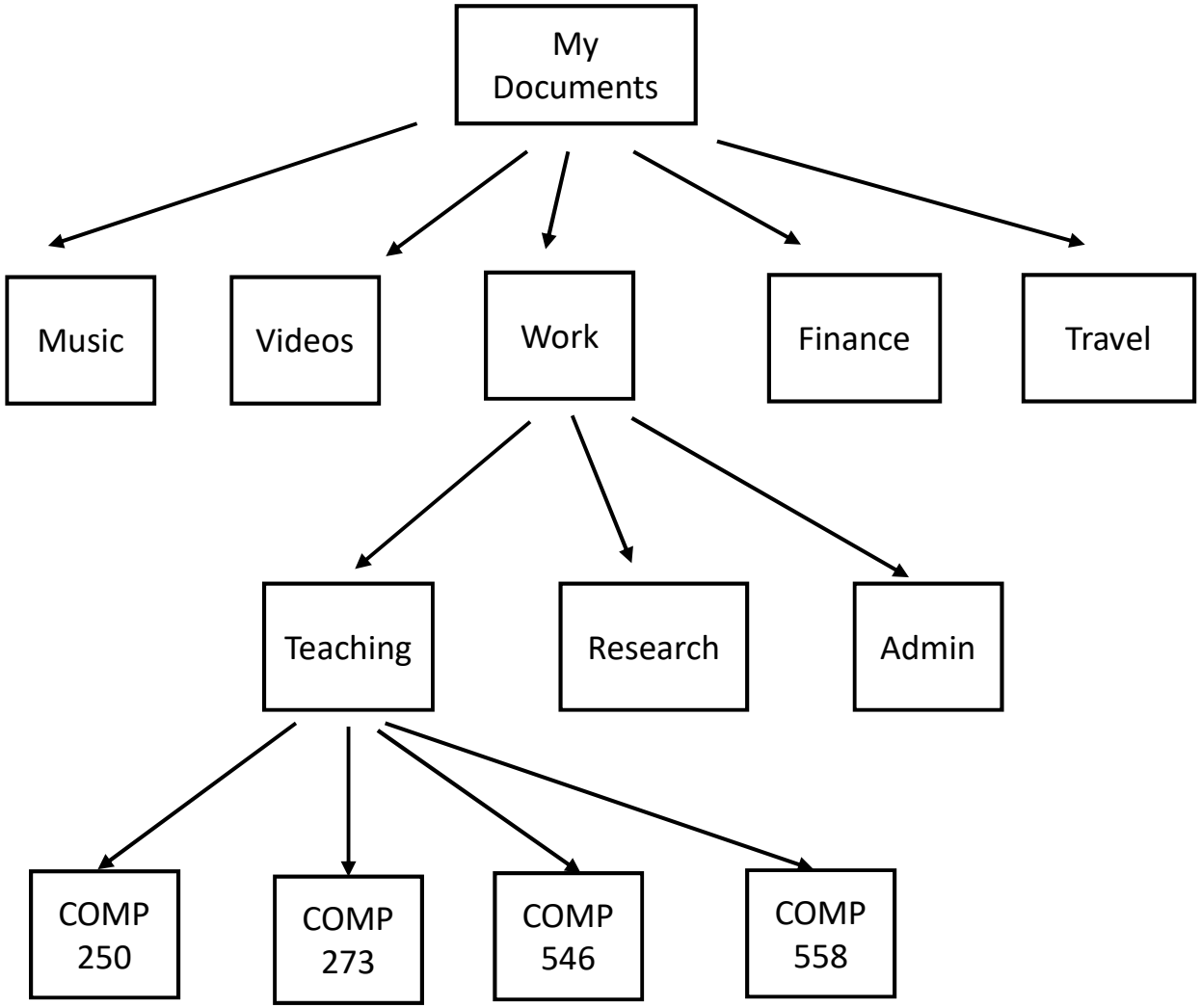
COMP 250

Lecture 24

tree traversal

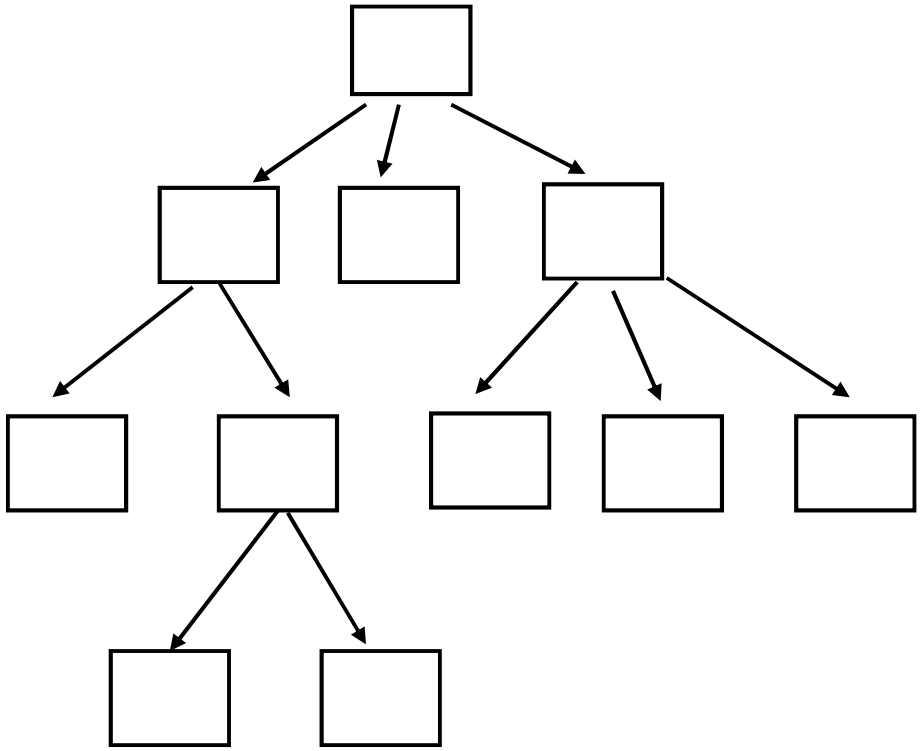
March 9, 2022

Tree: Example



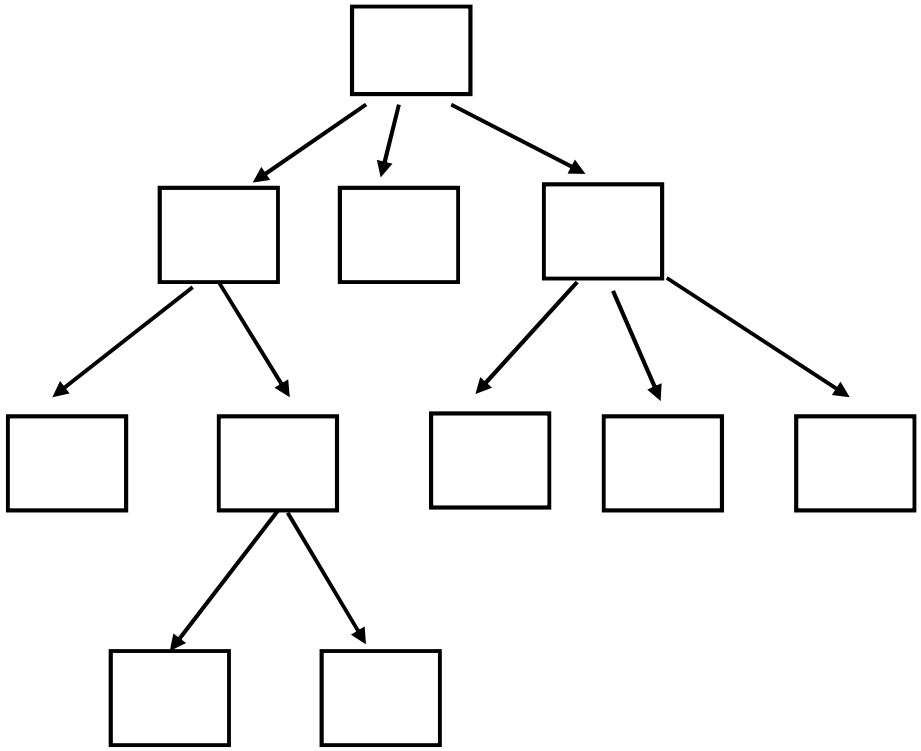
Tree Traversal

How to “visit” / “traverse” / “iterate through” the nodes in a tree ?



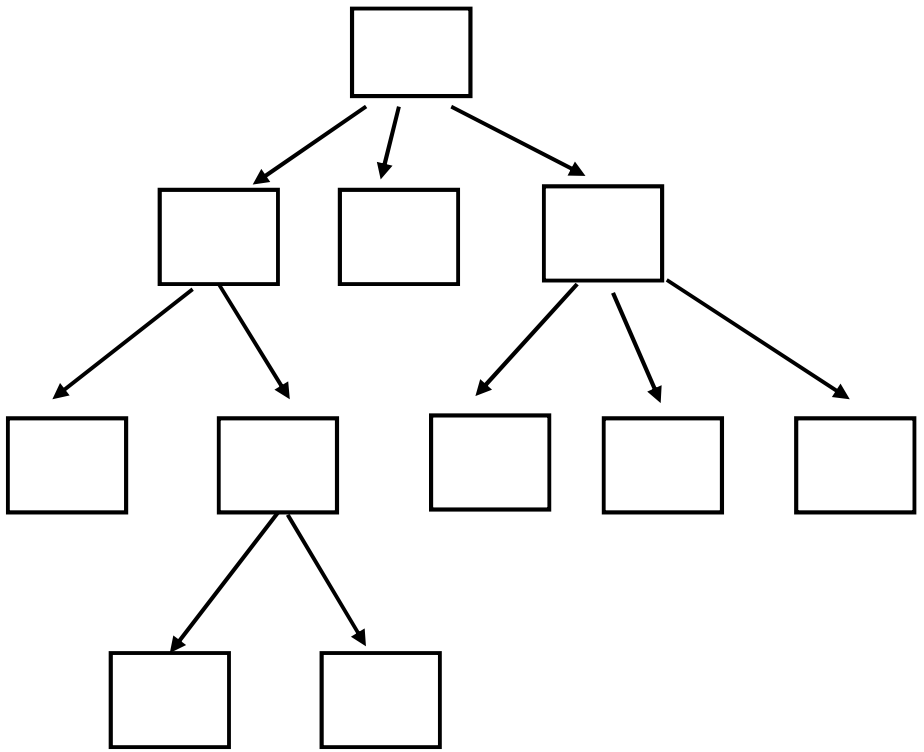
Tree Traversal

How to “visit” / “traverse” / “iterate through” the nodes in a tree ?



Recall – depth of a node

The depth or level of a node is the length of the path *from the root to the node*.



Tree Traversal

- Depth first traversal (also called depth first *search*)

  Start from root and follow paths to the leaves, backtracking only when a leaf is found

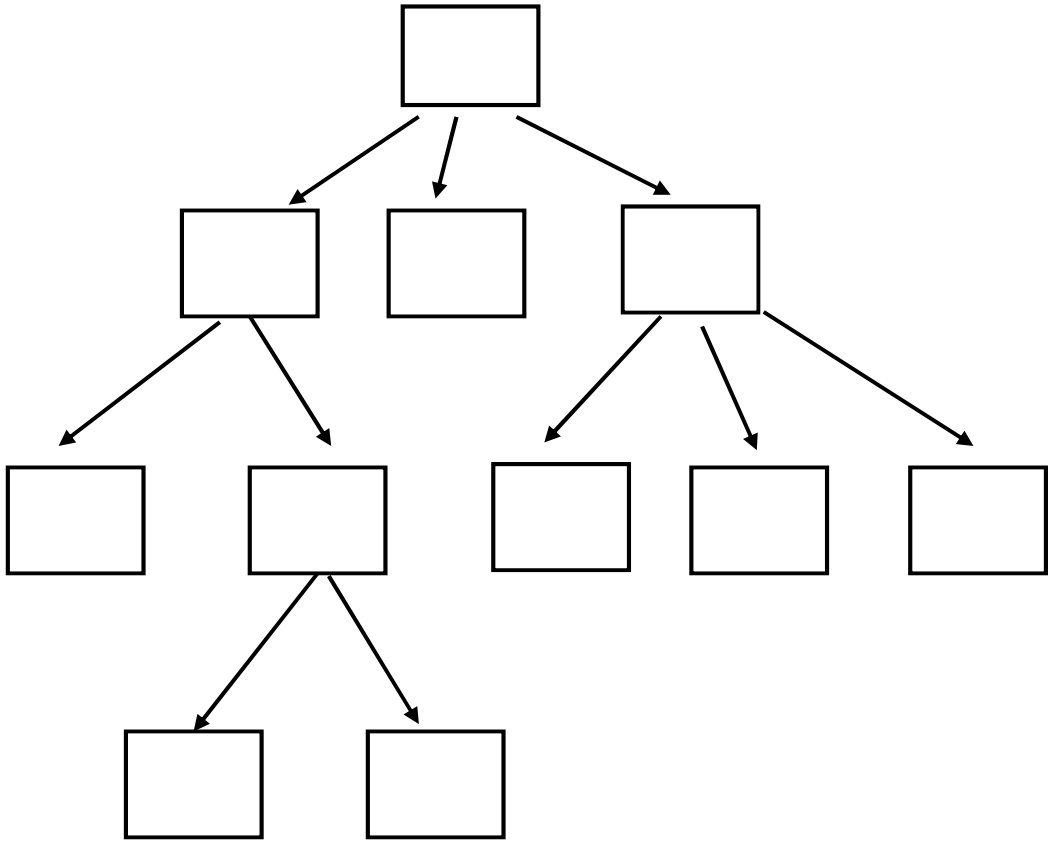
- Breadth first traversal (also called breadth first *search*)

  Start from root and visit all nodes at depth k before any nodes at depth k+1.

"Visit" a node implies that you do something at that node.

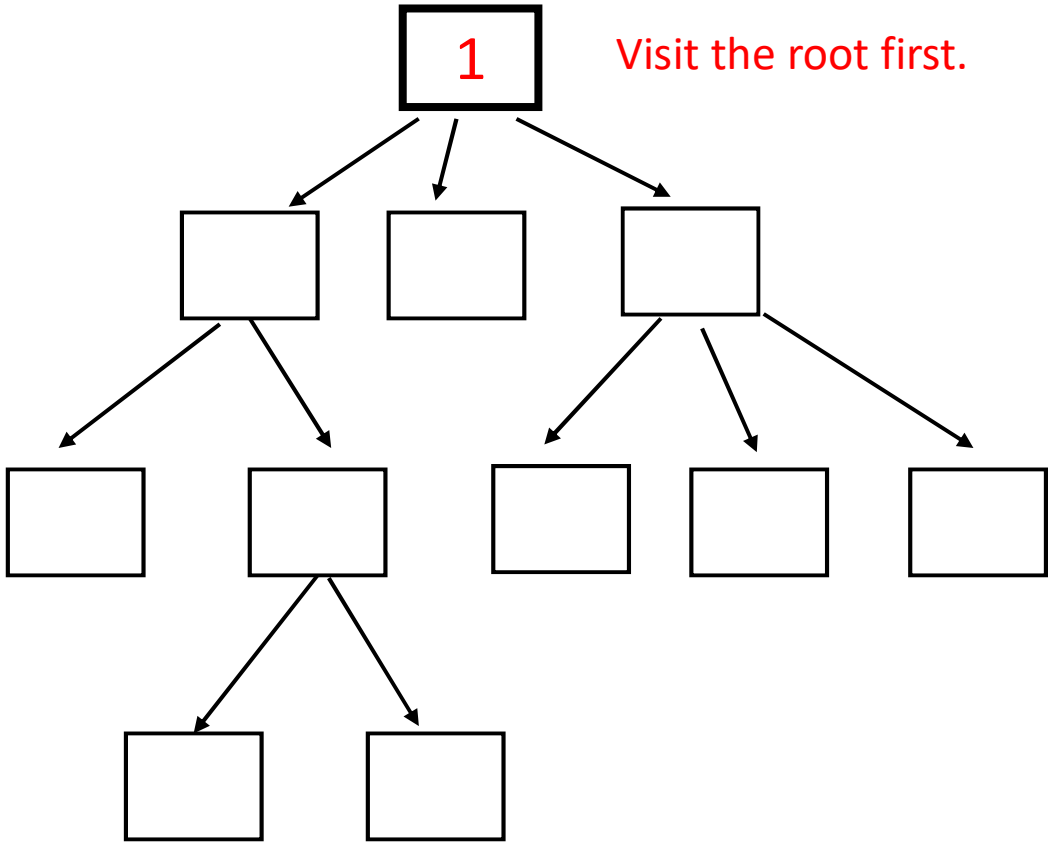
We will see some examples later.

```
depthfirst (root){
    visit root
    for each child of root
        depthfirst( child )
}
```



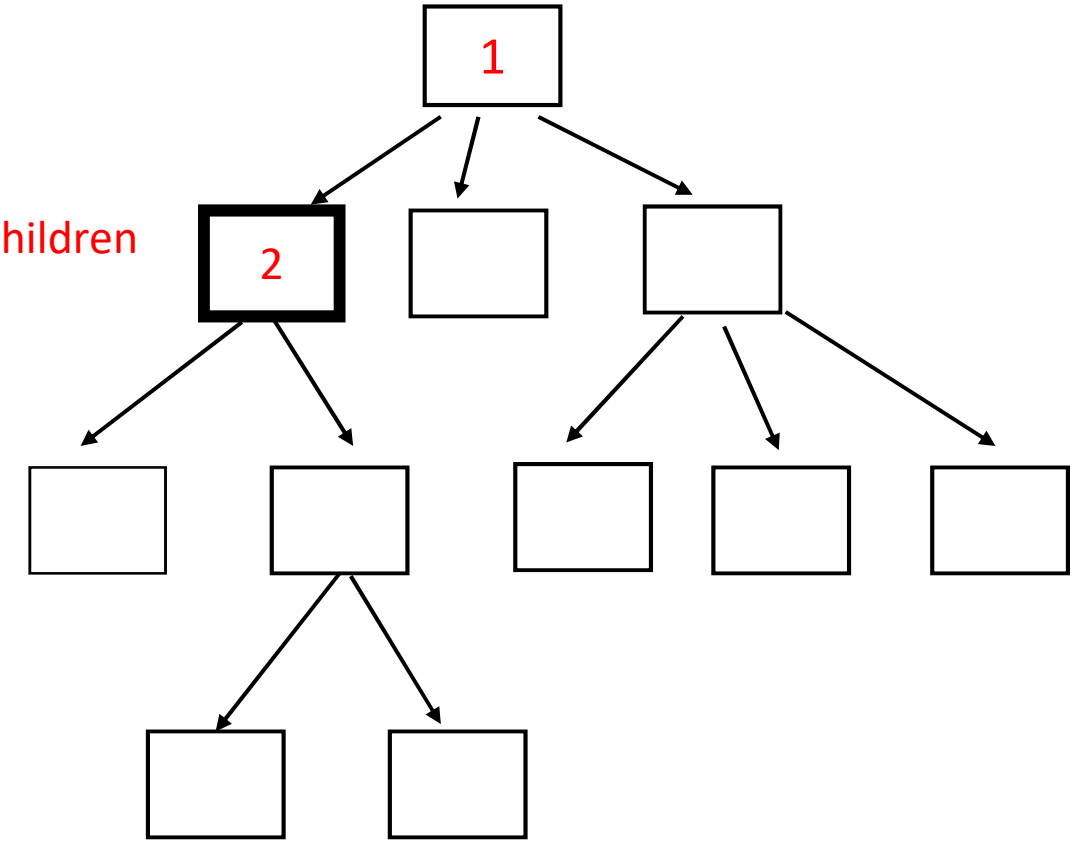
```
depthfirst (root){
    visit root
    for each child of root
        depthfirst( child )
}
```

"preorder" traversal: visit the root before the children



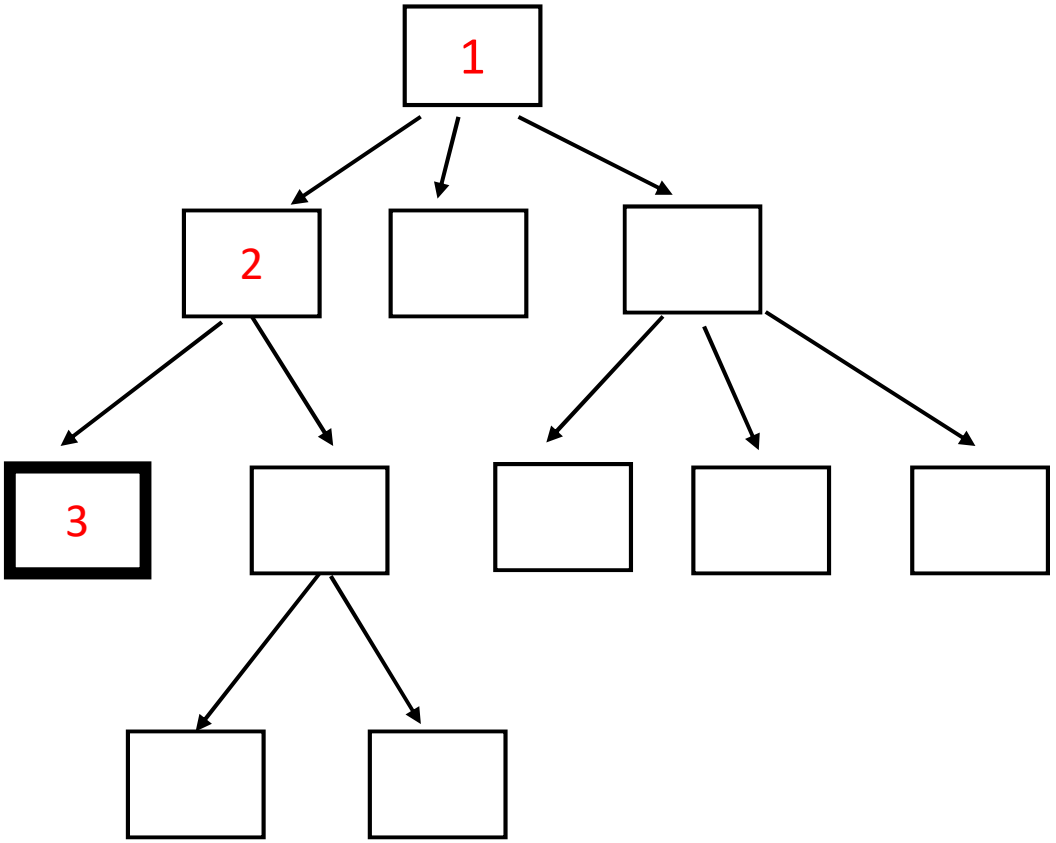
```
depthfirst (root){
    visit root
    for each child of root
        depthfirst( child )
}
```

Assume we visit children from left to right.

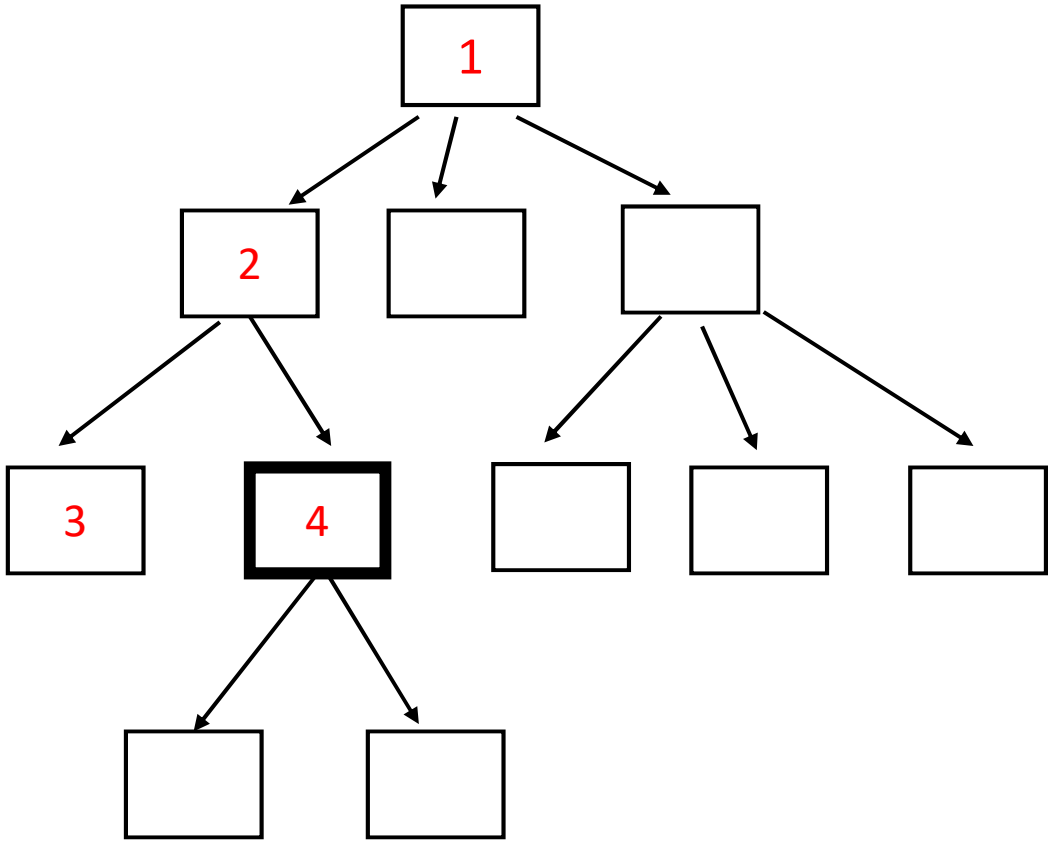


```
depthfirst (root){
    visit root
    for each child of root
        depthfirst( child )
}
```

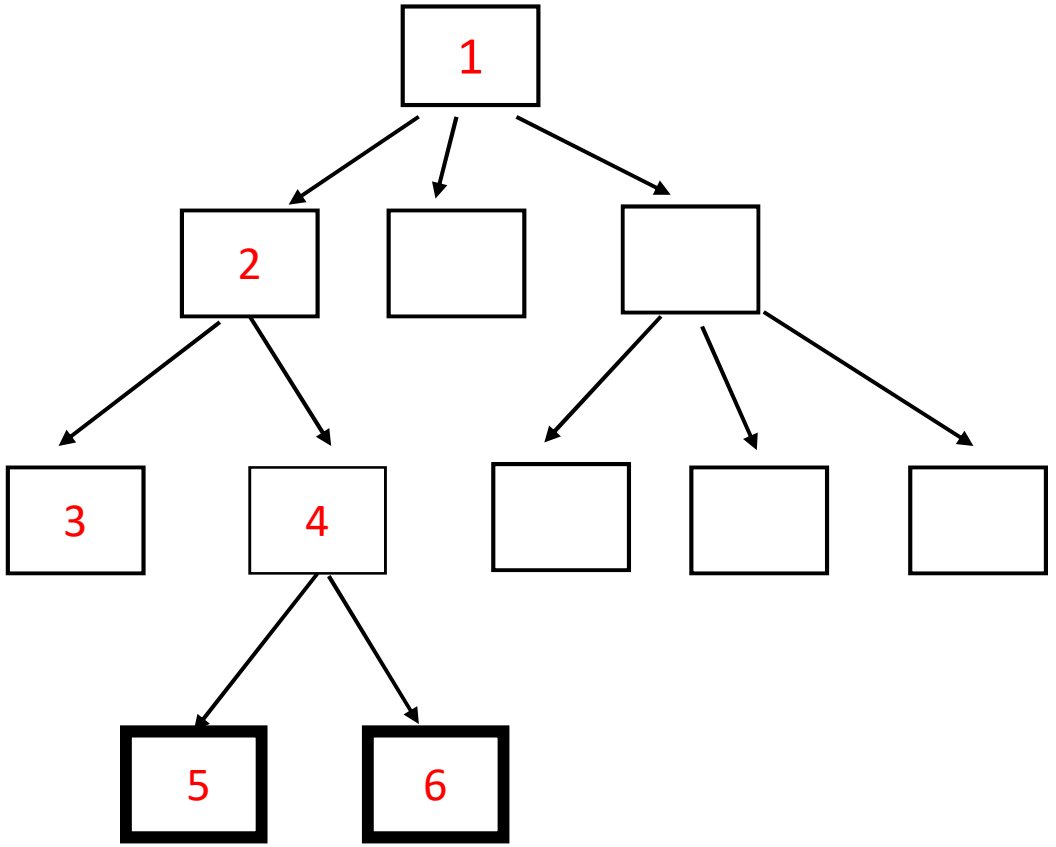
This node has no children.



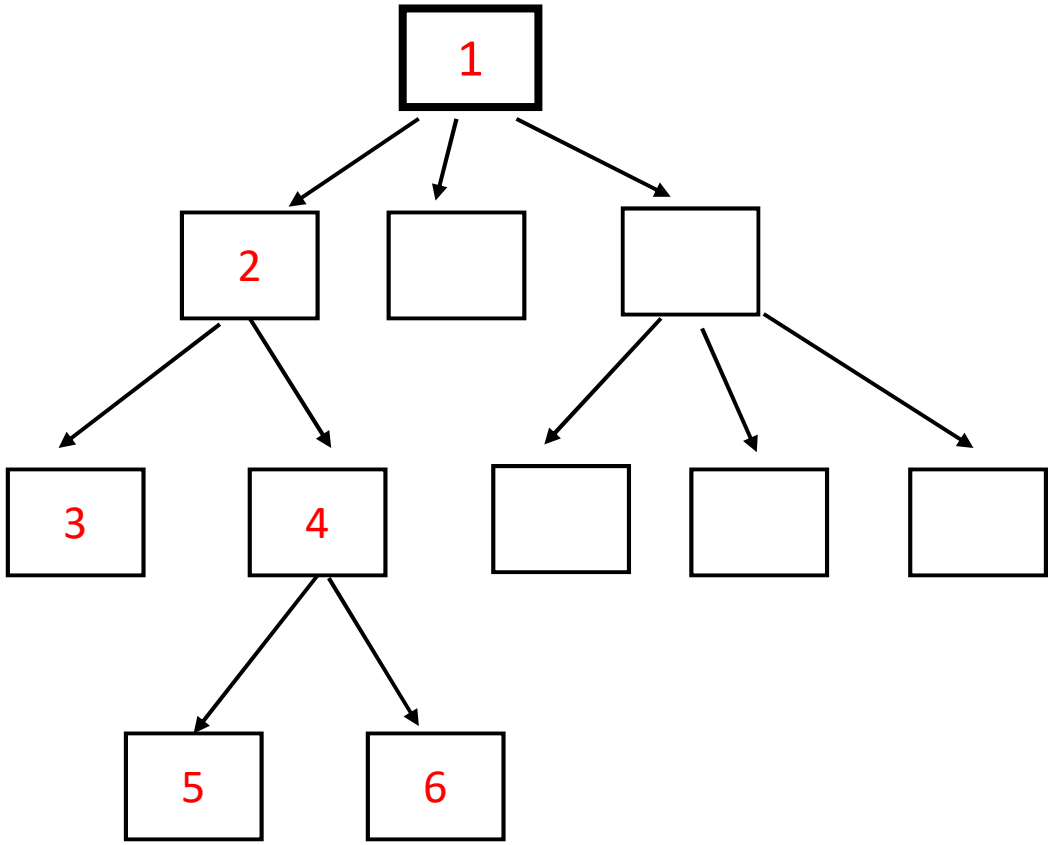
```
depthfirst (root){
    visit root
    for each child of root
        depthfirst( child )
}
```



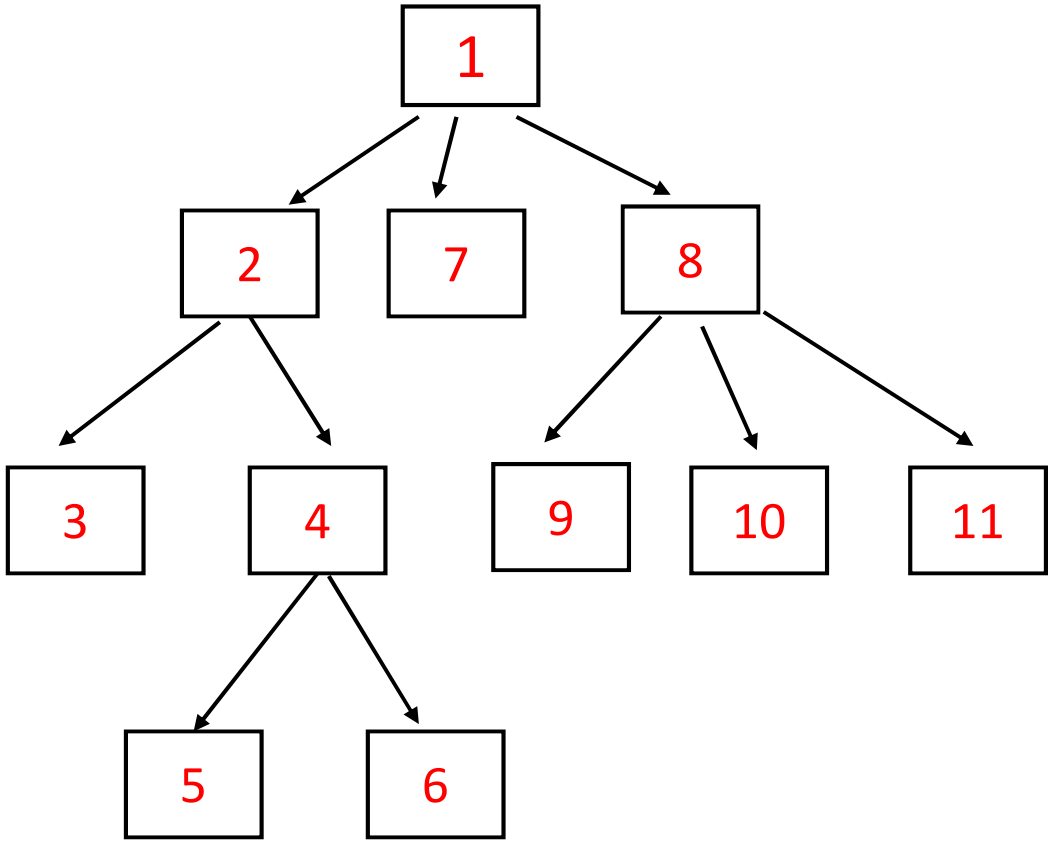
```
depthfirst (root){
    visit root
    for each child of root
        depthfirst( child )
}
```



```
depthfirst (root){
    visit root
    for each child of root
        depthfirst( child )
}
```



```
depthfirst (root){
    visit root
    for each child of root
        depthfirst( child )
}
```

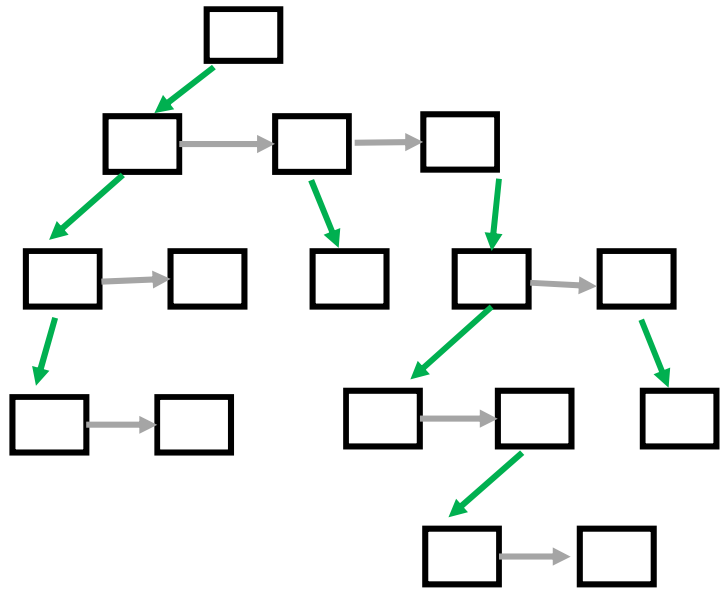


Implementation details

Recall the "first child, next sibling" implementation

Then when we write

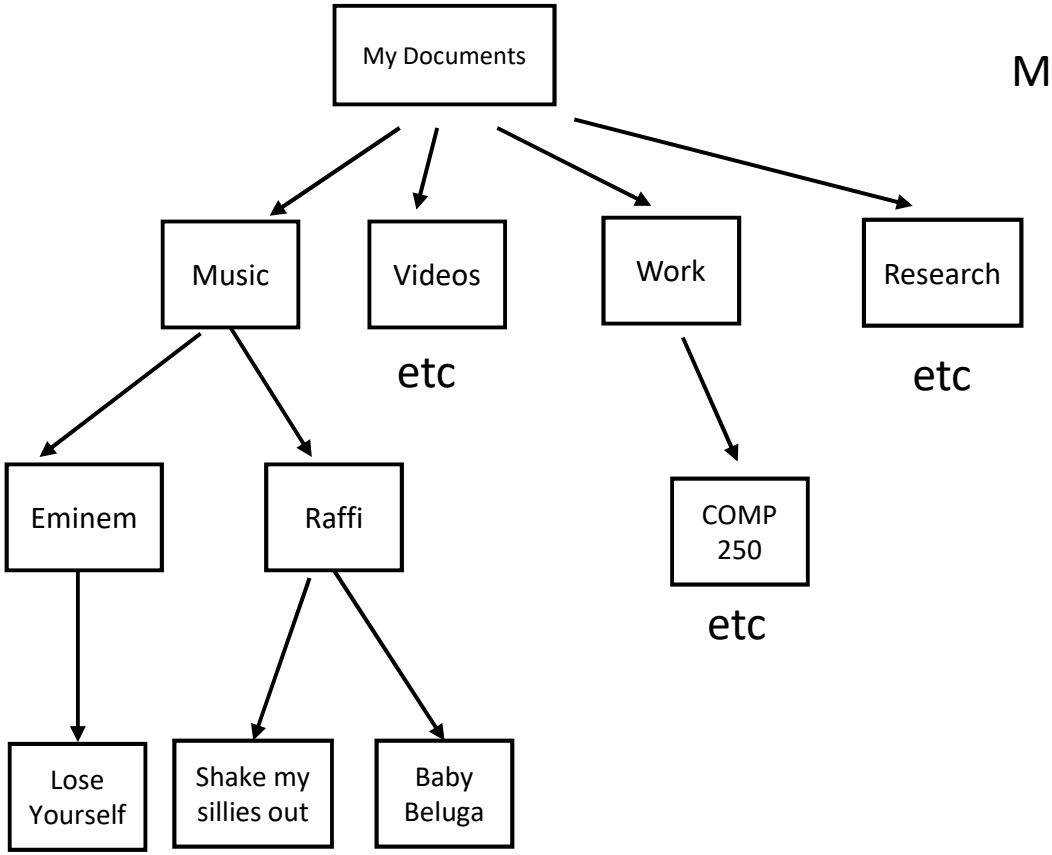
```
for each child of root {
        :
}
```



we would mean:

```
child = root.firstChild
while  (child !=null) {
        :
        child = child.nextSibling
}
```

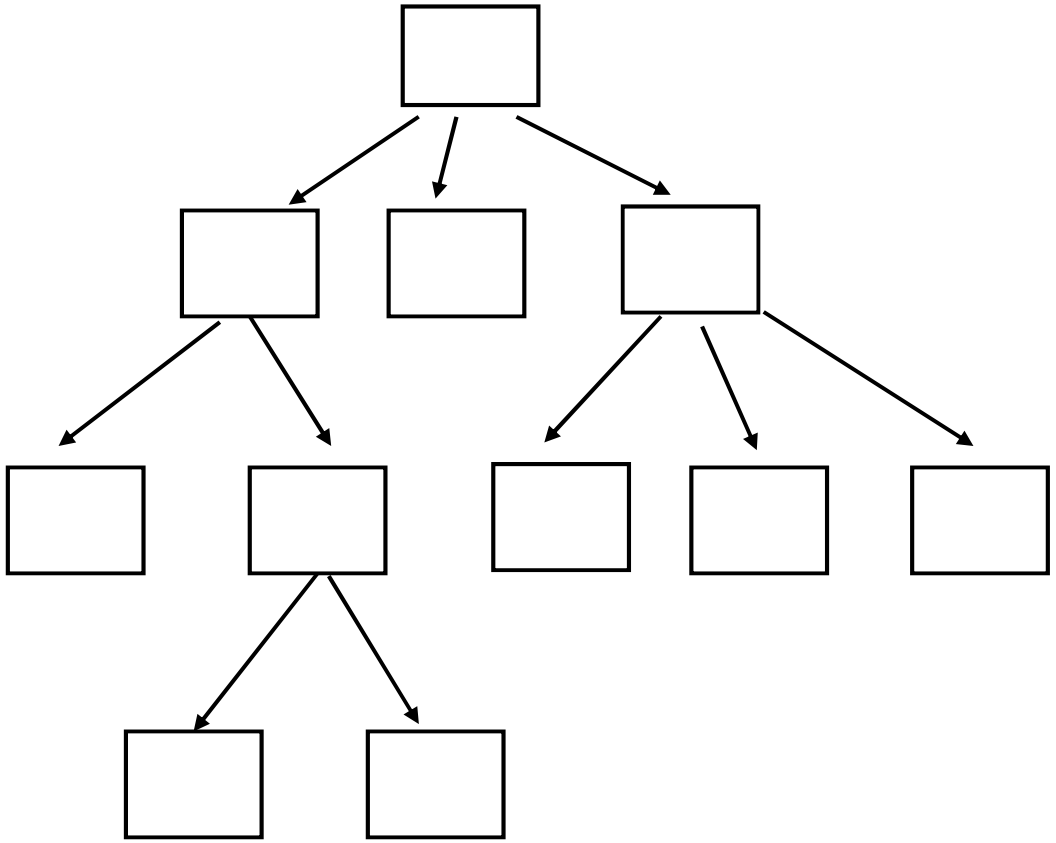
Example of Preorder Traversal:
printing a hierarchical file system
(visit = print directory or file name)



My 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)
  - :

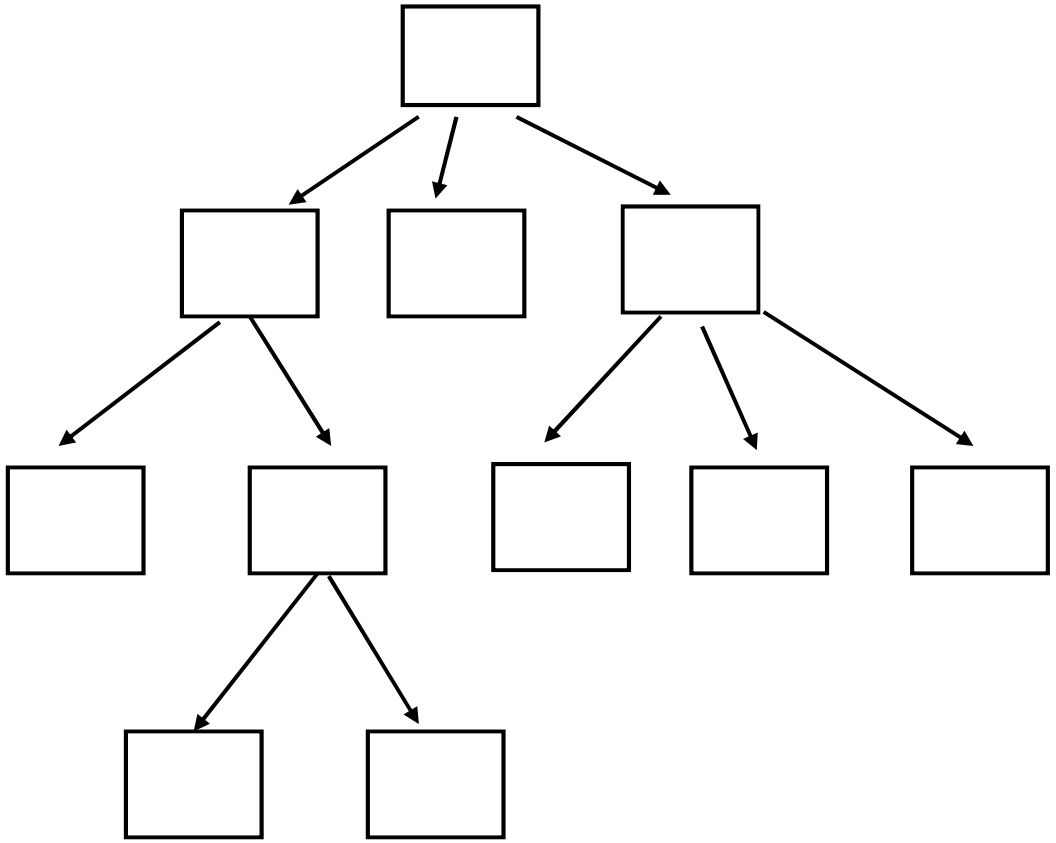
```
depthfirst (root){
    for each child of root
        depthfirst( child )
    visit root
}
```

"postorder" traversal: visit the root *after* the children



```
depthfirst (root){
    for each child of root
        depthfirst( child )
    visit root
}
```

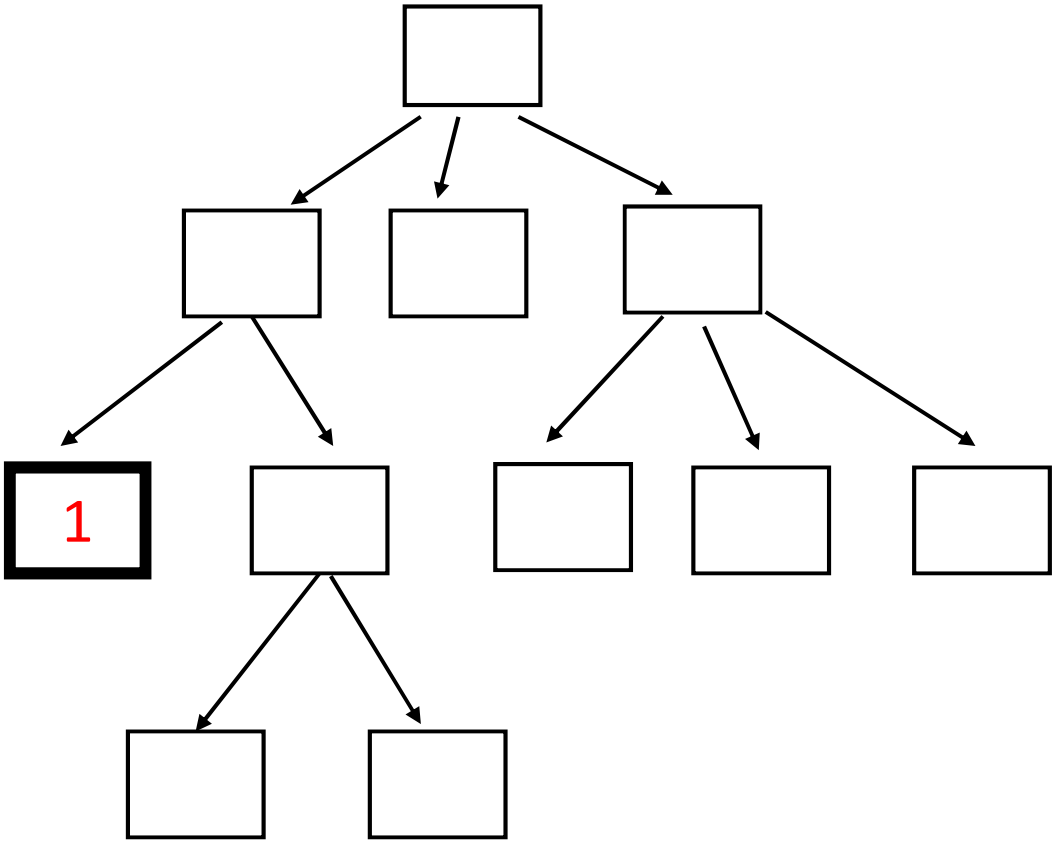
"postorder" traversal: visit the root after the children



Q: Which node is visited first?

```
depthfirst (root){
    for each child of root
        depthfirst( child )
    visit root
}
```

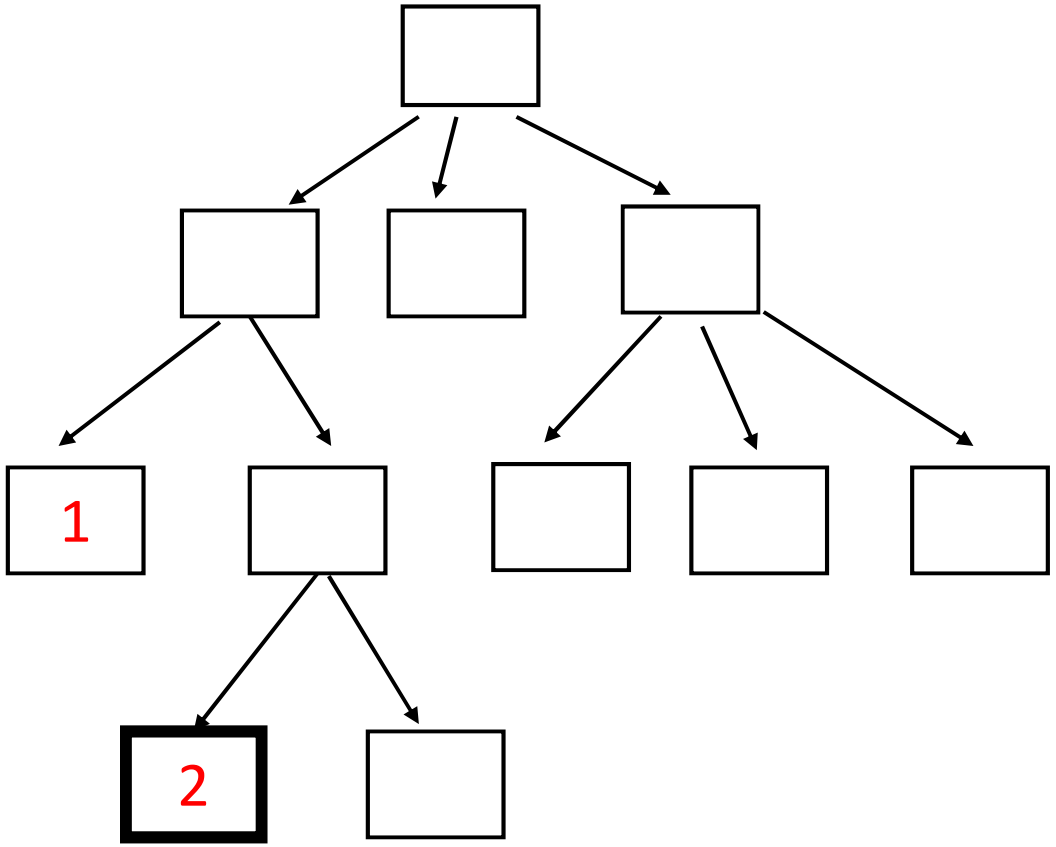
"postorder" traversal: visit the root after the children



Q: Which node is visited second?

```
depthfirst (root){
    for each child of root
        depthfirst( child )
    visit root
}
```

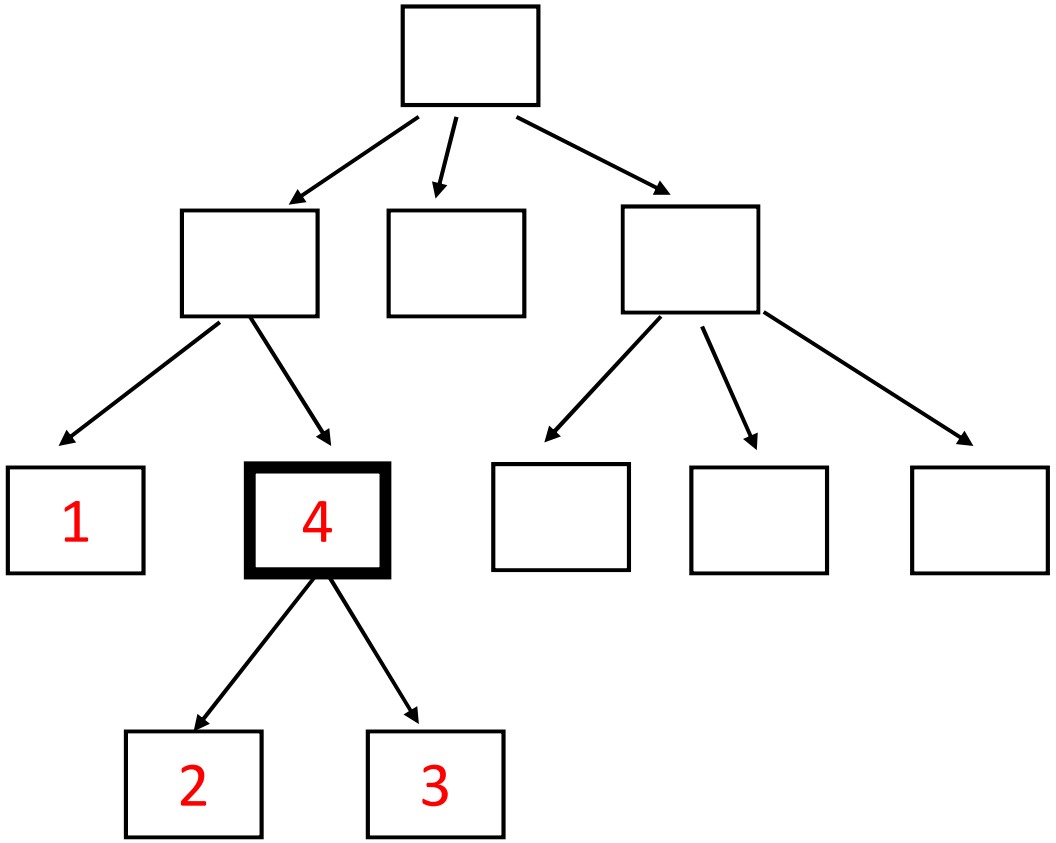
"postorder" traversal: visit the root after the children



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

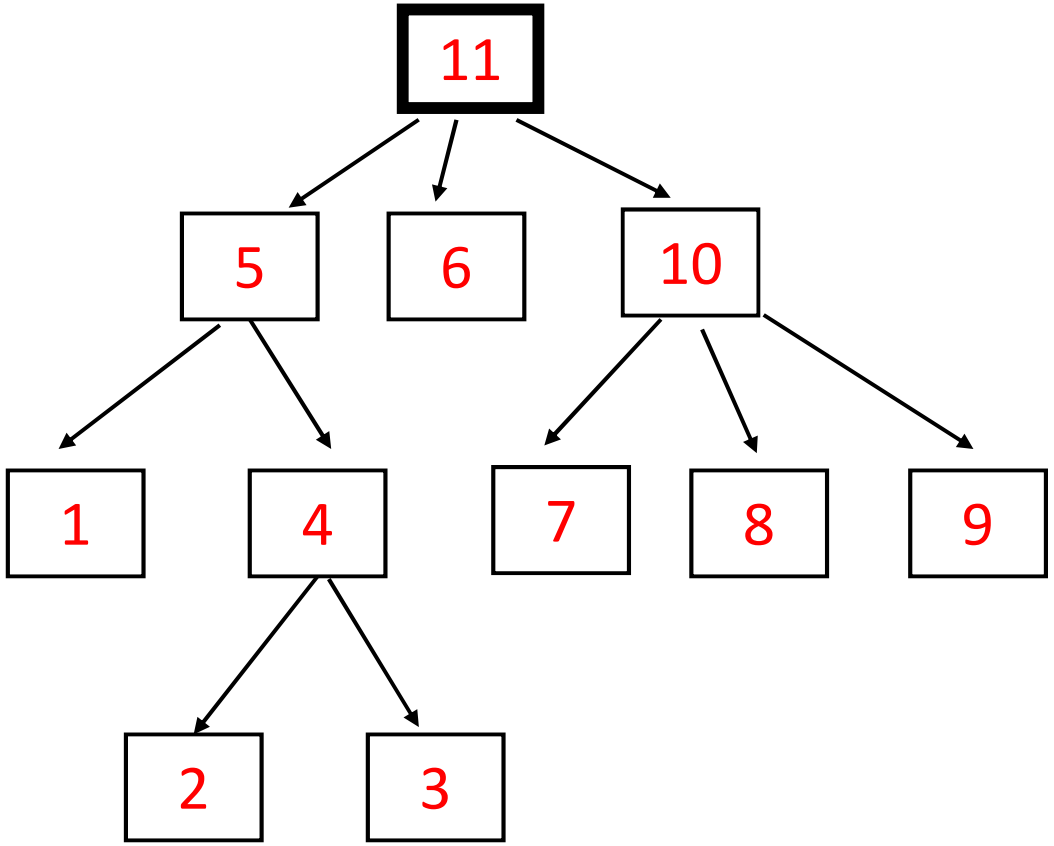
```
depthfirst (root){
    if (root is not empty){
        for each child of root
            depthfirst( child )
        visit root
    }
}
```

"postorder" traversal: visit the root after the children

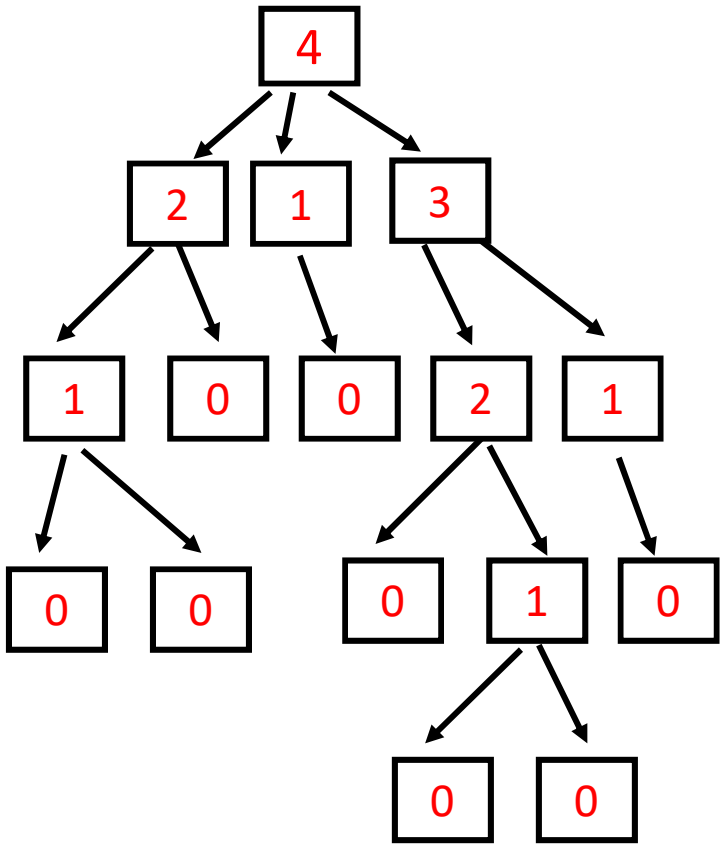


```
depthfirst (root){
    for each child of root
        depthfirst( child )
    visit root
}
```

"postorder" traversal: visit the root after the children



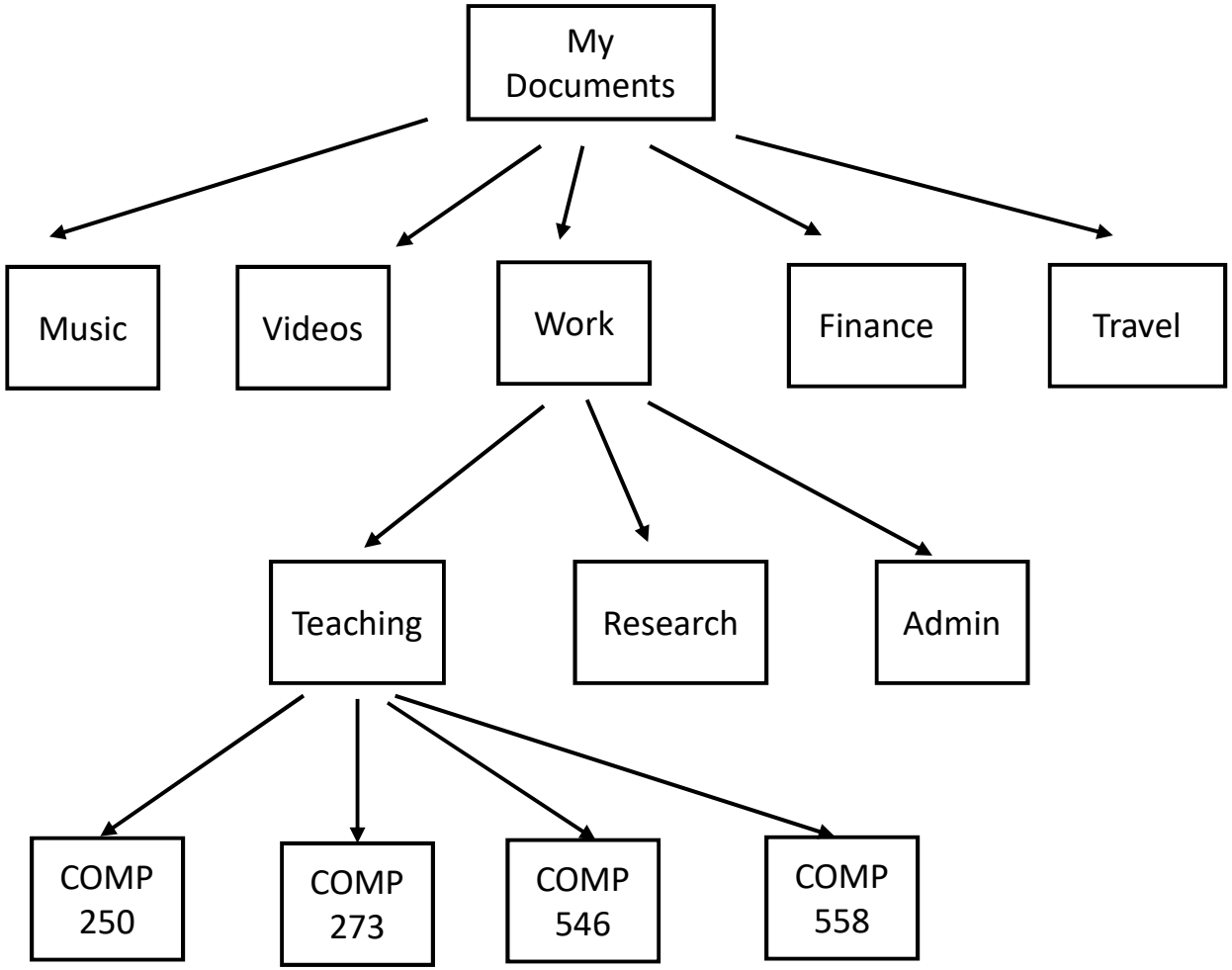
Example 1 postorder: height




```
height(v){
    if (v has no children)
        return 0
    else{
        h = 0
        for each child w of v
            h = max(h, height(w))
        return 1 + h
    }
}
```

`visit' a node here means `determine height of' the node

Example 2 Postorder:
total number of bytes (in a file or in some directory including subdirectories)



```
numBytes(root){
    if root has no children
        return number of bytes at root   //  0, if root is directory
    else {
        
    }
}
```


'visit' here means determine the number of bytes in subtree that is rooted at that node.

“preorder” traversal

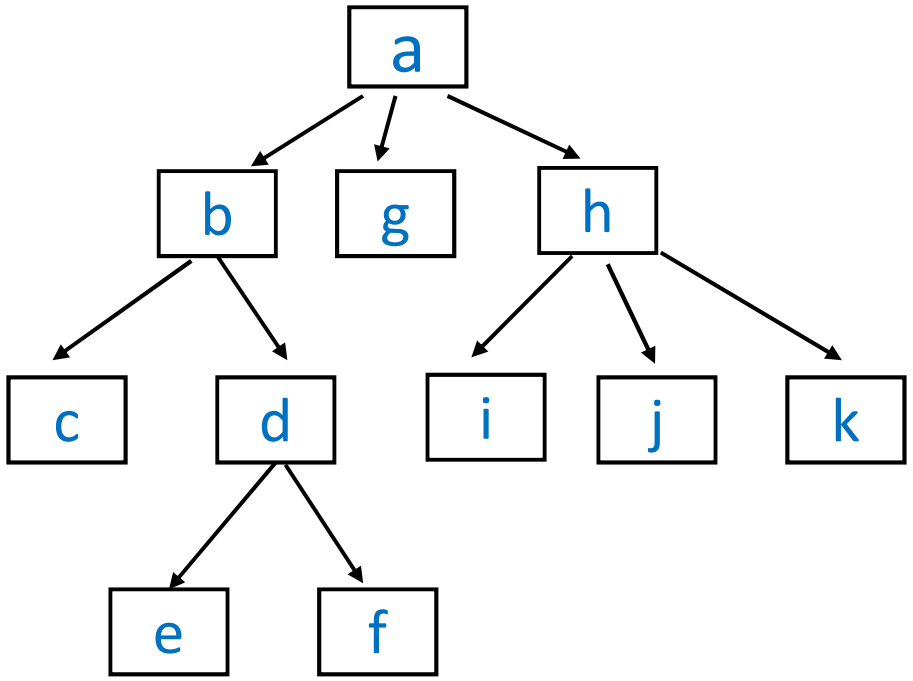
```
depthfirst (root){
    visit root
    for each child of root
        depthfirst( child )
}
```

“postorder” traversal

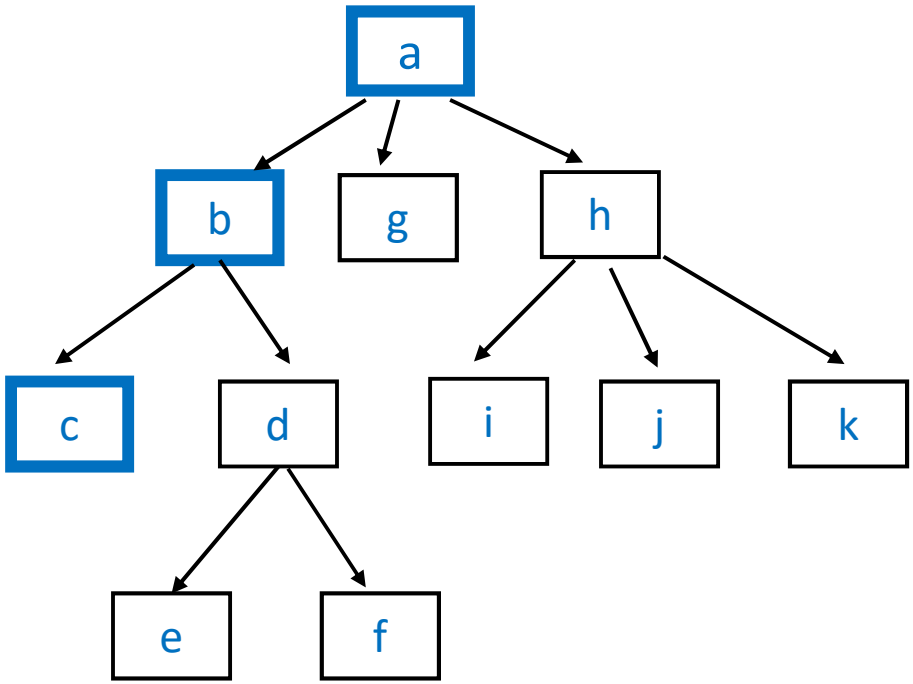
```
depthfirst (root){
    for each child of root
        depthfirst( child )
    visit root
}
```

The same depthfirst(root) *call sequence* occurs for preorder and postorder. Only the visiting order changes.

In example below, the letter order corresponds to the depthfirst(root) *call order*. Let's next examine the call stack.

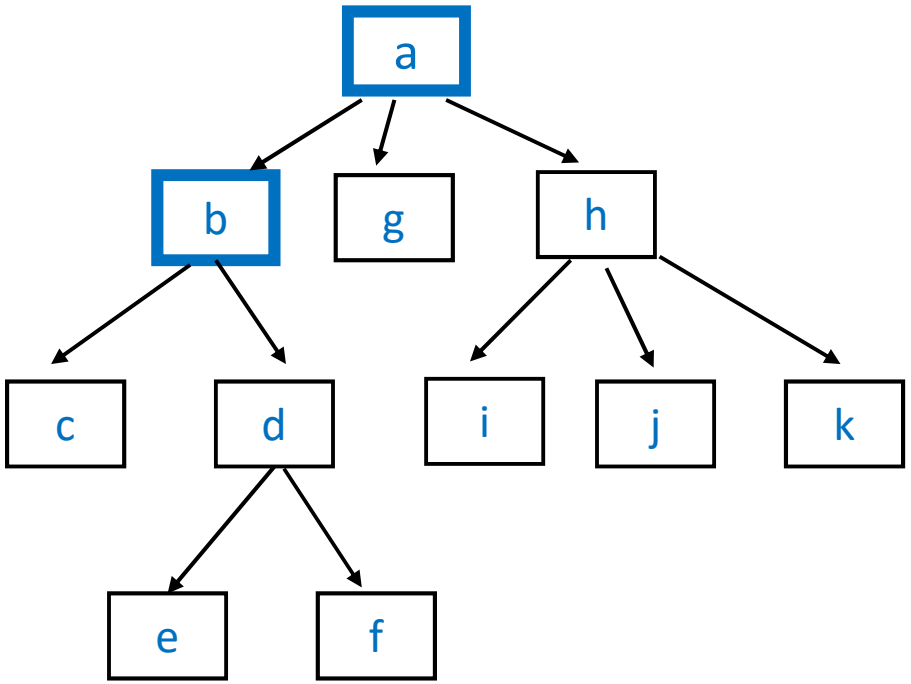


Call stack for depthfirst(root)



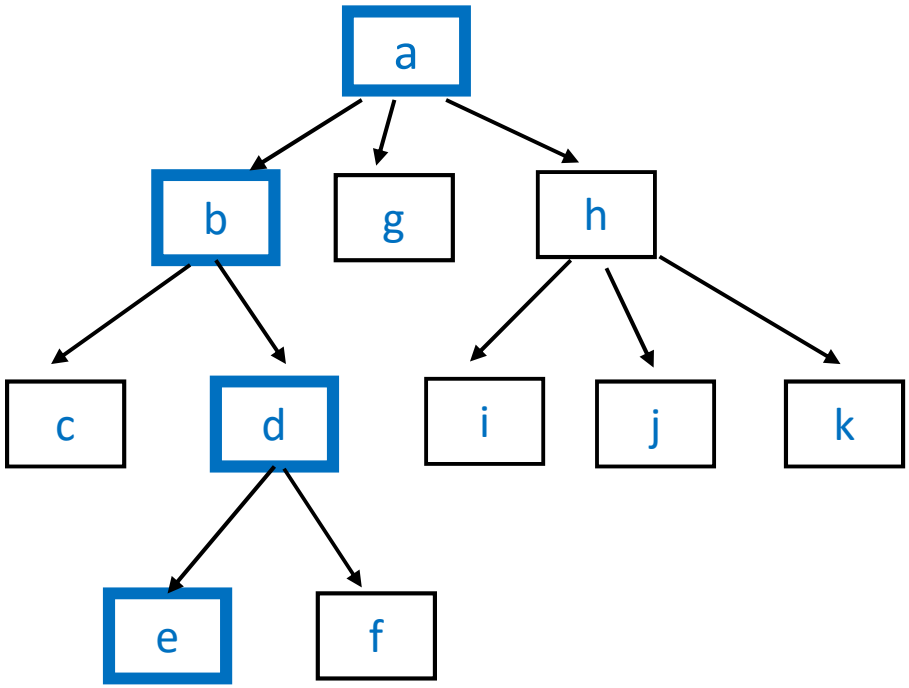
The “call stack” figure below only shows the root parameter.

Call stack for depthfirst(root)



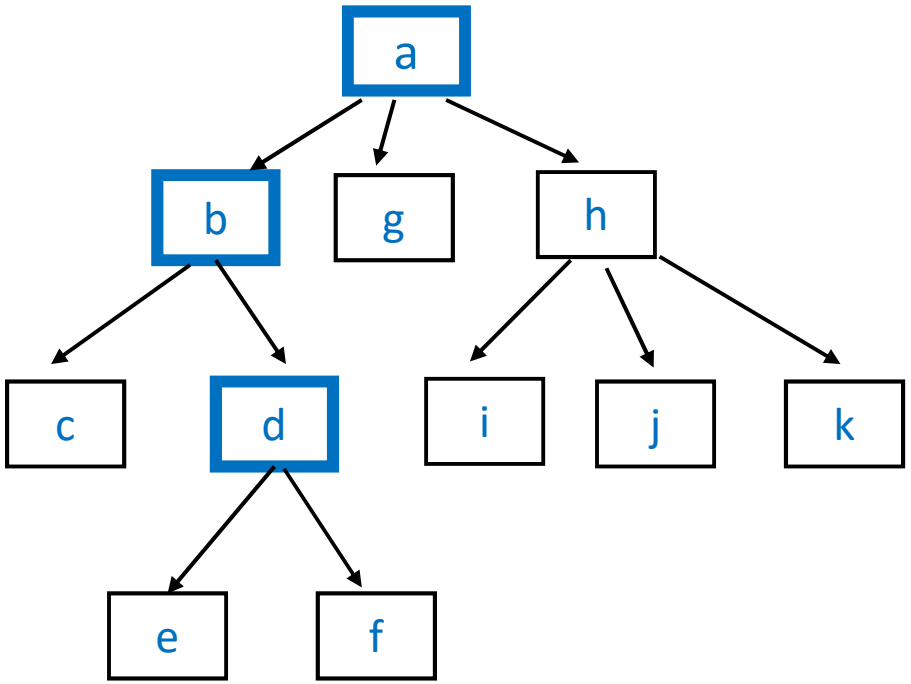
The “call stack” figure below only shows the root parameter.

Call stack for depthfirst(root)

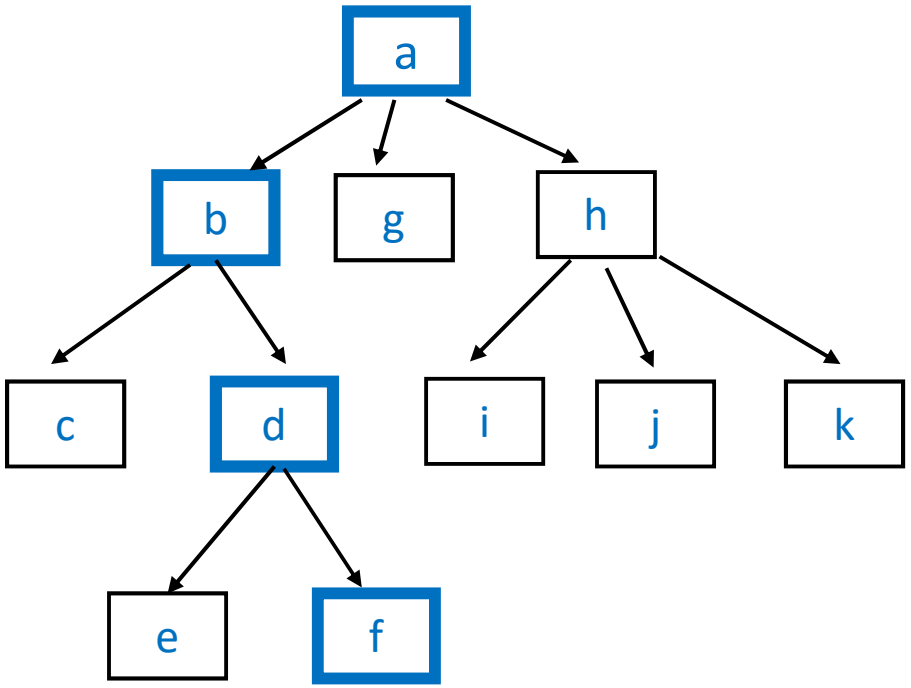


The “call stack” figure below only shows the root parameter.

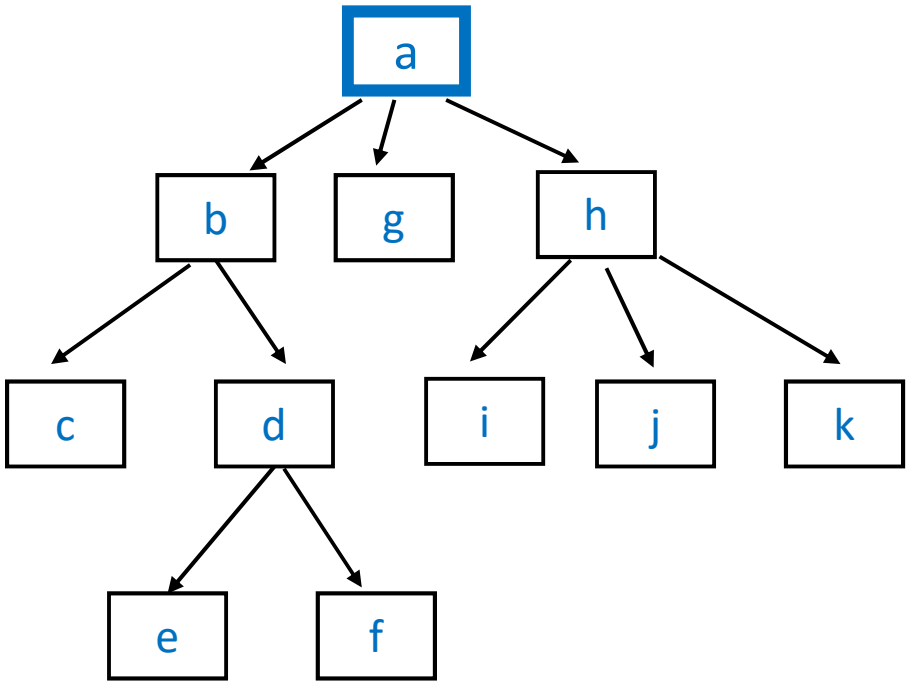
Call stack for depthfirst(root)

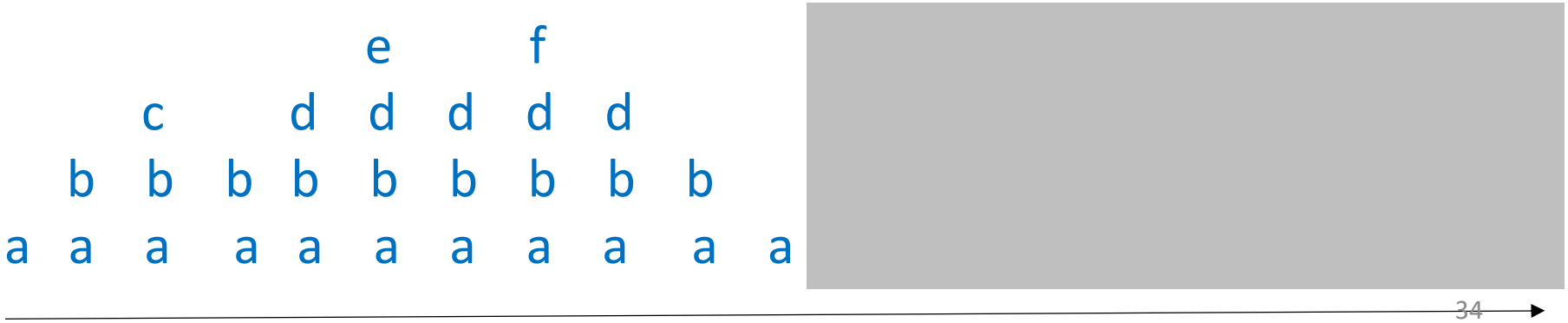


Call stack for depthfirst(root)

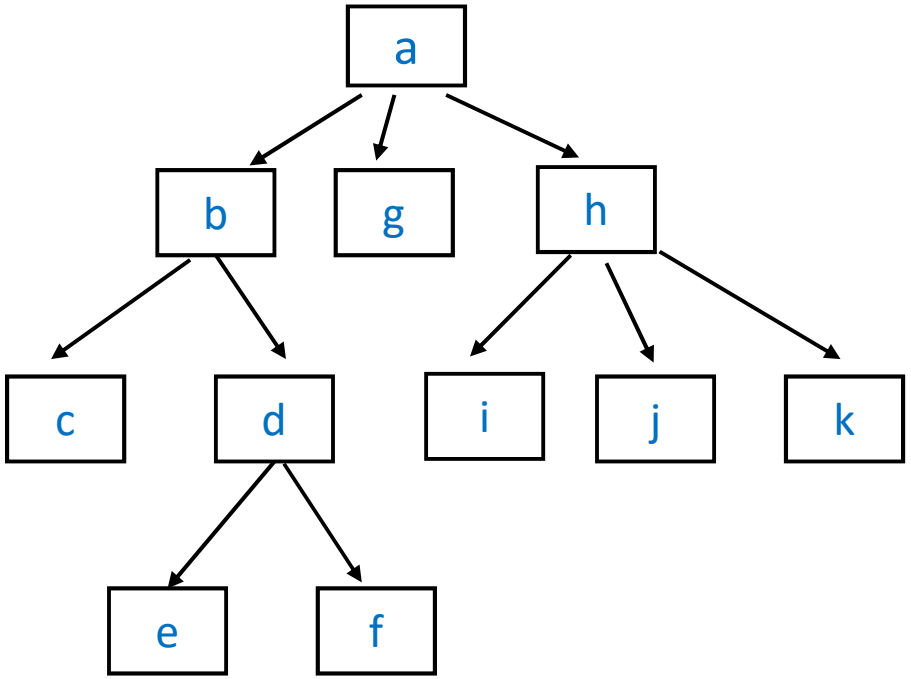


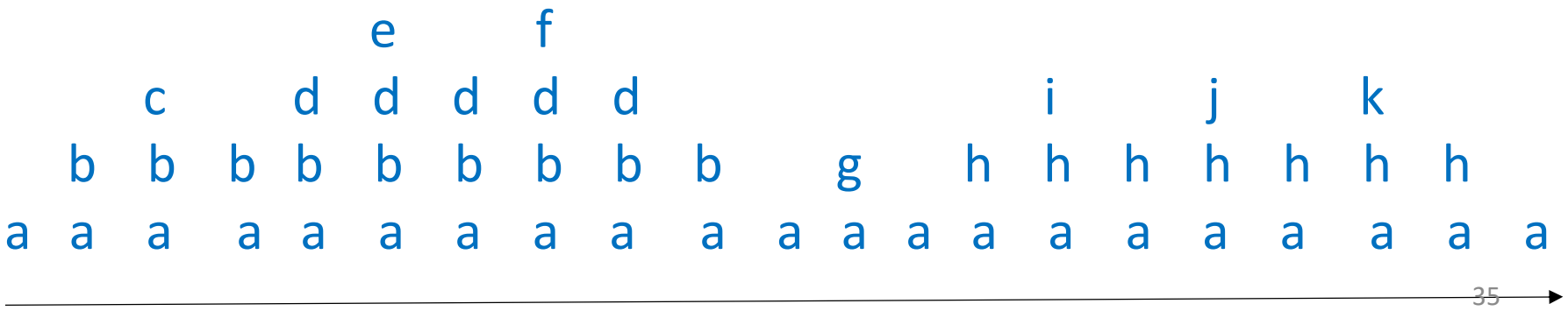
Call stack for depthfirst(root)





Call stack for depthfirst(root)





Tree traversal

Recursive

- depth first (pre- versus post-order)

Non-Recursive

- using a stack
- using a queue

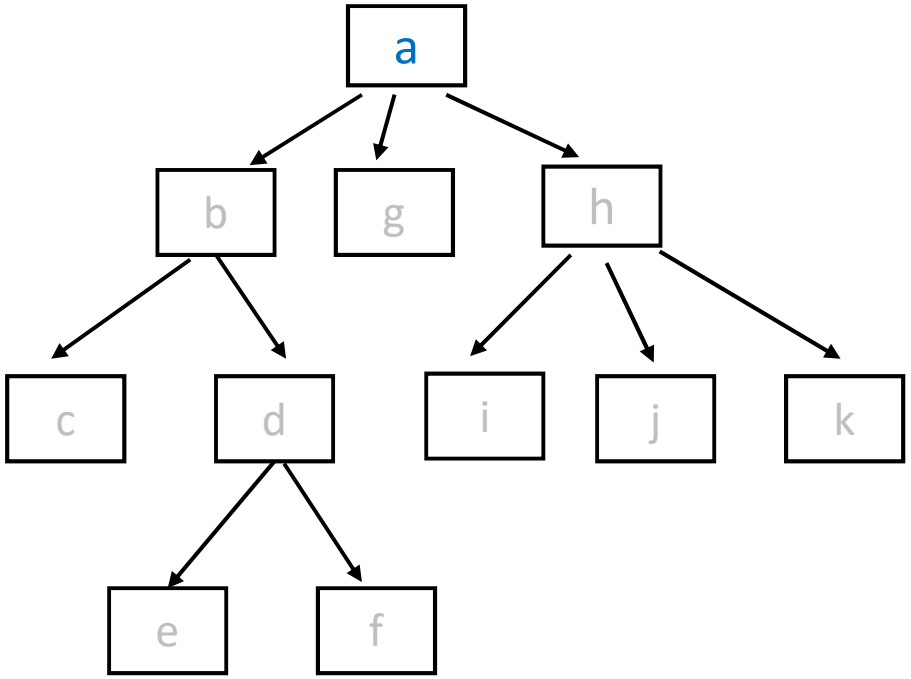
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    
}
```

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

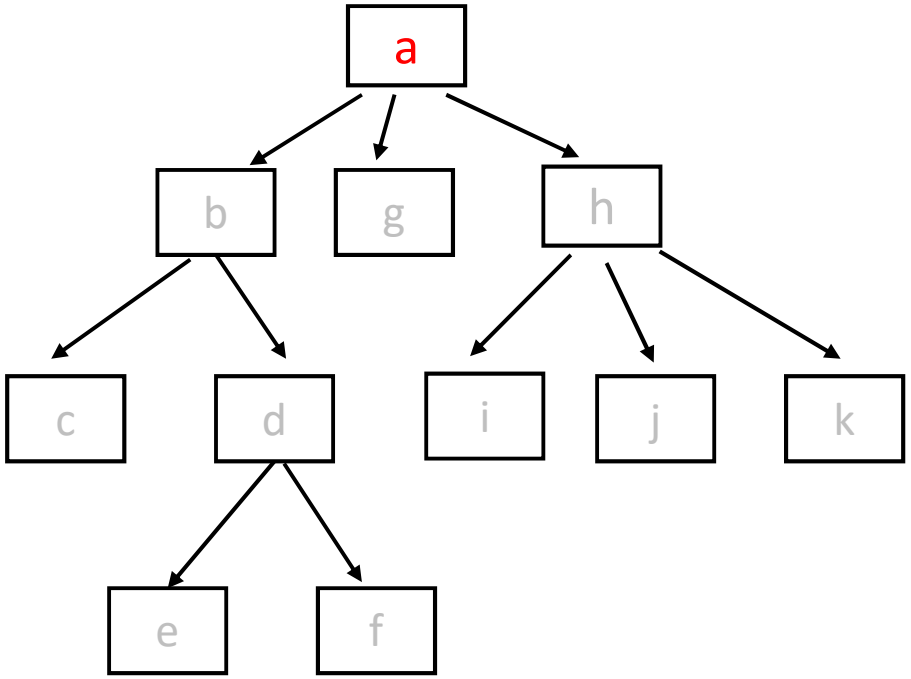
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```

What is the order of nodes visited by treeTraversalUsingStack() ?

```
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)
    }
}
```

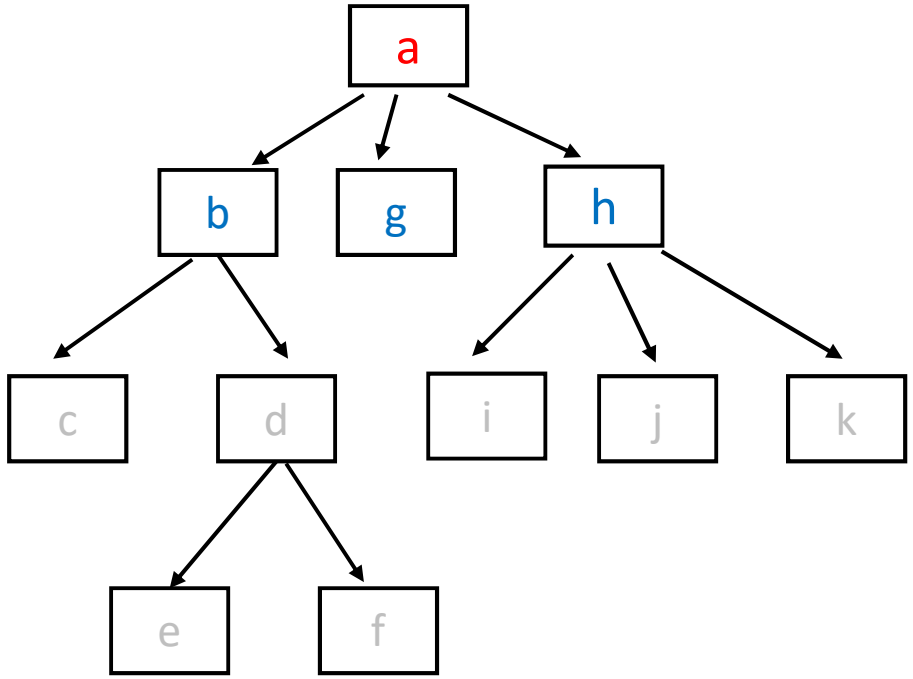


```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```

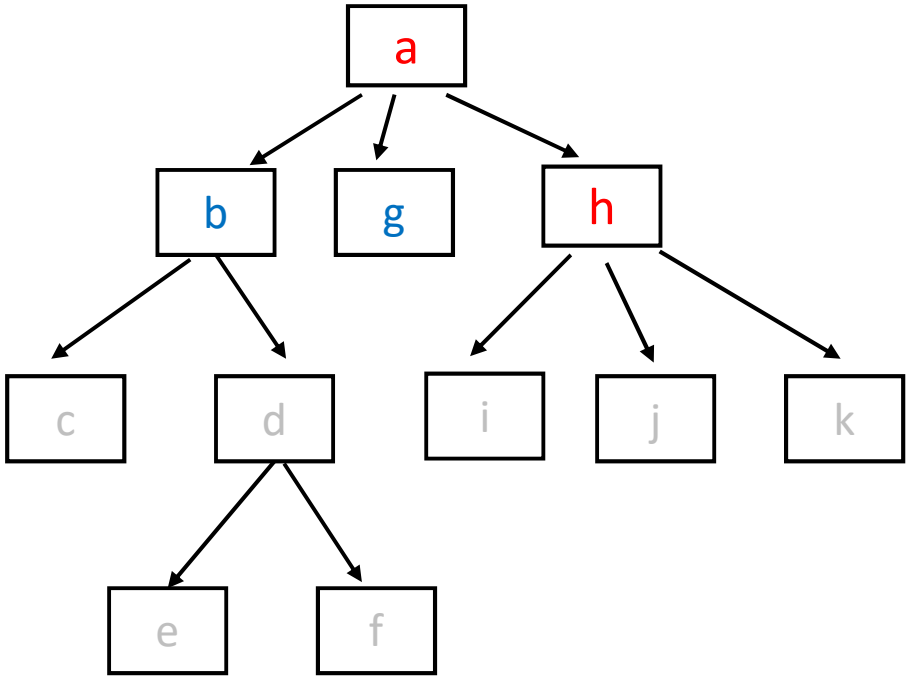


a _

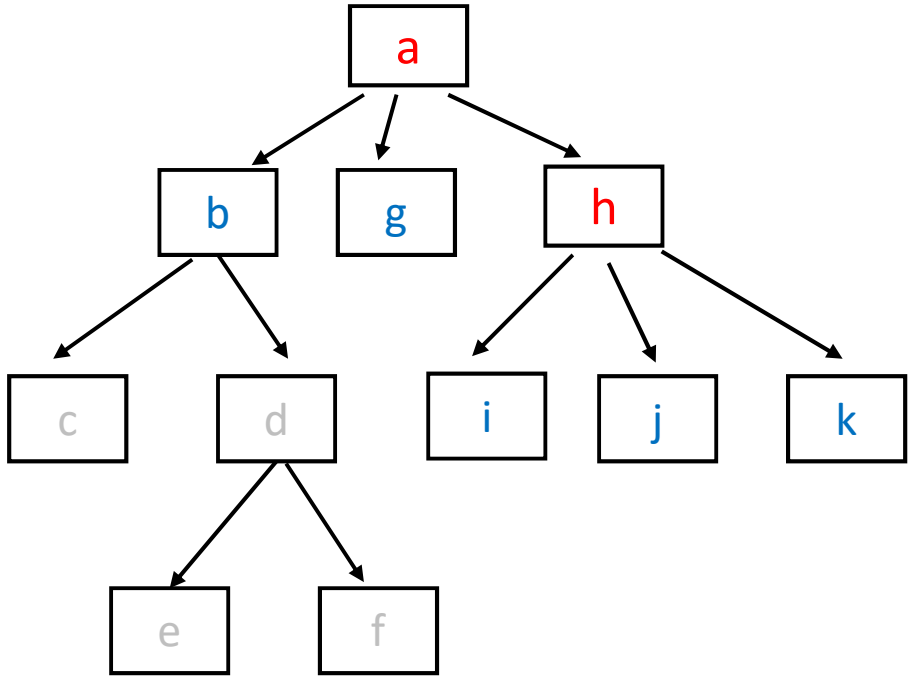
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```



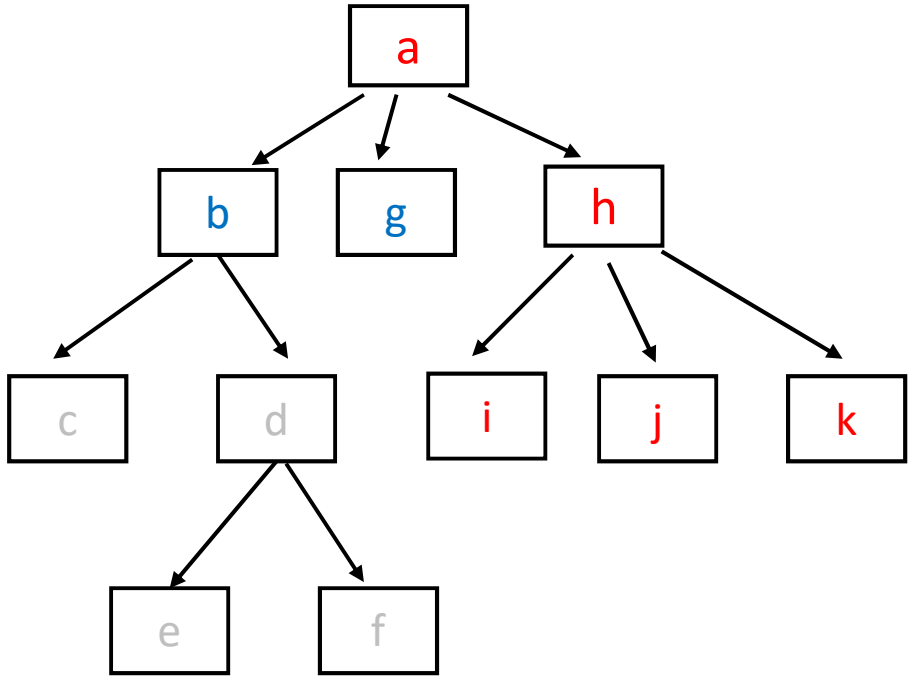
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```

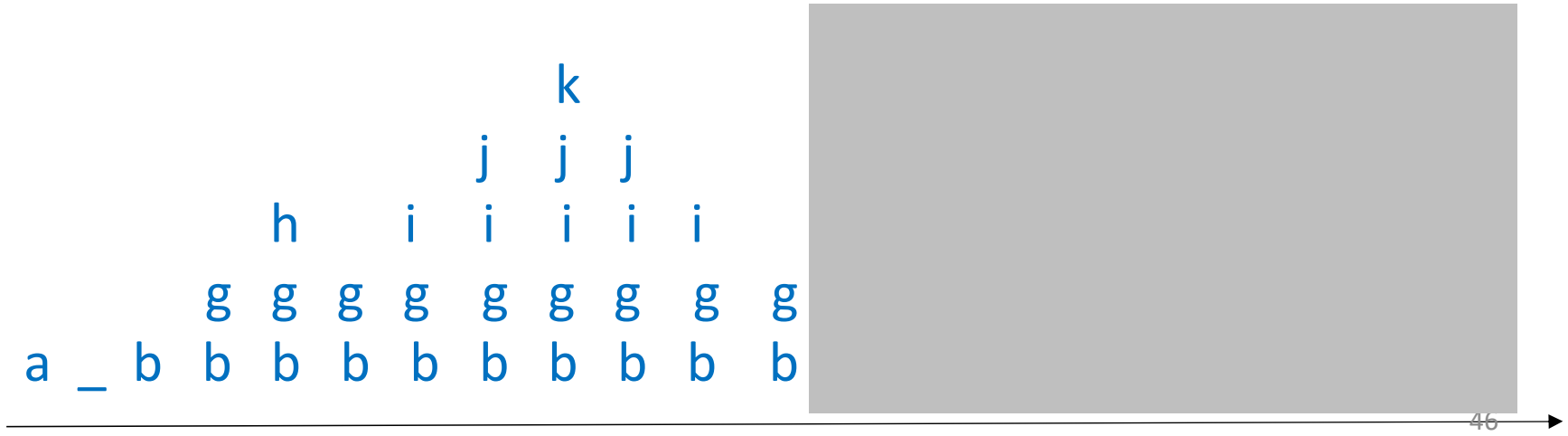


```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```

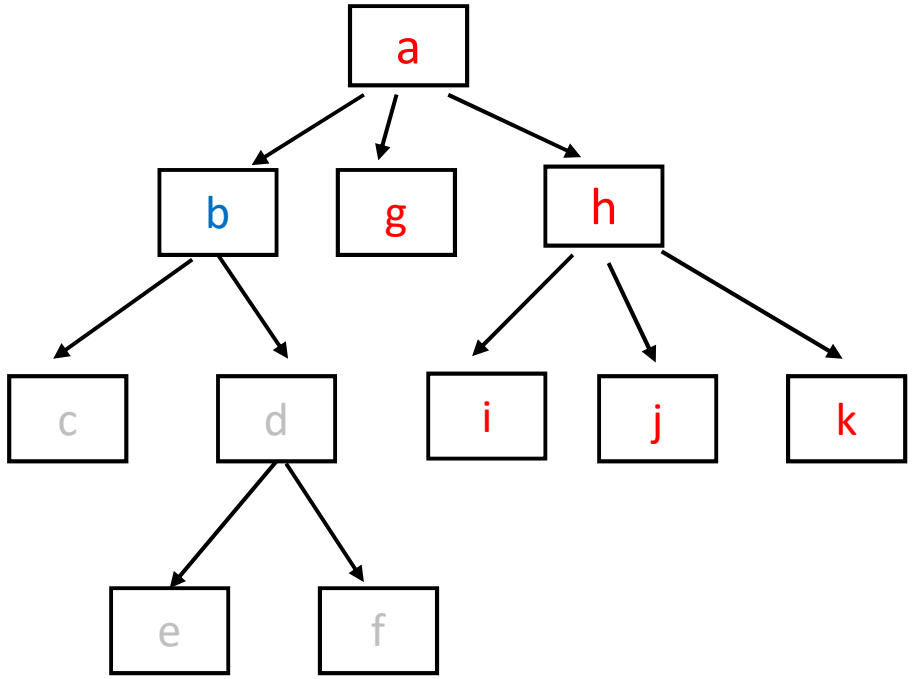


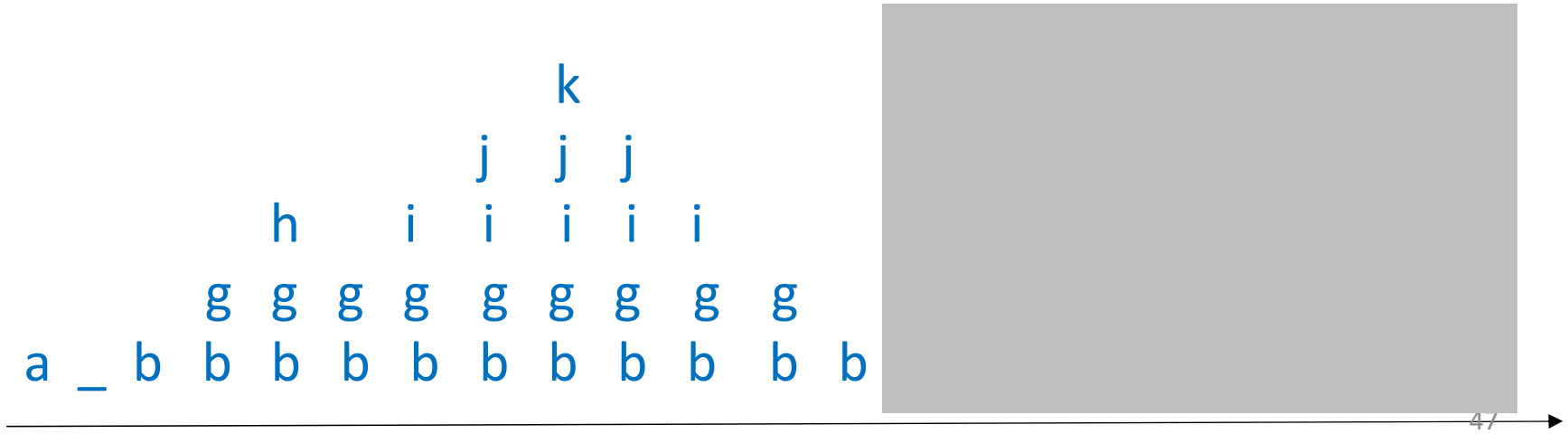
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```



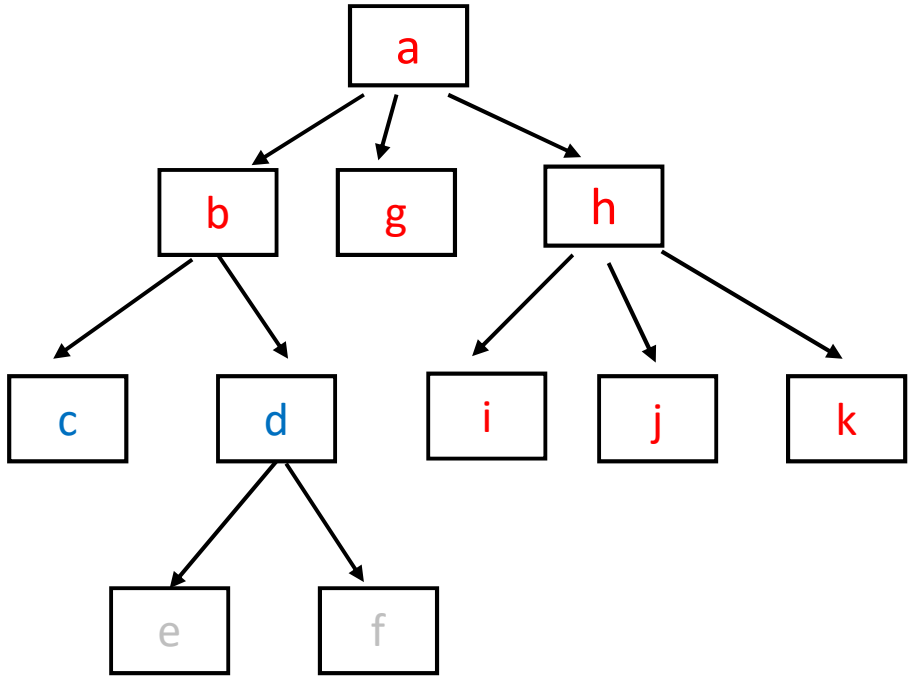


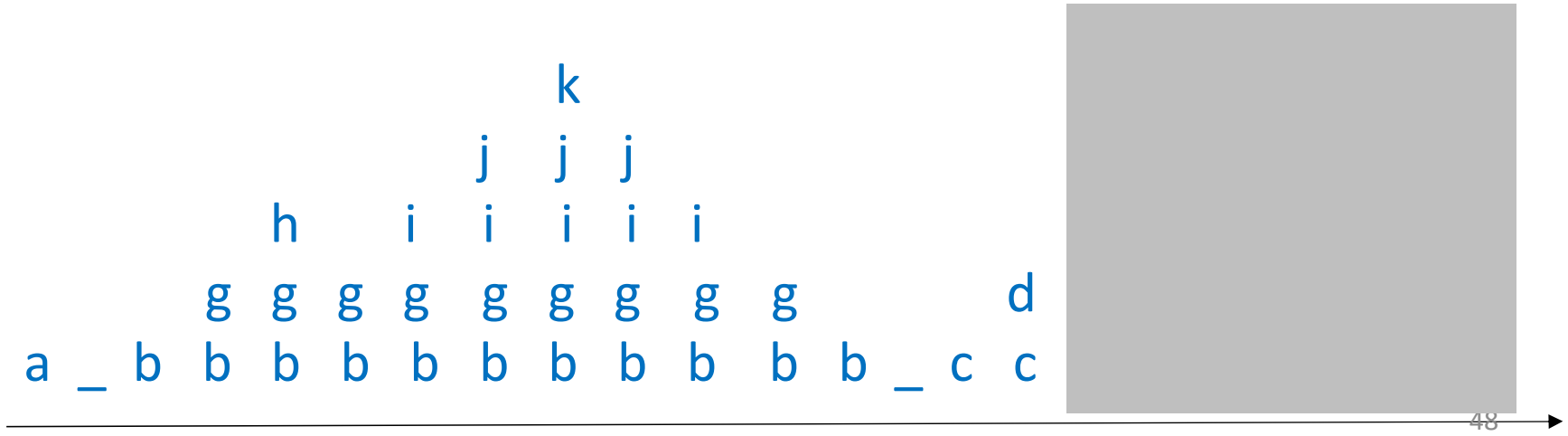
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```



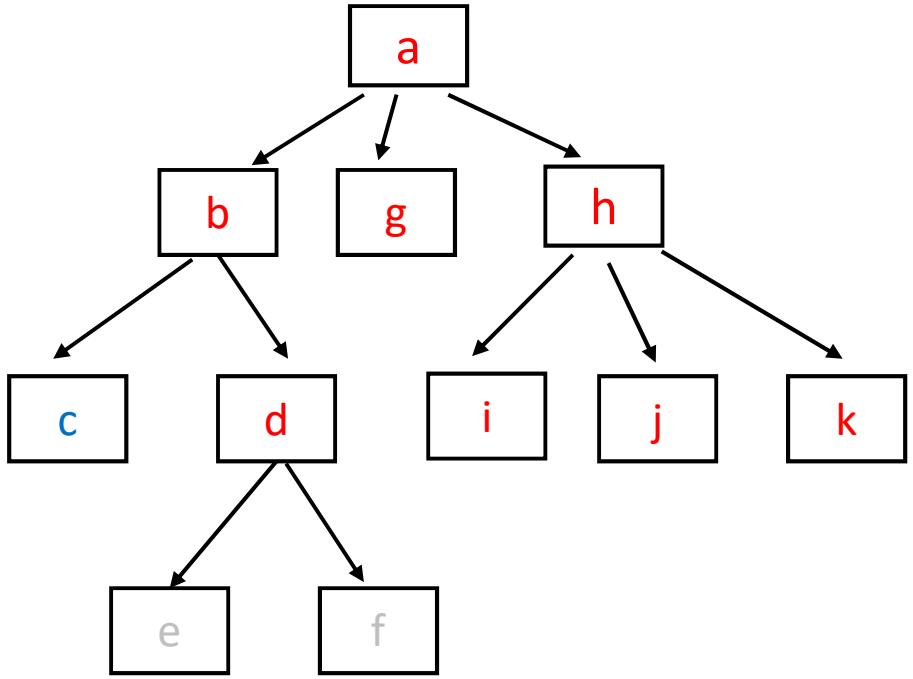


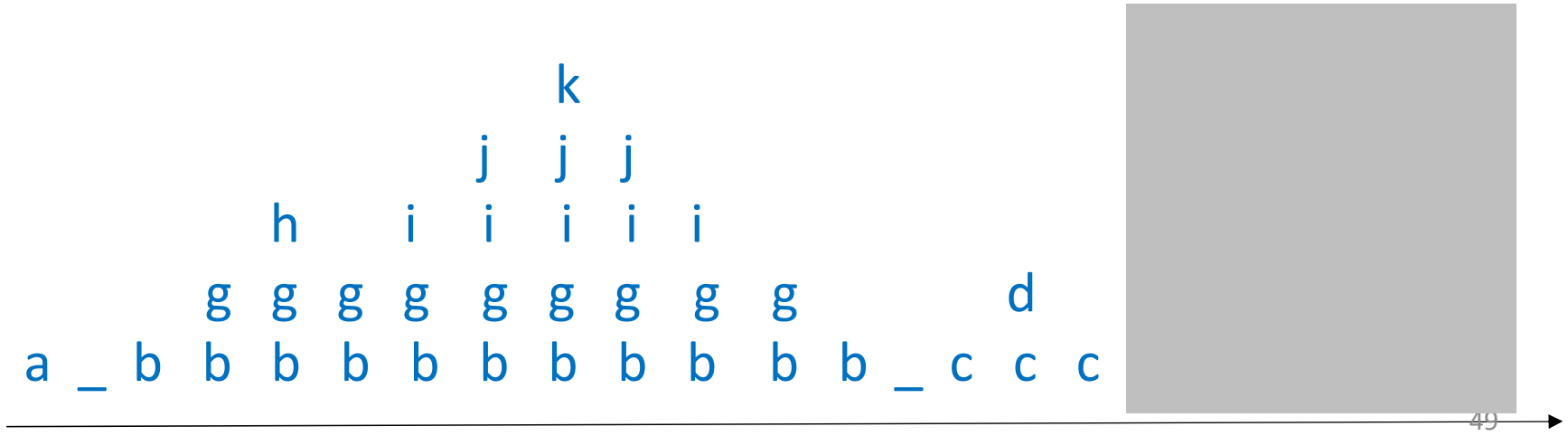
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```



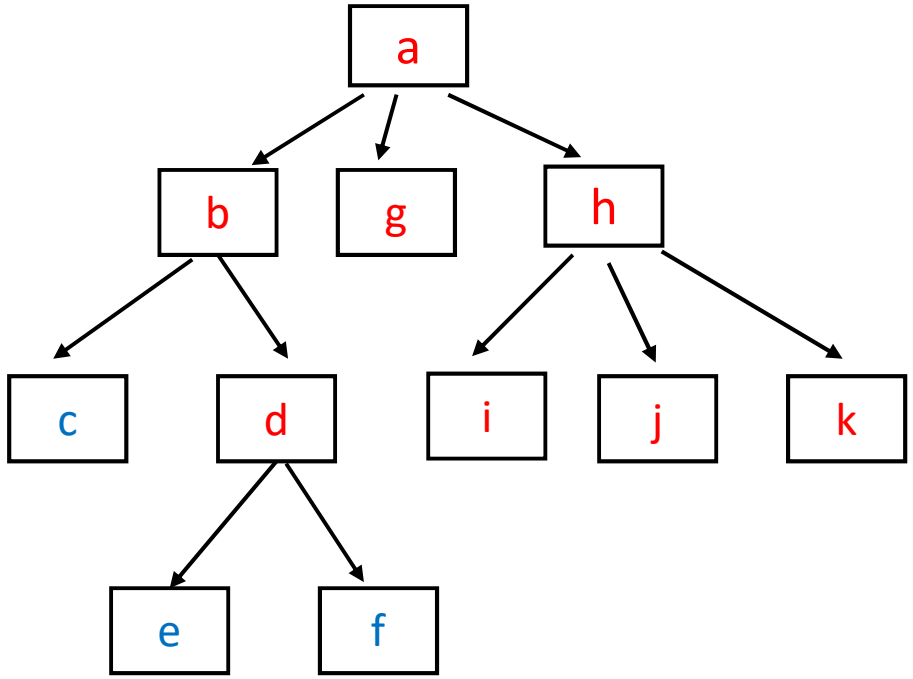


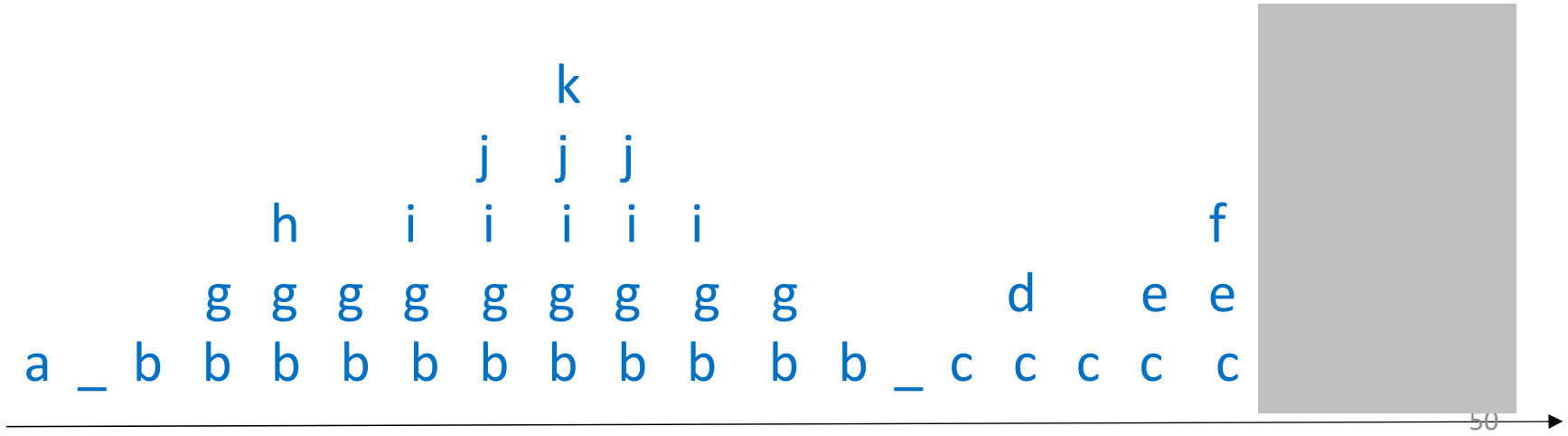
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```



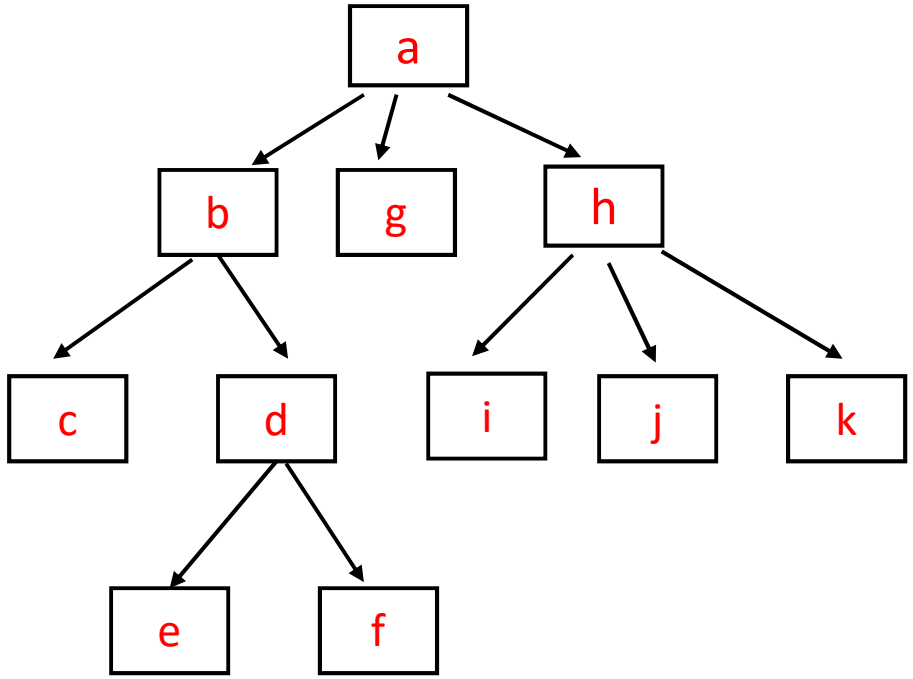


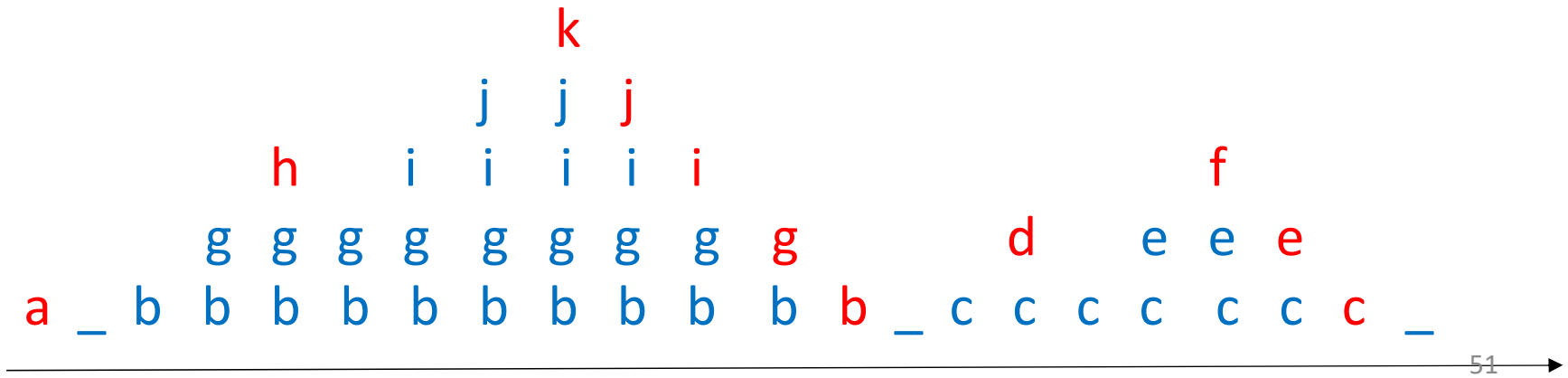
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```



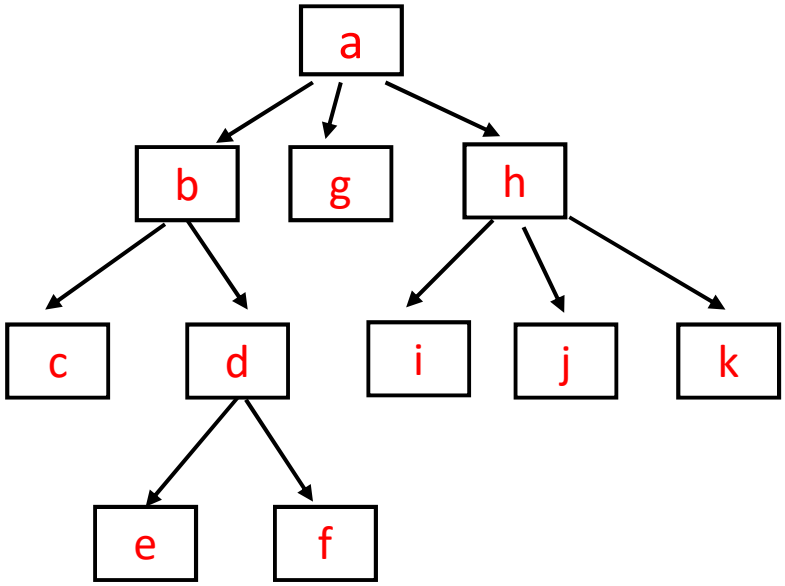


```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```





The stack based method is also depth first, but we are visiting children from right to left



recursive preorder : abcdefghijk

recursive postorder : cefdbgijkha

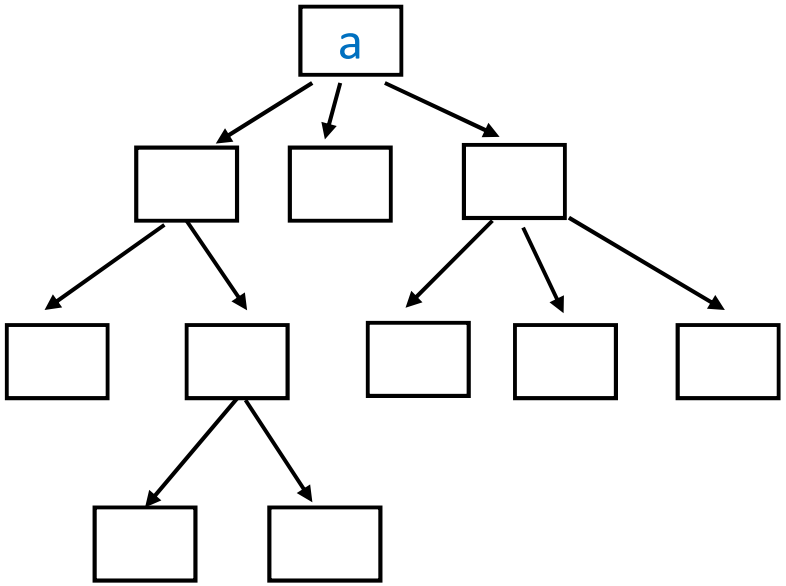
non-recursive (stack) : ahkjigbdfec

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)
    }
}
```

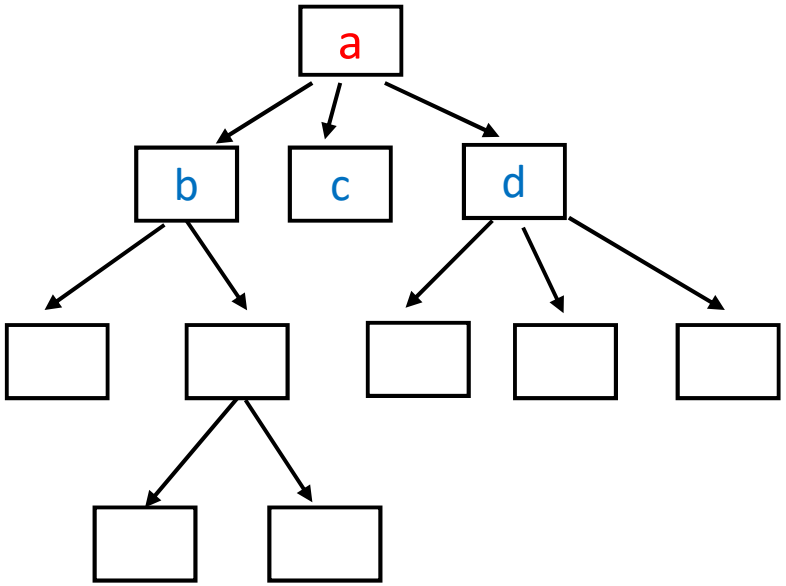
```
treeTraversalUsingQueue(root){
    initialize empty queue  q
    q.enqueue(root)
    while q is not empty {
        cur = q.dequeue()
        visit cur
        for each child of cur
            q.enqueue(child)
    }
}
```



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)
    }
}
```

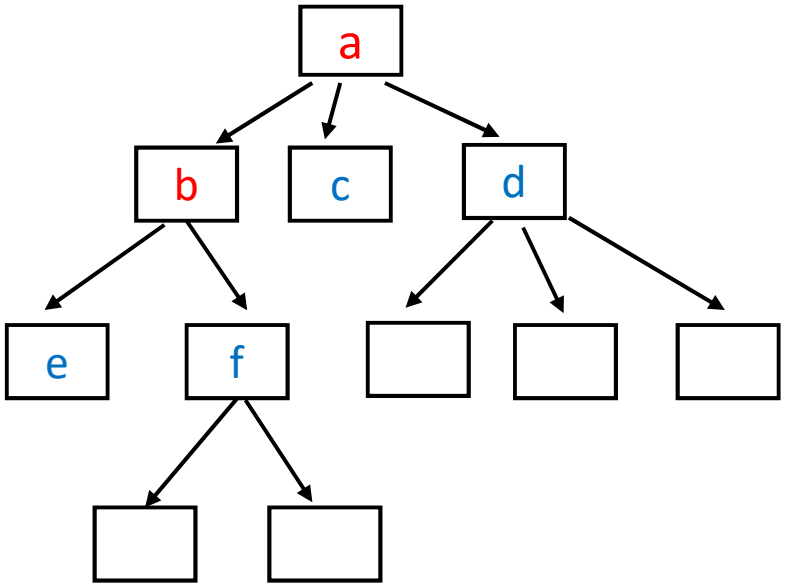


Queue state at start of the while loop

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)
    }
}
```



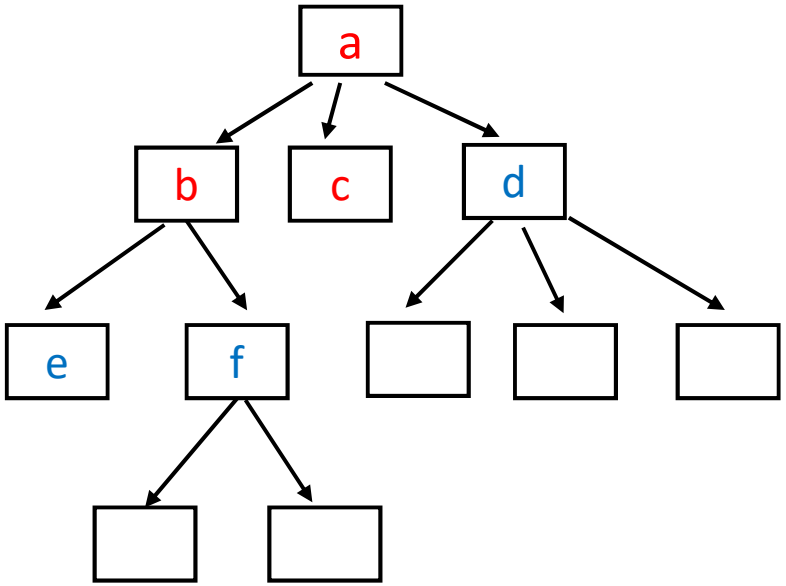
Queue state at start of the while loop

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

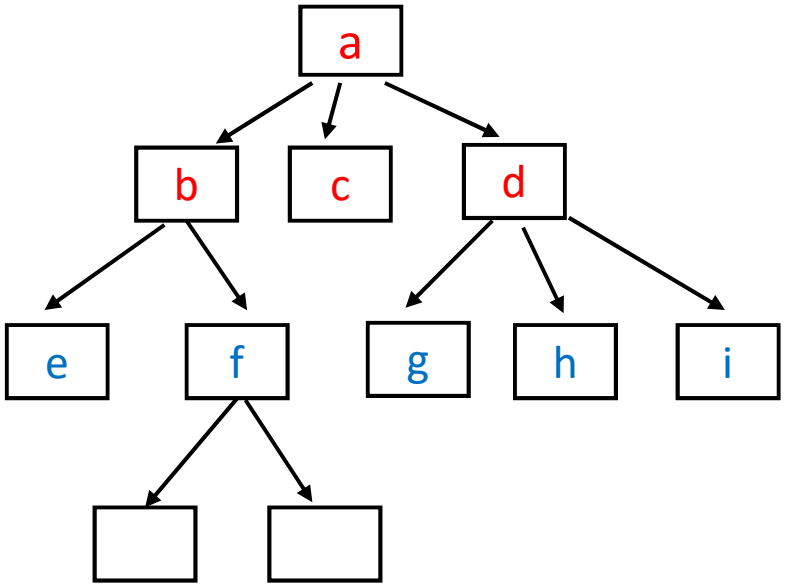
a

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

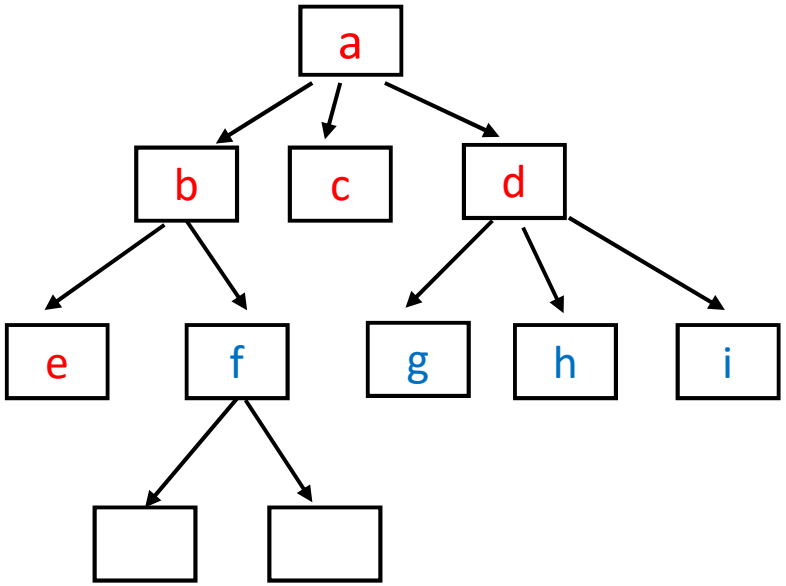
b c d

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)
    }
}
```



Queue state at start of the while loop

a

b c d

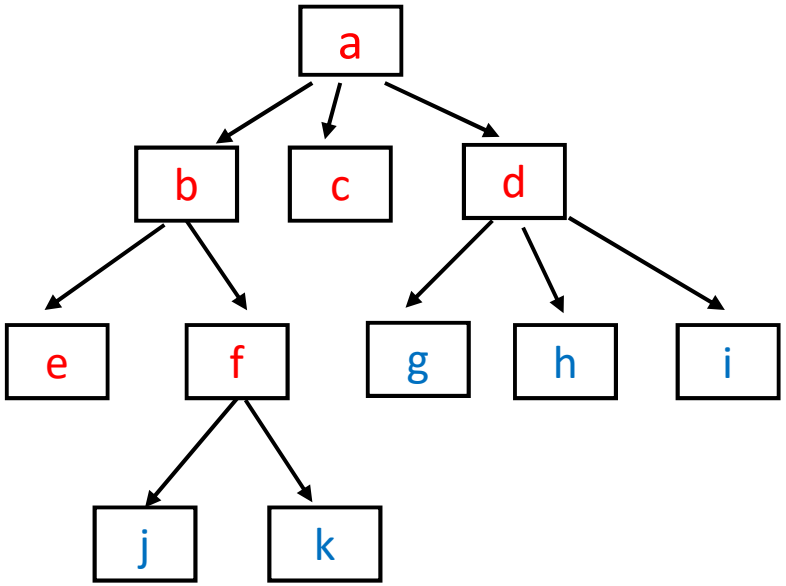
c d e f

d e f

e f g h i

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)
    }
}
```



Queue state at start of the while loop

a

b c d

c d e f

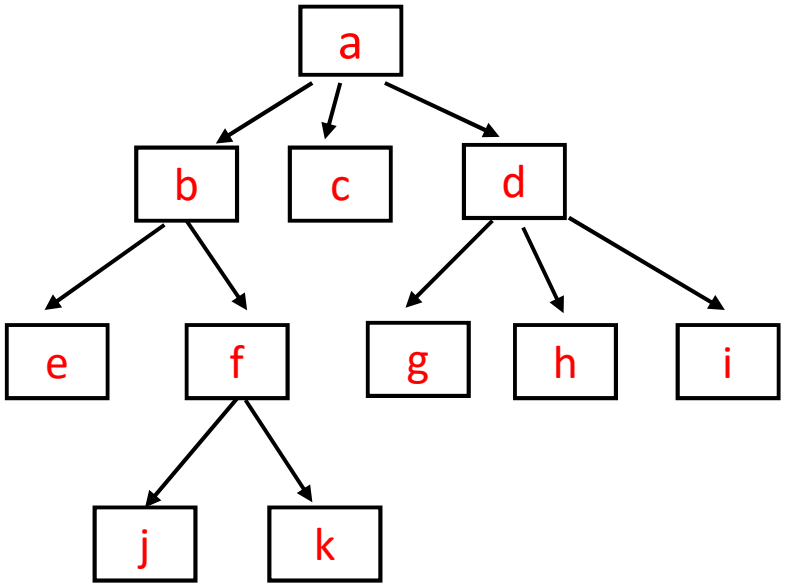
d e f

e f g h i

f g h i

g h i j k

```
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

f g h i

g h i j k

h i j k

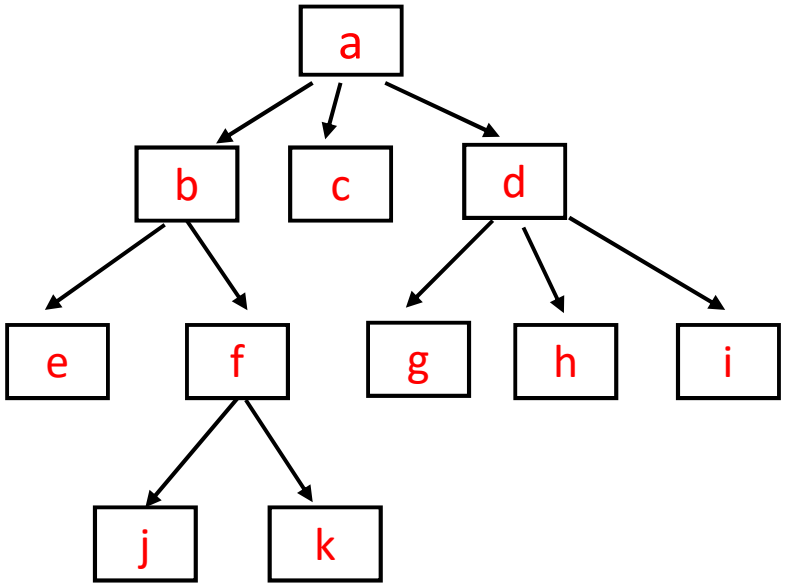
i j k

j k

k

breadth first traversal

for each level i
    visit all nodes at level i



order visited: abcdefghijk

Coming up...

Lectures

Fri. March 11 (lecture 25)
Expression Trees

Mon. March 14
Binary Search Trees

Wed & Fri. March 16 & 18
Heaps

Tutorial + Assessments

Tutorial for Assignment 3
today at 6 pm

Assignment 3
due Wed. March 16

Quiz 4 (lectures 20-25)
Fri. March 18