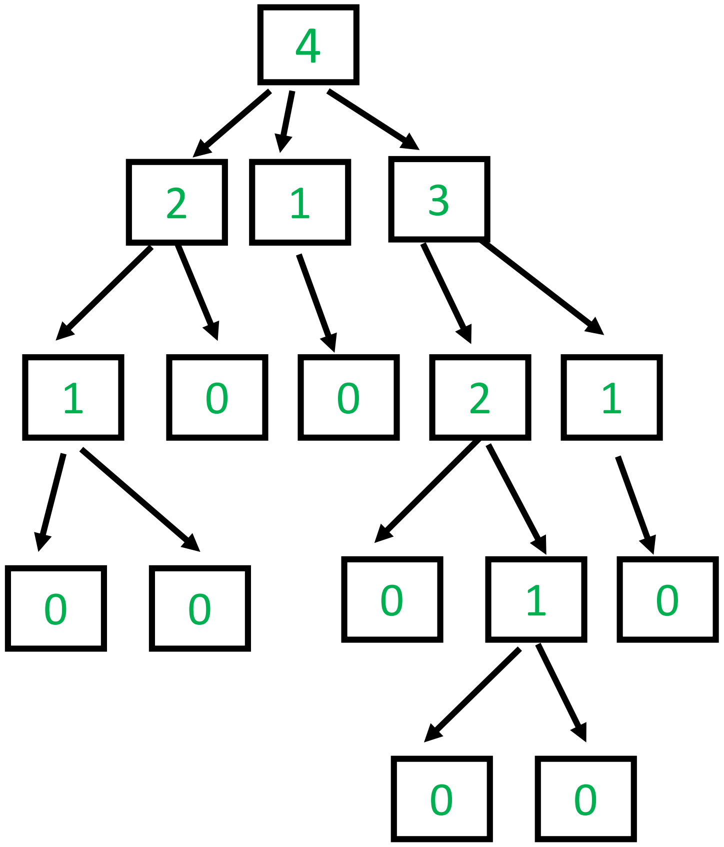


```
depthfirst (root){
    if (root is not empty){
        for each child of root
            depthfirst( child )
        visit root
    }
}
```

"postorder" traversal: visit the root after the children

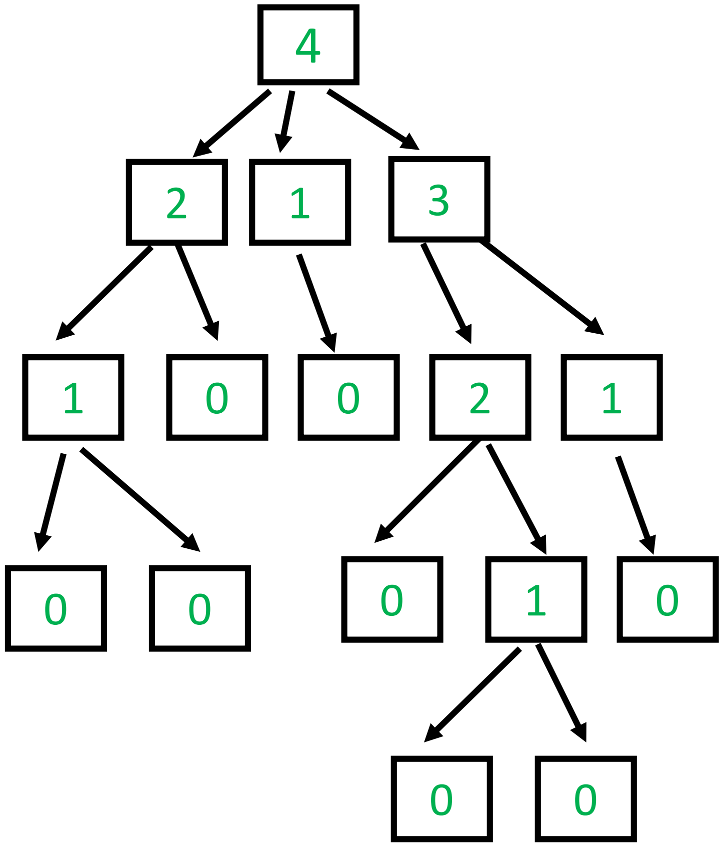


Example 1 postorder



```
height(v){

                ?

}
```

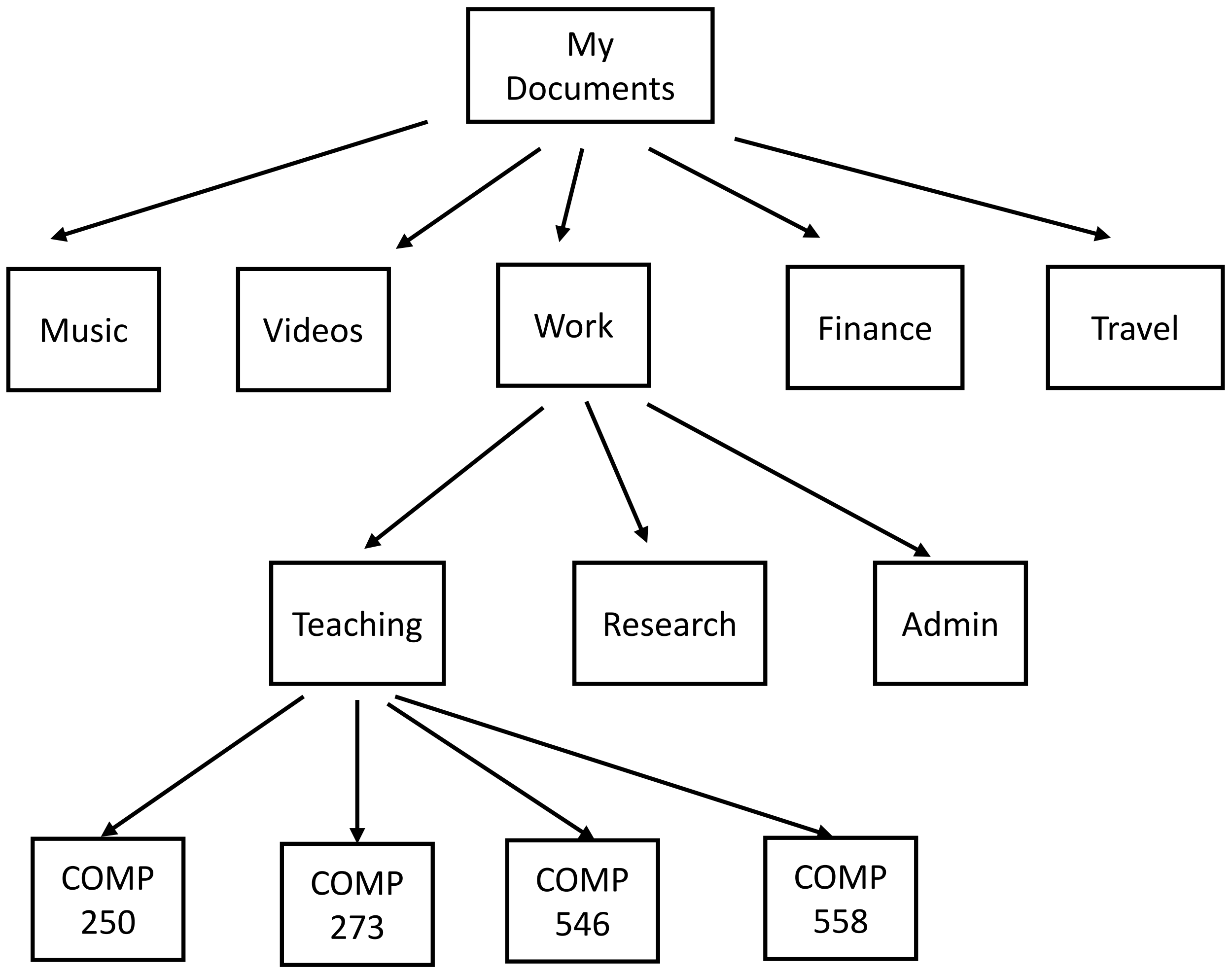
Example 1 postorder



```
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.

“preorder” traversal

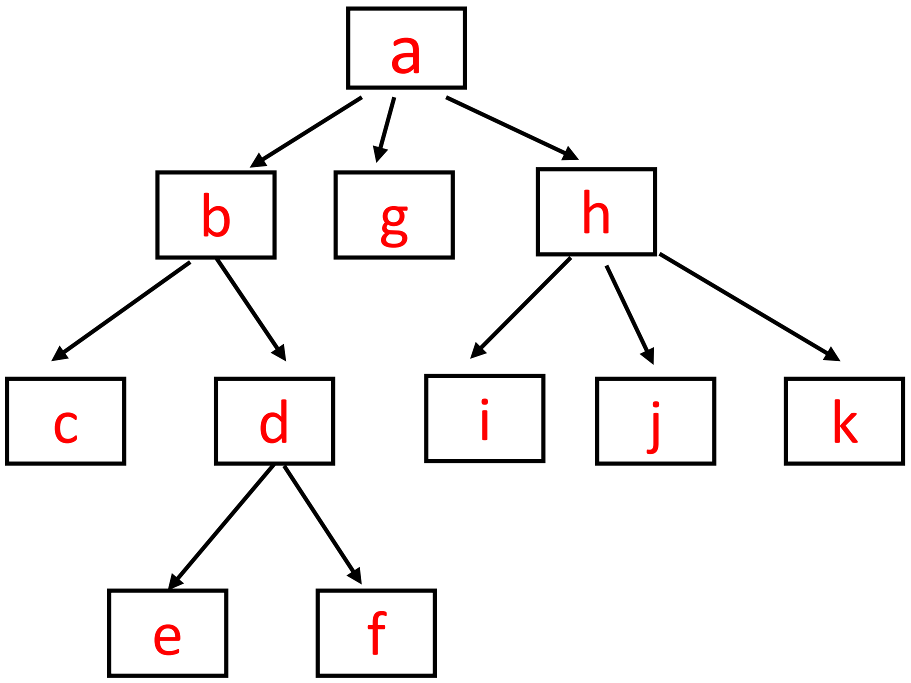
```
depthfirst (root){
    if (root is not empty){
        visit root
        for each child of root
            depthfirst( child )
    }
}
```

“postorder” traversal

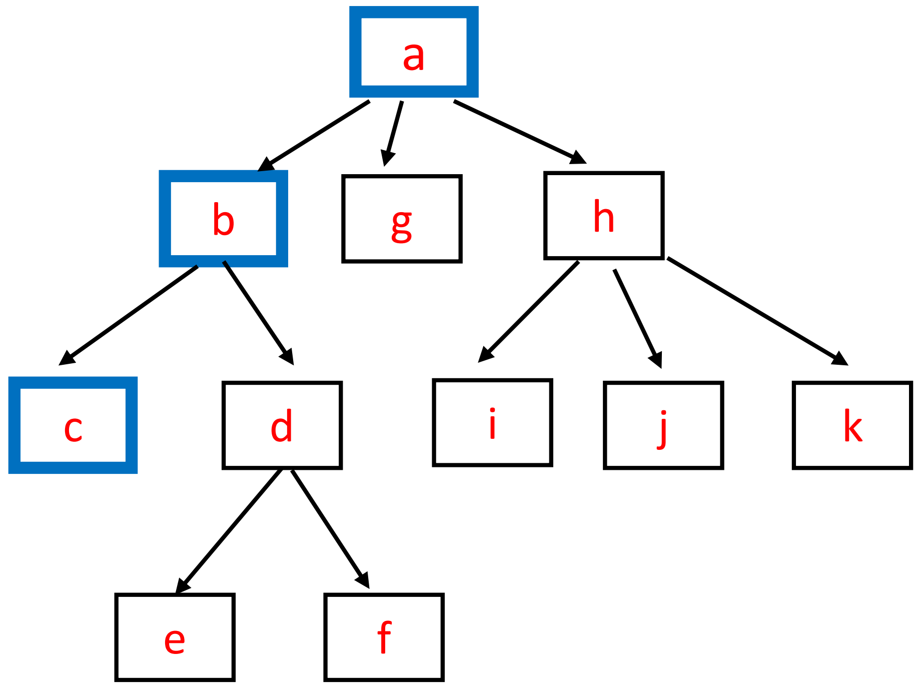
```
depthfirst (root){
    if (root is not empty){
        for each child of root
            depthfirst( child )
        visit root
    }
}
```

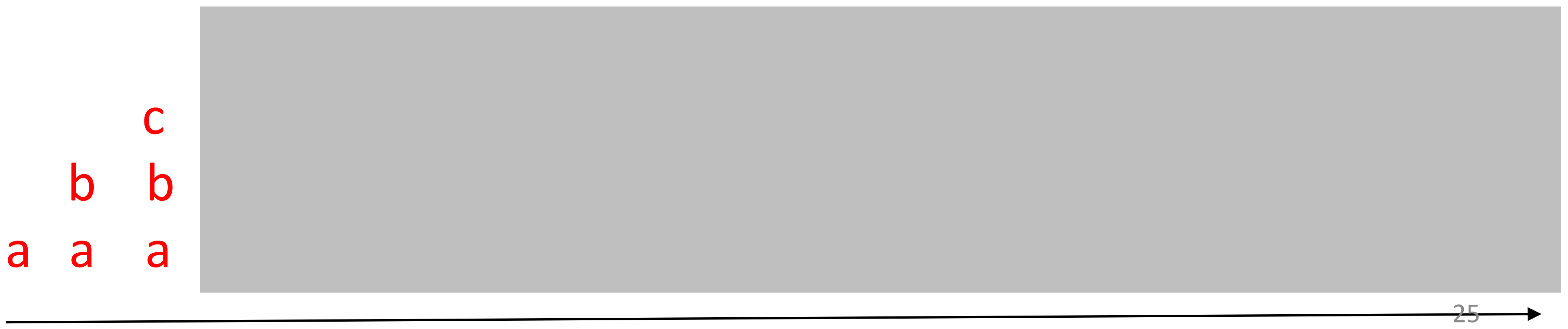
Same depthfirst(root) call sequence occurs for preorder vs postorder.

In example below, the letter order corresponds to depthfirst(root) call order.

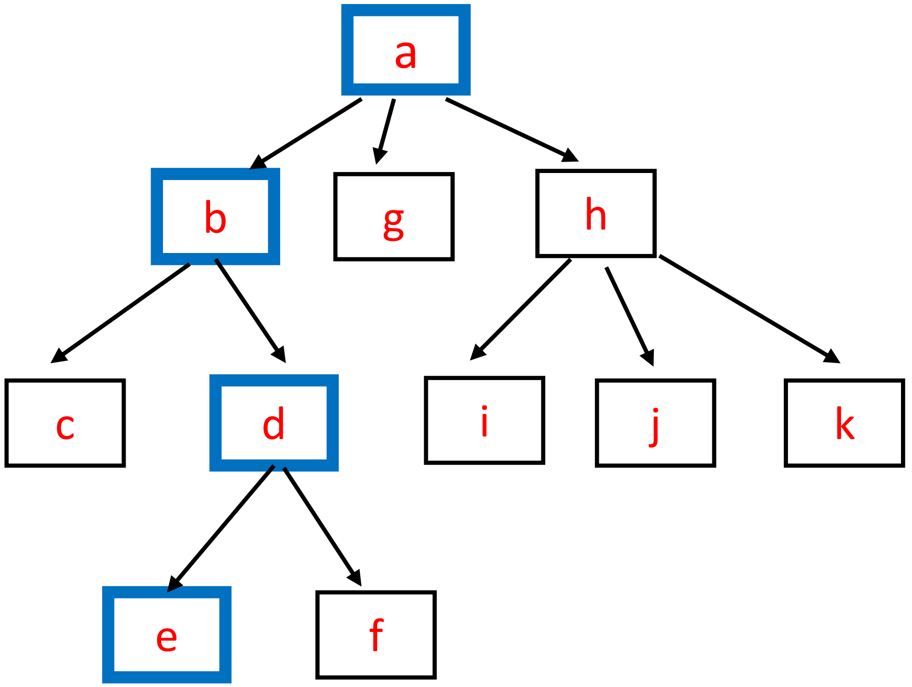


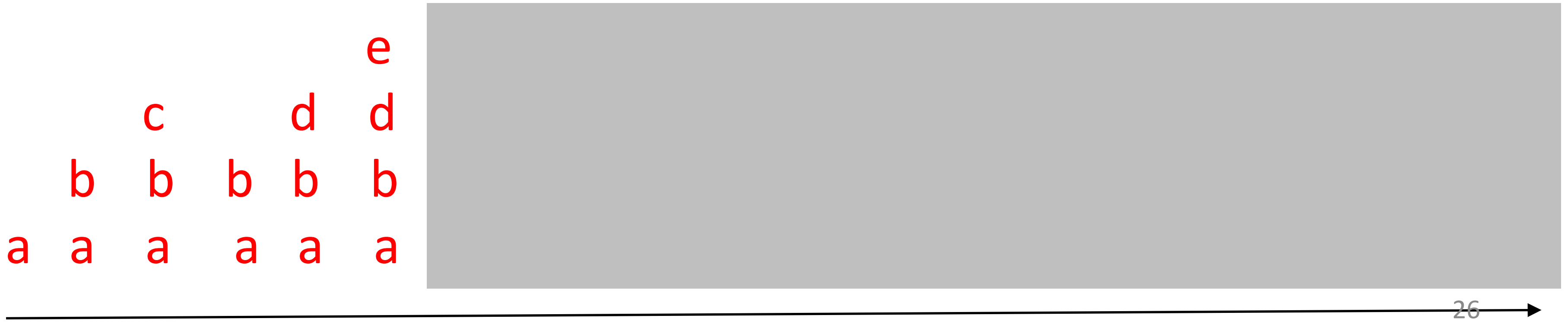
Call stack for depthfirst(root)



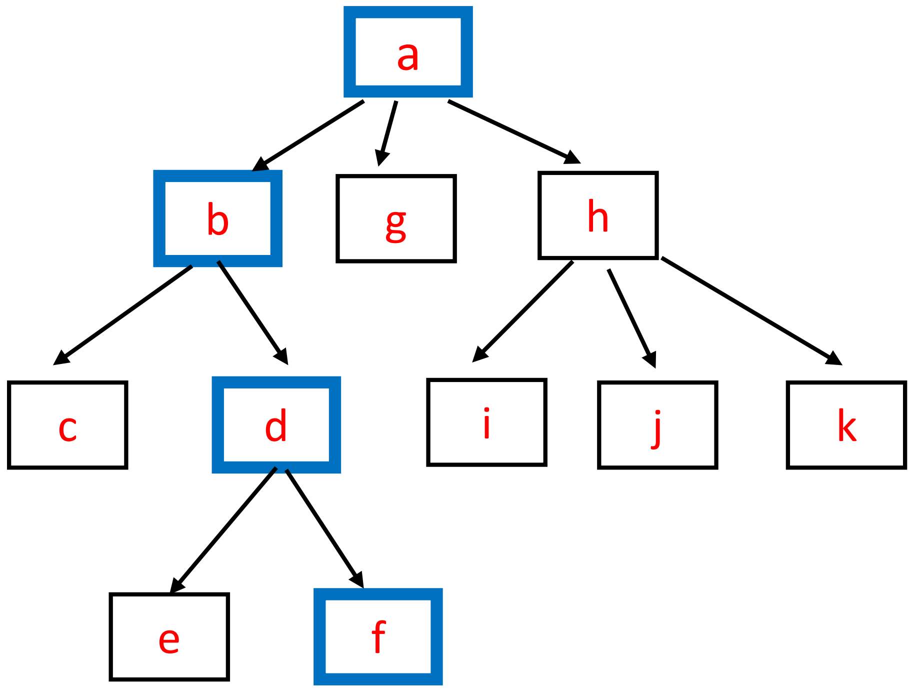


Call stack for depthfirst(root)



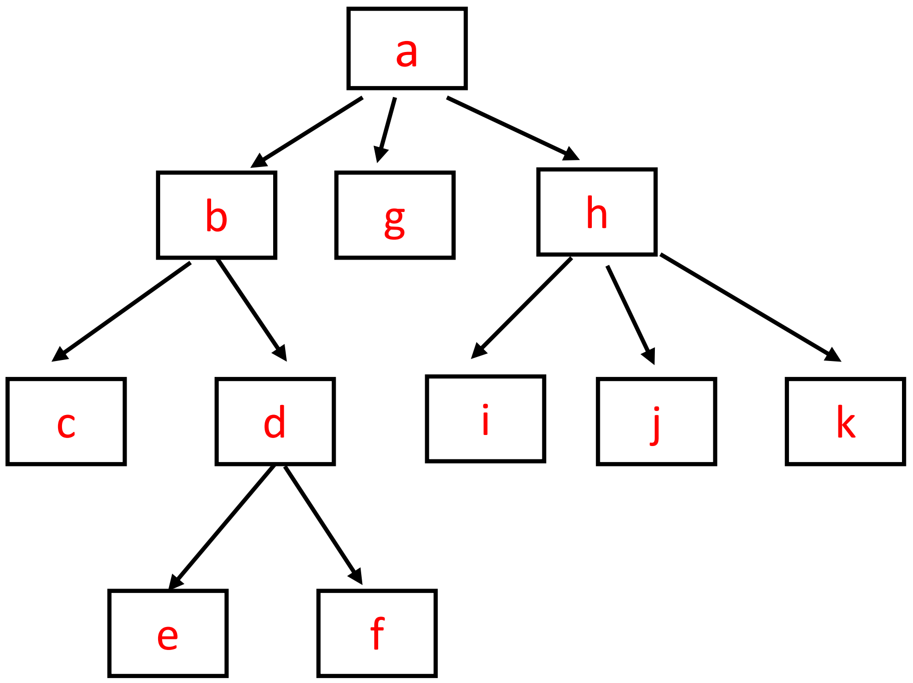


Call stack for depthfirst(root)

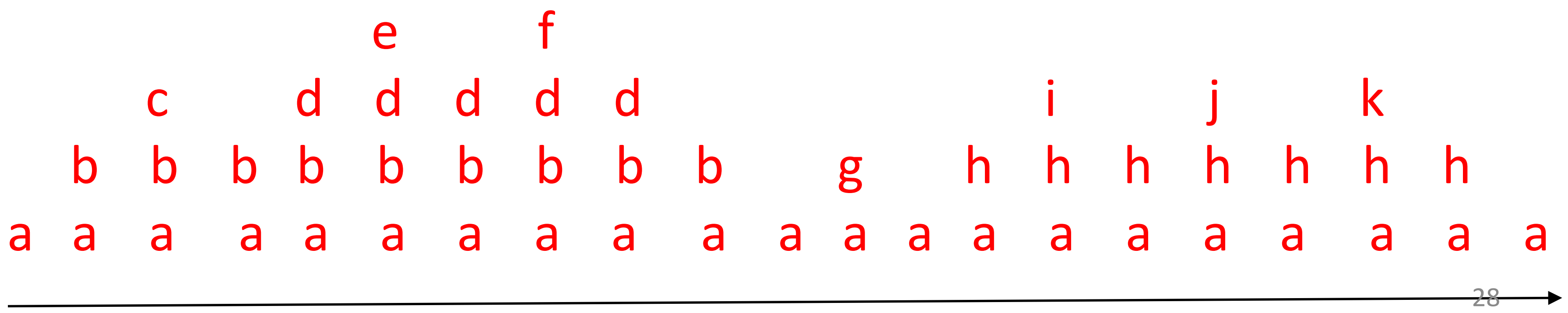


					e		f
		c		d	d	d	d
	b	b	b	b	b	b	b
a	a	a	a	a	a	a	a

Call stack for depthfirst(root)



Notation: the letters indicate call order of depthFirst(root)


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 a

Tree traversal

Recursive

- depth first (pre- versus post-order)

Non-Recursive

- using a stack
- using a queue

Assume algorithm has access to root of tree.

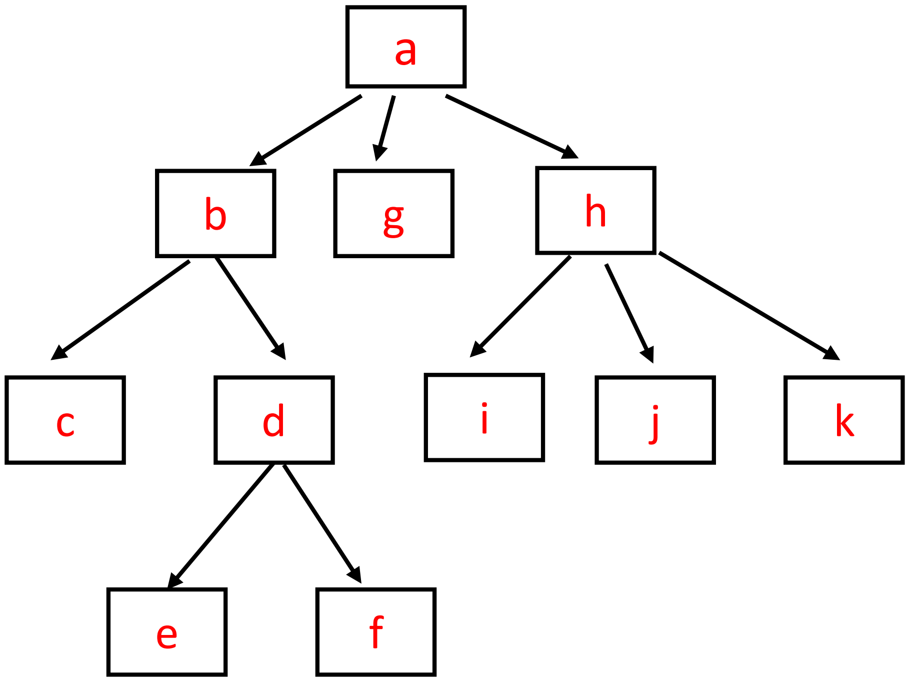
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)







}
```

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



    }
}
```

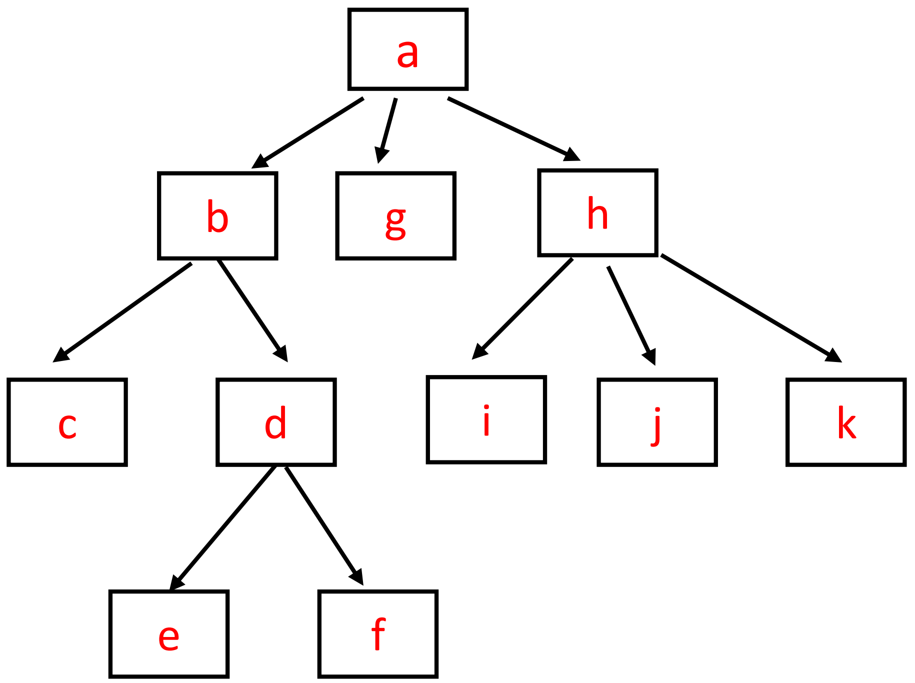
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```

What is the order of nodes visited ?

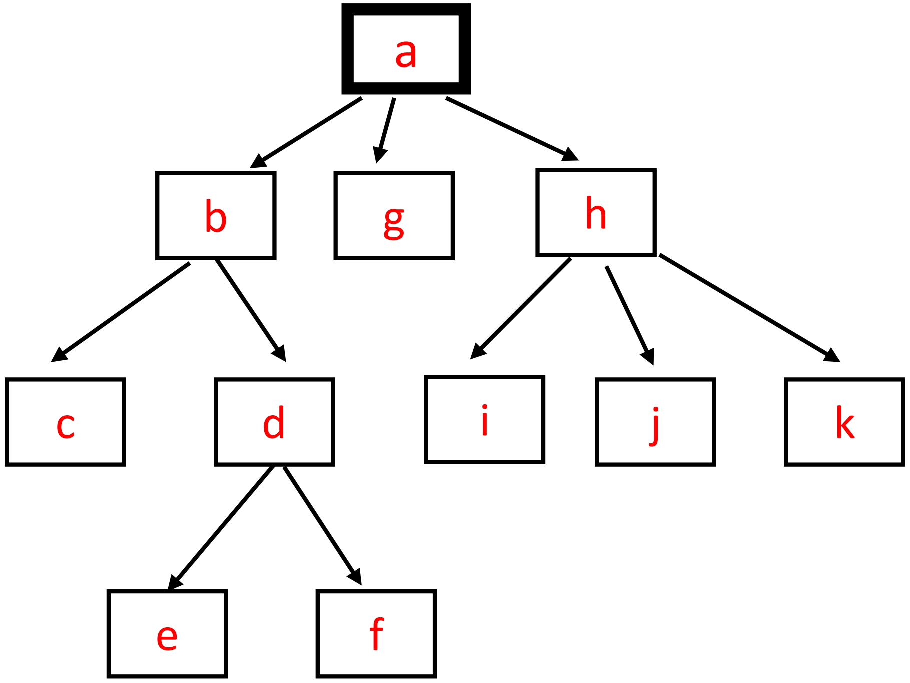
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```

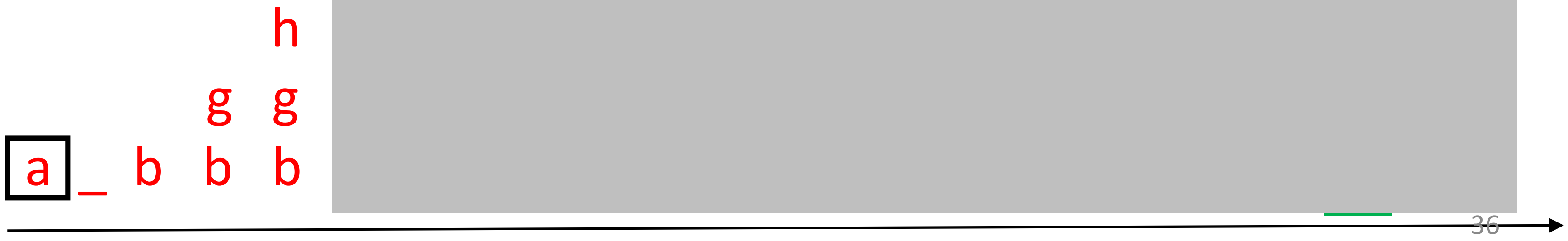


```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```

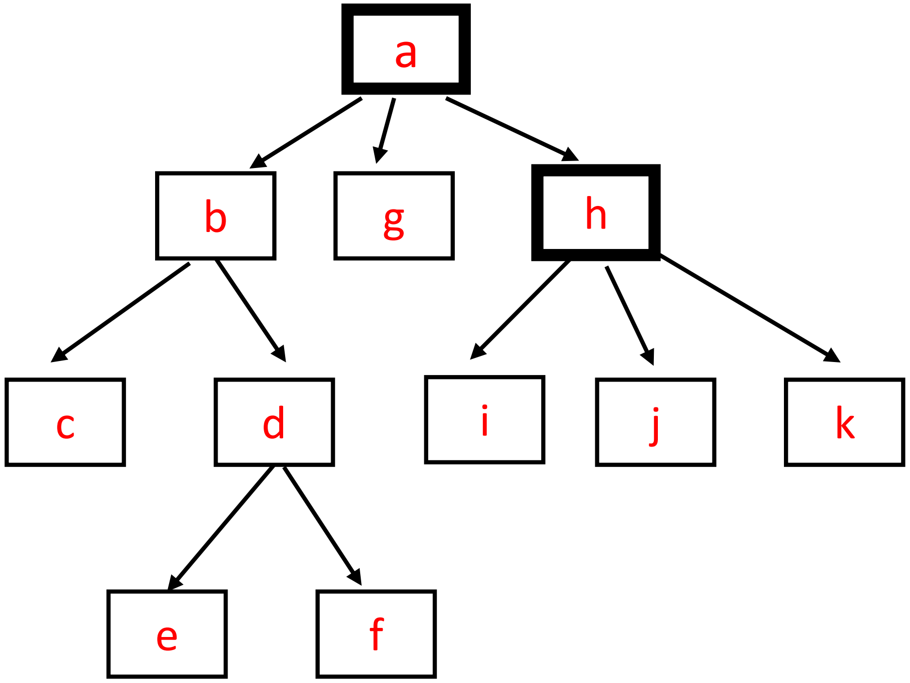


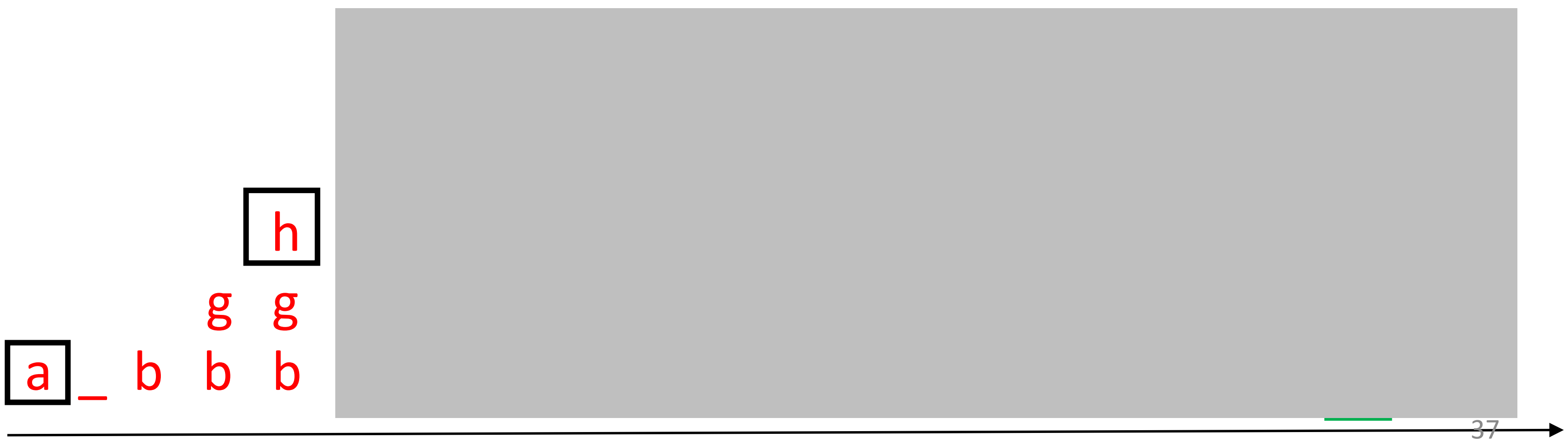
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```



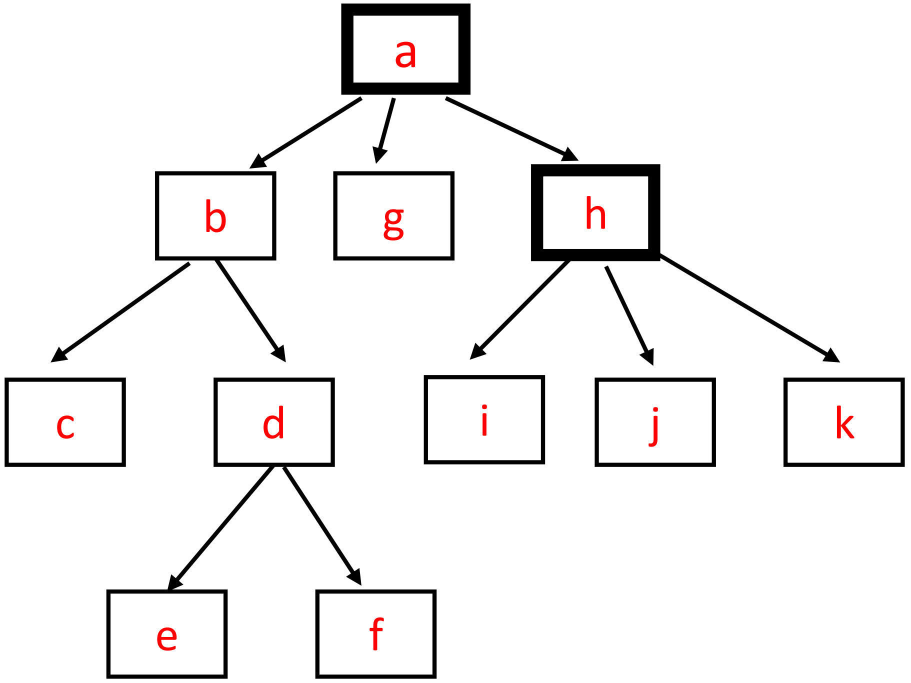


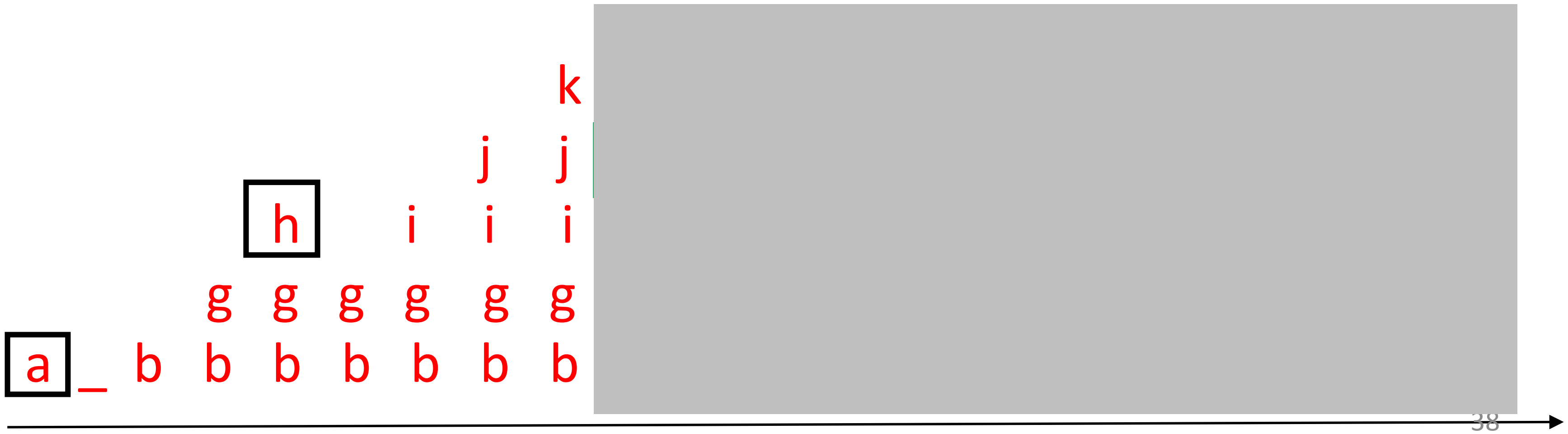
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```



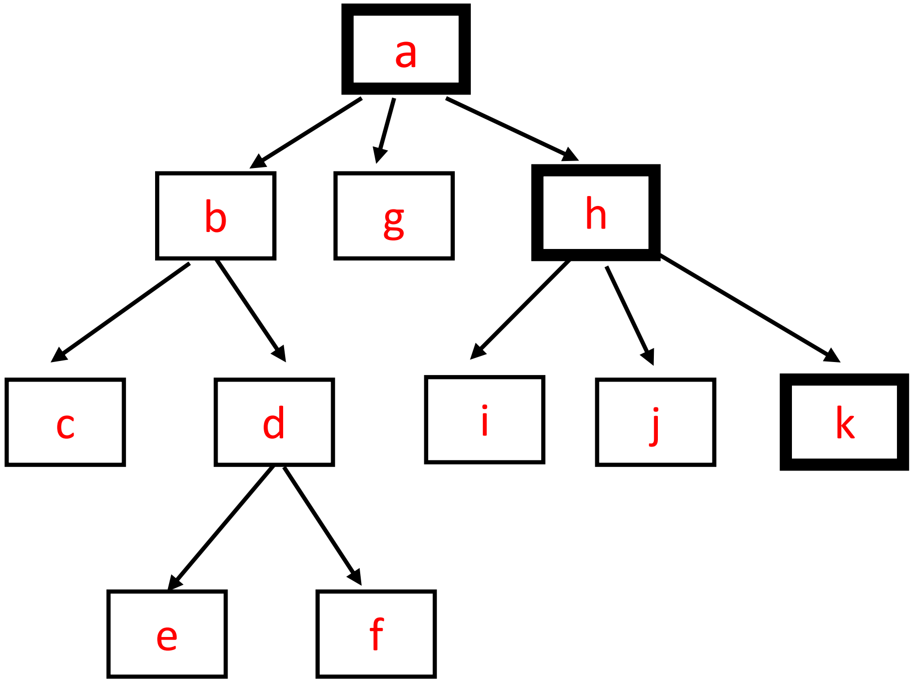


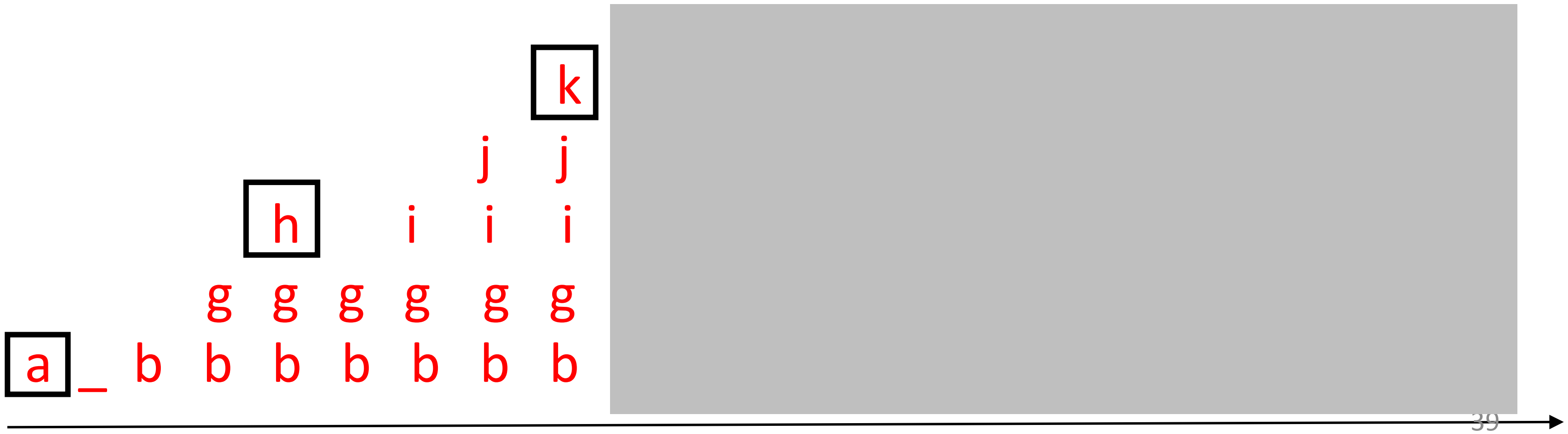
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```



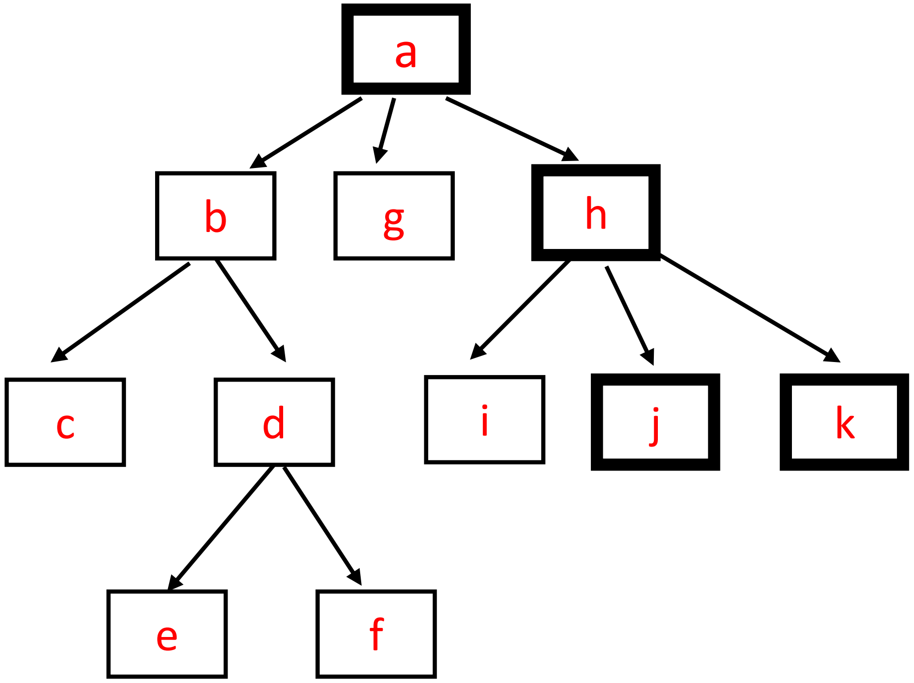


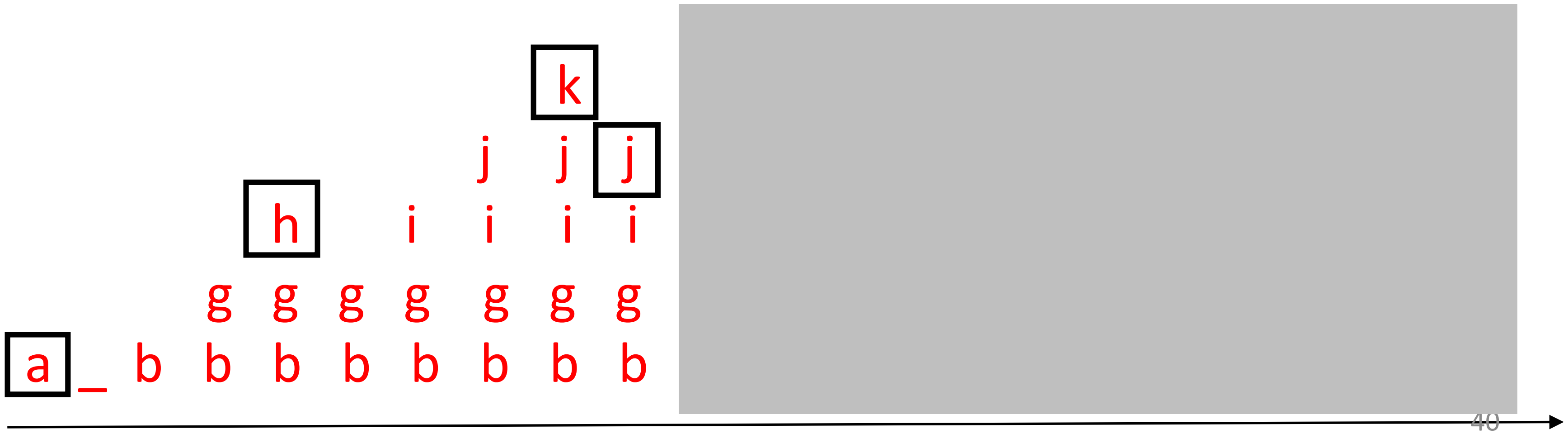
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```



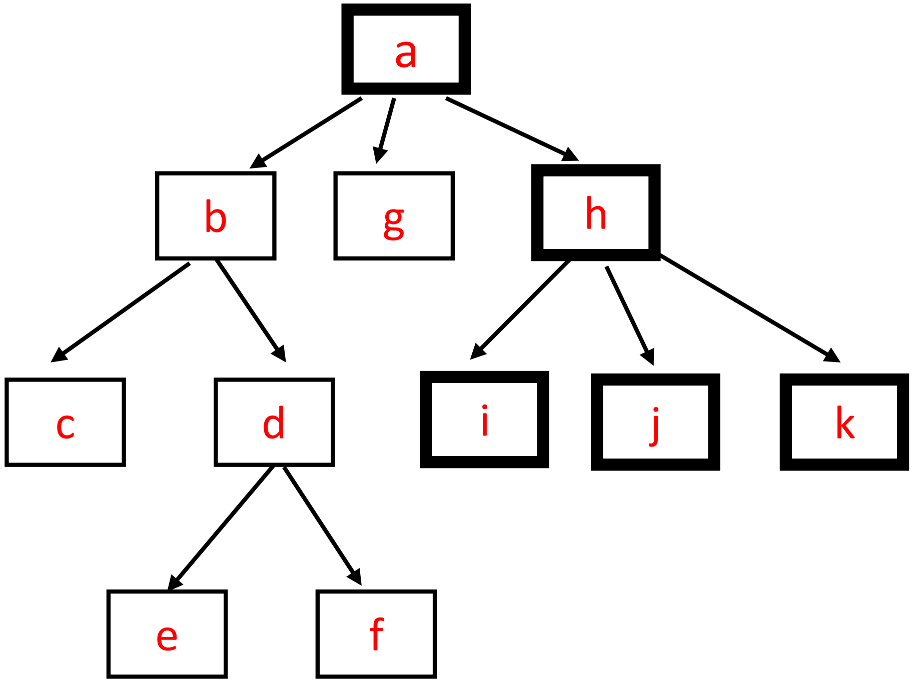


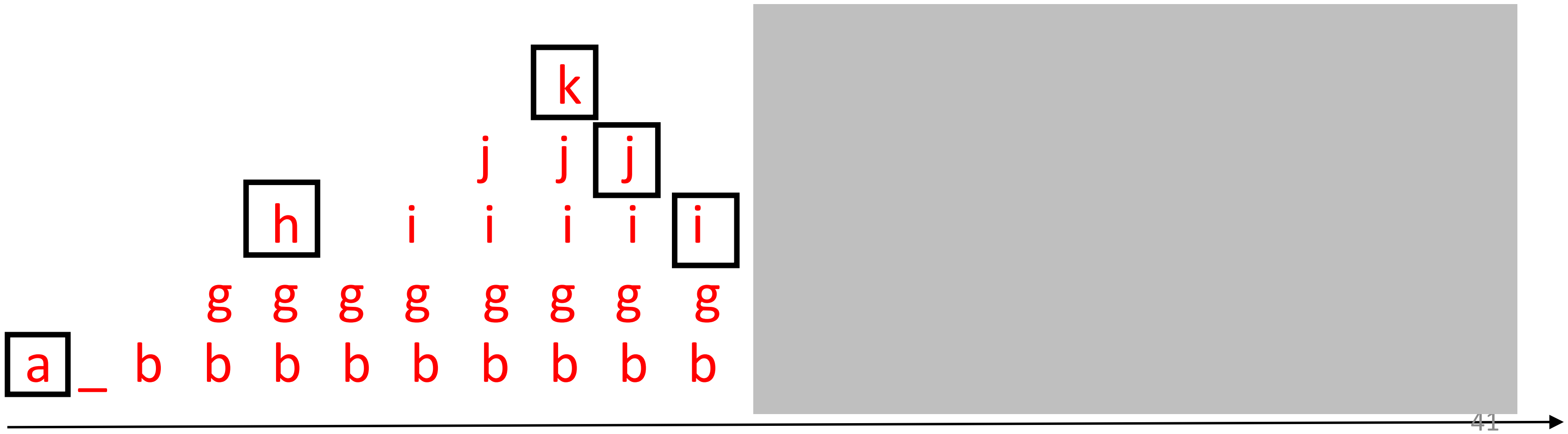
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```



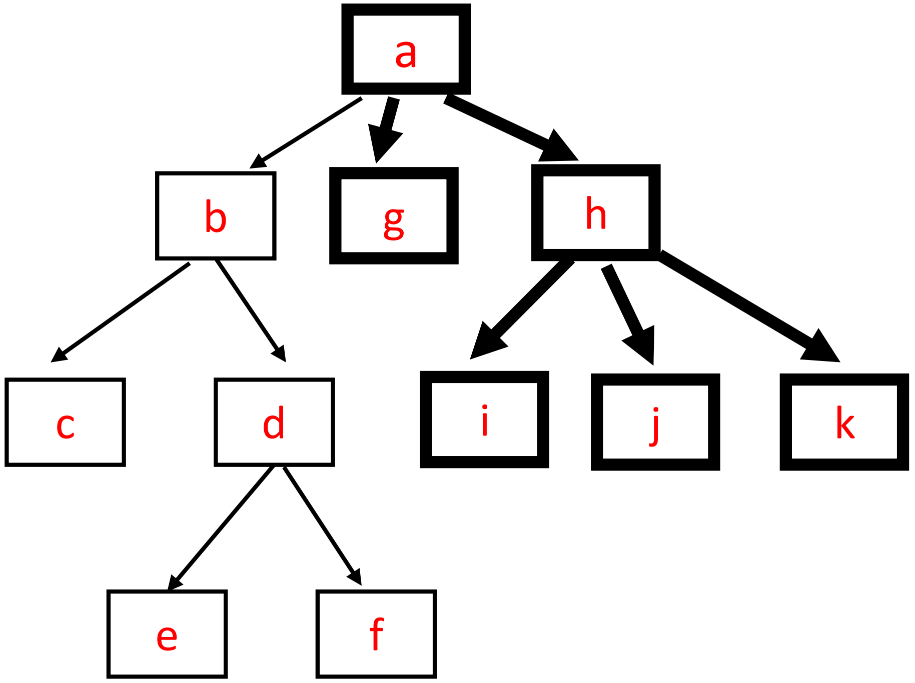


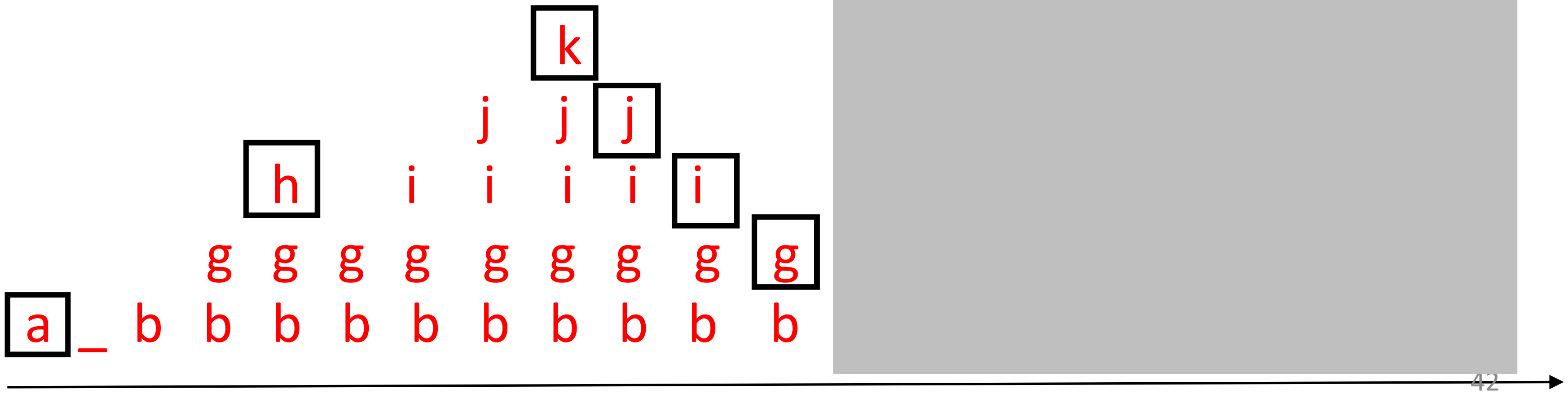
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```



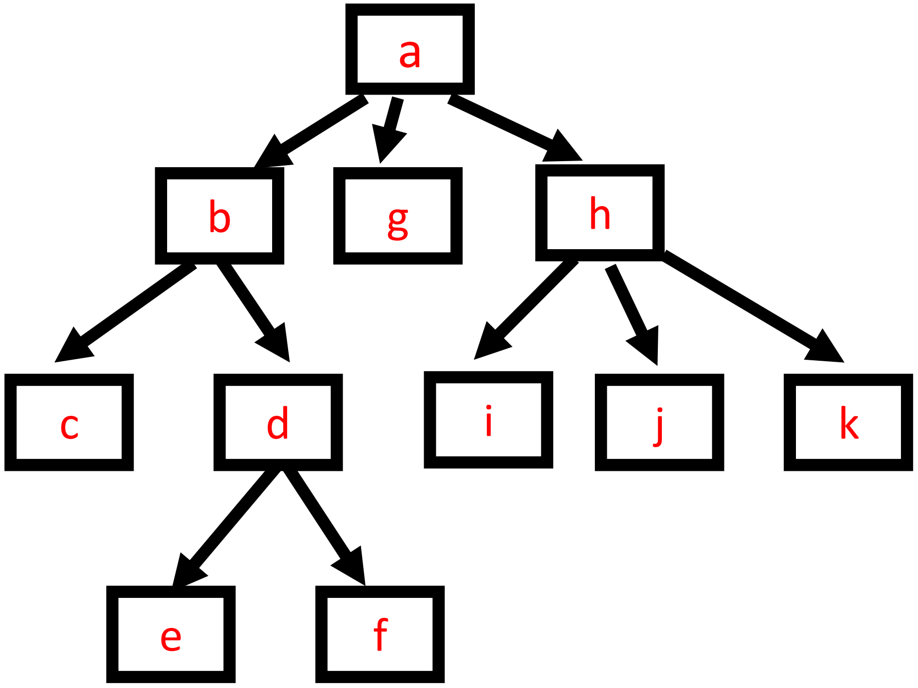


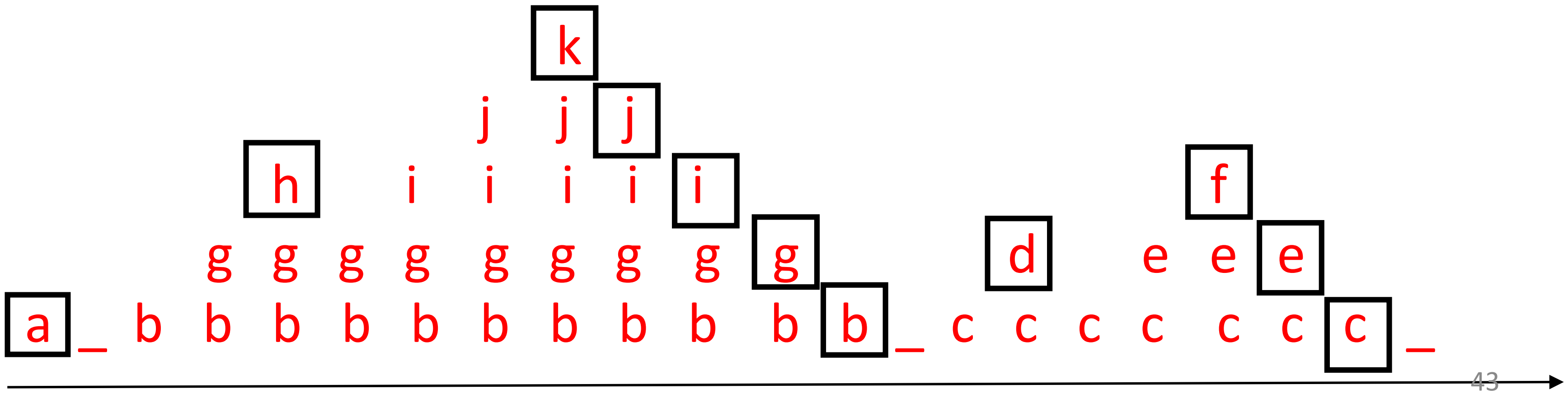
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```



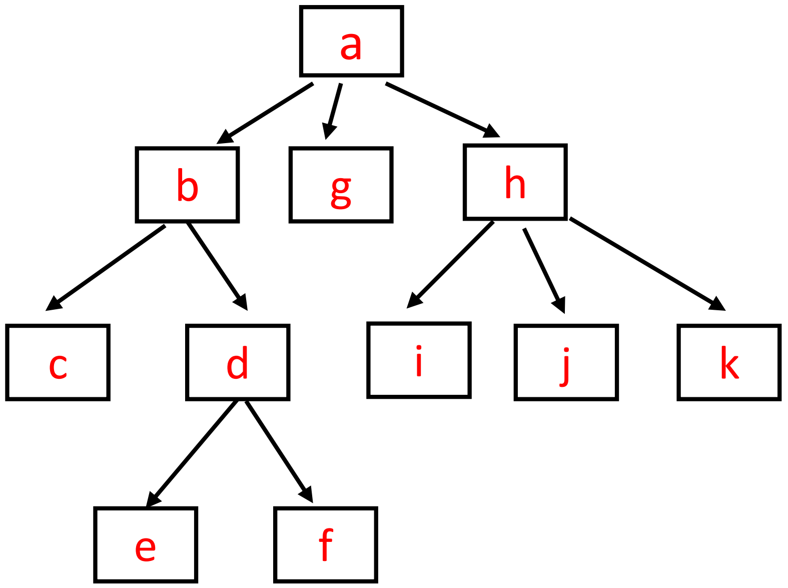


```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```





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



recursive preorder	abcdefghijk
recursive postorder	cefdbgijkha
non-recursive (stack)	ahkjigbdfec

```
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.

It is still pre-order.

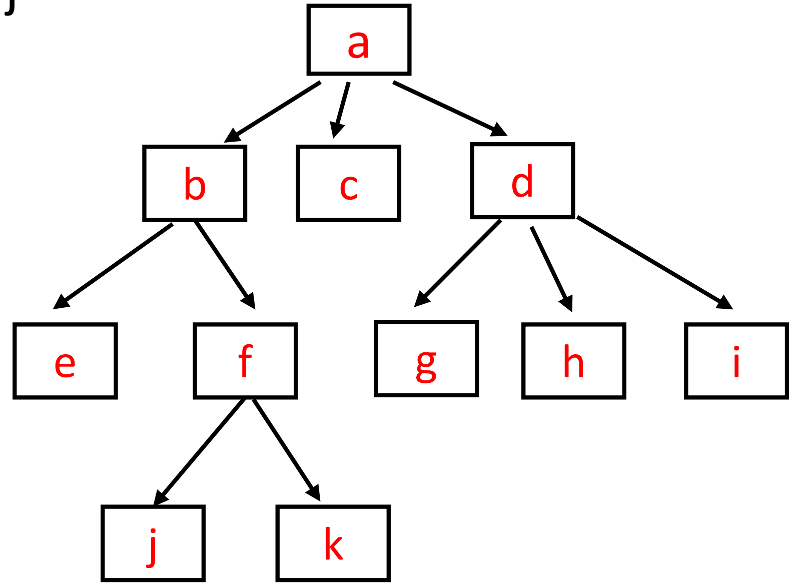
Why?

What if we use a queue instead?

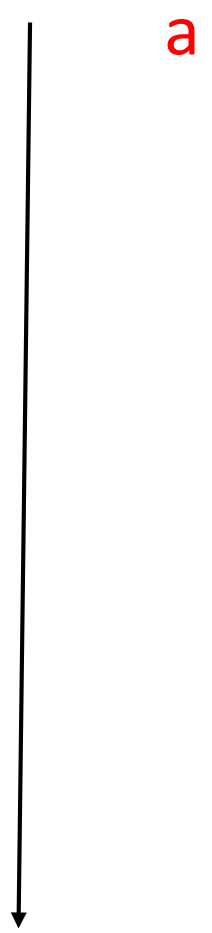
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```

```
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)
    }
}
```

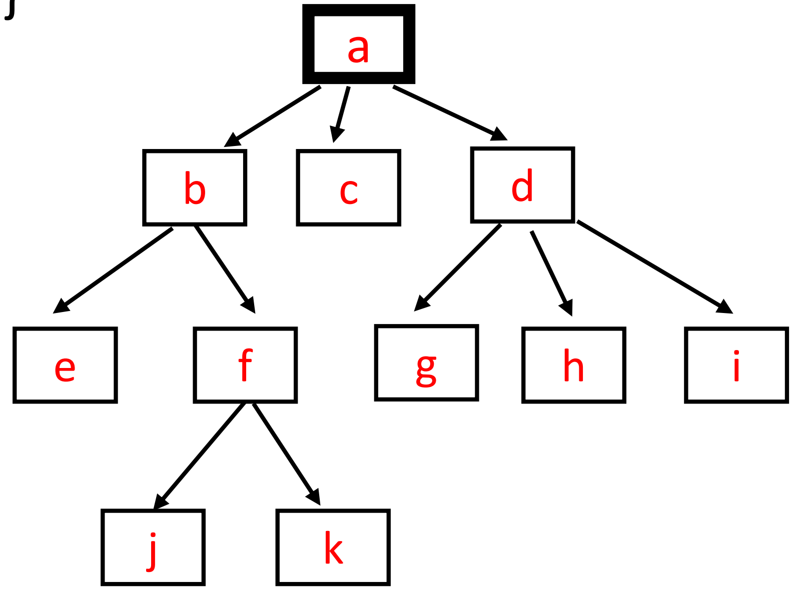
```
treeTraversalUsingQueue(root){
    initialize empty queue  q
    q.enqueue(root)
    while q is not empty {
       cur = q.dequeue()
       visit cur
       for each child of cur
            q.enqueue(child)
     }
}
```



Queue state at start of the while loop



```
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)
    }
}
```

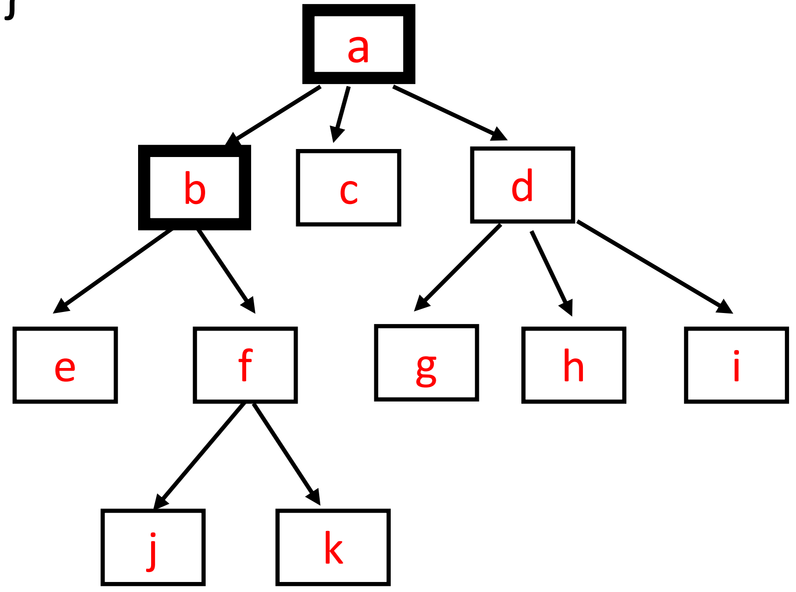


Queue state
at start of the
while loop

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)
    }
}
```



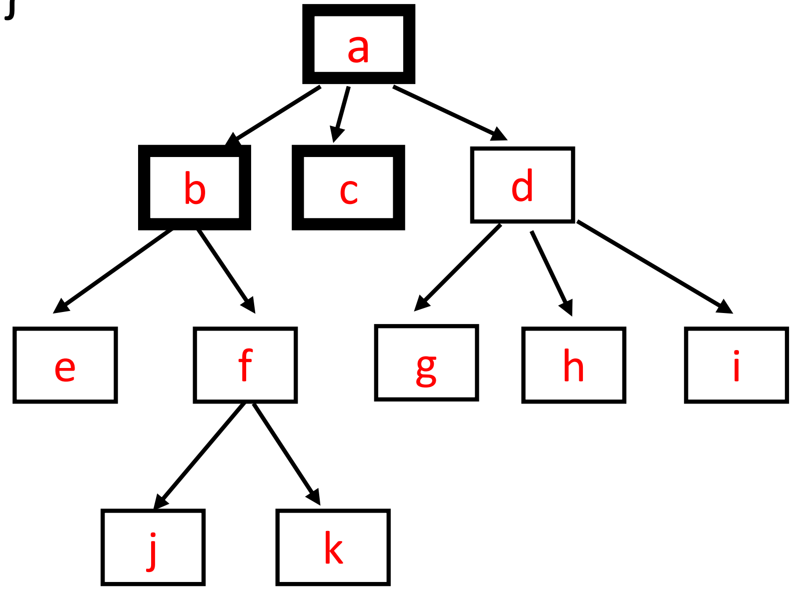
Queue state at start of the while loop

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

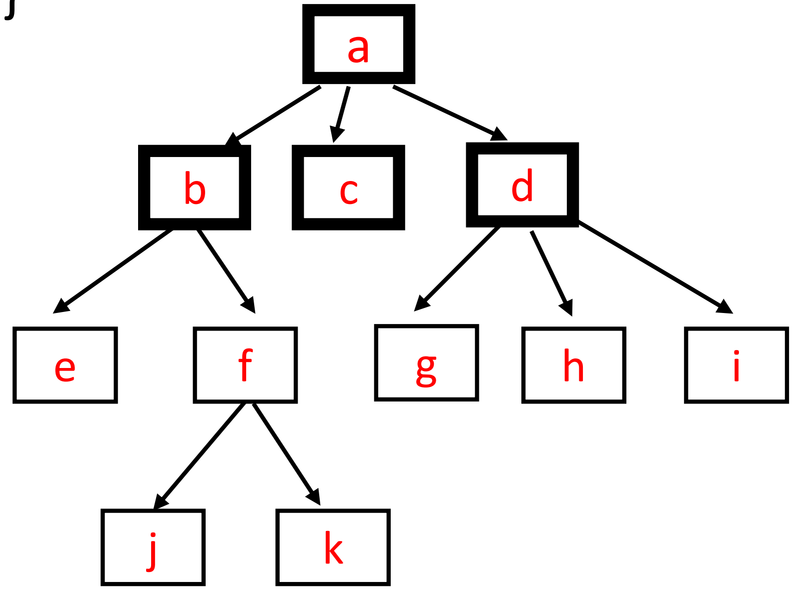
a

b c d

c d e f

d e f

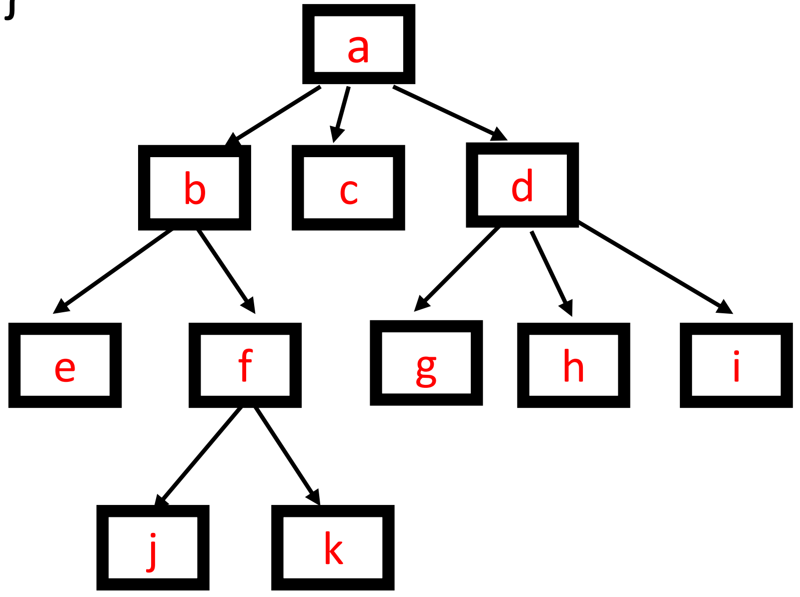
```
treeTraversalUsingQueue(root){
    initialize empty queue  q
    q.enqueue(root)
    while q is not empty {
       cur = q.dequeue()
       visit cur
       for each child of cur
            q.enqueue(child)
     }
}
```



Queue state
at start of the
while loop

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

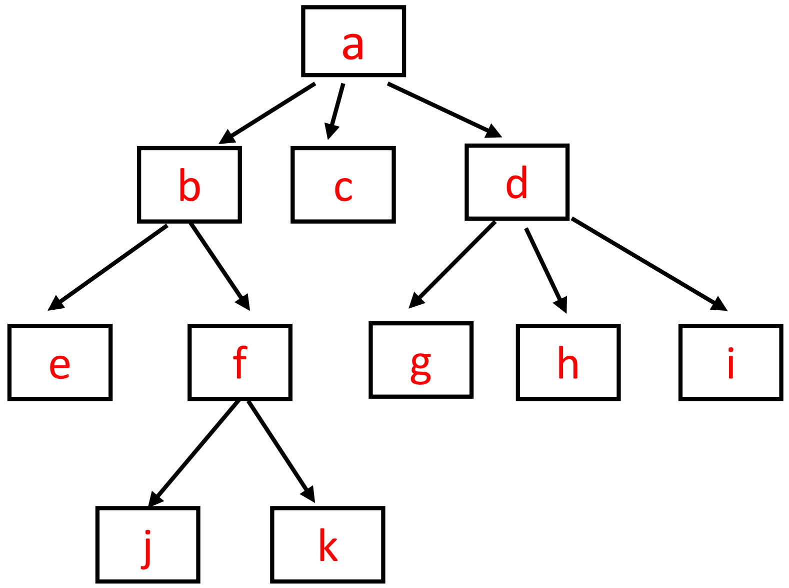
```
treeTraversalUsingQueue(root){
    initialize empty queue  q
    q.enqueue(root)
    while q is not empty {
       cur = q.dequeue()
       visit cur
       for each child of cur
            q.enqueue(child)
     }
}
```



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