COMP 250

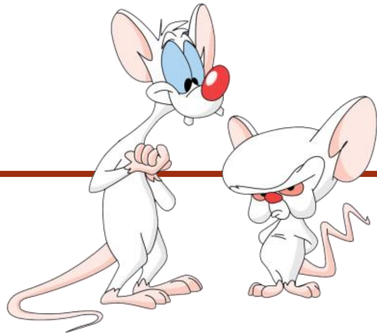
INTRODUCTION TO COMPUTER SCIENCE

Week 11-2 : Tree Traversals

Giulia Alberini, Fall 2020

Slides adapted from Michael Langer’s

WHAT ARE WE GOING TO DO IN THIS VIDEO?

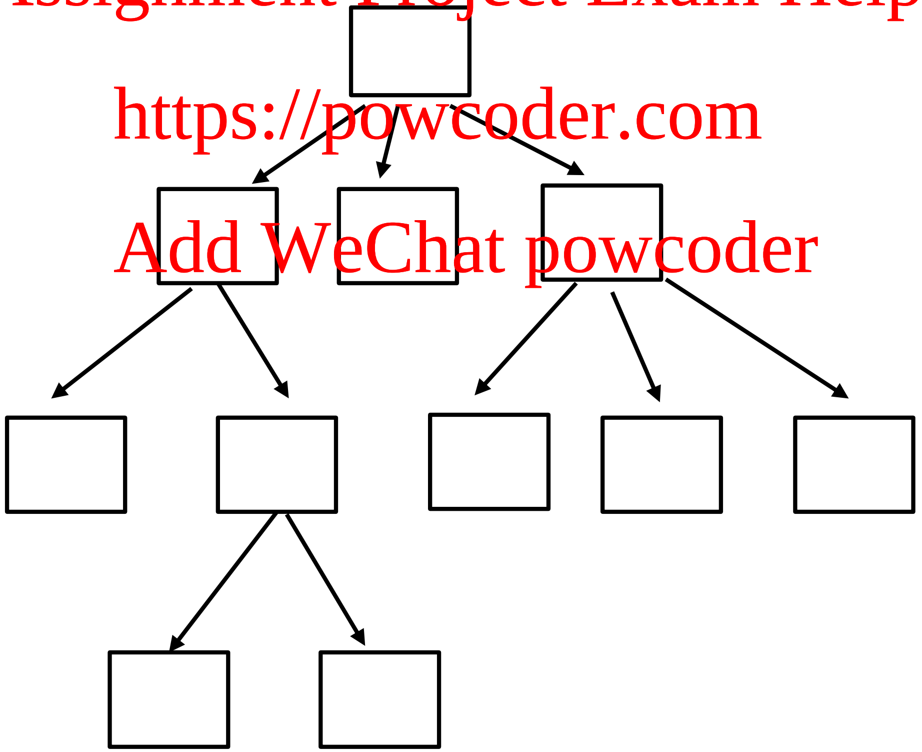


- Tree traversals
  - Depth first VS Breadth first
  - Recursive and Non-recursive (with stack or with queue)

TRAVERSALS

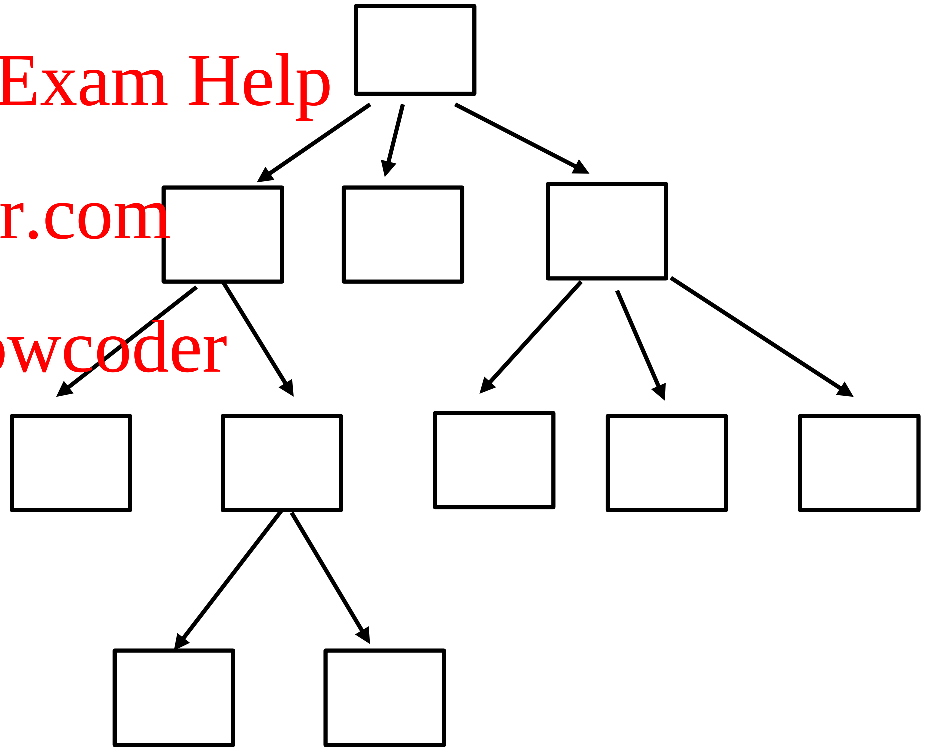
TREE TRAVERSAL

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



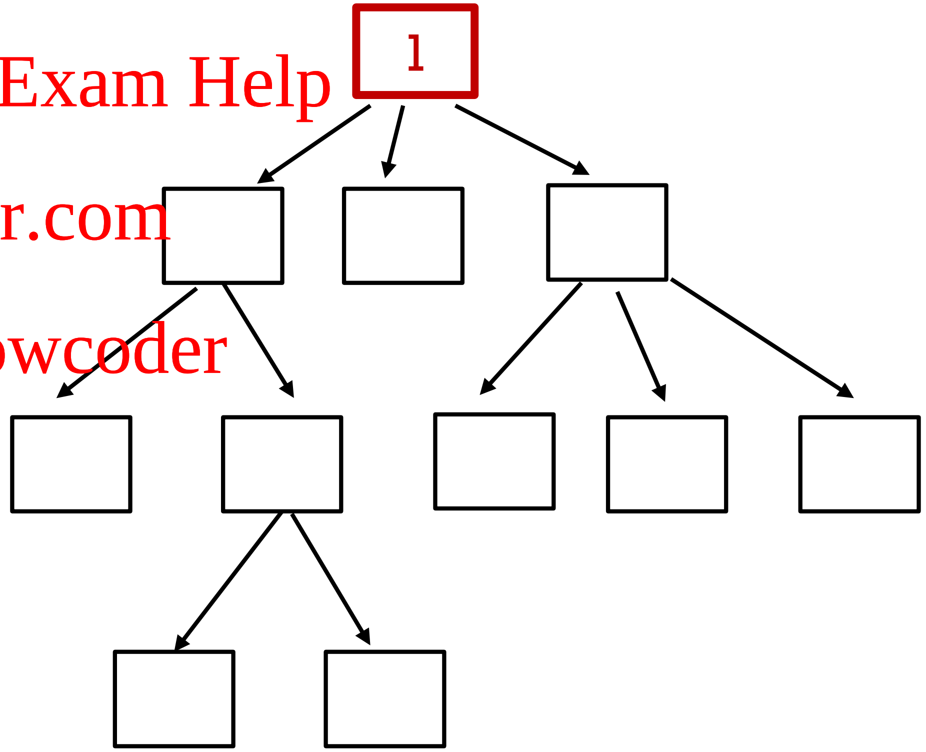
TREE TRAVERSAL – DEPTH FIRST "PREORDER"

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



TREE TRAVERSAL – DEPTH FIRST "PREORDER"

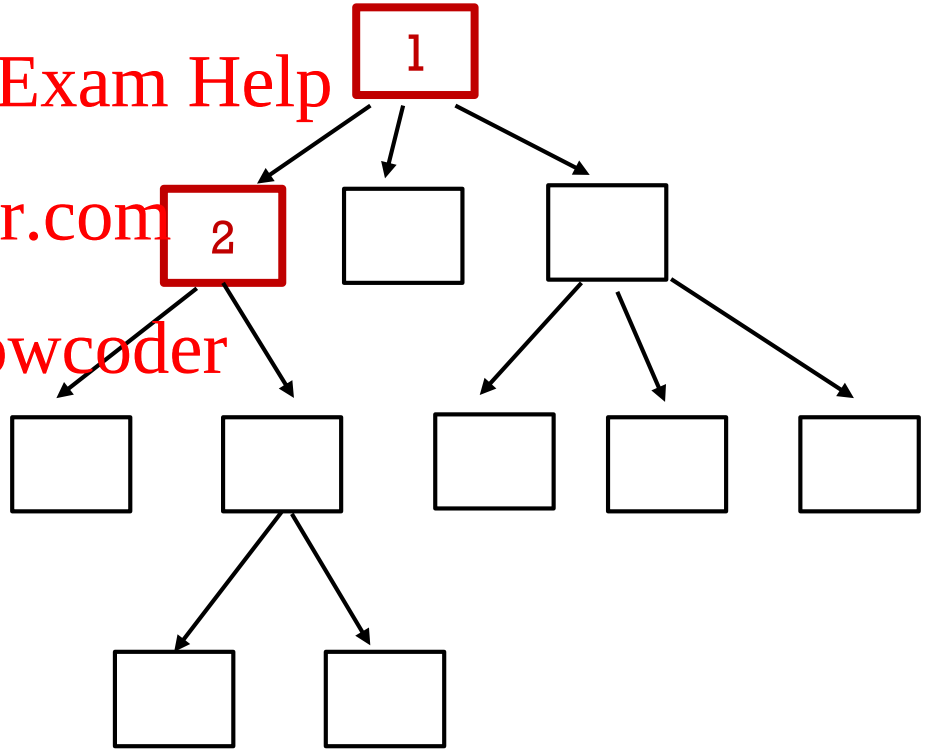
```
depthFirst (root){
    if (root is not empty){
        visit root
        for each child of root
            depthfirst( child )
    }
}
```



"preorder" traversal: visit the root before the children

TREE TRAVERSAL – DEPTH FIRST "PREORDER"

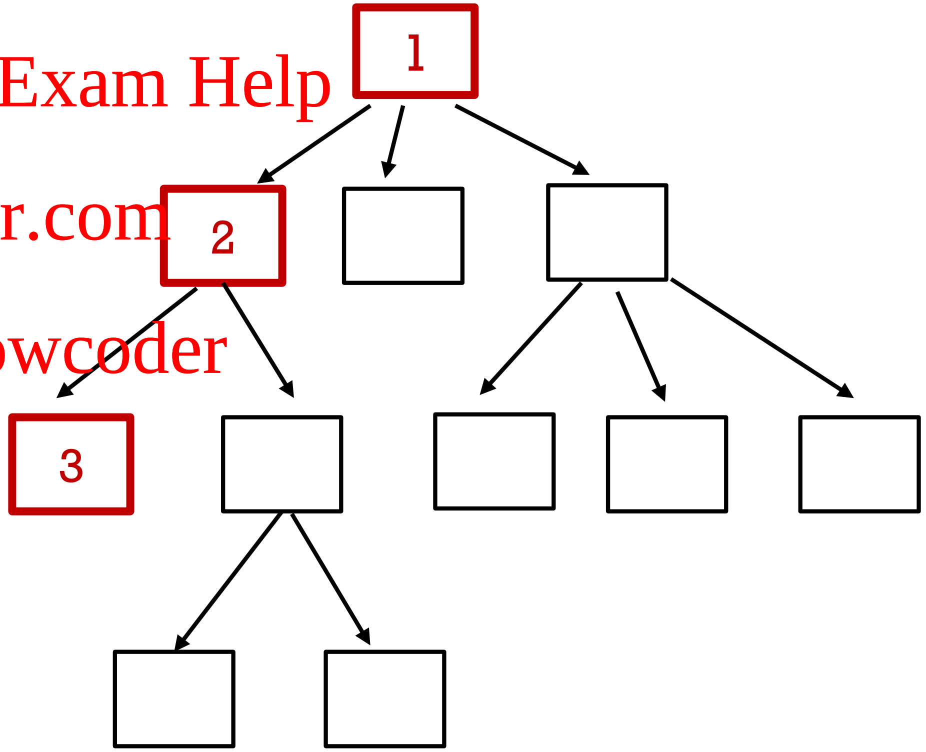
```
depthFirst (root){
    if (root is not empty){
        visit root
        for each child of root
            depthfirst( child )
    }
}
```



Note that here we are assuming that we iterate through the children nodes from left to right.

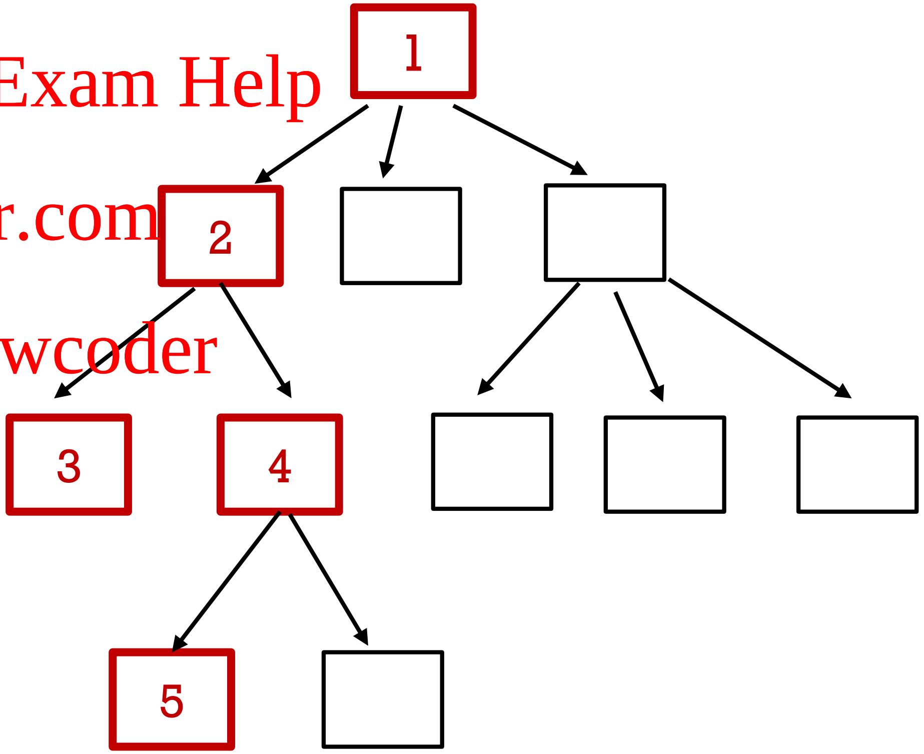
TREE TRAVERSAL – DEPTH FIRST “PREORDER”

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



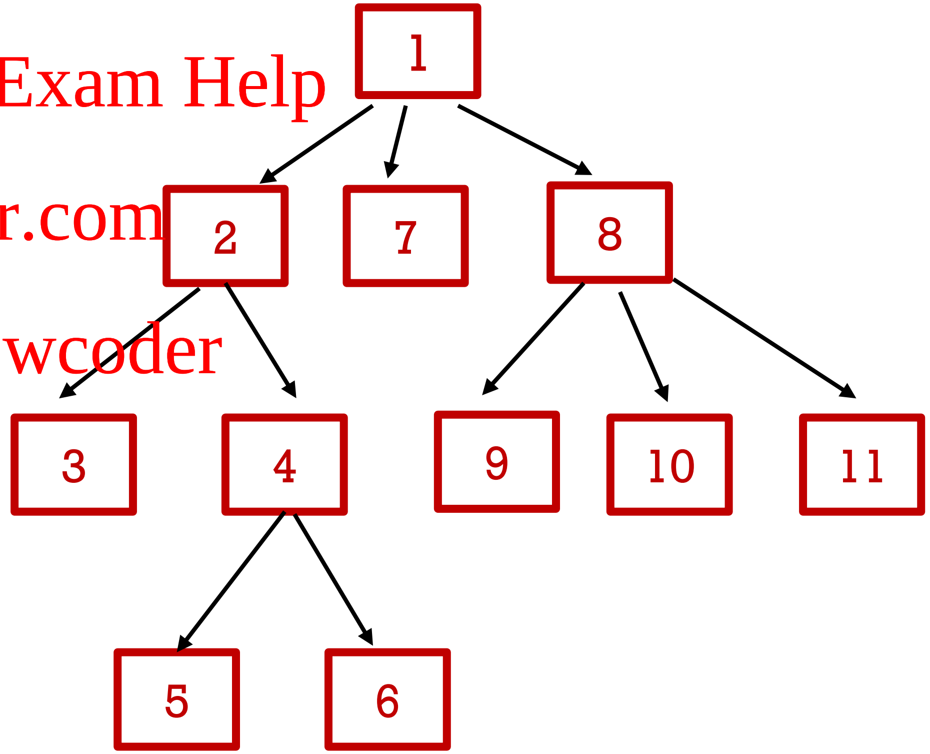
TREE TRAVERSAL – DEPTH FIRST “PREORDER”

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



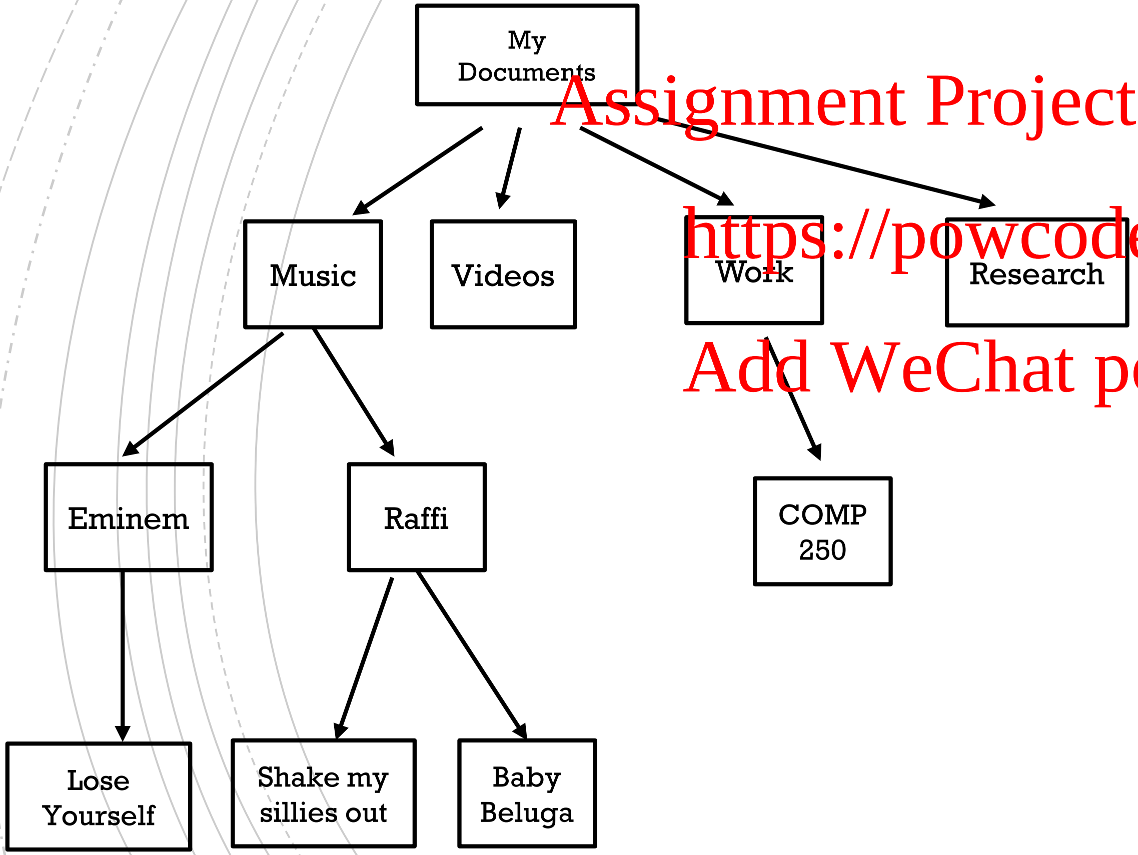
TREE TRAVERSAL – DEPTH FIRST “PREORDER”

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



EXAMPLE OF USING A PREORDER TRAVERSAL

We would like to print 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" A NODE

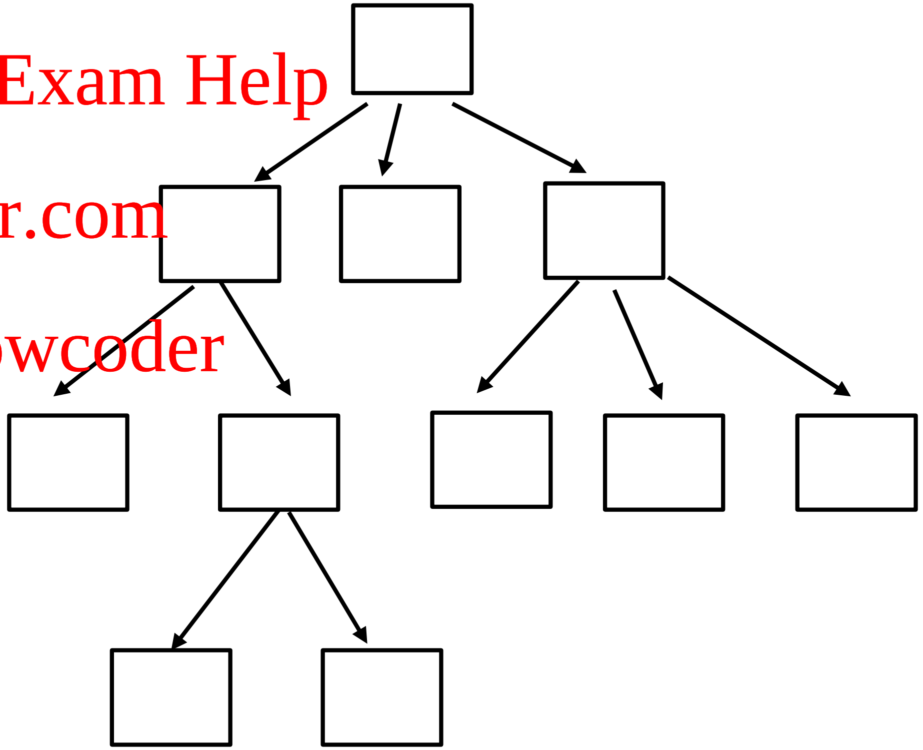
"Visit" implies that you do something at that node.

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

TREE TRAVERSAL – DEPTH FIRST "POSTORDER"

Q: Which node is visited first?

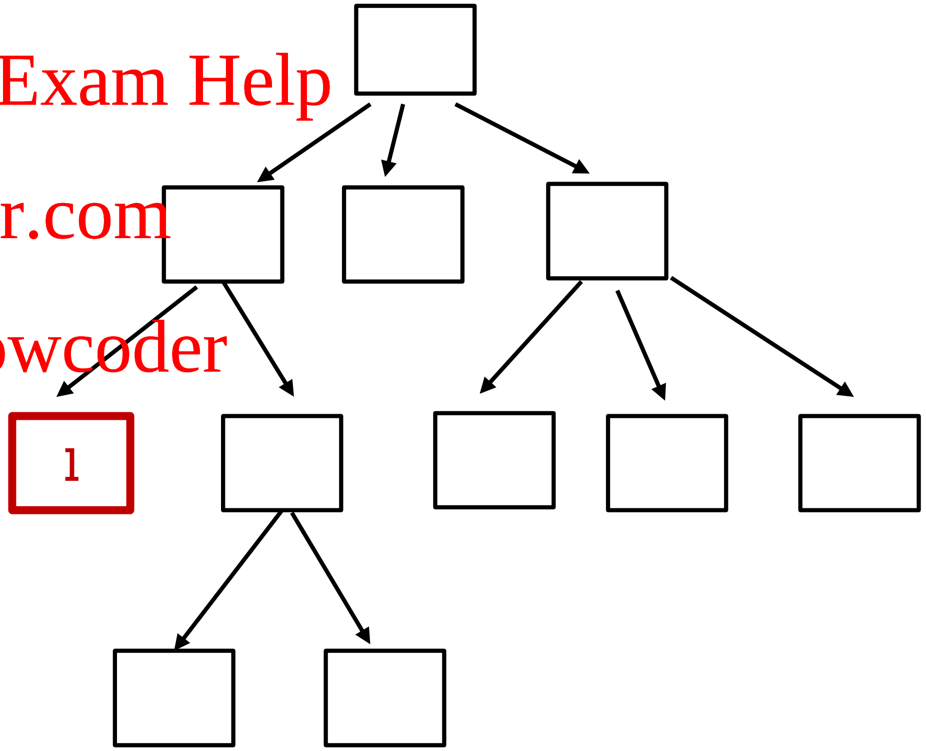
```
depthFirst (root){
    if (root is not empty){
        for each child of root
            depthfirst( child )
        visit root
    }
}
```



"postorder" traversal: visit the root after the children

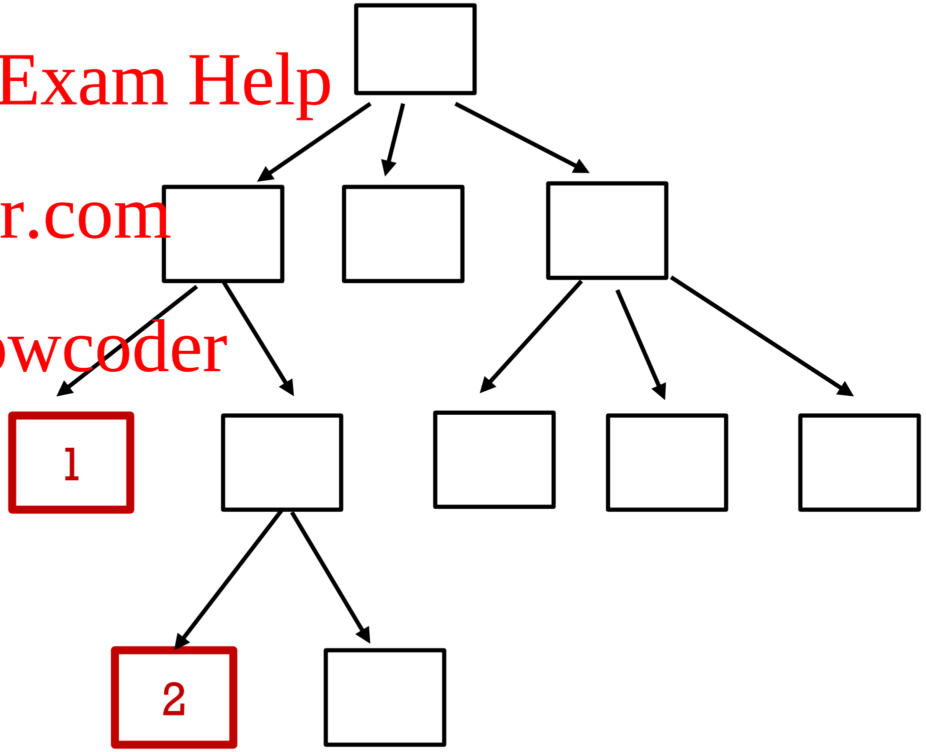
TREE TRAVERSAL – DEPTH FIRST "POSTORDER"

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



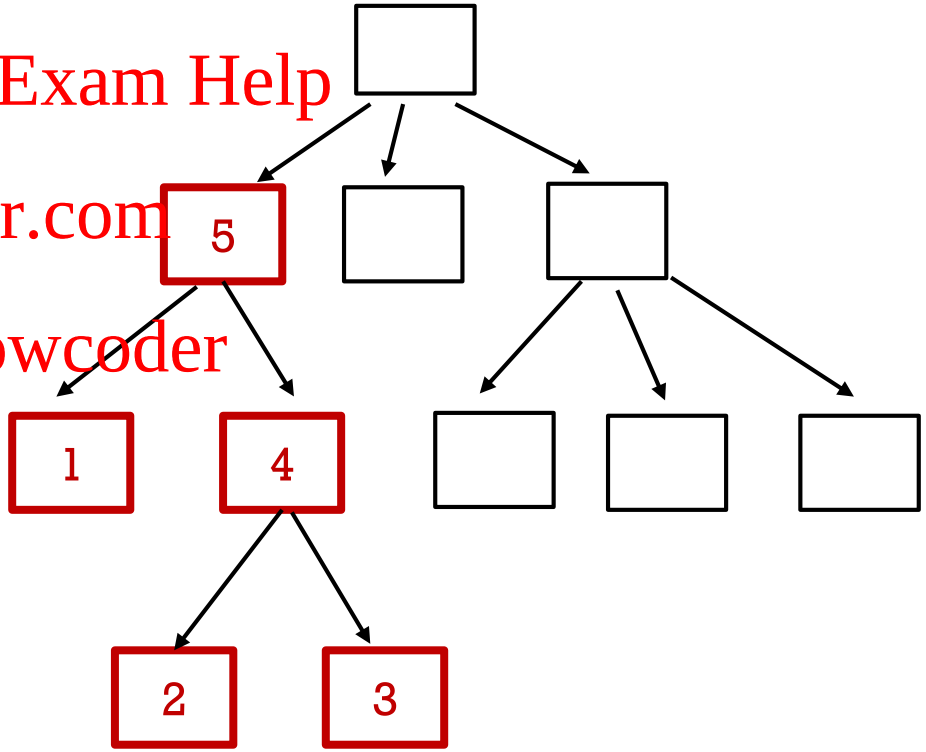
TREE TRAVERSAL – DEPTH FIRST “POSTORDER”

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



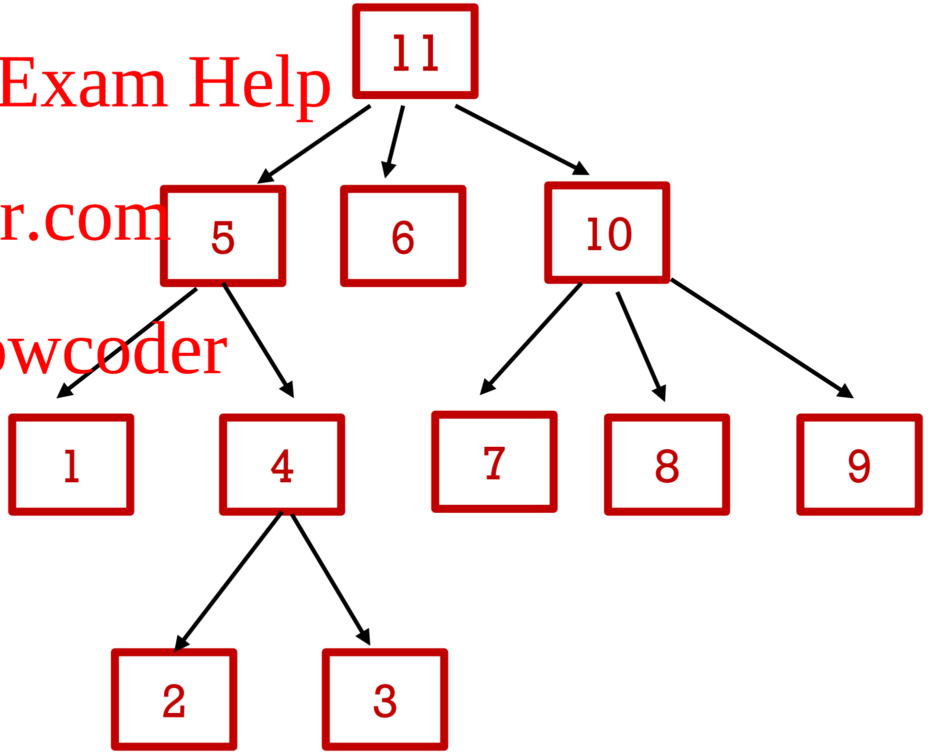
TREE TRAVERSAL – DEPTH FIRST "POSTORDER"

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

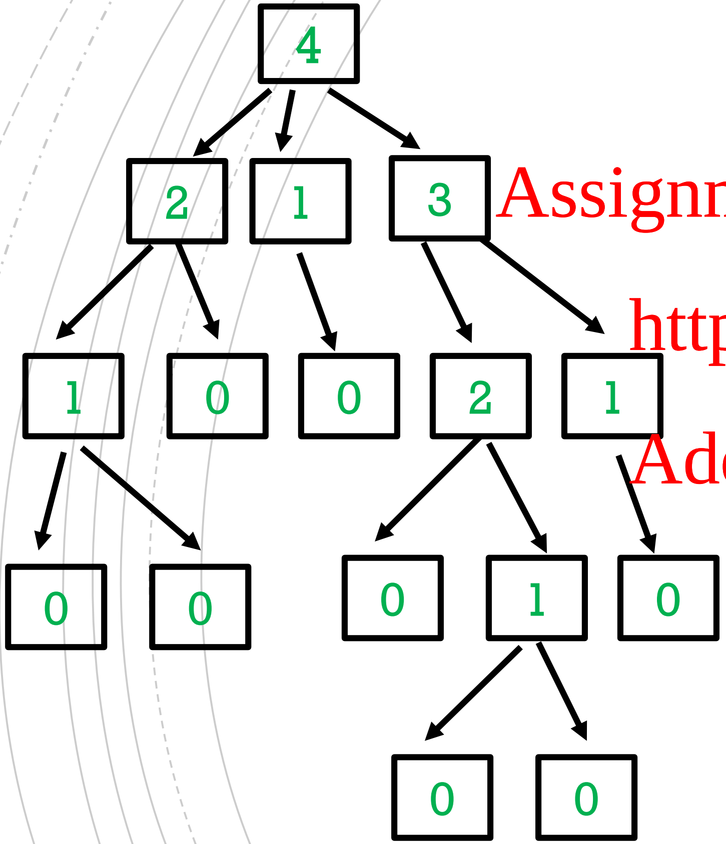


TREE TRAVERSAL – DEPTH FIRST “POSTORDER”

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



EXAMPLE 1 OF USING A POSTORDER TRAVERSAL: height(v)

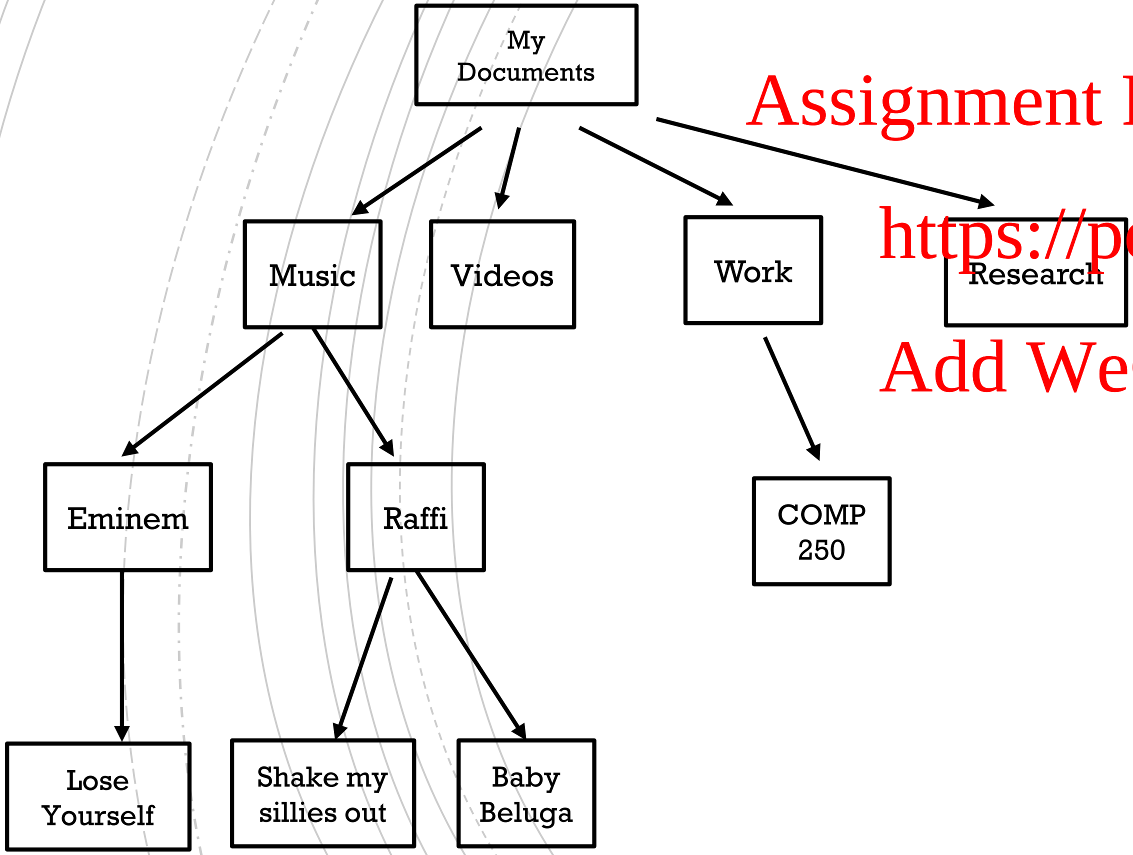


```
height(v){
   if (v is a leaf)
      return 0
   else{
      h = 0
      for each child w of v
         h = max(h, height(w))
      return 1 + h
   }
}
```

visit = return value of height

EXAMPLE 2 OF USING A POSTORDER TRAVERSAL

What is the total number of bytes in all files in a directory?



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

visit = determining the number of bytes for a node, e.g. If we were to store 'sum' at the node.

depthFirst() – PREORDER VS POSTORDER TRAVERSAL

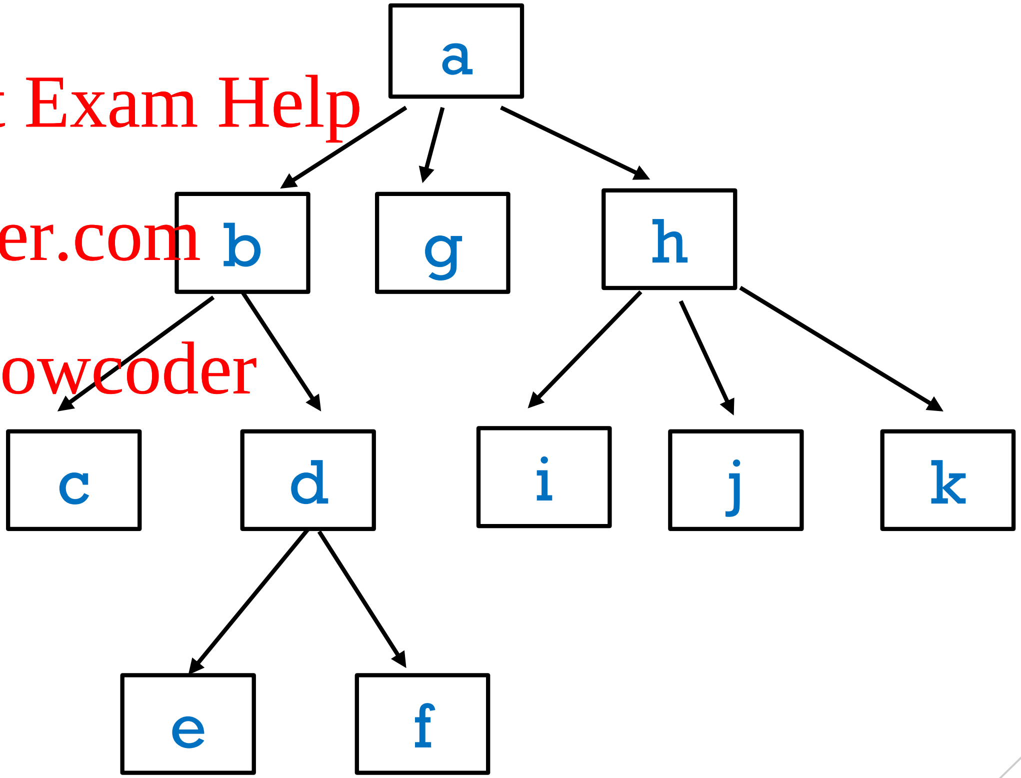
```
depthFirst (root){
    if (root is not empty){
        visit root
        for each child of root
            depthfirst( child )
    }
}
```

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

CALL SEQUENCE OF `depthFirst()`

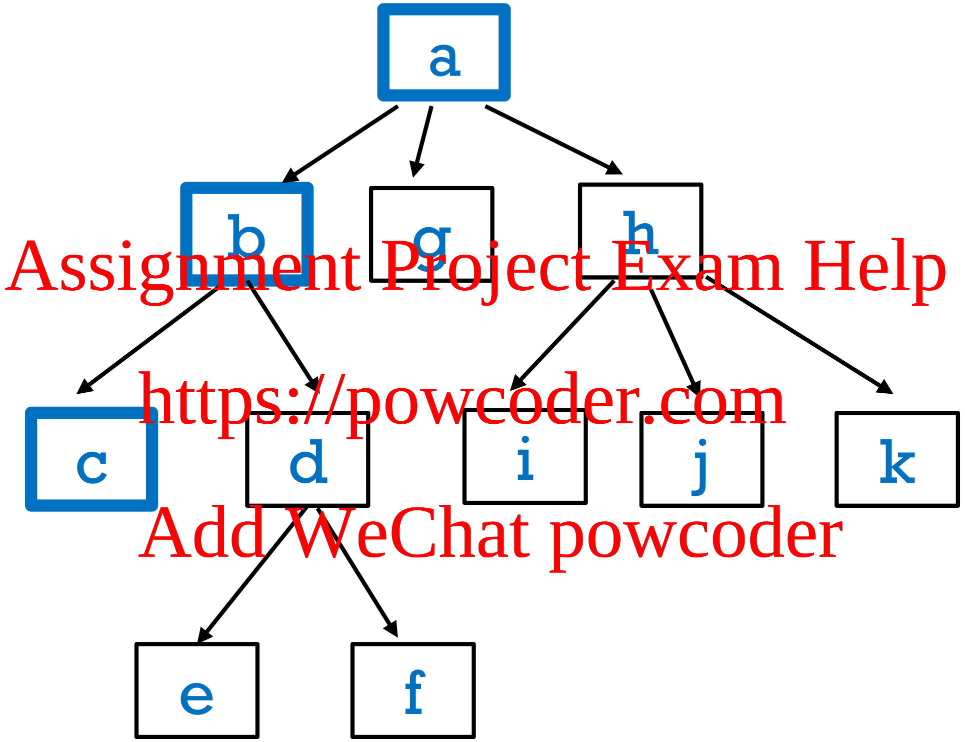
When we call `depthFirst(root)`, the same call sequence occurs for preorder vs postorder implementation.

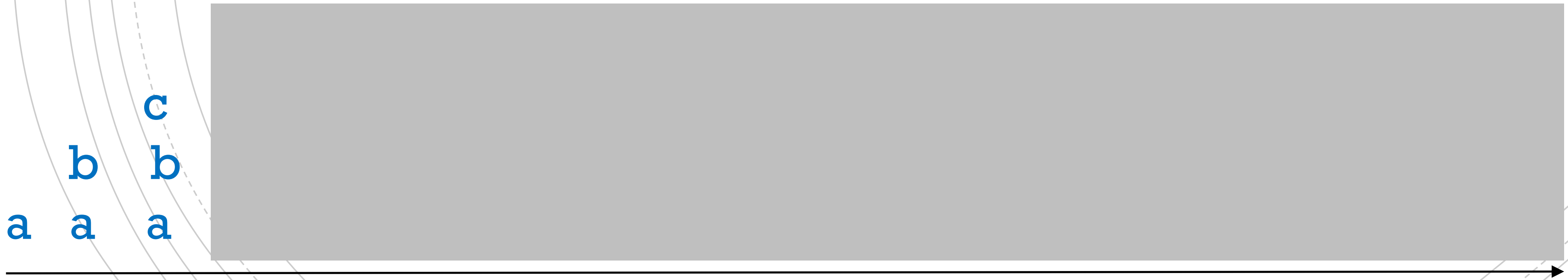
In the example on the rigth, the letter order corresponds to depthFirst(root) call order.



Note that the call stack stores information about the active method calls in a program.

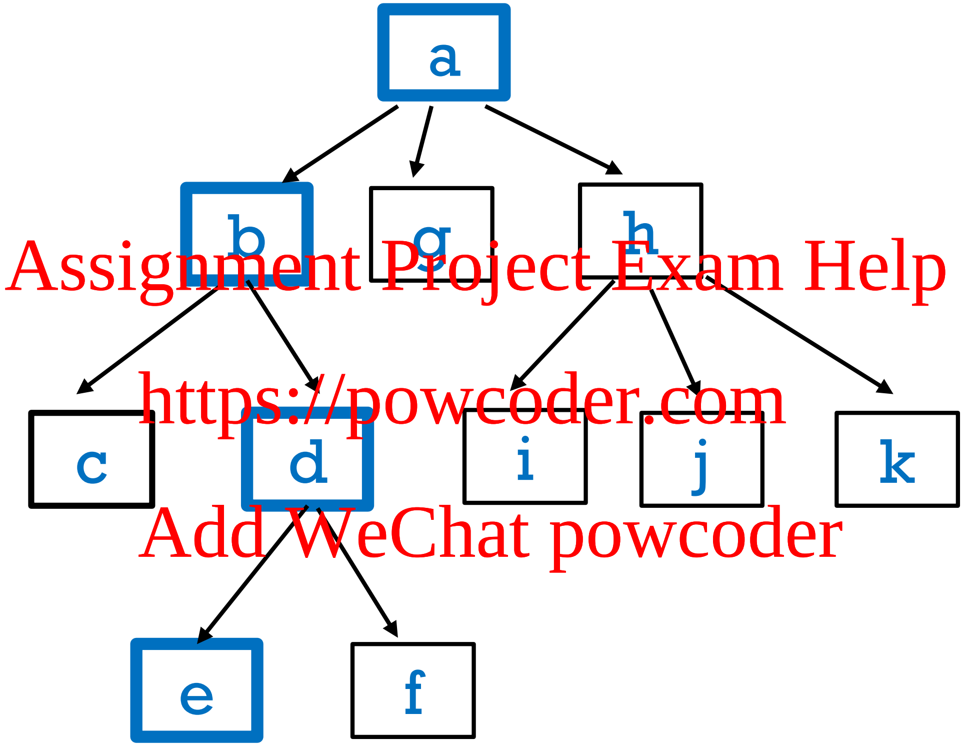
CALL STACK FOR depthFirst(root)

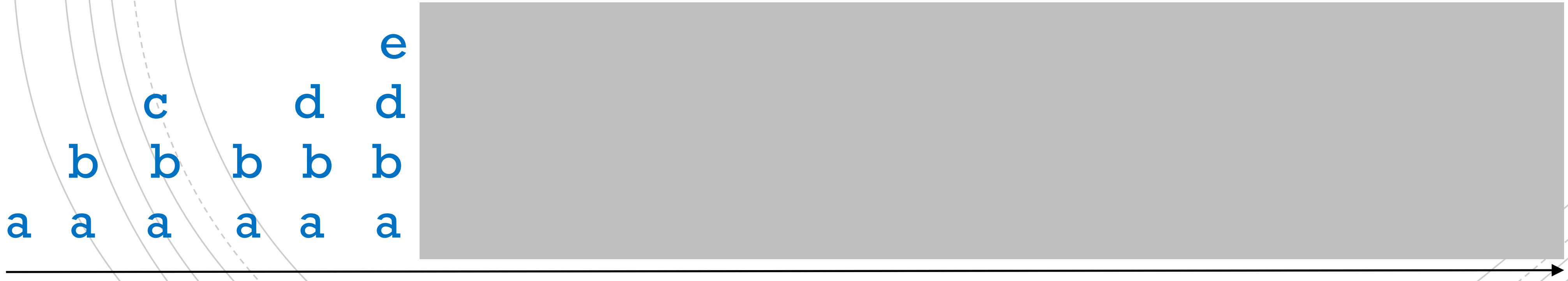




Note that the call stack stores information about the active method calls in a program.

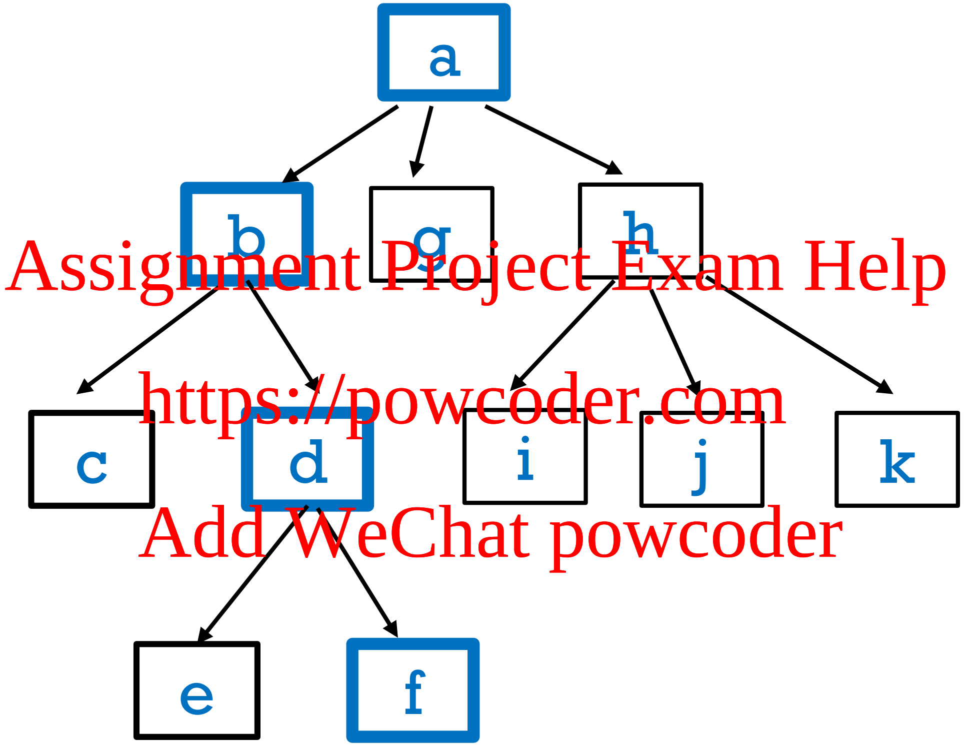
CALL STACK FOR depthFirst(root)





Note that the call stack stores information about the active method calls in a program.

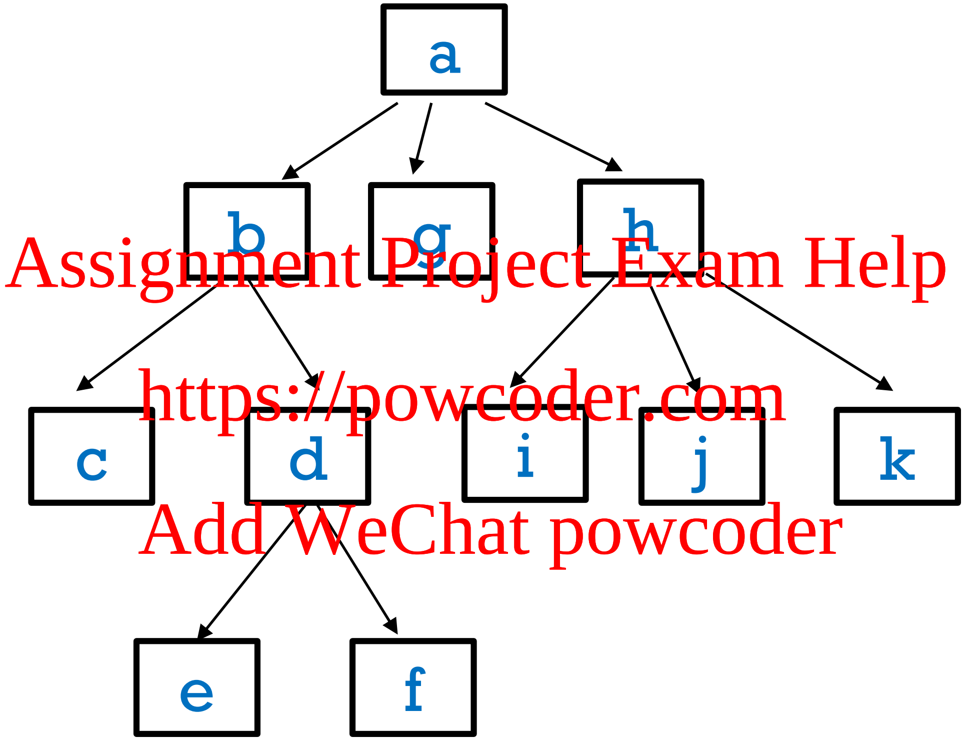
CALL STACK FOR depthFirst(root)



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

Note that the call stack stores information about the active method calls in a program.

CALL STACK FOR depthFirst(root)

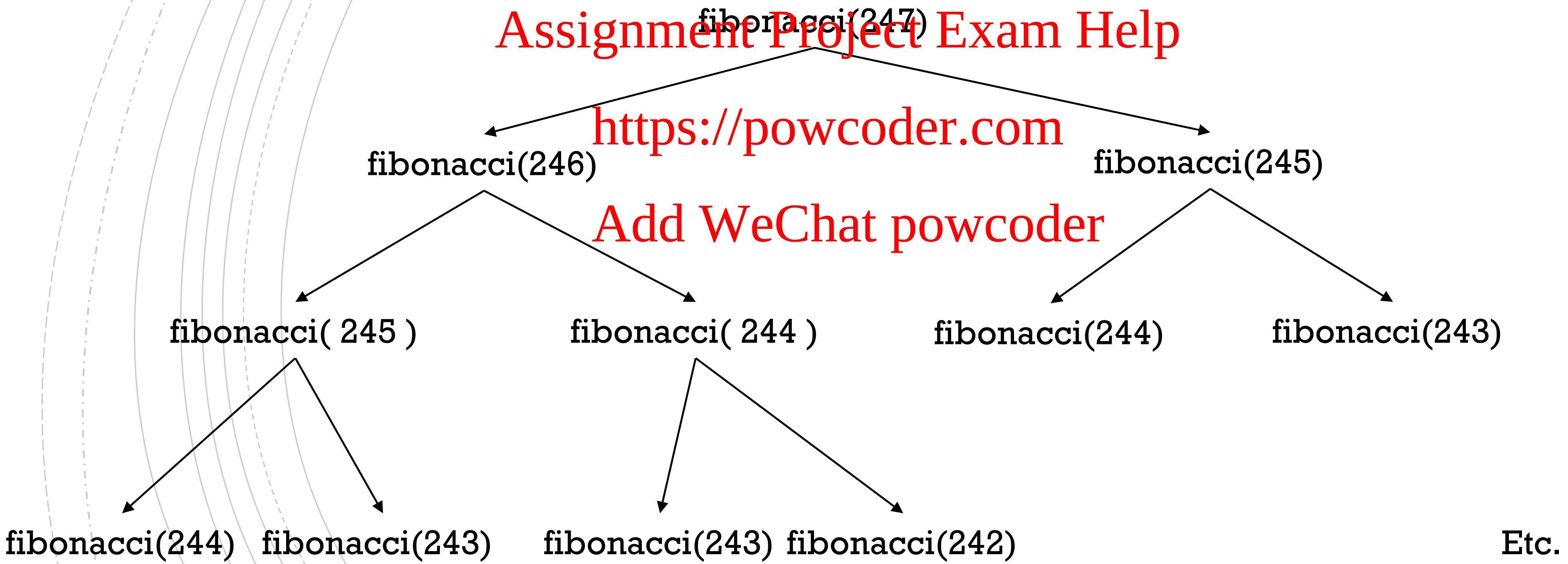


Notation: the letters indicate the call order of `depthFirst(root)`

					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

EXAMPLE

We used a tree to represent the call stack of the recursive Fibonacci method.



fibonacci(247)

fibonacci(246) fibonacci(245)

fibonacci(245) fibonacci(244) fibonacci(244) fibonacci(243)

fibonacci(244) fibonacci(243) fibonacci(243) fibonacci(242) Etc.

TREE TRAVERSAL IMPLEMENTATIONS

Recursive

- depth first (pre- versus post-order)

Non-Recursive

- using a stack
- using a queue

TREE TRAVERSAL – WITH A STACK

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







}
```

TREE TRAVERSAL – WITH A STACK

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



    }
}
```

TREE TRAVERSAL – WITH A STACK

```
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)
    }
}
```

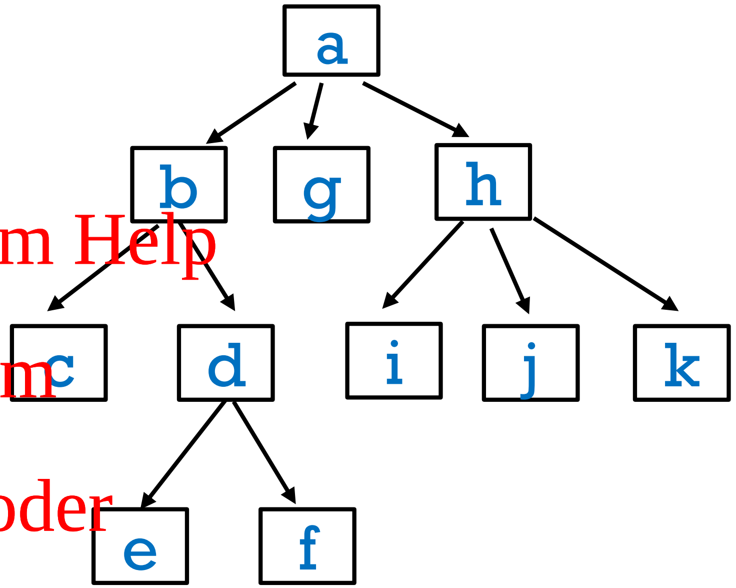
TREE TRAVERSAL – WITH A STACK

```
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 in which the nodes are visited?

TREE TRAVERSAL – WITH A STACK

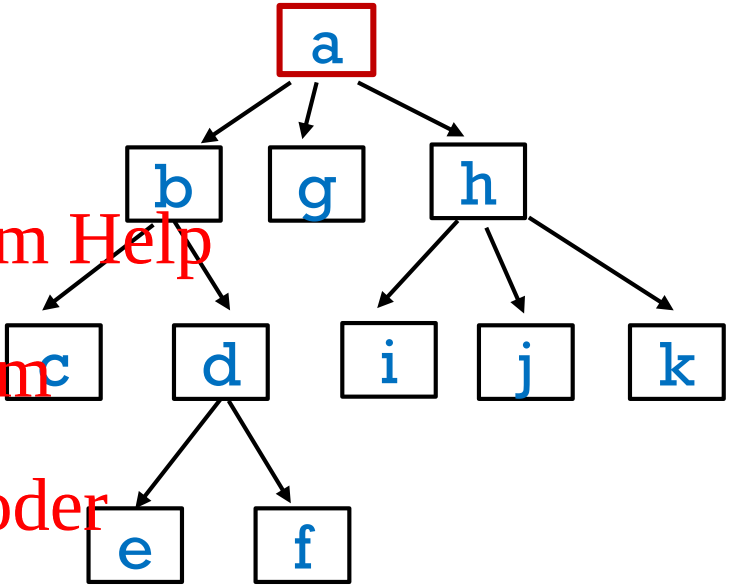
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```

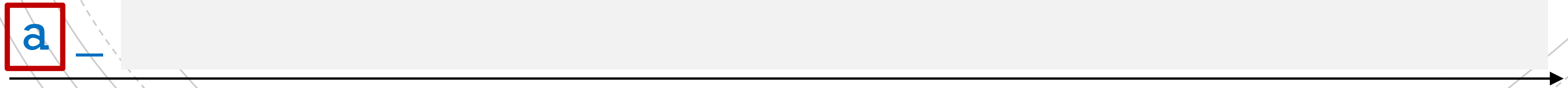


a

TREE TRAVERSAL – WITH A STACK

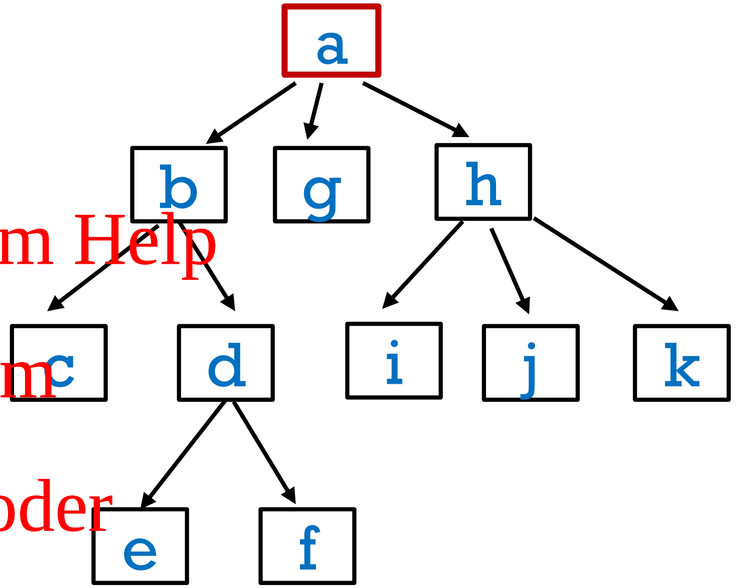
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```

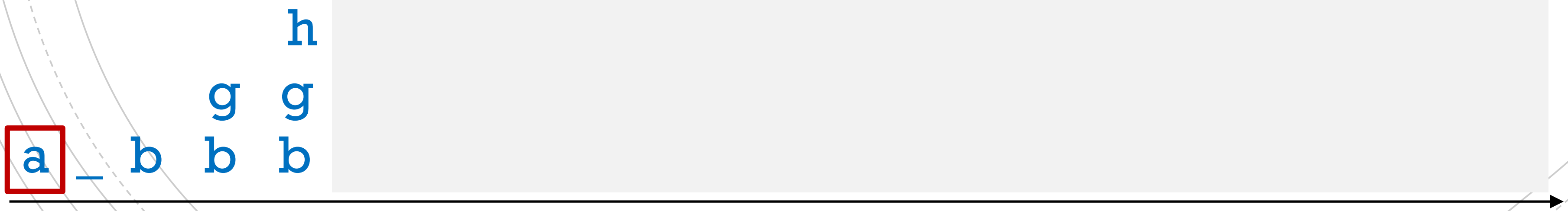




TREE TRAVERSAL – WITH A STACK

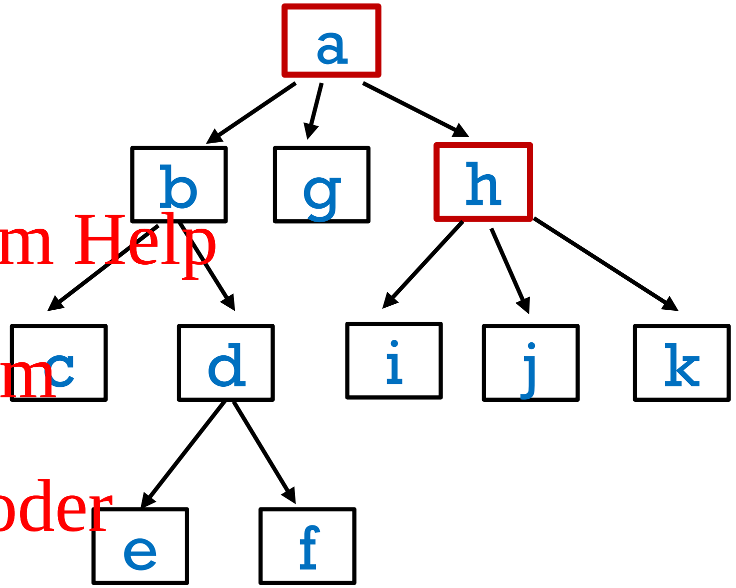
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```

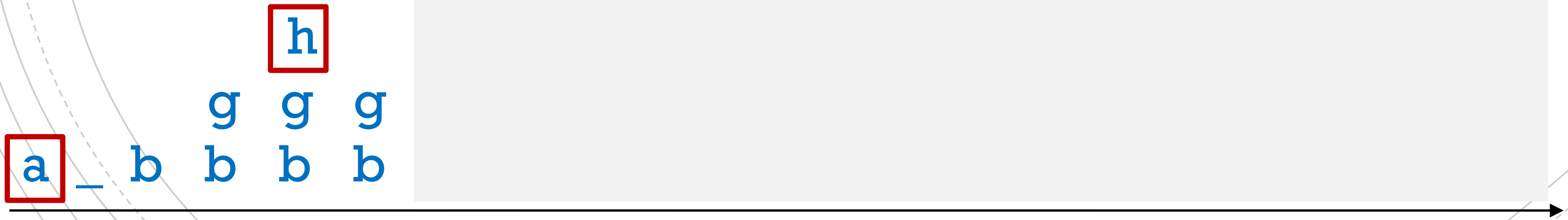




TREE TRAVERSAL – WITH A STACK

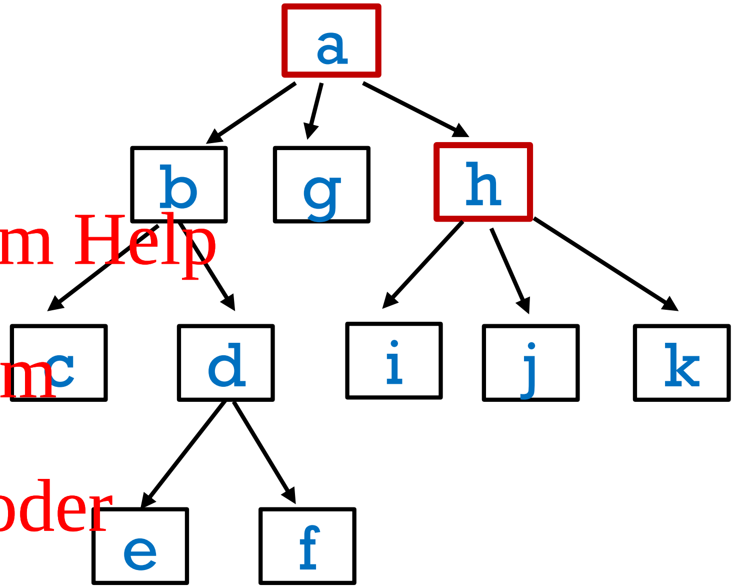
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```

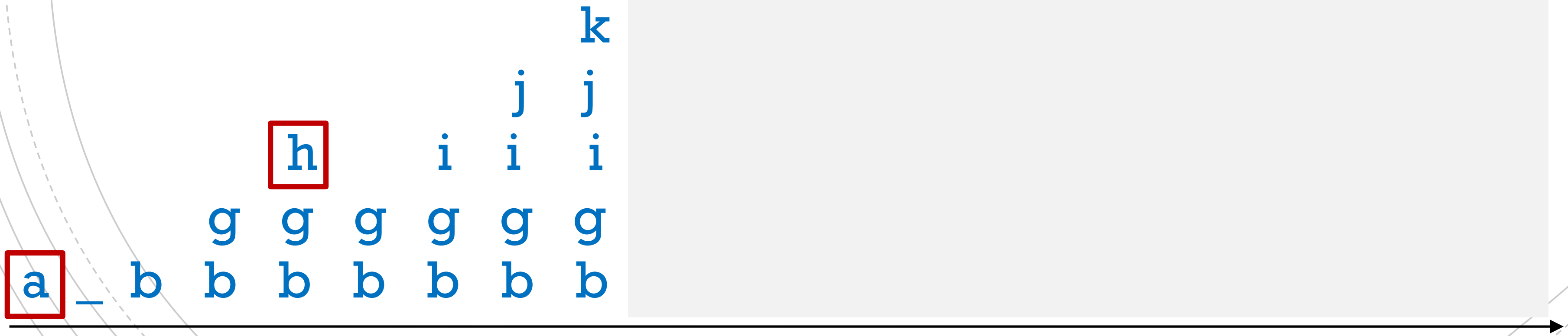




TREE TRAVERSAL – WITH A STACK

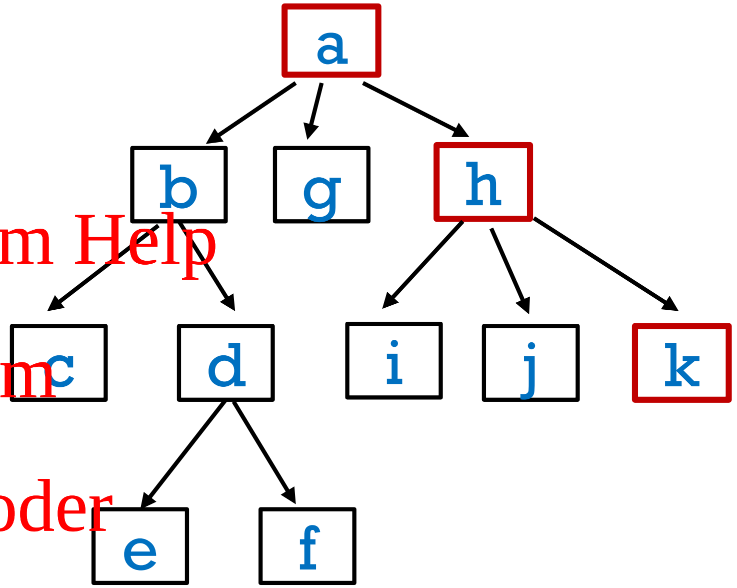
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```

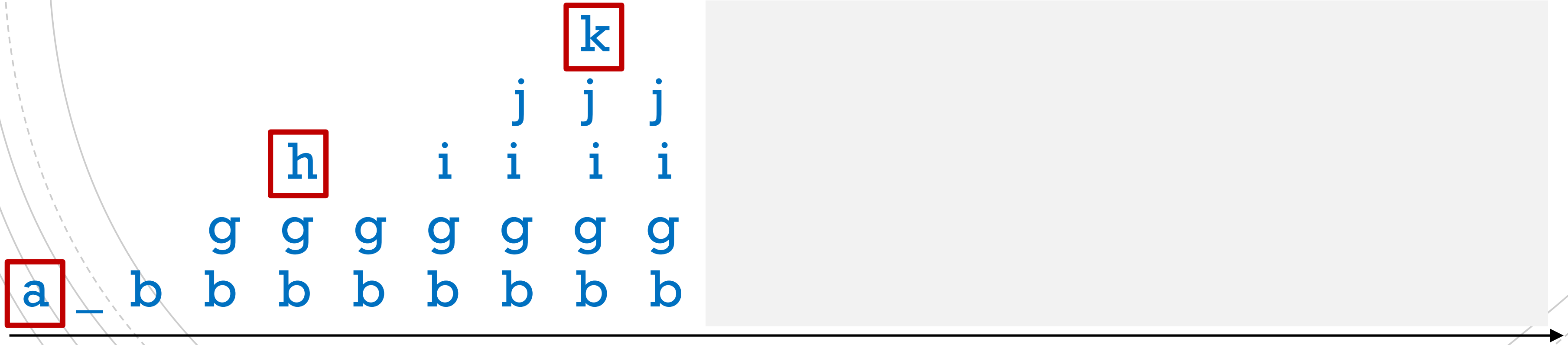




TREE TRAVERSAL – WITH A STACK

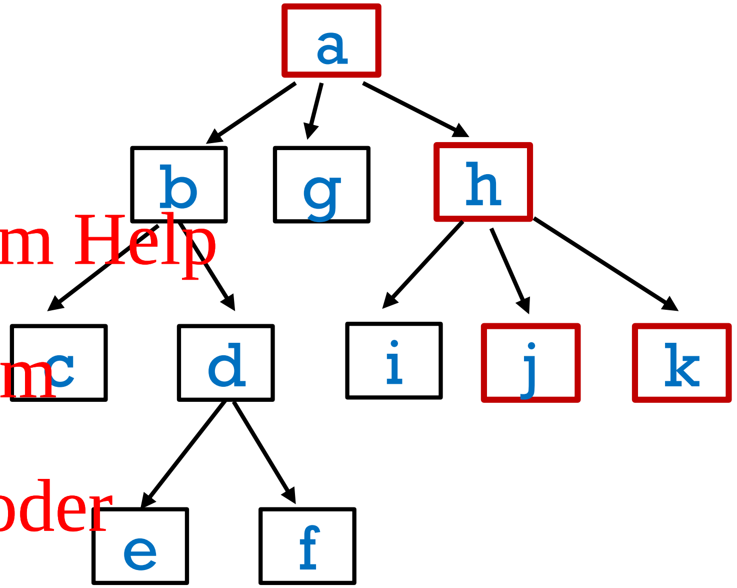
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```

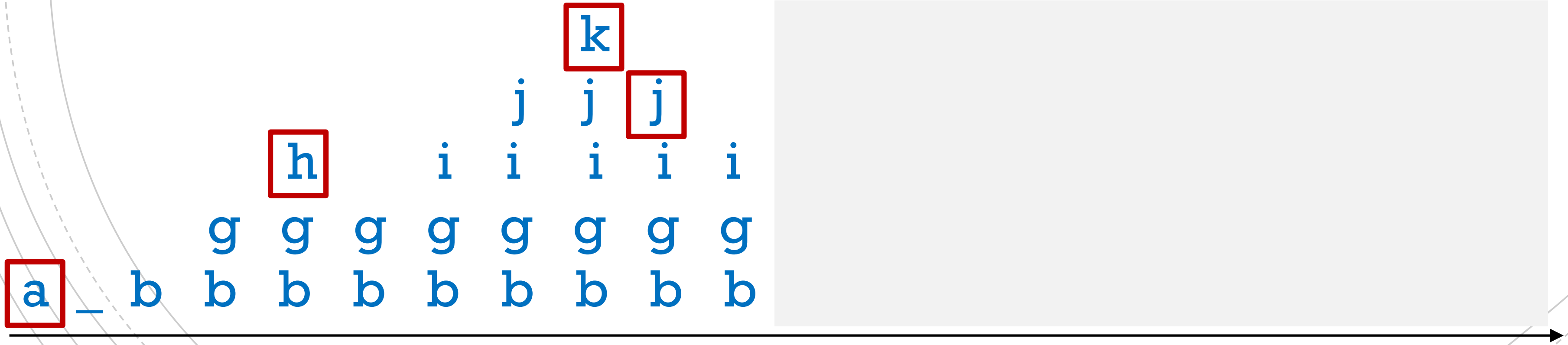




TREE TRAVERSAL – WITH A STACK

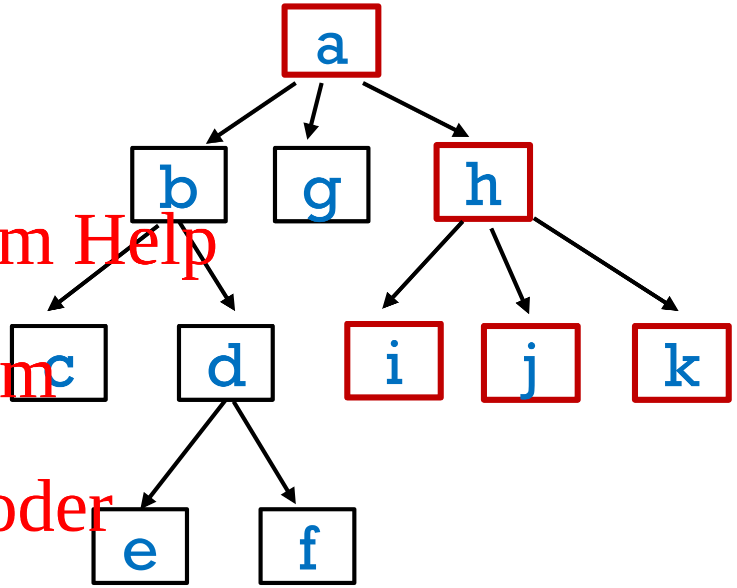
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```

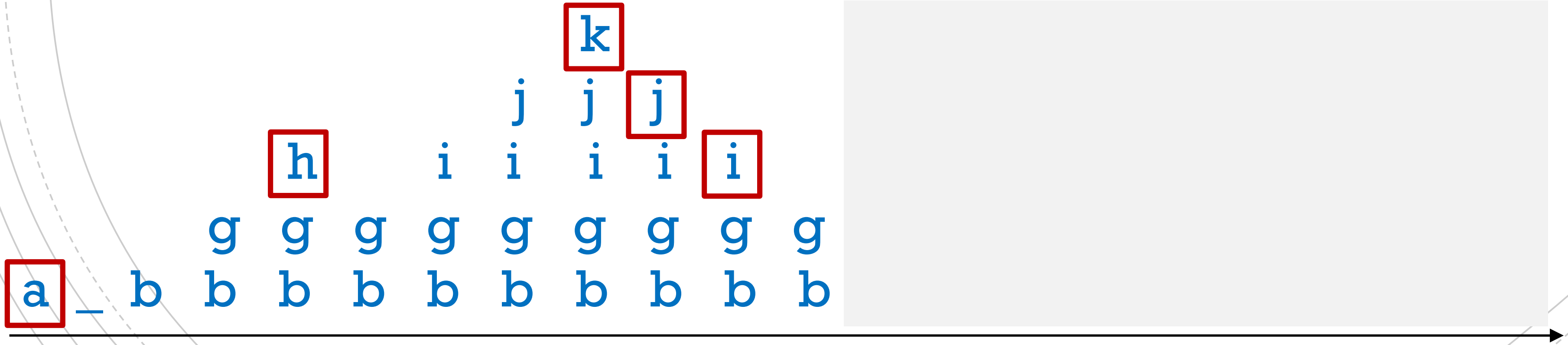




TREE TRAVERSAL – WITH A STACK

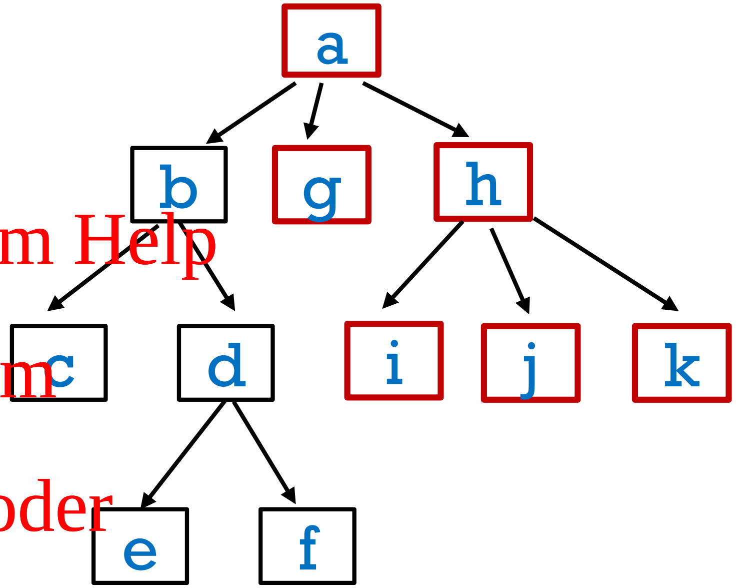
```
treeTraversalUsingStack(root){
   initialize empty stack s
   s.push(root)
   while s is not empty {
      cur = s.pop()
      visit cur
      for each child of cur
         s.push(child)
   }
}
```

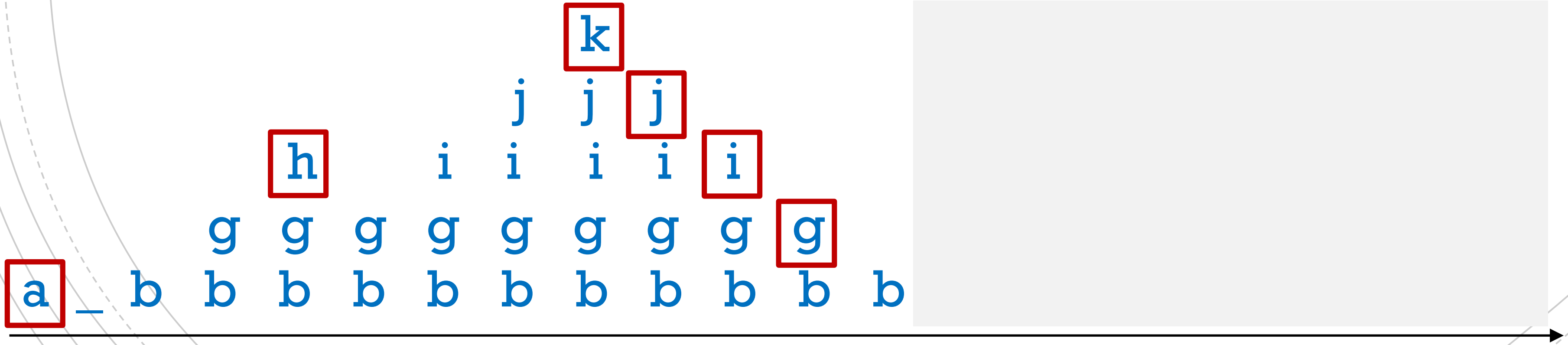




TREE TRAVERSAL – WITH A STACK

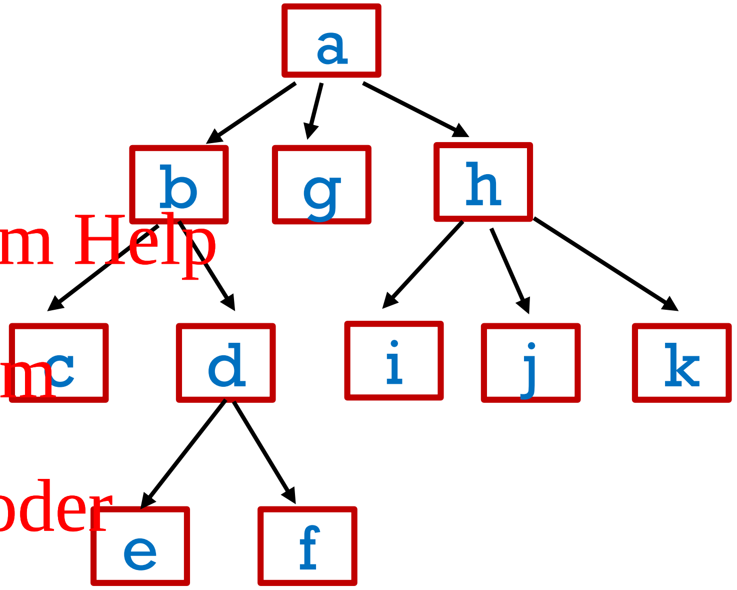
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```

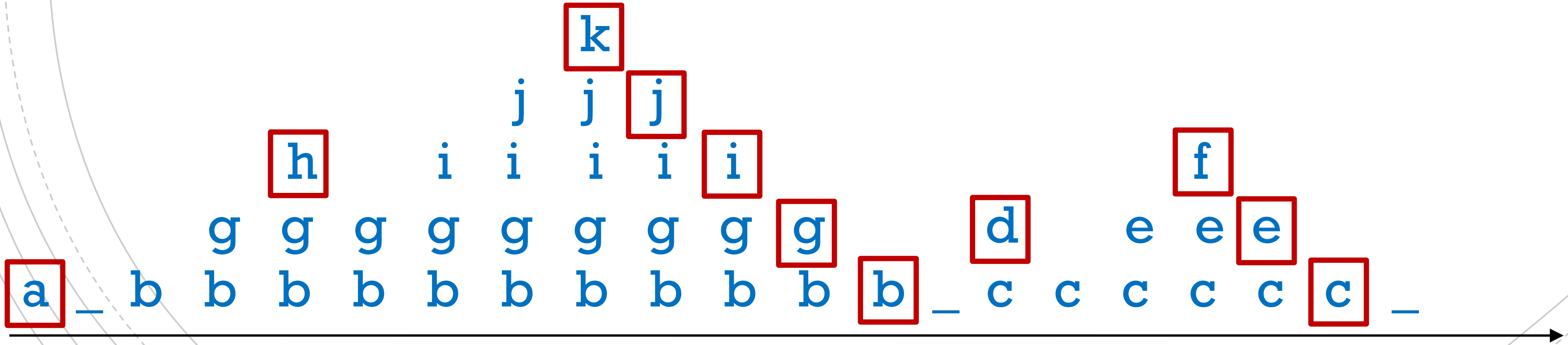




TREE TRAVERSAL – WITH A STACK

```
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)
   }
}
```

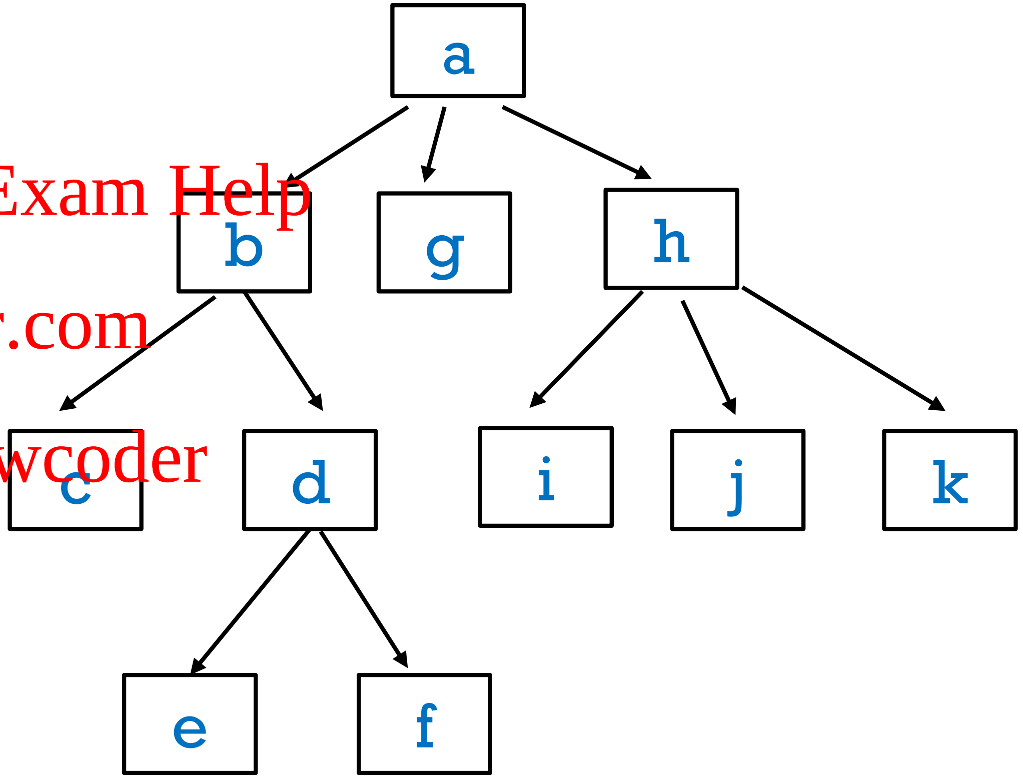



k
j j j
h i i i i i f
g g g g g g g g g d e e e
a _ b b b b b b b b b b b _ c c c c c c _

TREE TRAVERSAL – WITH A STACK

Q: Is it depth first?

A: Yes, but it visits the children "from right to left"



Recursive preorder: abcdefghijk

Recursive postorder: cefdbgijkha

Non-recursive (stack): ahkjigbdfec

TREE TRAVERSAL – WITH A STACK

Q: Is it preorder or postorder?

A: It’s preorder.

Q: Would move the visit change that?

A: No… why?

```
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)

   }
}
```

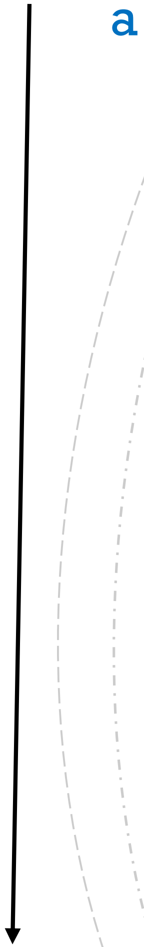
WHAT IF WE USED A QUEUE INSTEAD?

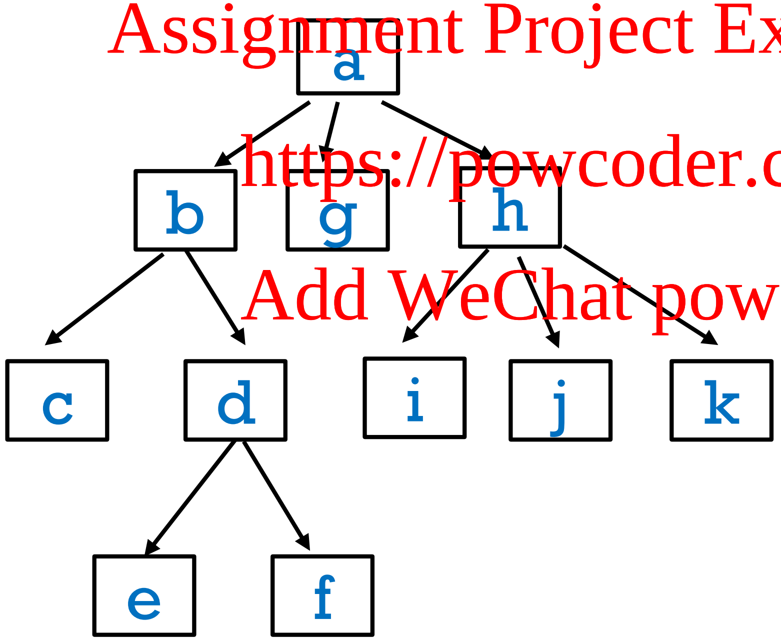
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```

```
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)
    }
}
```

WHAT IF WE USED A QUEUE INSTEAD?

Queue state at start of the while loop





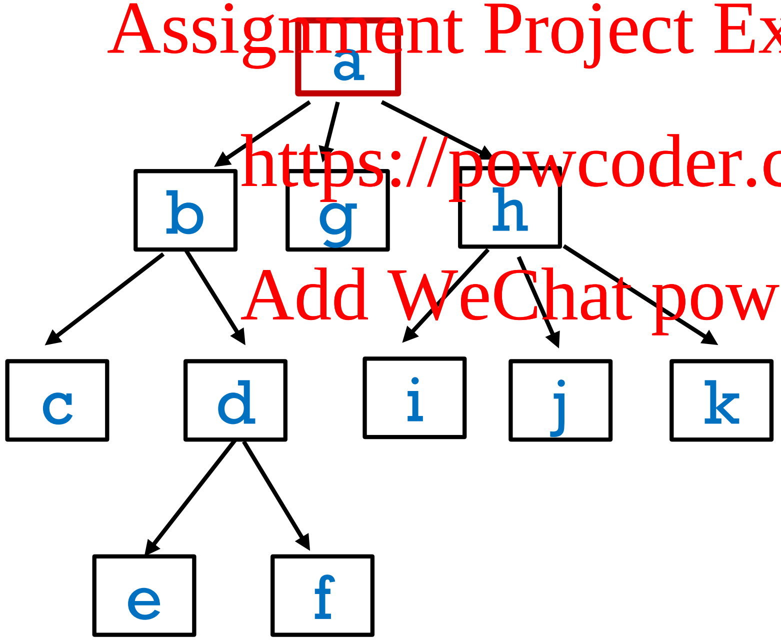
```
treeTraversalUsingQueue(root){
    initialize empty queue q
    q.enqueue(root)
    while s is not empty {
       cur = q.dequeue()
       visit cur
       for each child of cur
          q.enqueue(child)
    }
}
```

WHAT IF WE USED A QUEUE INSTEAD?

Queue state at start of the while loop

a

b g h



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

WHAT IF WE USED A QUEUE INSTEAD?

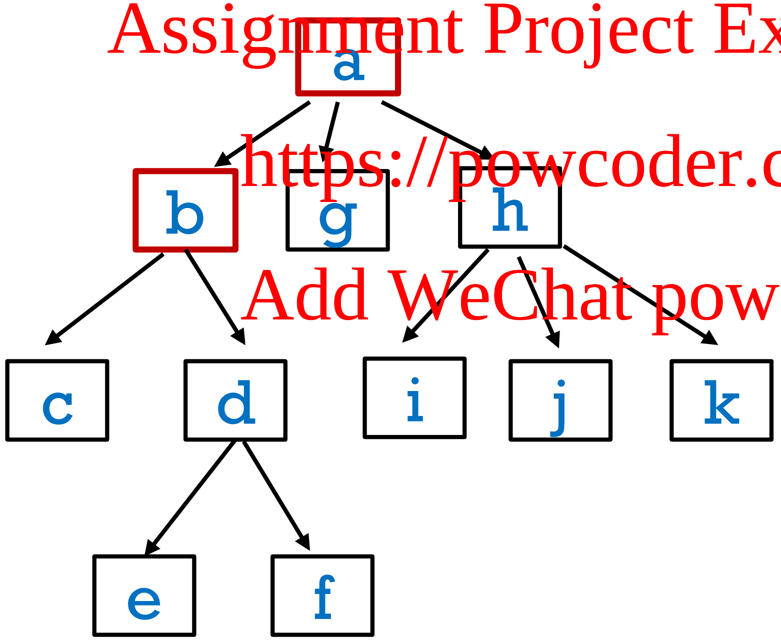
Queue state at start of the while loop

a

b g h

g h

g h c d



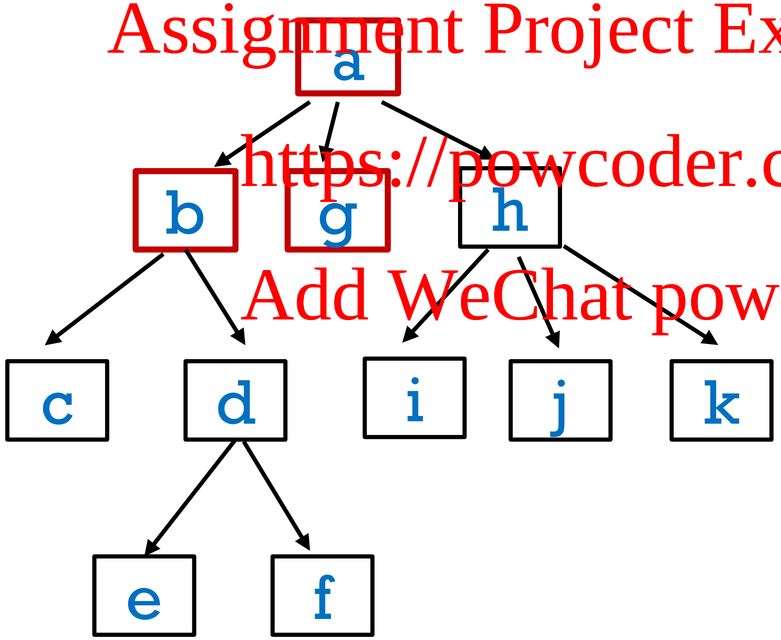
```
treeTraversalUsingQueue(root){
    initialize empty queue q
    q.enqueue(root)
    while s is not empty {
        cur = q.dequeue()
        visit cur
        for each child of cur
            q.enqueue(child)
    }
}
```

WHAT IF WE USED A QUEUE INSTEAD?

Queue state at start of the while loop

a

b g h
g h
g h c d
h c d



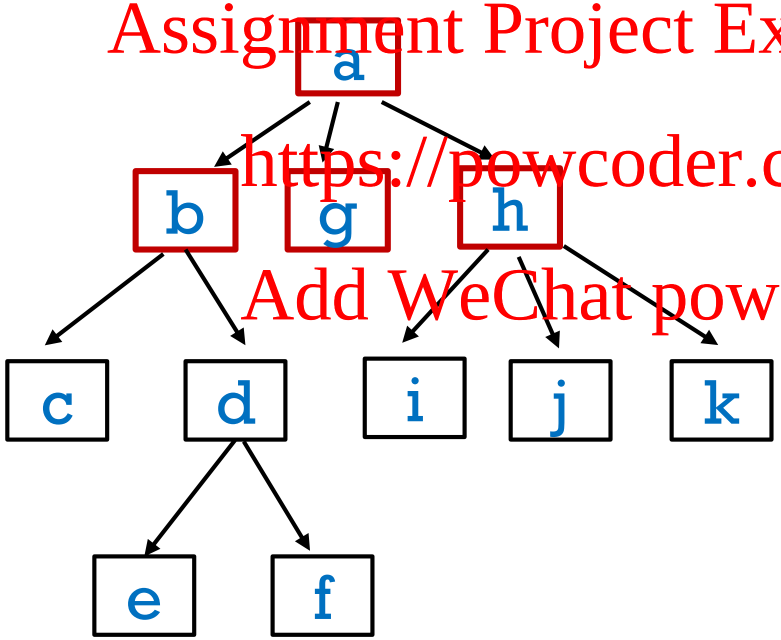
```
treeTraversalUsingQueue(root){
    initialize empty queue q
    q.enqueue(root)
    while s is not empty {
        cur = q.dequeue()
        visit cur
        for each child of cur
            q.enqueue(child)
    }
}
```

WHAT IF WE USED A QUEUE INSTEAD?

Queue state at start of the while loop

a

b g h
g h
g h c d
h c d
c d
c d i j k



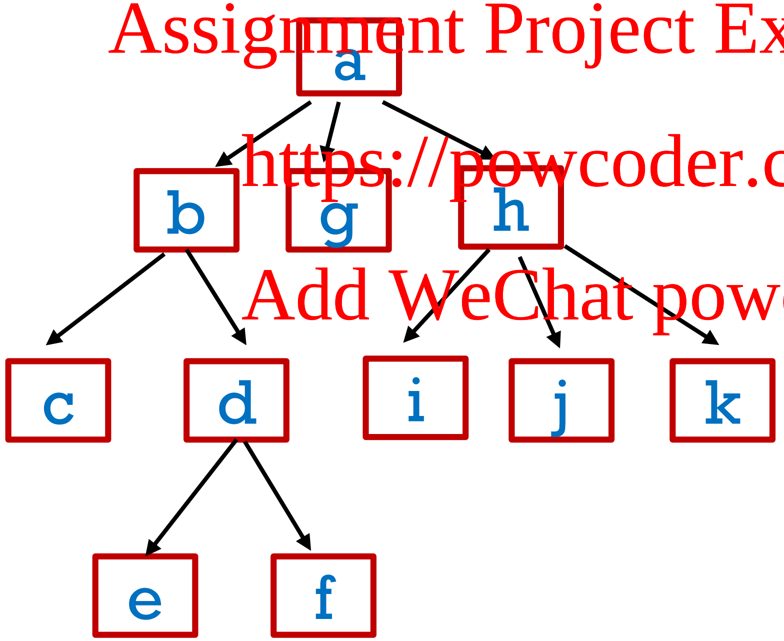
```
treeTraversalUsingQueue(root){
    initialize empty queue q
    q.enqueue(root)
    while s is not empty {
        cur = q.dequeue()
        visit cur
        for each child of cur
            q.enqueue(child)
    }
}
```

WHAT IF WE USED A QUEUE INSTEAD?

Queue state at start of the while loop

a

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



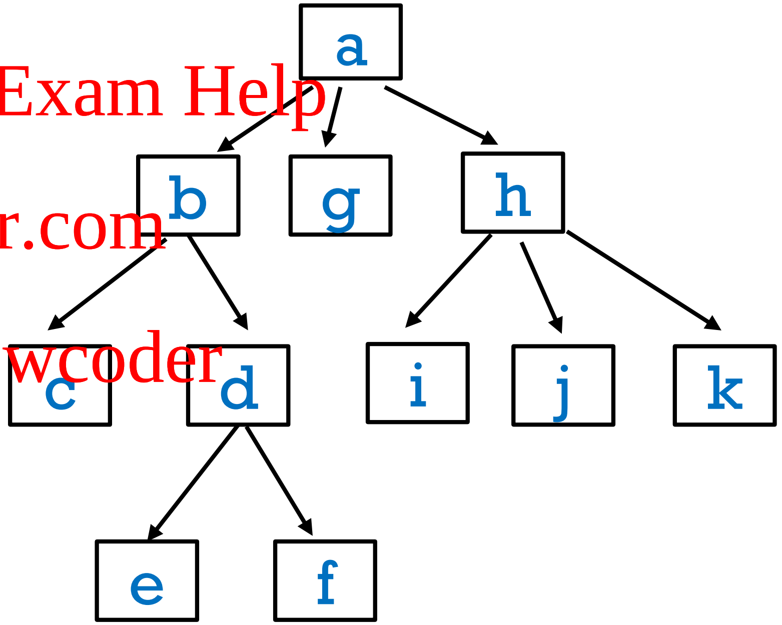
```
treeTraversalUsingQueue(root){
    initialize empty queue q
    q.enqueue(root)
    while s is not empty {
        cur = q.dequeue()
        visit cur
        for each child of cur
            q.enqueue(child)
    }
}
```

BREADTH FIRST TRAVERSAL

For each level i

visit all nodes at level i

Order visited: abghcdijkef

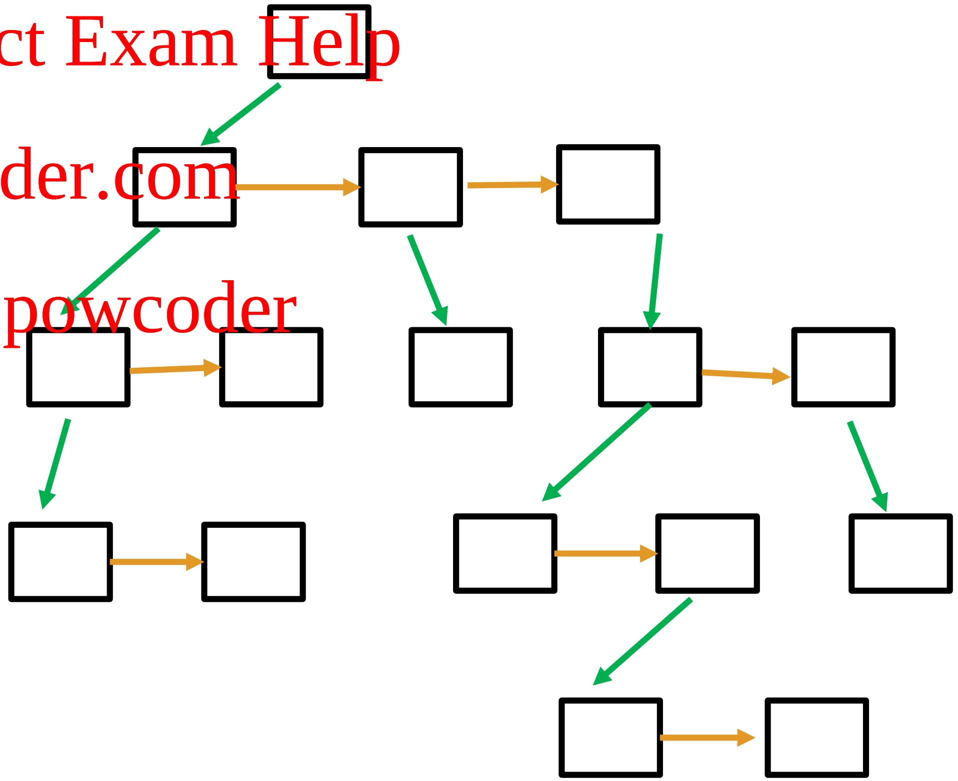


IMPLEMENTATION DETAILS

Recall the "first child, next sibling" implementation

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

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



IMPLEMENTATION DETAILS

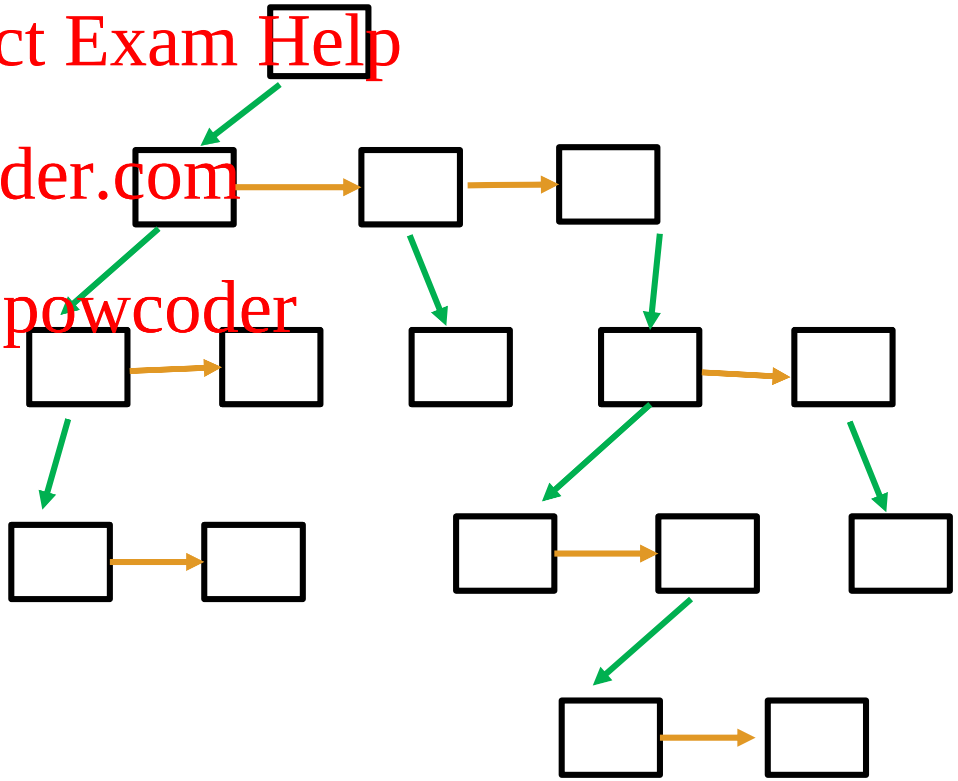
Recall the "first child, next sibling" implementation

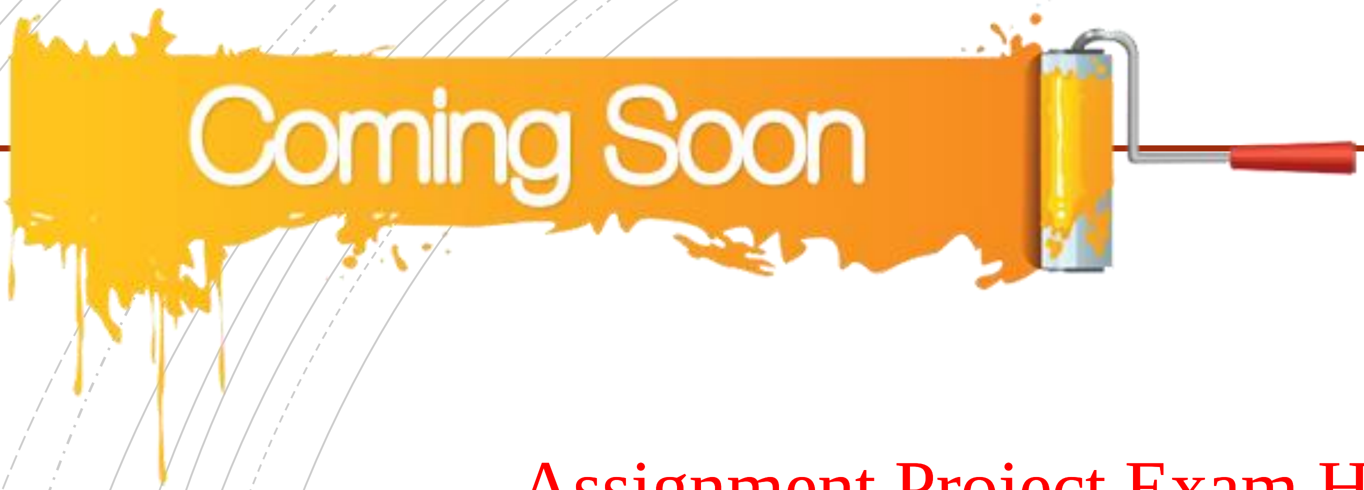
Then when we write

```
for each child {
        :
}
```

it means

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





In the next video:

- Binary Trees