Tree Data Structures

Data Structures for Computer Professionals

Patiwet Wuttisarnwattana, Ph.D.

patiwet@eng.cmu.ac.th

Computer Engineering, Chiang Mai University

Why do we need Tree?

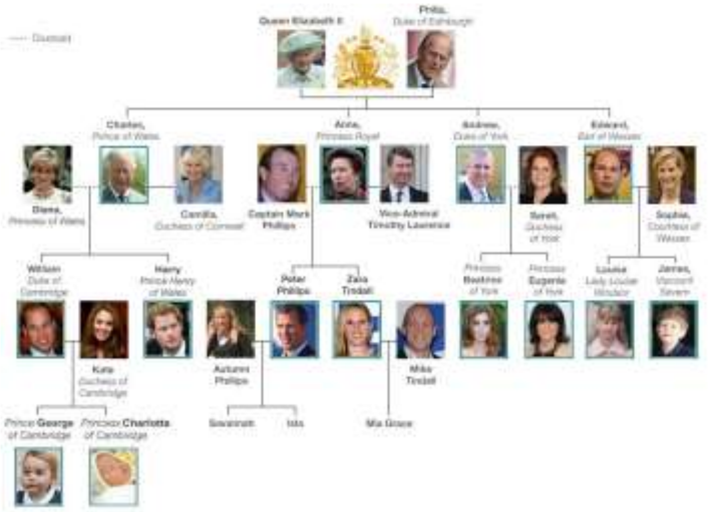
- Lists, Stacks, and Queues are linear relationships
  - Accessing time can be super fast O(1)
  - But searching for a key takes O(n)
  - Some say this is too slow
  - Can you find something that the search is faster than O(n)
- Well, Tree can be the answer

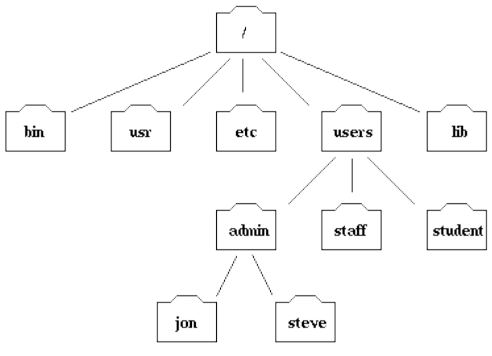
What is Tree?

- What is Tree
  - Tree consists of nodes that contain data
  - Tree can contain no node (empty)
  - **A node can point to one or more nodes**
  - A node can point to null (leaf node)
  - A node that has no other node point to it; is called **root** node
  - **Tree contains no loop/cycle of nodes**

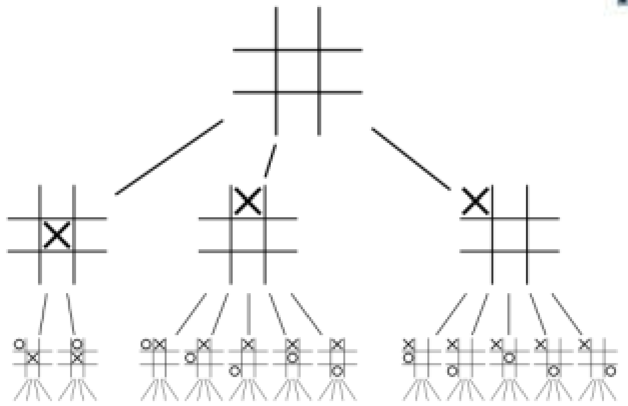
Trees

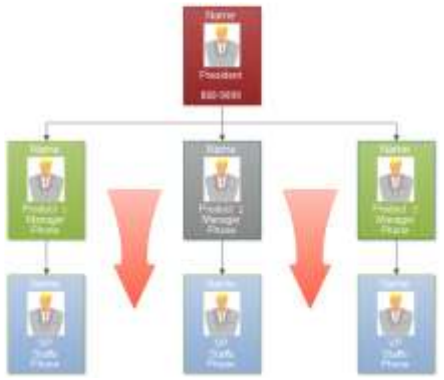
- Information often contains hierarchical relationships
  - Family
  - File directories or folders
  - Moves in game
  - Hierarchies in organizations





Part of the filesystem tree

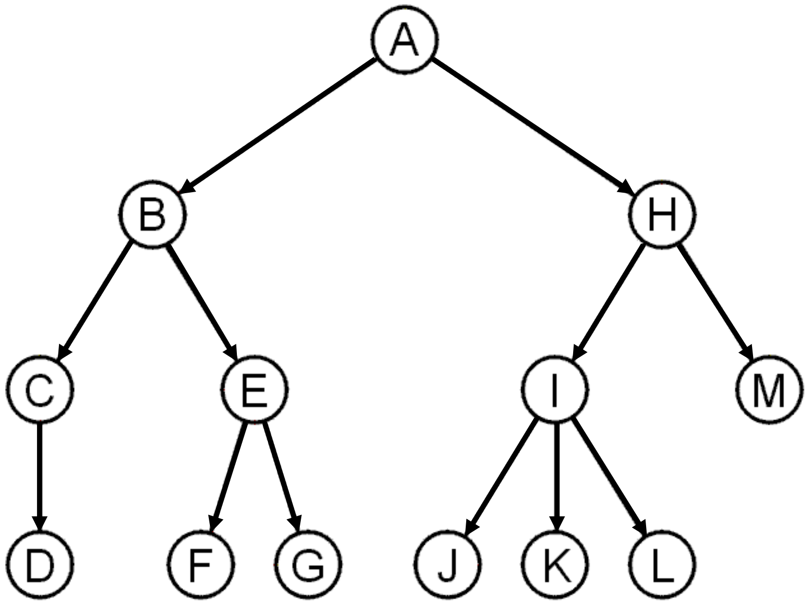




Tree Terminology

A tree data structure stores information in *nodes*

- Similar to linked lists:
  - There is a first node, or *root*
  - Each node has variable number of references to successors
  - Each node, other than the root, has exactly one node pointing to it
  - In the diagram, the line that connected between nodes is called edge
    - The edge should be directed (either be unidirected or bidirected)



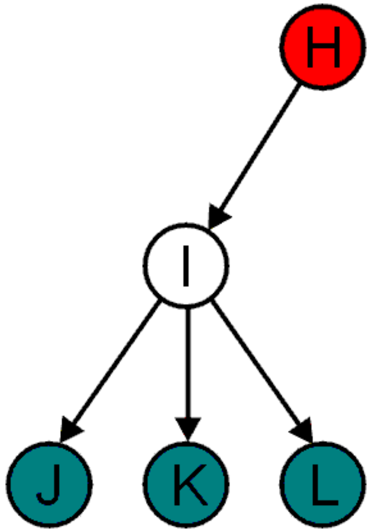
Parent and Child

All nodes will have zero or more child nodes or *children*

- I has three children: J, K and L

For all nodes other than the root node, there is one parent node

- H is the parent I

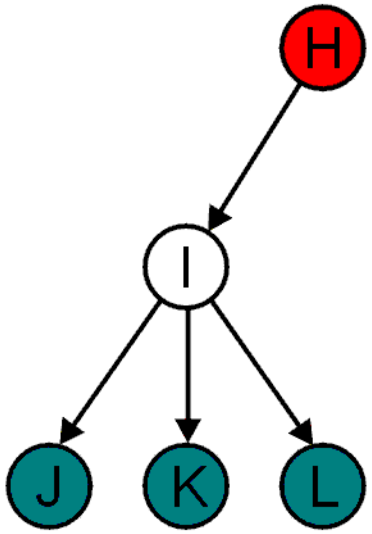


Node Degree

The *degree* of a node is defined as the number of its children:
deg(I) = 3

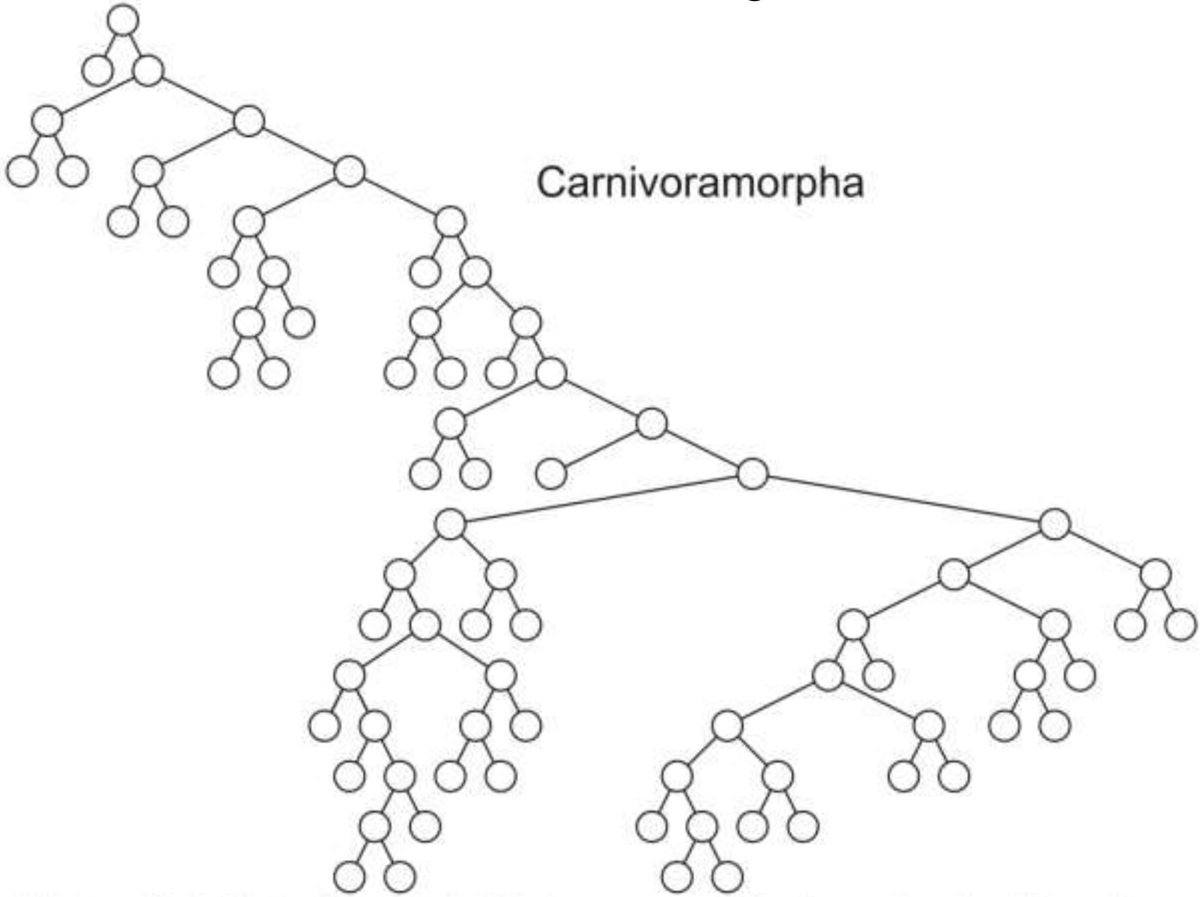
Nodes with the same parent are *siblings*

– J, K, and L are siblings



Phylogenetic tree

Phylogenetic trees have nodes with degree 2 or 0:

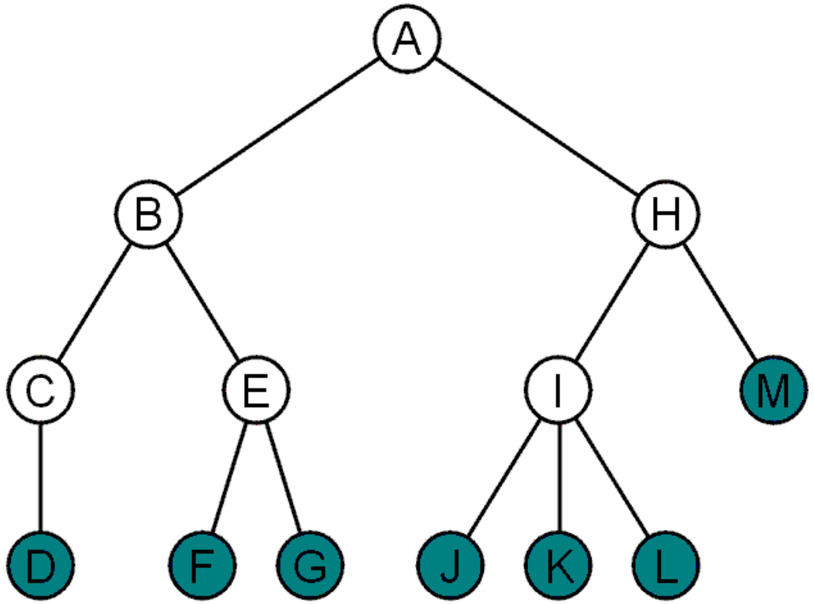


Wesley-Hunt, G. D.; Flynn, J. J. "Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of 'Miacoidea'

Leaf Nodes and Internal Nodes

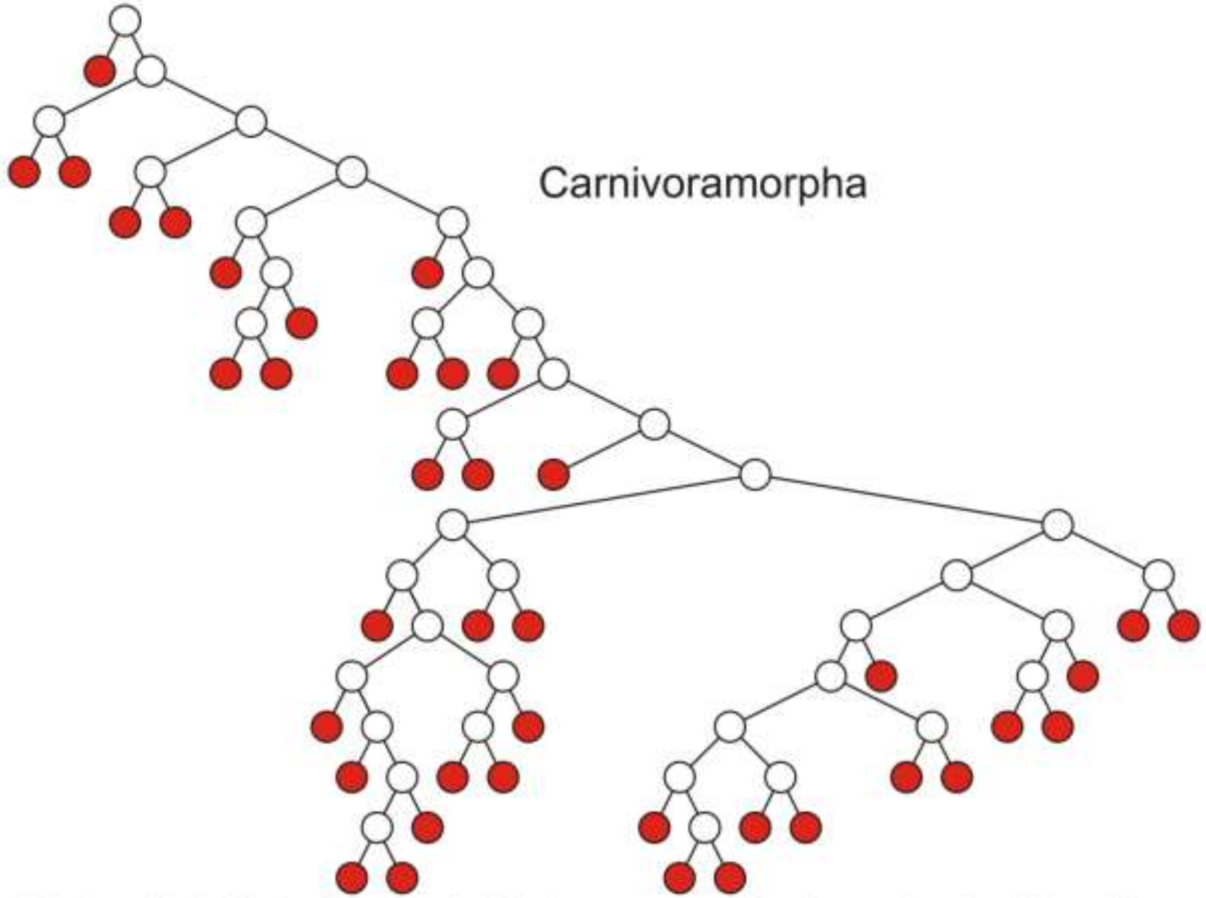
Nodes with degree zero are also called *leaf nodes*

All other nodes are said to be *internal nodes*, that is, they are internal to the tree



Leaf Nodes

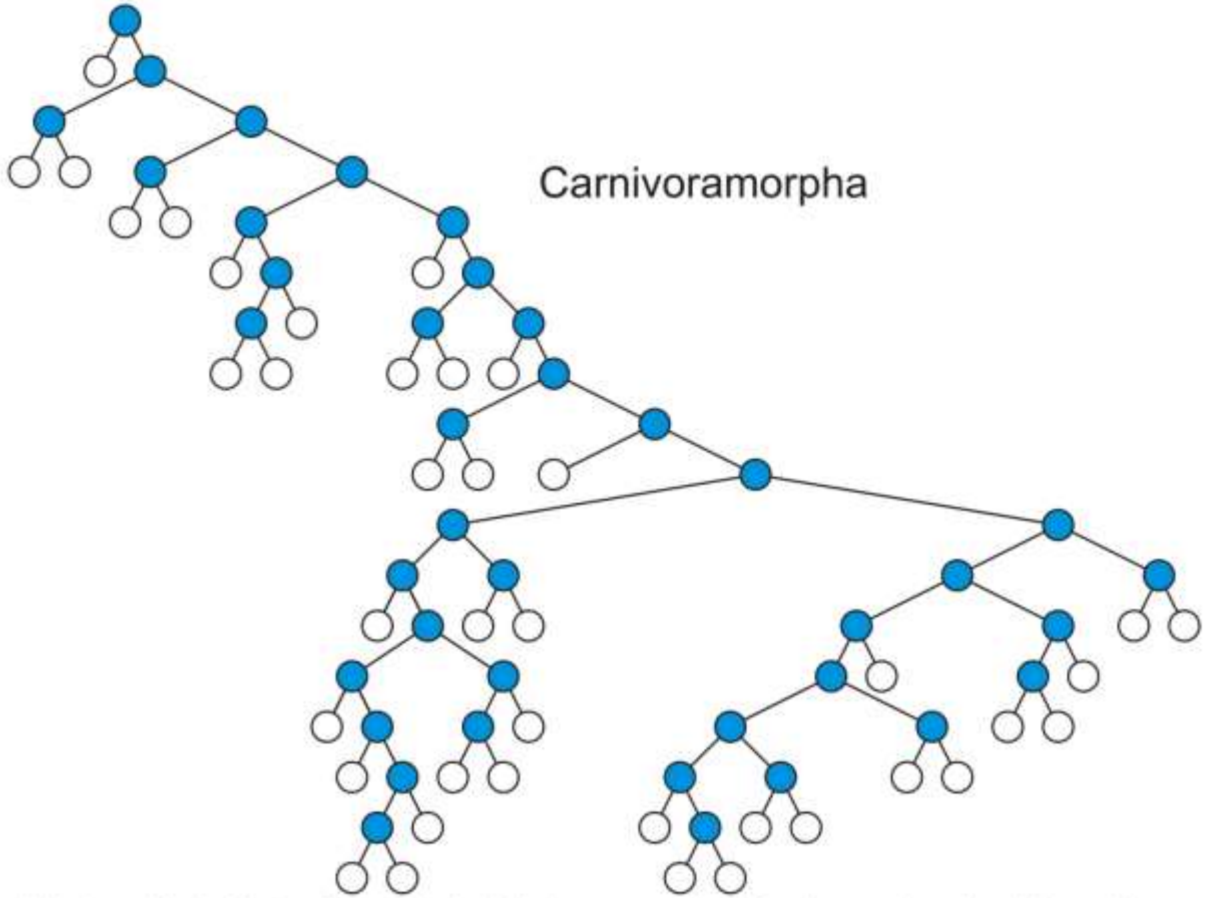
Leaf nodes:



Wesley-Hunt, G. D.; Flynn, J. J. "Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of 'Miacoidea'

Internal Nodes

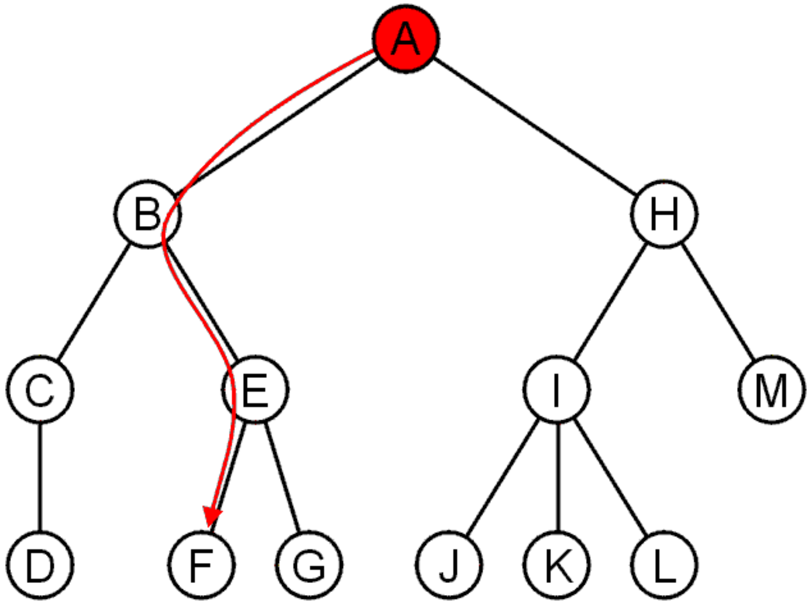
Internal nodes:



Wesley-Hunt, G. D.; Flynn, J. J. "Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of 'Miacoidea'

Root node

- The shape of a rooted tree gives a natural flow from the *root node*, or just *root*
- Every node grows from the root
- Tree in Data Structure grows upside down



Path

A **path** is a sequence of nodes

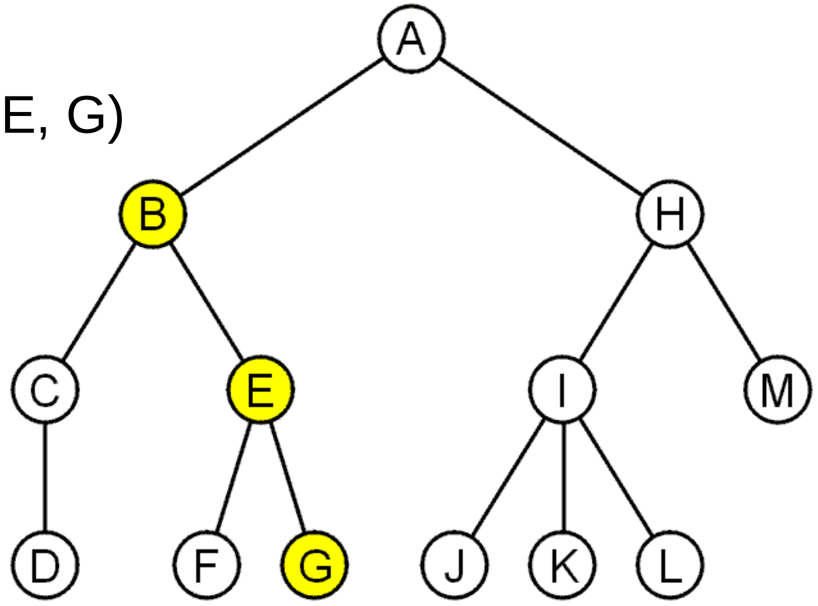
$(a_0, a_1, ..., a_n)$

where a_{k+1} is a child of a_k is

The **length** of this path is n

E.g., the path (B, E, G) has length 2

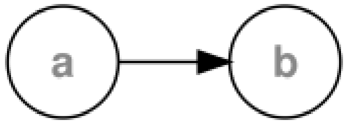
A tree with N nodes always has N-1 edges



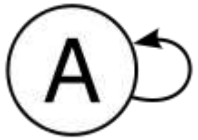
Loops in Tree?

- Two nodes in a tree have at most one path between them
- Can a non-zero path from node N reach node N again?

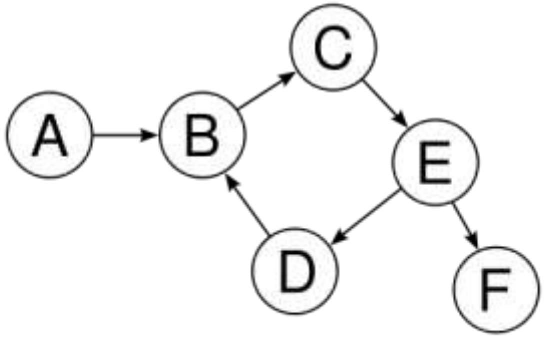
No. Trees can never have cycles (loops)



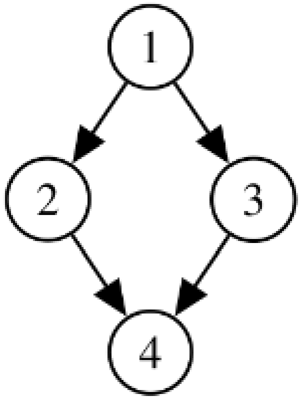
Tree?



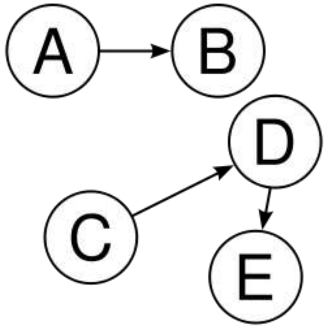
Tree?



Tree?



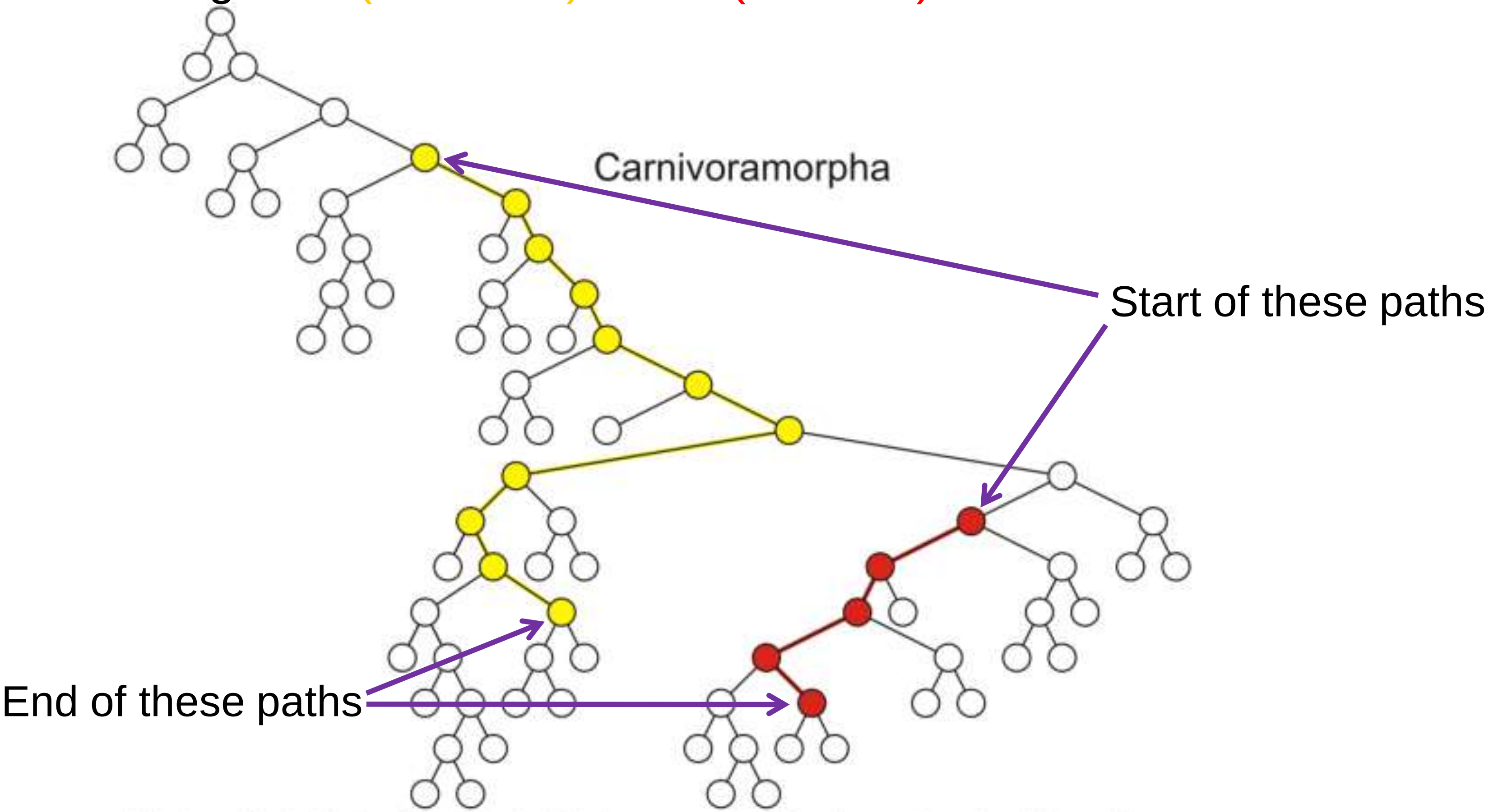
Tree?



Tree?

Path Example

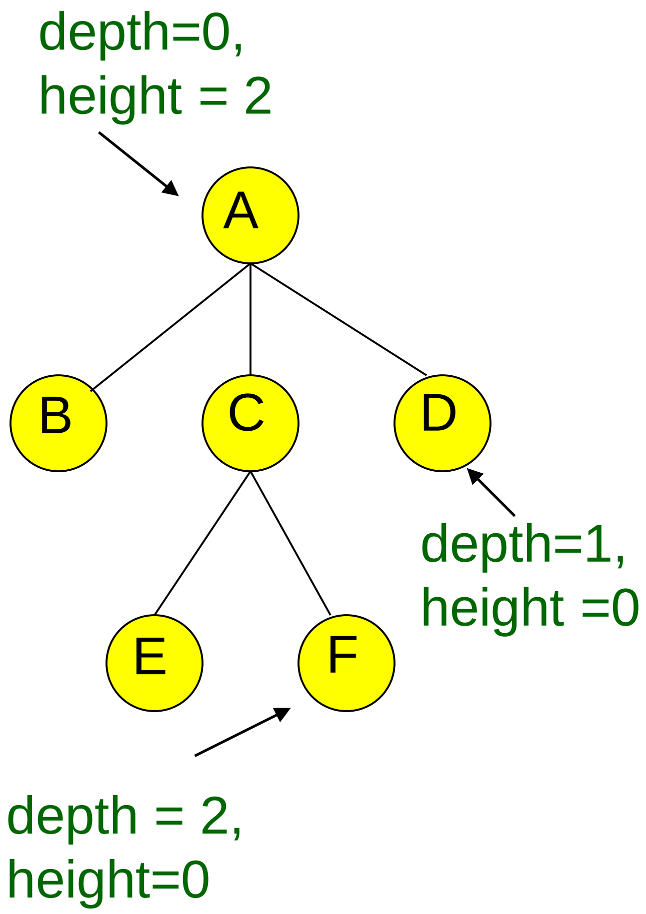
Paths of length 10 (11 nodes) and 4 (5 nodes)



Wesley-Hunt, G. D.; Flynn, J. J. "Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of 'Miacoidea'

Distance Measurement

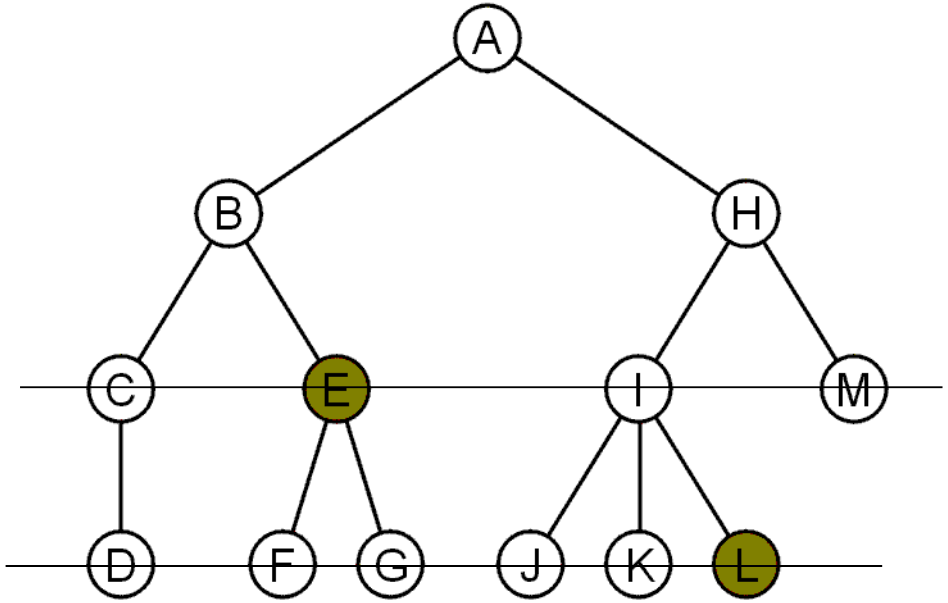
- **Length** of a path = number of edges
- **Depth** of a node N = length of path from root to N
- **Height** of node N = length of longest path from N to a leaf
- **Depth of tree** = depth of deepest node
- **Height of tree** = height of root



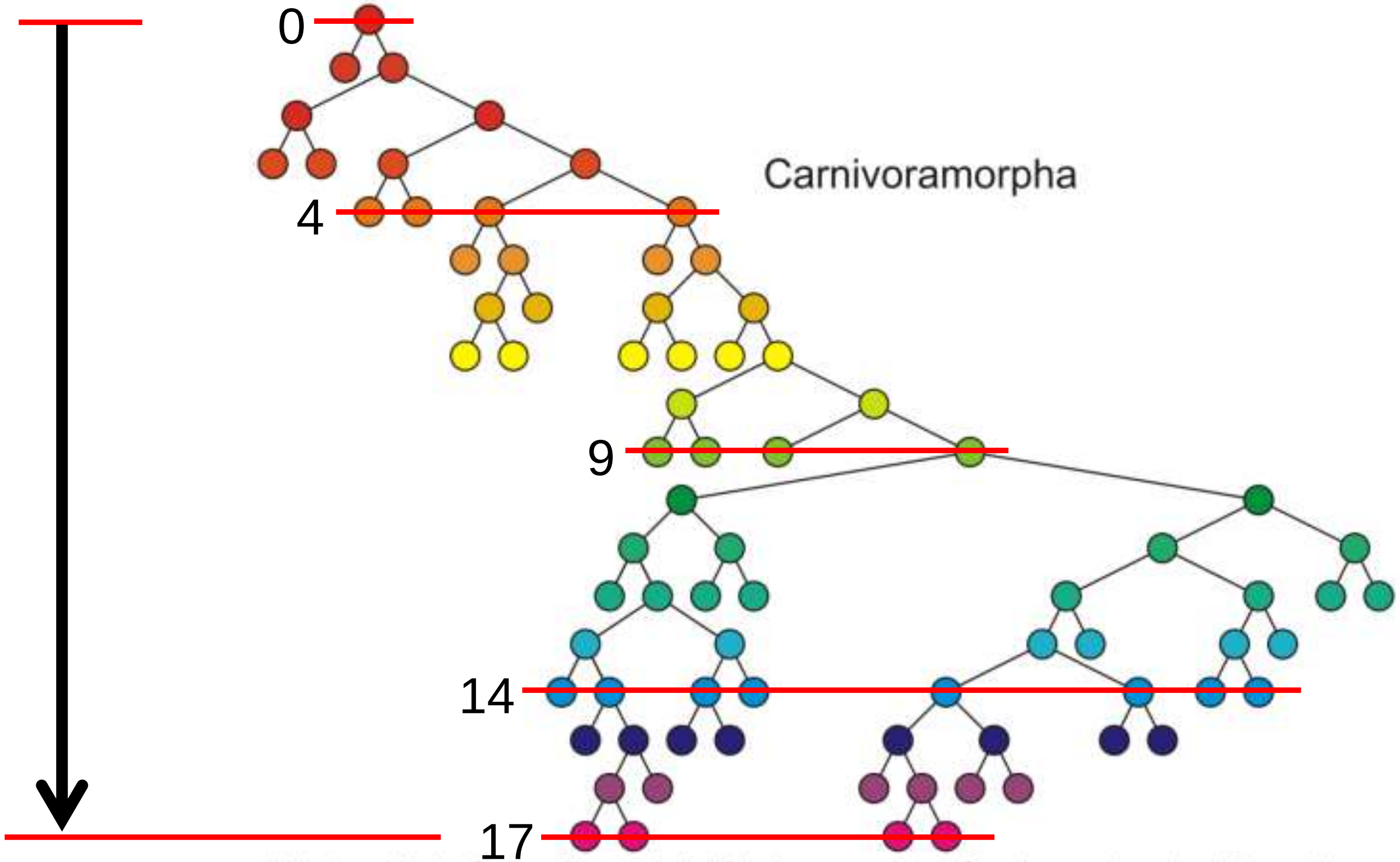
Node Depth

For each node in a tree, there exists a unique ***path from the root node to that node***

- What is depth of the node E?
  - ❑ 2
- What is depth of the node L?
  - ❑ 3



Node Depth and Tree Depth

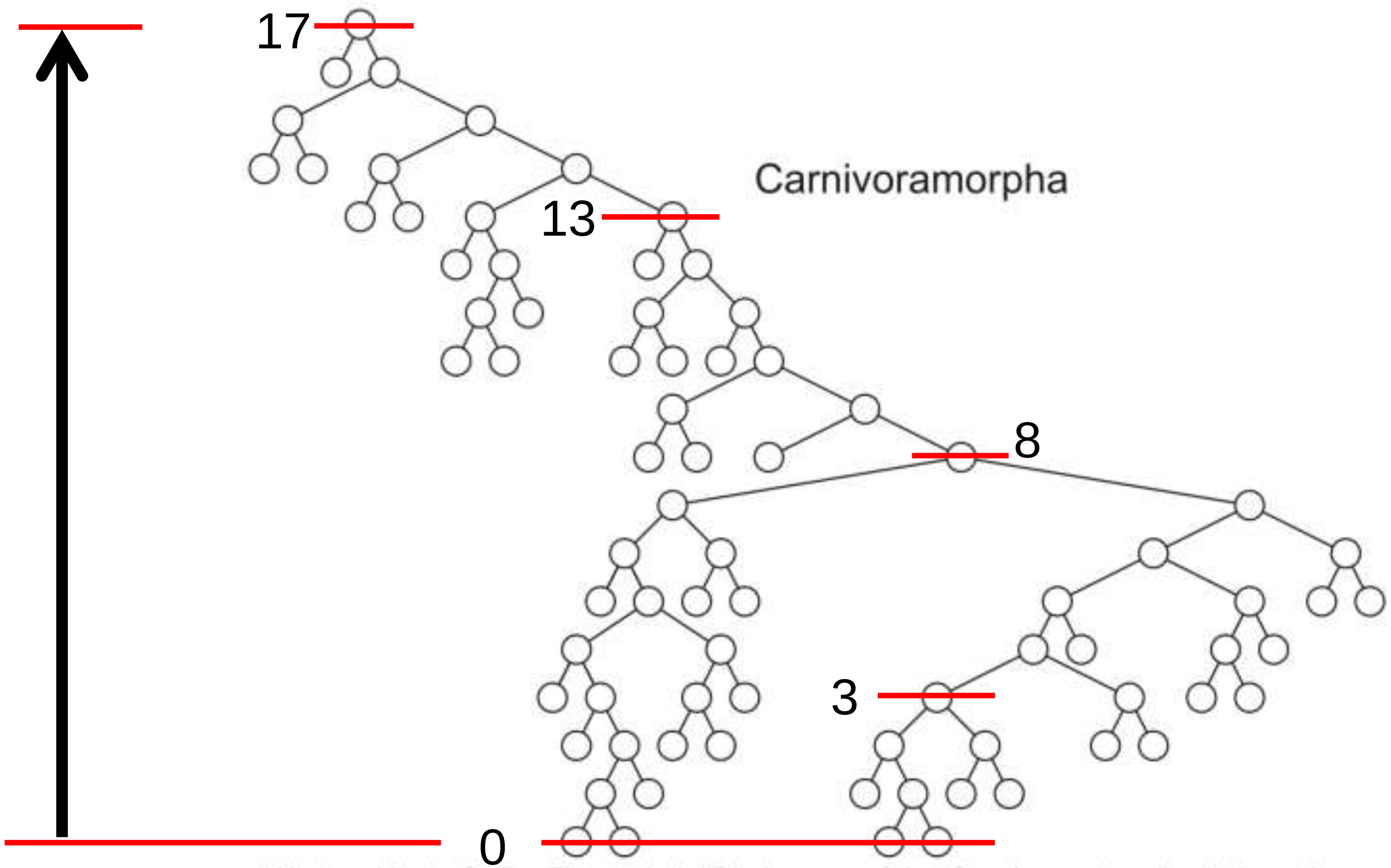


Wesley-Hunt, G. D.; Flynn, J. J. "Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of 'Miacoidea'

Height of Node and Tree

- The height of a node is the path length from the node to the deepest descendant.
- The height of a tree is the path length from root node to the deepest leaf.
- The height of a tree with a single node is 0
- The height of a leaf node is 0
- The height of a tree is equal to the depth of the tree
- For convenience, we define the height of the empty tree to be -1

Node Height vs Tree Height



Wesley-Hunt, G. D.; Flynn, J. J. "Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of 'Miacoidea'

Ancestor and Descendent

If a path exists from node a to node b:

- a is an *ancestor* of b
- b is a *descendent* of a

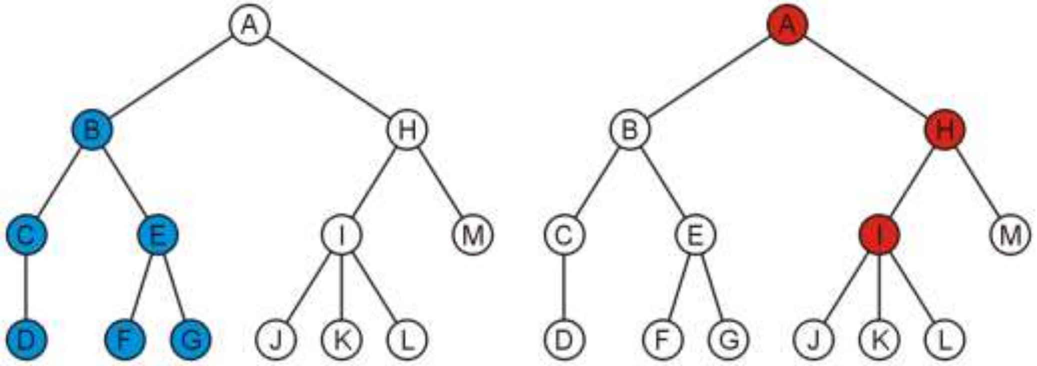
Thus, a node is both an ancestor and a descendant of itself

- We can add the adjective *strict* to exclude equality: a is a *strict descendent* of b if a is a descendant of b but $a \neq b$

The root node is an ancestor of all nodes

Ancestor and Descendent

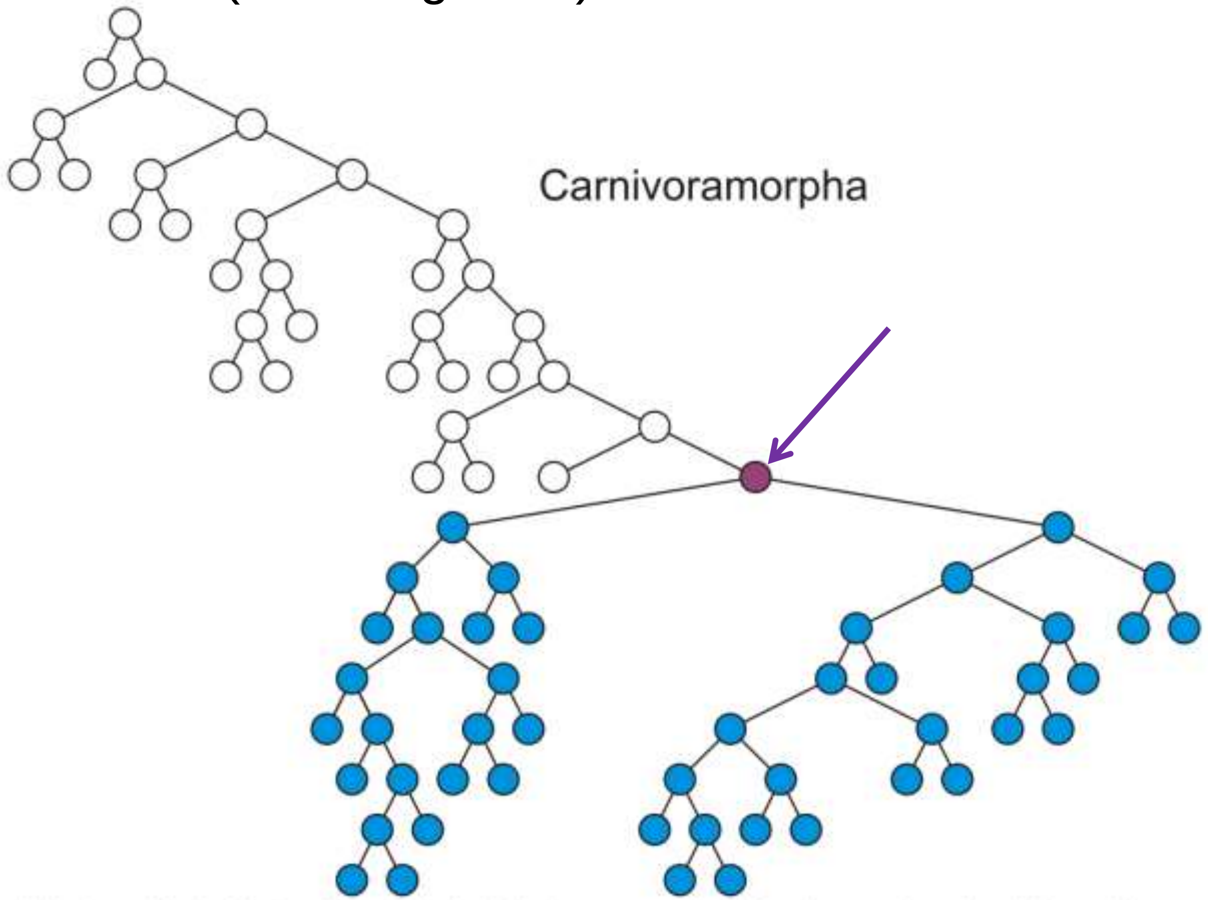
The descendants of node B are B, C, D, E, F, and G:



The ancestors of node I are I, H, and A:

Descendants

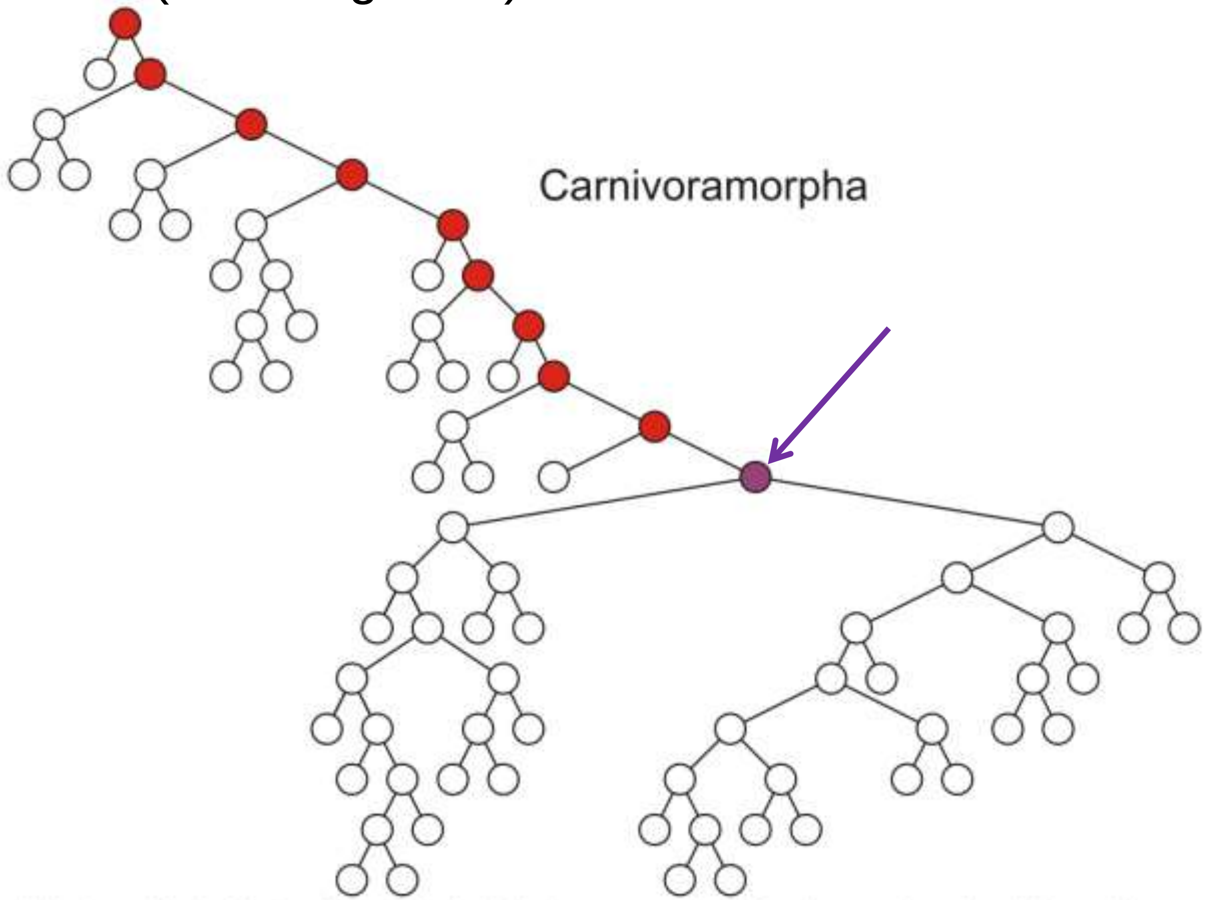
All descendants (including itself) of the indicated node



Wesley-Hunt, G. D.; Flynn, J. J. "Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of 'Miacoidea'

Ancestors

All ancestors (including itself) of the indicated node

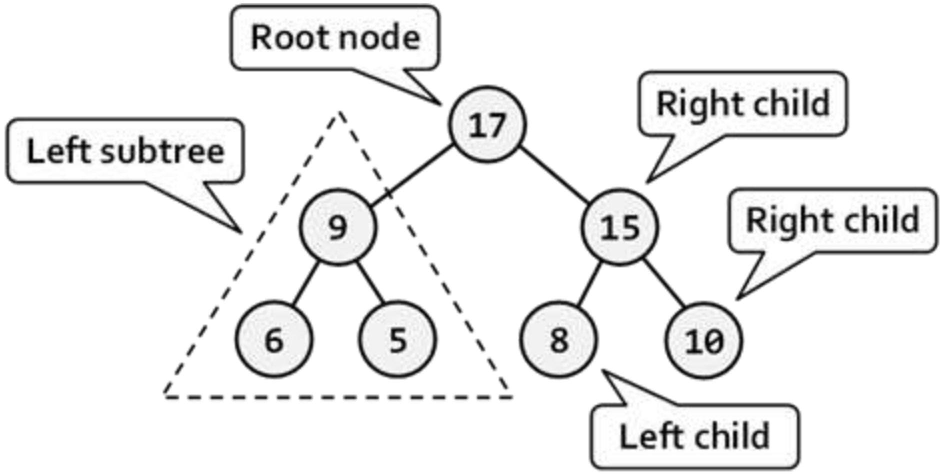


Wesley-Hunt, G. D.; Flynn, J. J. "Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of 'Miacoidea'

Alternative Definition of Tree

Another approach to a tree is to define the tree recursively:

- A tree can be represented by a root note
- Any node can be a root node
- A child node of a node is also a root node of a tree (sub-tree)



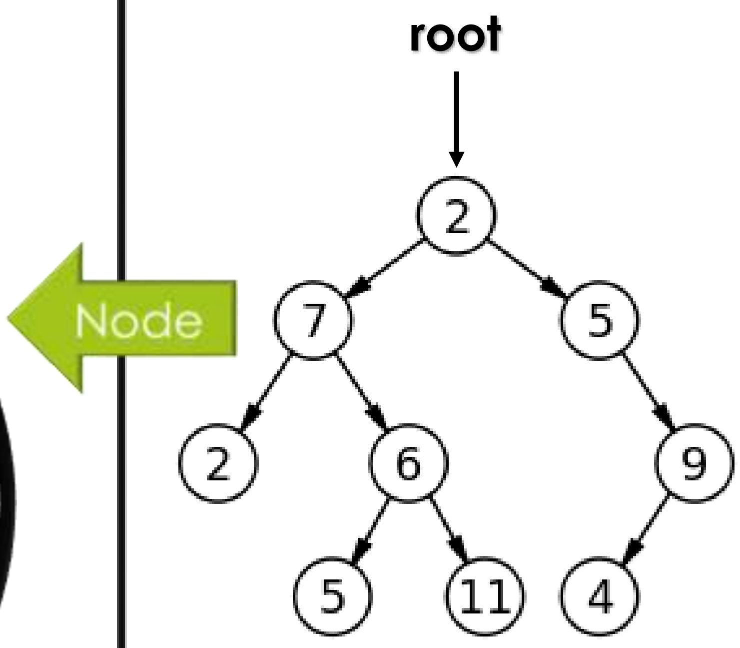
Implementation of Trees (K-ary Tree)

Pointer-based Implementation

class Node

// **Required**:
Object Key;
// K-ary Tree
Node child1, child2, ..., childK

// **Optional**:
// For bidirectional tree
Node parent;

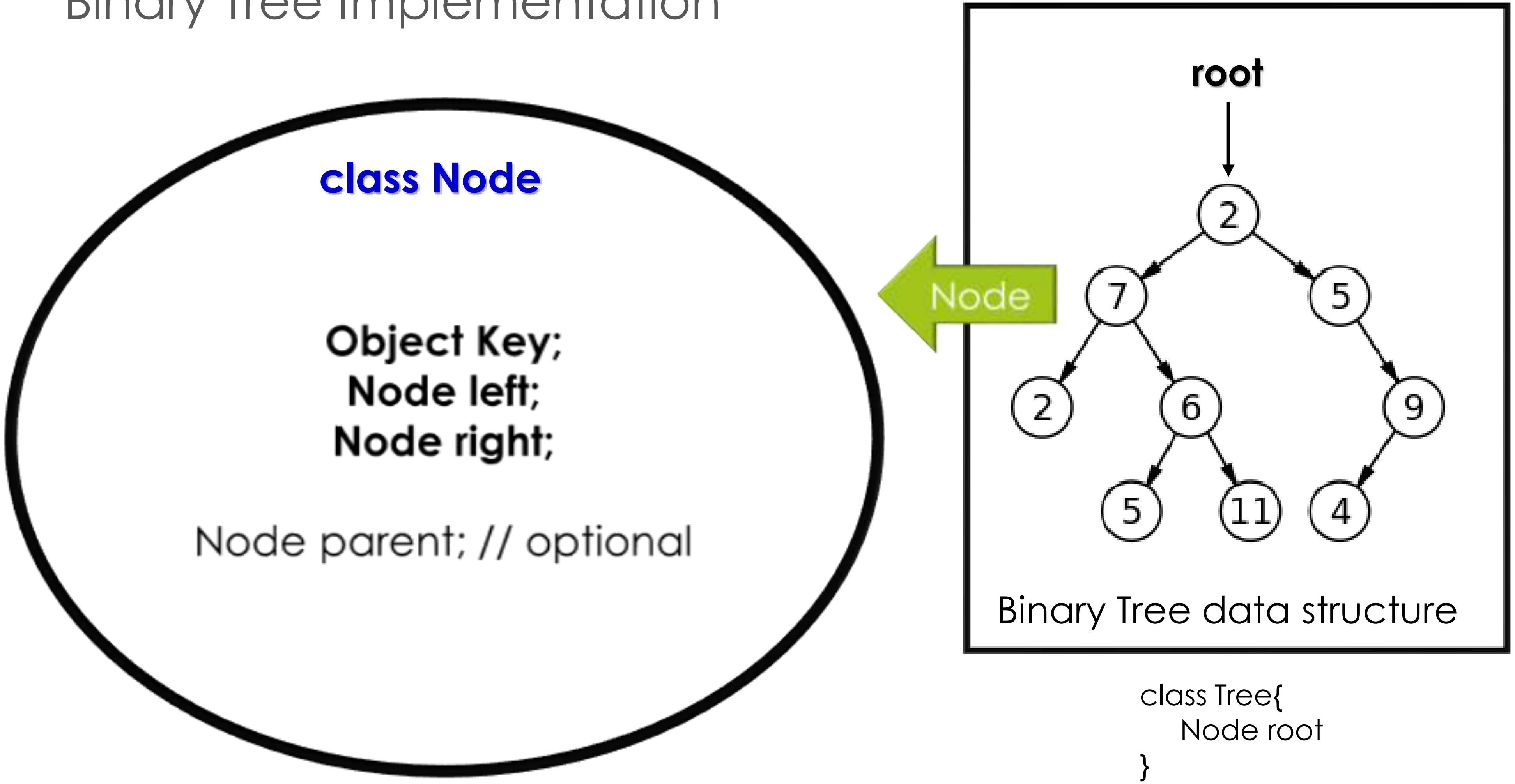


Tree data structure

```
class Tree{
    Node root
}
```

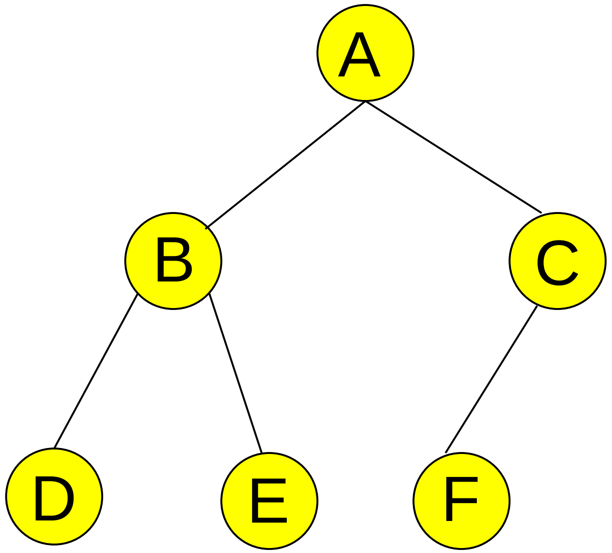
Implementation of Binary Trees

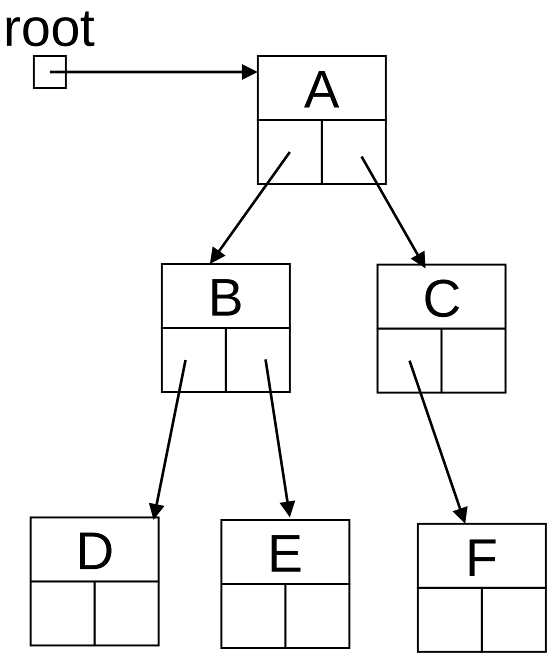
Binary Tree Implementation

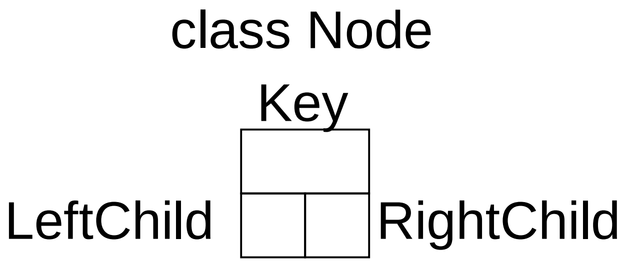



```
class Tree{
    Node root
}
```

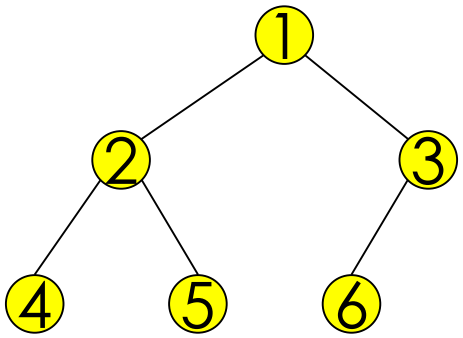
Binary Tree Structure (applicable to K-nary tree)

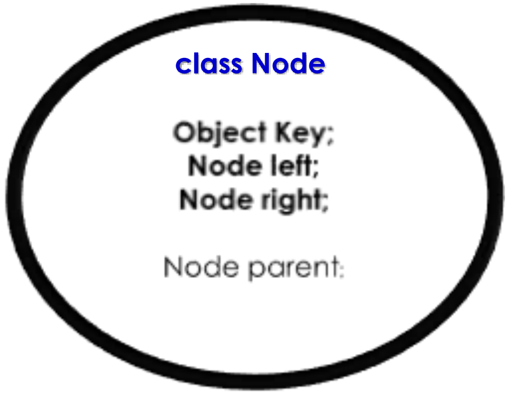


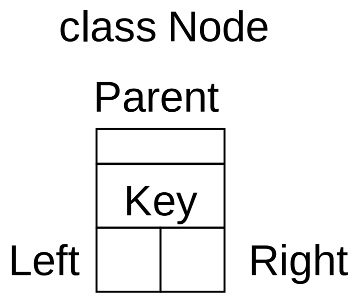




Binary Tree







- Every node has at most two children
- Most popular tree in computer engineering field
- If I do not say otherwise, we will assume that we are talking about Binary Trees for the rest of the course.
- Given a binary tree with depth ***d***, what is the possible number of nodes (**N**)?
- Given a number of nodes **N**, what is the minimum depth of a binary tree?

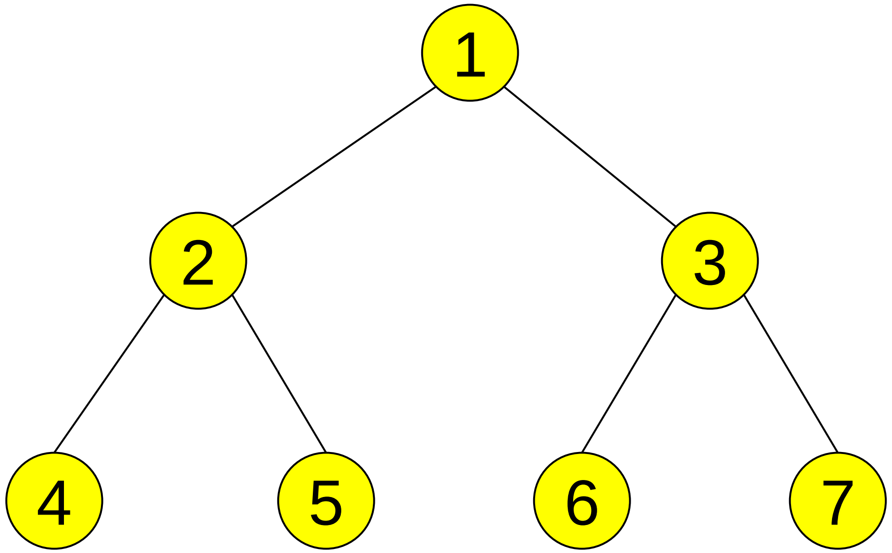
Minimum depth vs Node count

- At depth d, you can have $N = d+1$ to $2^{d+1}-1$ nodes
- minimum depth d is $\Theta(\log N)$

T(n) = Θ(f(n)) means
T(n) = O(f(n)) and f(n) = O(T(n)),
i.e. T(n) and f(n) have the same
growth rate

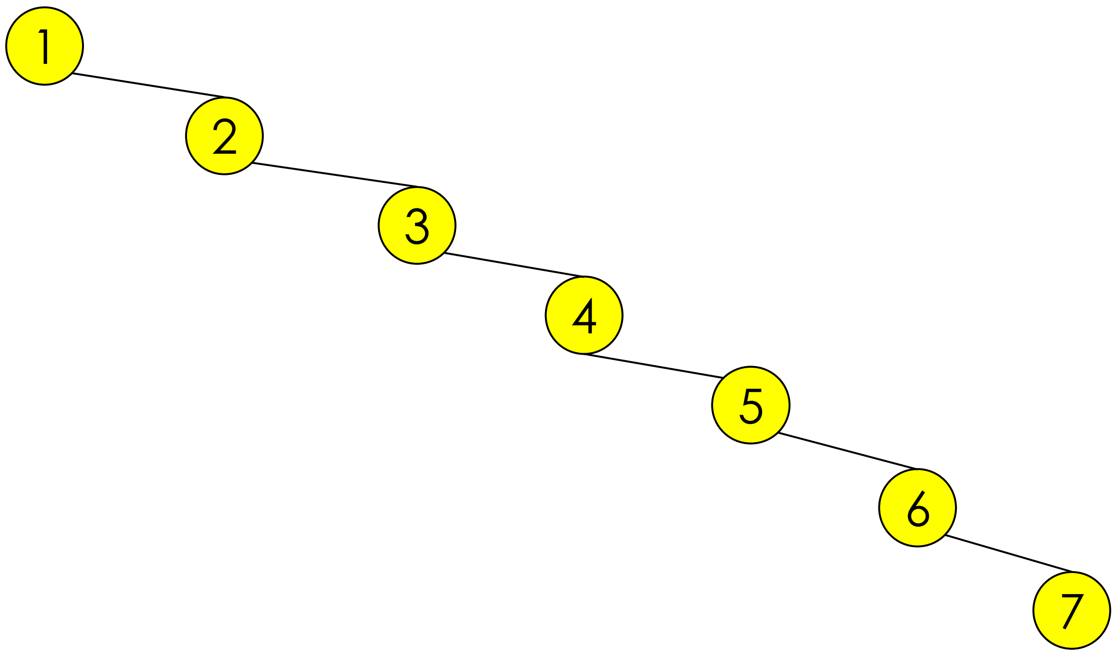
d=2

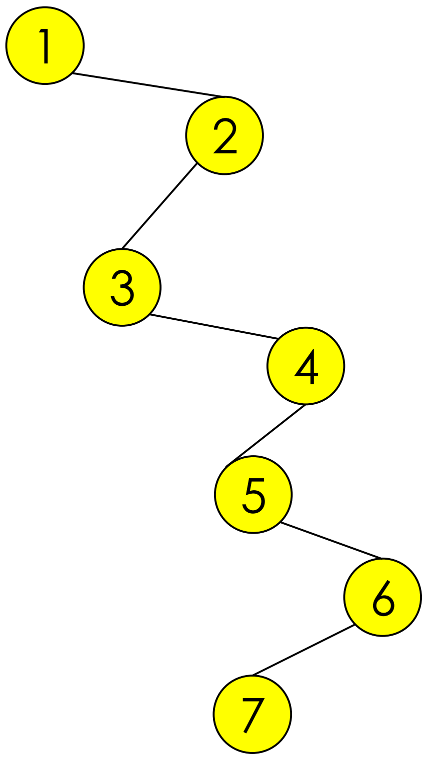
N= 3 to 2^3-1 (i.e, 3 to 7 nodes)



Minimum depth vs Node count

- What is the maximum depth of a binary tree?
  - Degenerate case: Tree is a linked list!
  - Maximum depth = N – 1

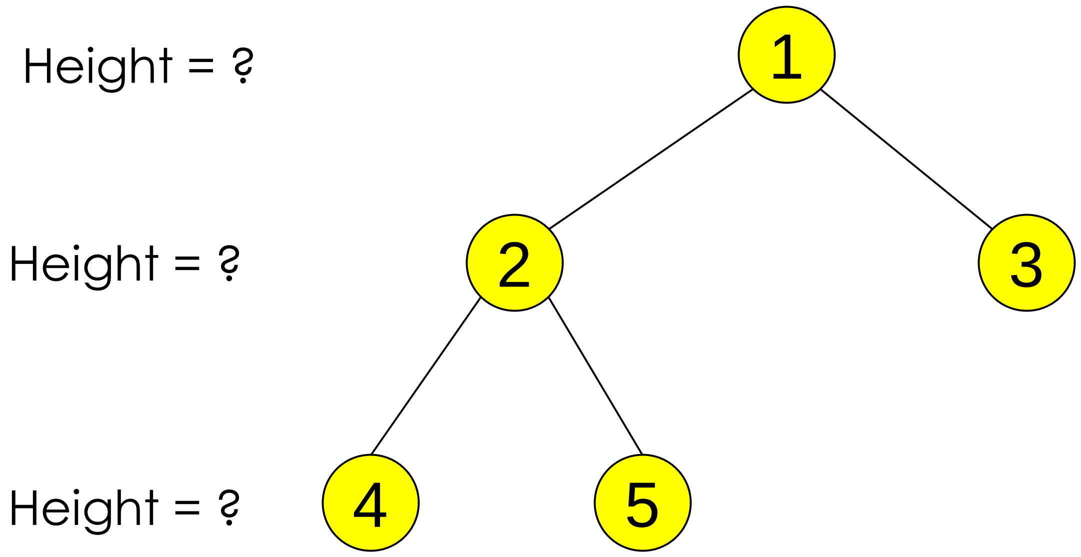




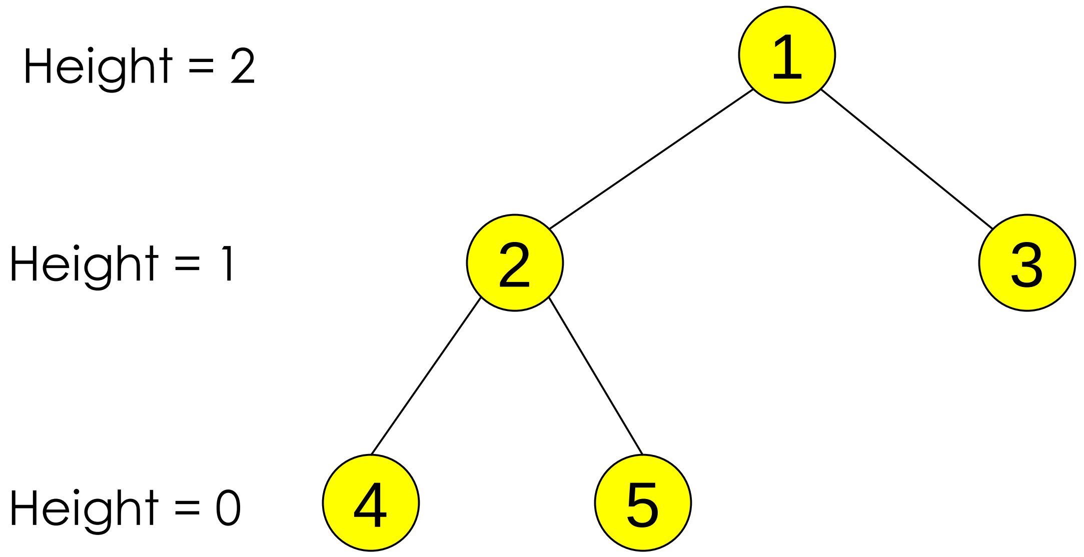
Thought Questions

- If a target key is often stored at the leaf nodes
  - What is the complexity of the searching for a key in a tree with the maximum depth (degenerate tree)?
  - What is the complexity of the searching for a key in a tree with the minimum depth (complete tree)?

Height Measurement



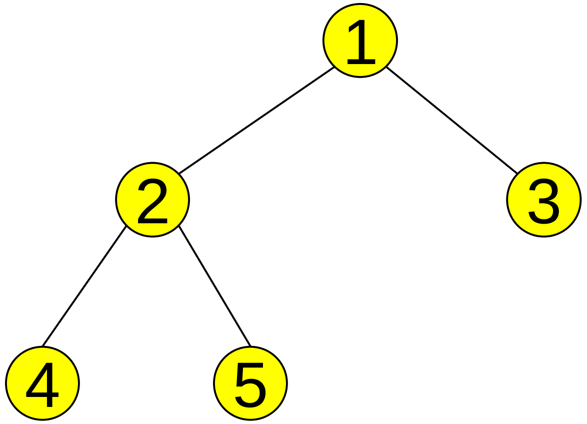
Height Measurement Algorithm



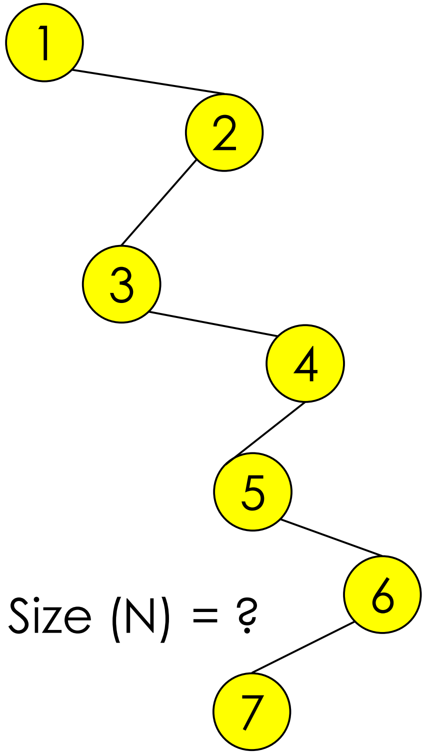
int Height(Node node)

```
if node == null
    return -1
else
    return 1 + max( Height(node.left), Height(node.right) )
```

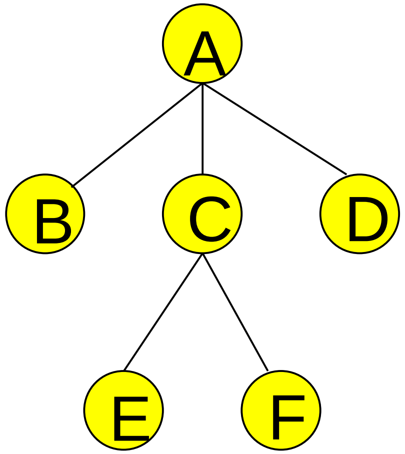
Tree Size Measurement



Tree Size (N) = ?

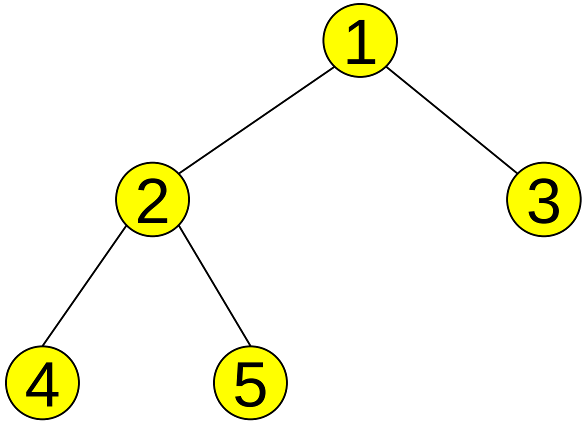


Tree Size (N) = ?

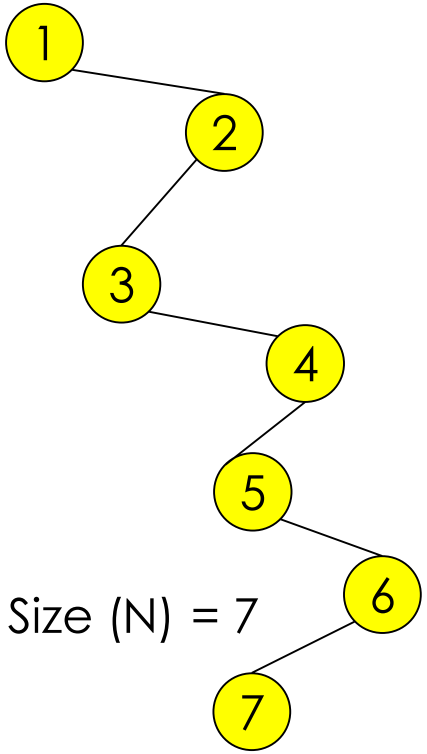


Tree Size (N) = ?

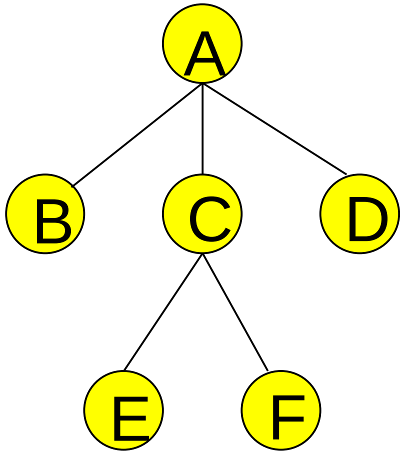
Tree Size Measurement Algorithm



Tree Size (N) = 5



Tree Size (N) = 7



Tree Size (N) = 6

int Size(Node node)

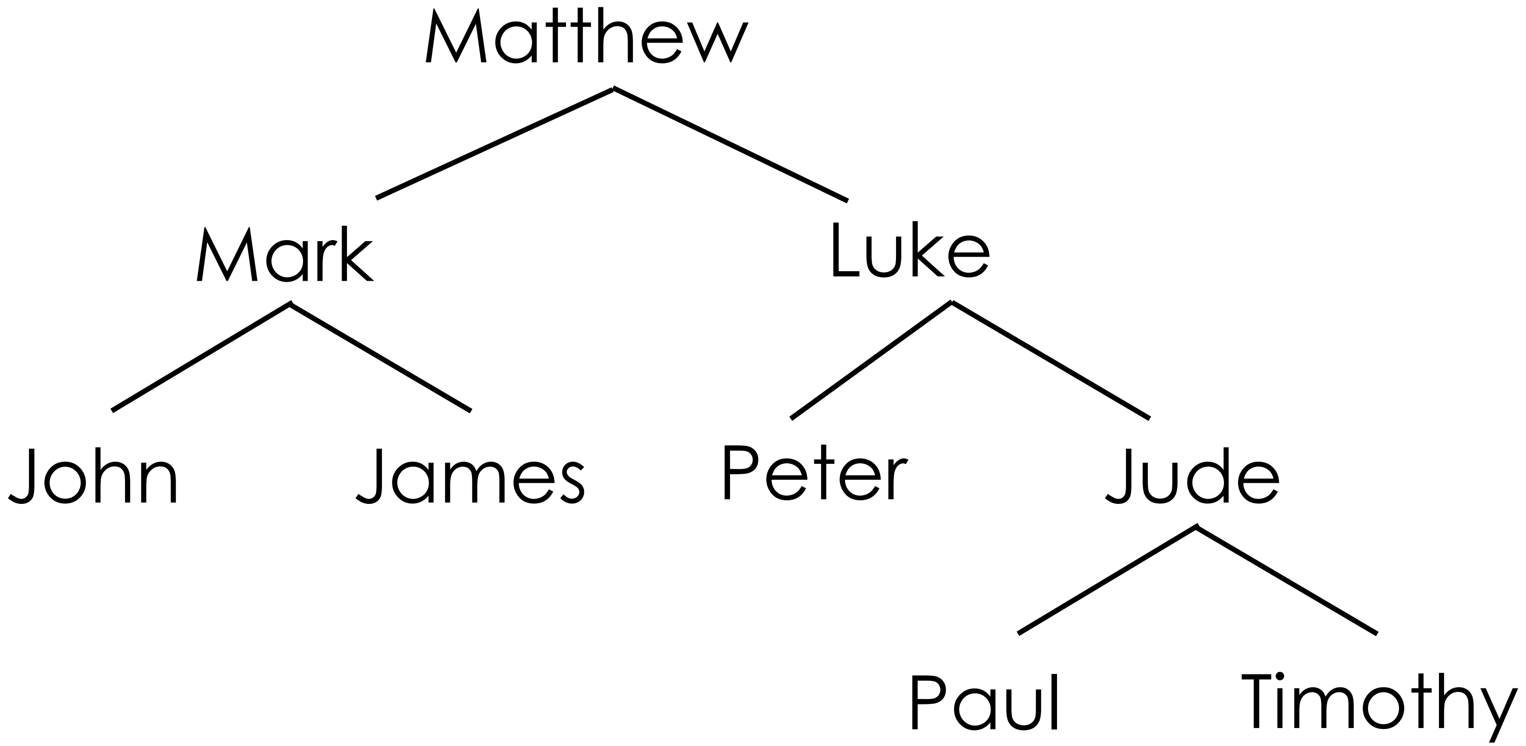
```
if node == null
    return  0
else
    return  1 + Size(node.left) + Size(node.right)
```

Tree Traversal (Walking a Tree)

- We want to visit all nodes of the tree
  - Searching for a key in an unordered tree (O(n))
  - Re-evaluate values of one or more nodes
  - Print all the values contains within the tree (Θ(n))

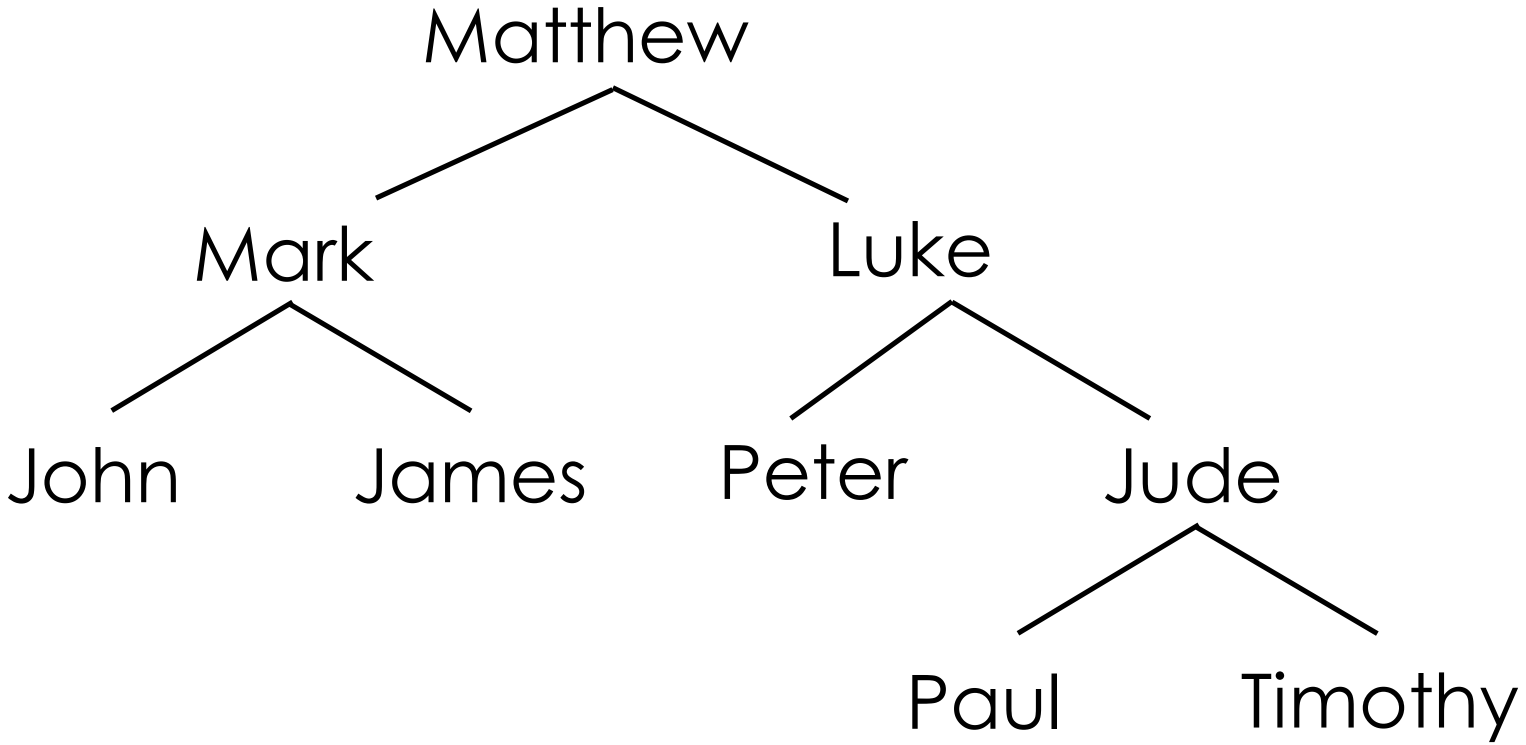
- Traversal order: Breadth-first and Depth-first
  - **Breadth-first**: We traverse all nodes at one level before progressing to the next level
    - e.g. Users tend to place files are usually places at the earlier folders rather than deeper subfolder
  - **Depth-first**: We completely traverse one sub-tree to before exploring a sibling sub-tree
    - e.g. Important data often stored at the leaves of the tree (such as operands) which should be read first before the internal nodes (operators)

Breadth-first Traversal (Level Traversal)



Output:

Breadth-first Traversal (Level Traversal)



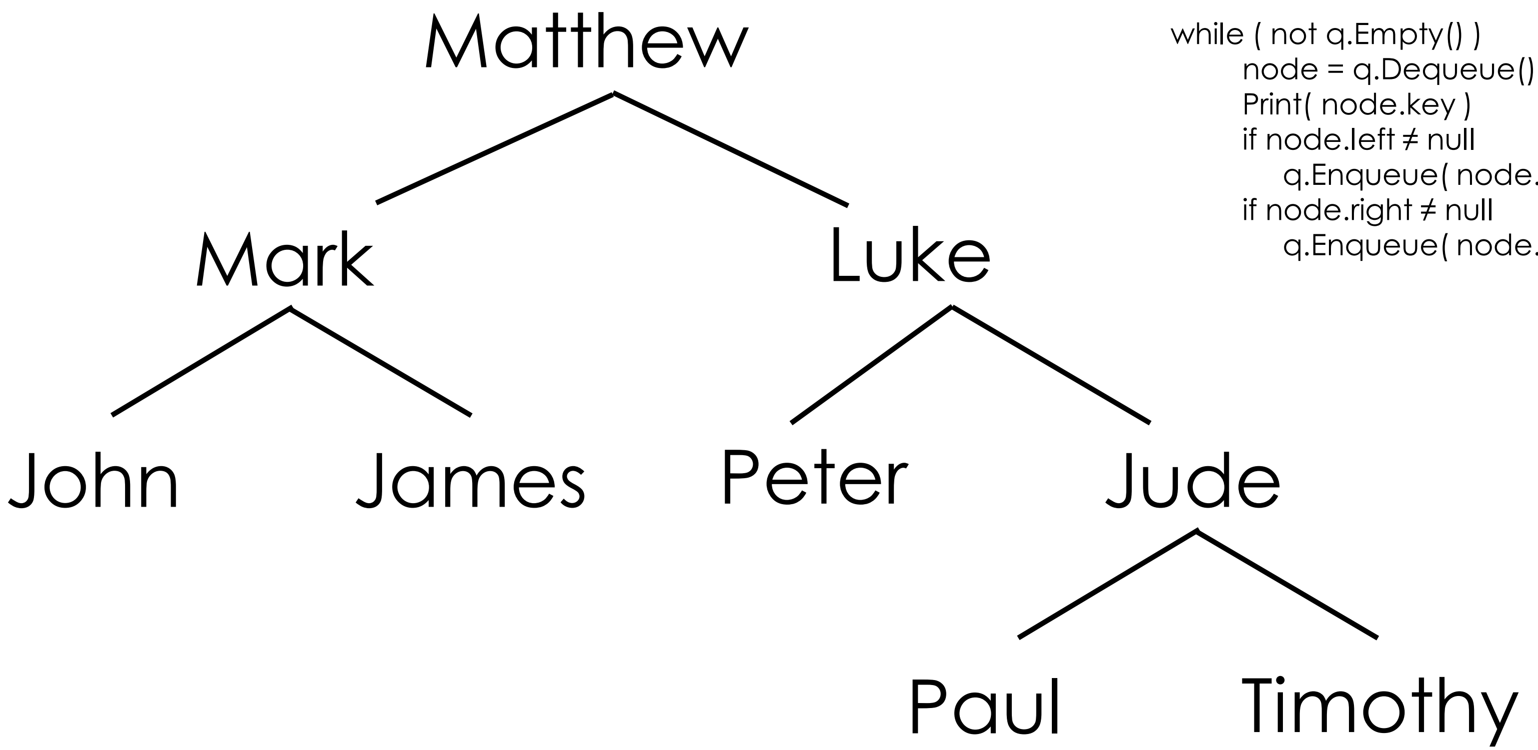
Output: Matthew, Mark, Luke, John, James, Peter, Jude, Paul, Timothy

Breadth-first Traversal implementation using Queue

LevelTraversal (Node node)

```
if node == null
    return
else
    Queue q
    q.Enqueue( node )
    while ( not q.Empty() )
        node = q.Dequeue()
        Print( node.key )  // Do something to the node
        if node.left ≠ null
            q.Enqueue( node.left )
        if node.right ≠ null
            q.Enqueue( node.right )
```

BFT using Q



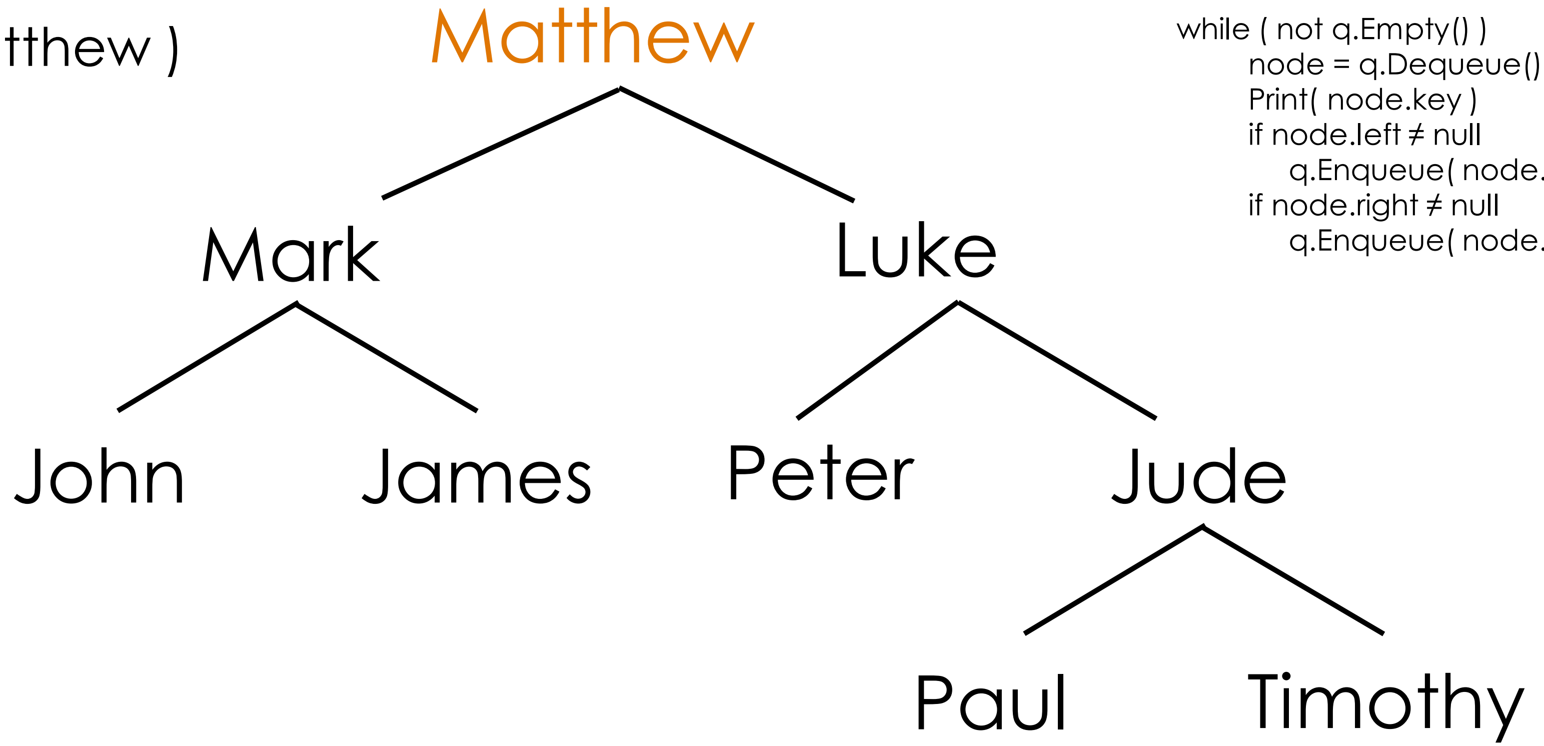

```
while ( not q.Empty() )
        node = q.Dequeue()
        Print( node.key )
        if node.left ≠ null
            q.Enqueue( node.left )
        if node.right ≠ null
            q.Enqueue( node.right )
```

Output:

Queue:

BFT using Q

Enqueue(Matthew)



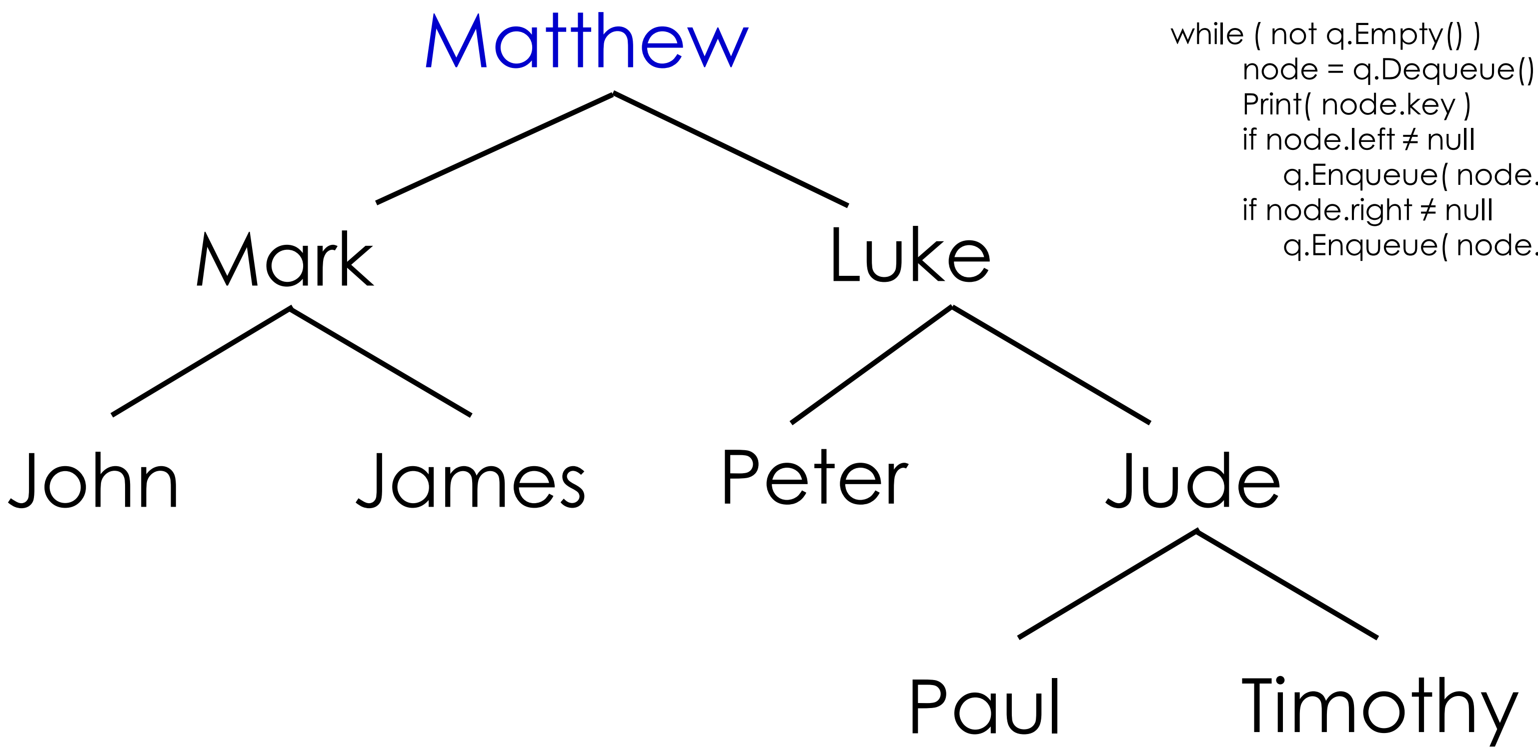

```
while ( not q.Empty() )
    node = q.Dequeue()
    Print( node.key )
    if node.left ≠ null
        q.Enqueue( node.left )
    if node.right ≠ null
        q.Enqueue( node.right )
```

Output:

Queue: Matthew

BFT using Q

Dequeue ()



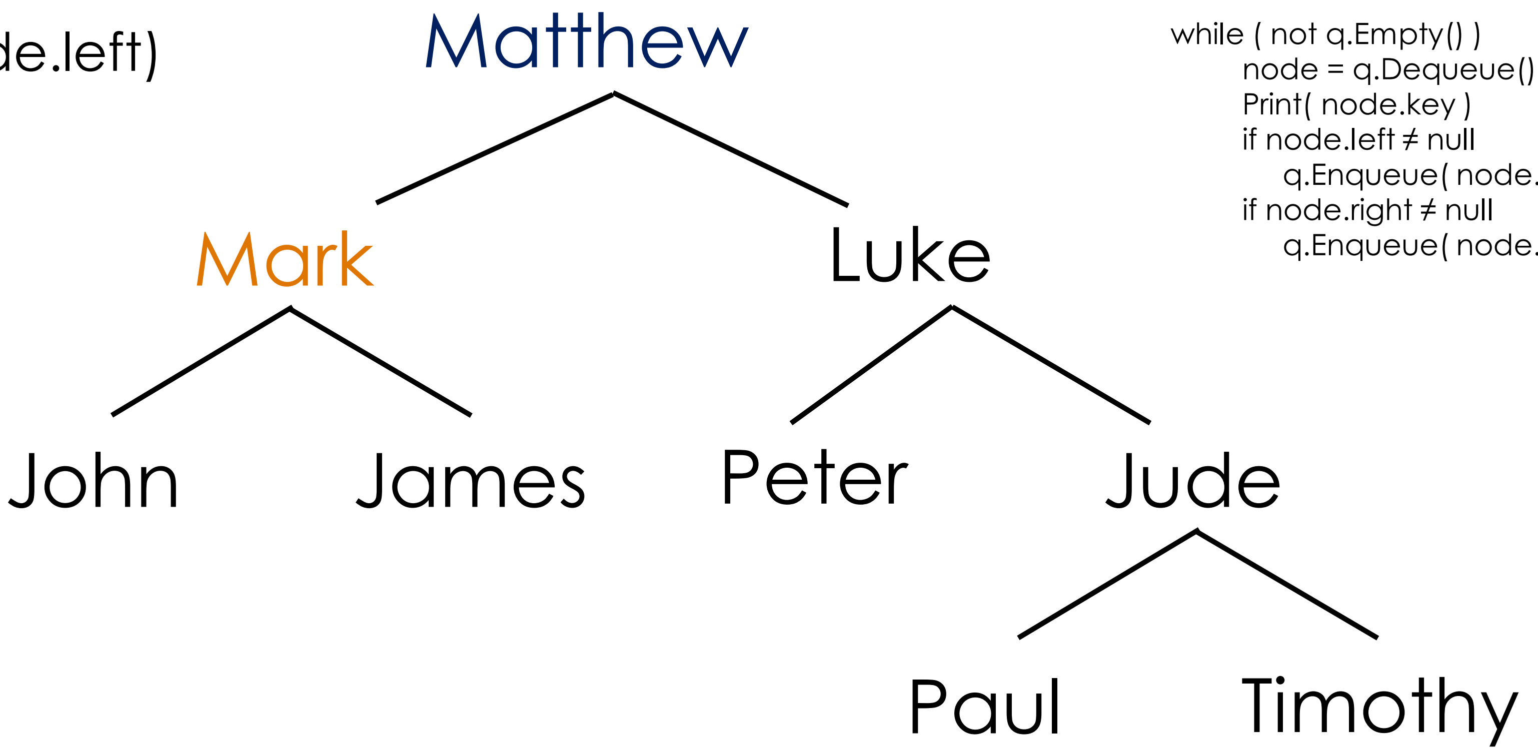

```
while ( not q.Empty() )
    node = q.Dequeue()
    Print( node.key )
    if node.left ≠ null
        q.Enqueue( node.left )
    if node.right ≠ null
        q.Enqueue( node.right )
```

Output: Matthew
node

Queue:

BFT using Q

Enqueue (node.left)




```
while ( not q.Empty() )
        node = q.Dequeue()
        Print( node.key )
        if node.left ≠ null
            q.Enqueue( node.left )
        if node.right ≠ null
            q.Enqueue( node.right )
```

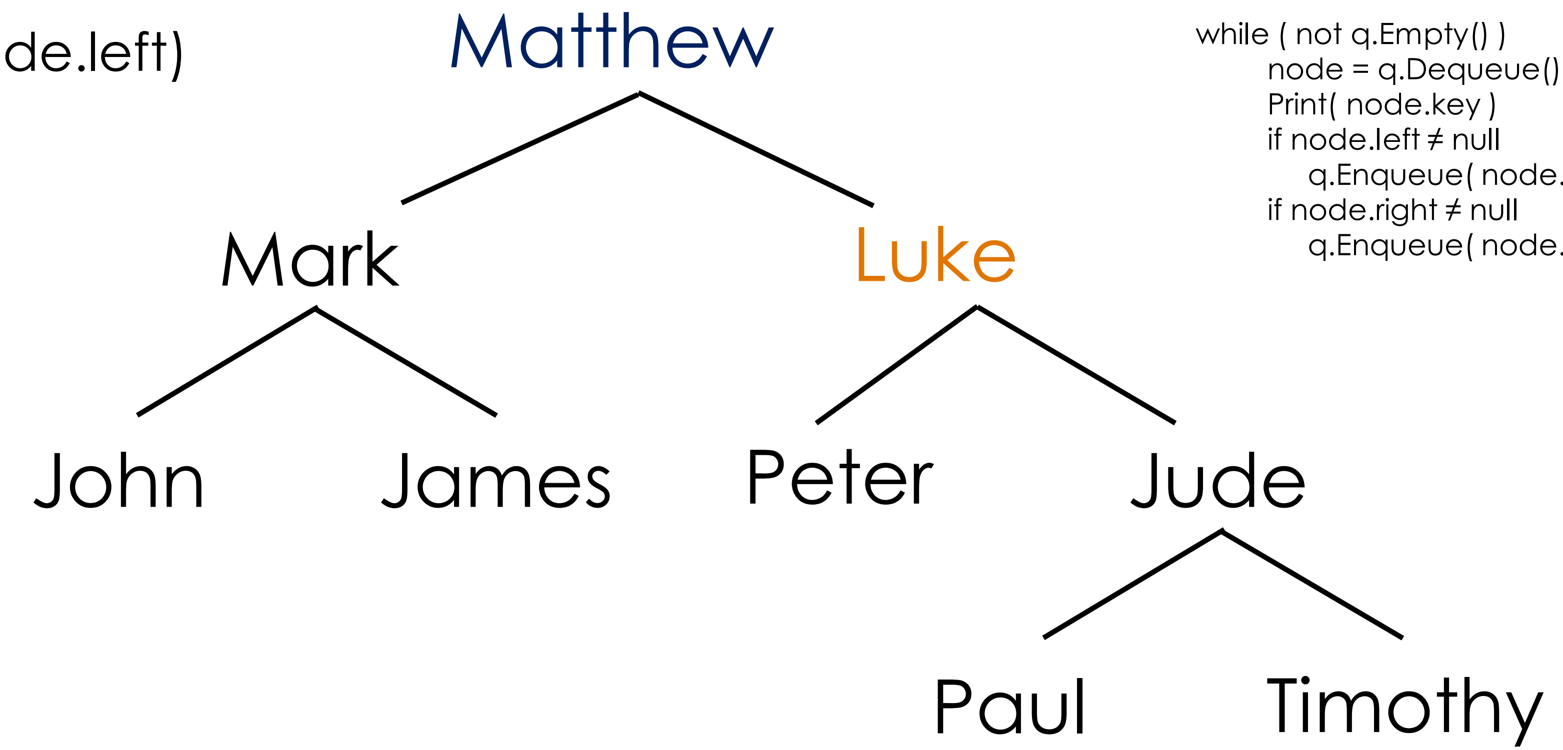
Output: Matthew

node

Queue: Mark

BFT using Q

Enqueue (node.left)




```
while ( not q.Empty() )
    node = q.Dequeue()
    Print( node.key )
    if node.left ≠ null
        q.Enqueue( node.left )
    if node.right ≠ null
        q.Enqueue( node.right )
```

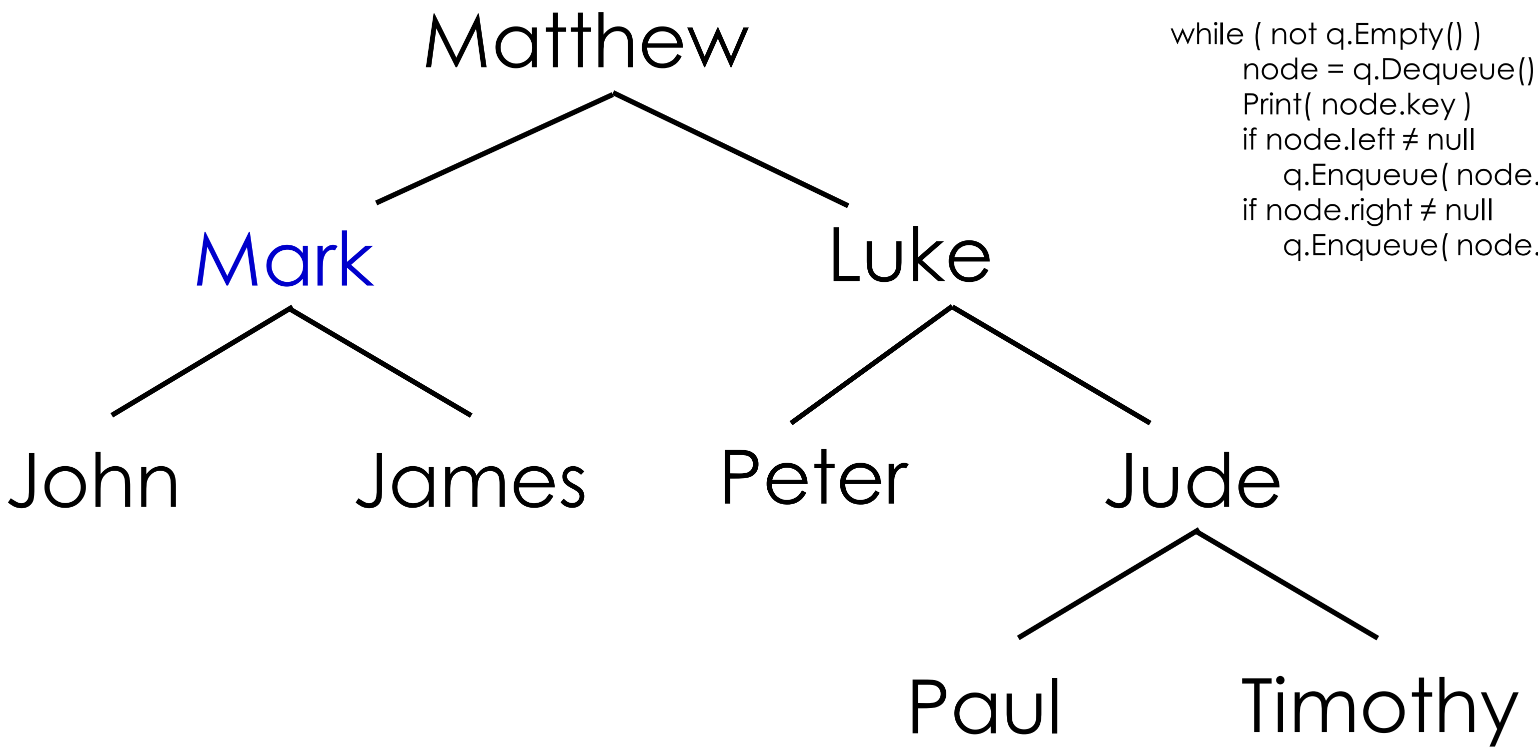
Output: Matthew

node

Queue: Mark, Luke

BFT using Q

Dequeue ()




```
while ( not q.Empty() )
    node = q.Dequeue()
    Print( node.key )
    if node.left ≠ null
        q.Enqueue( node.left )
    if node.right ≠ null
        q.Enqueue( node.right )
```

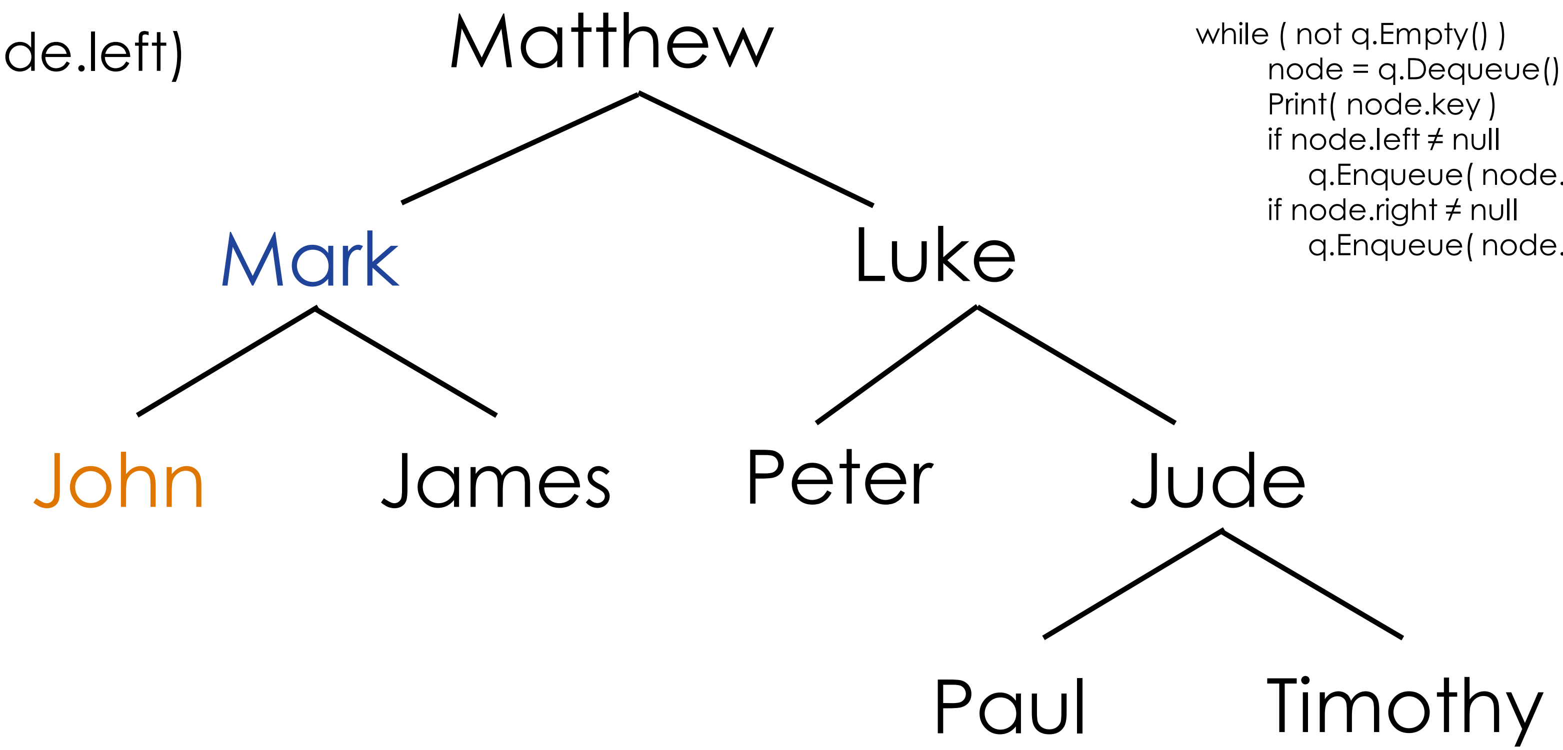
Output: Matthew, Mark

node

Queue: Luke

BFT using Q

Enqueue (node.left)




```
while ( not q.Empty() )
    node = q.Dequeue()
    Print( node.key )
    if node.left ≠ null
        q.Enqueue( node.left )
    if node.right ≠ null
        q.Enqueue( node.right )
```

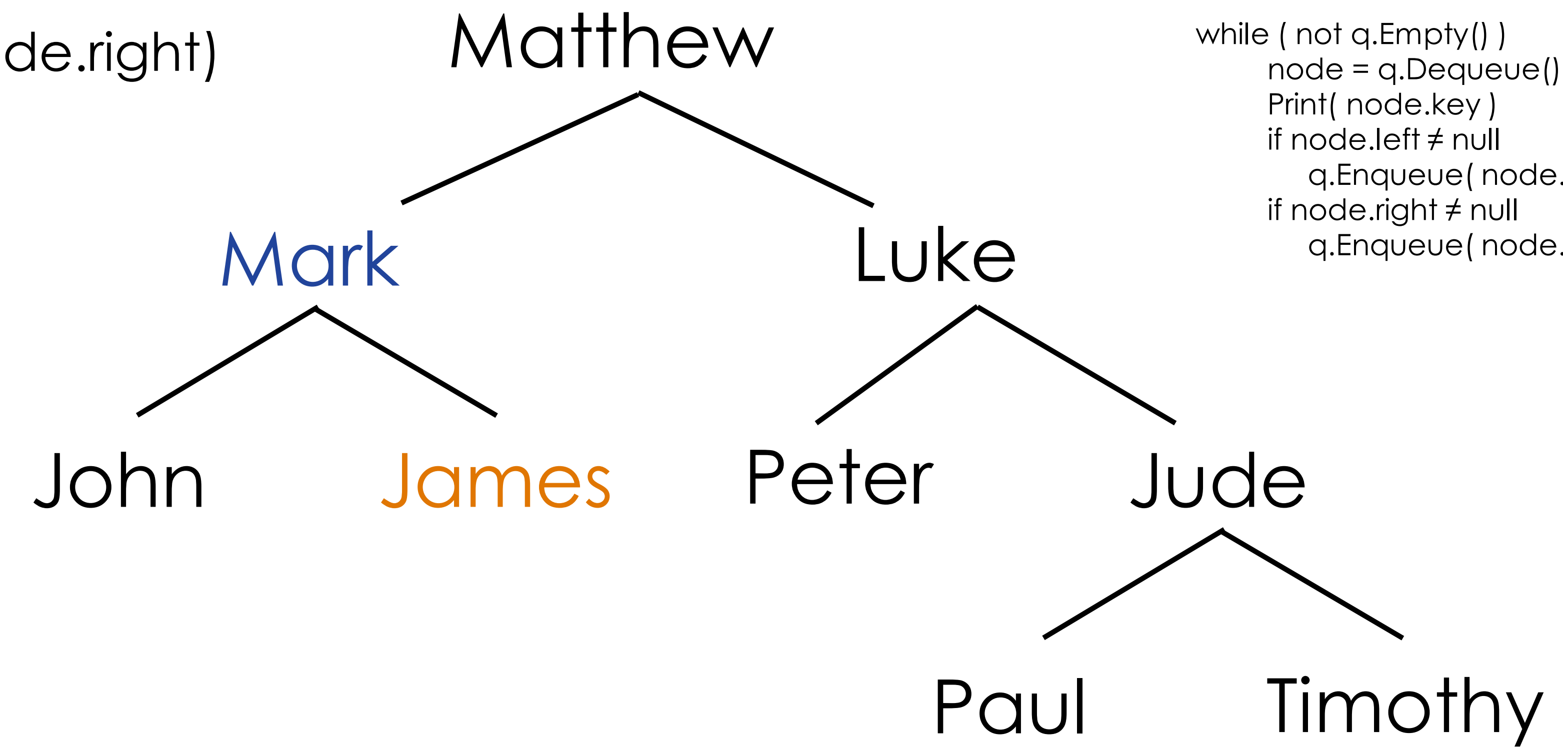
Output: Matthew, Mark

node

Queue: Luke, John

BFT using Q

Enqueue (node.right)




```
while ( not q.Empty() )
    node = q.Dequeue()
    Print( node.key )
    if node.left ≠ null
        q.Enqueue( node.left )
    if node.right ≠ null
        q.Enqueue( node.right )
```

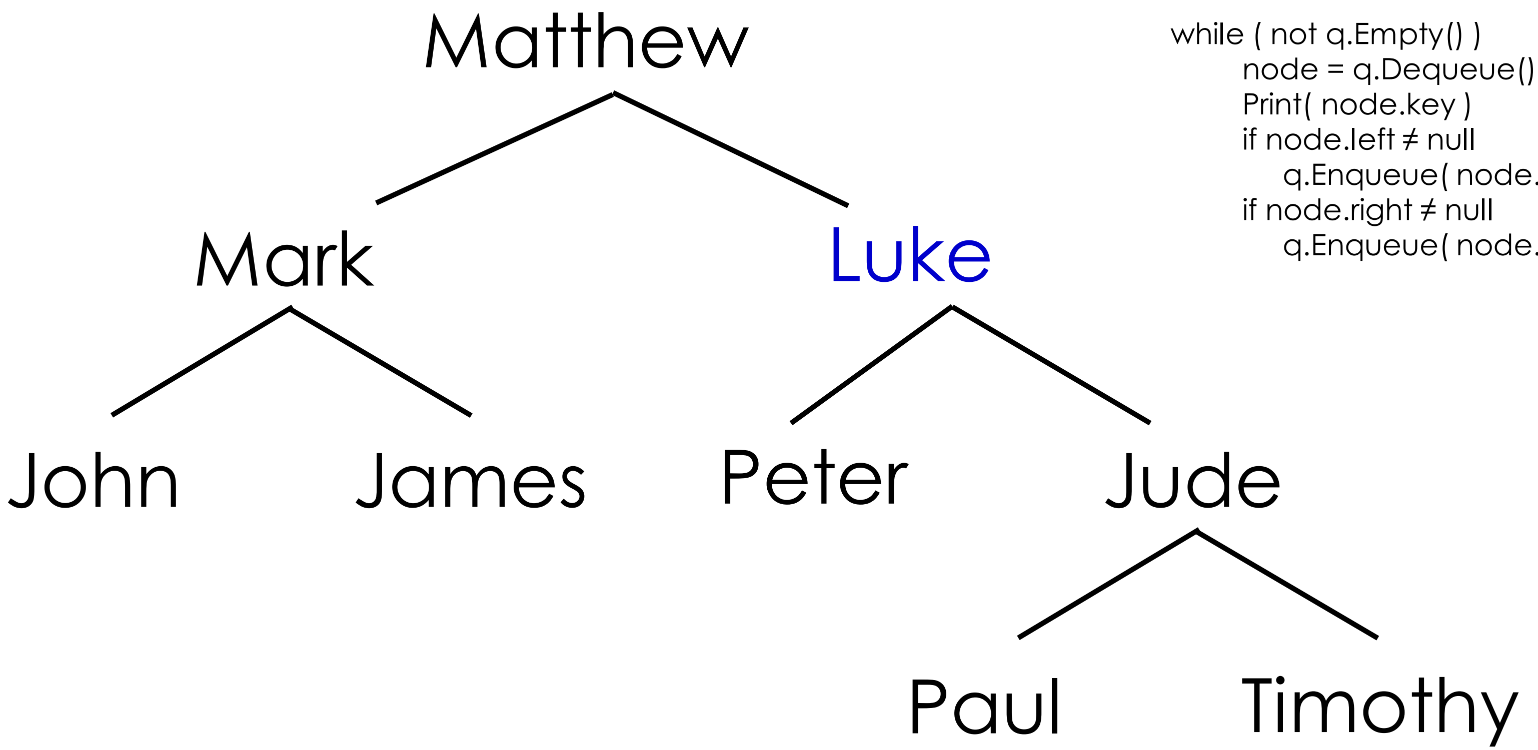
Output: Matthew, Mark

node

Queue: Luke, John, James

BFT using Q

Dequeue ()




```
while ( not q.Empty() )
        node = q.Dequeue()
        Print( node.key )
        if node.left ≠ null
            q.Enqueue( node.left )
        if node.right ≠ null
            q.Enqueue( node.right )
```

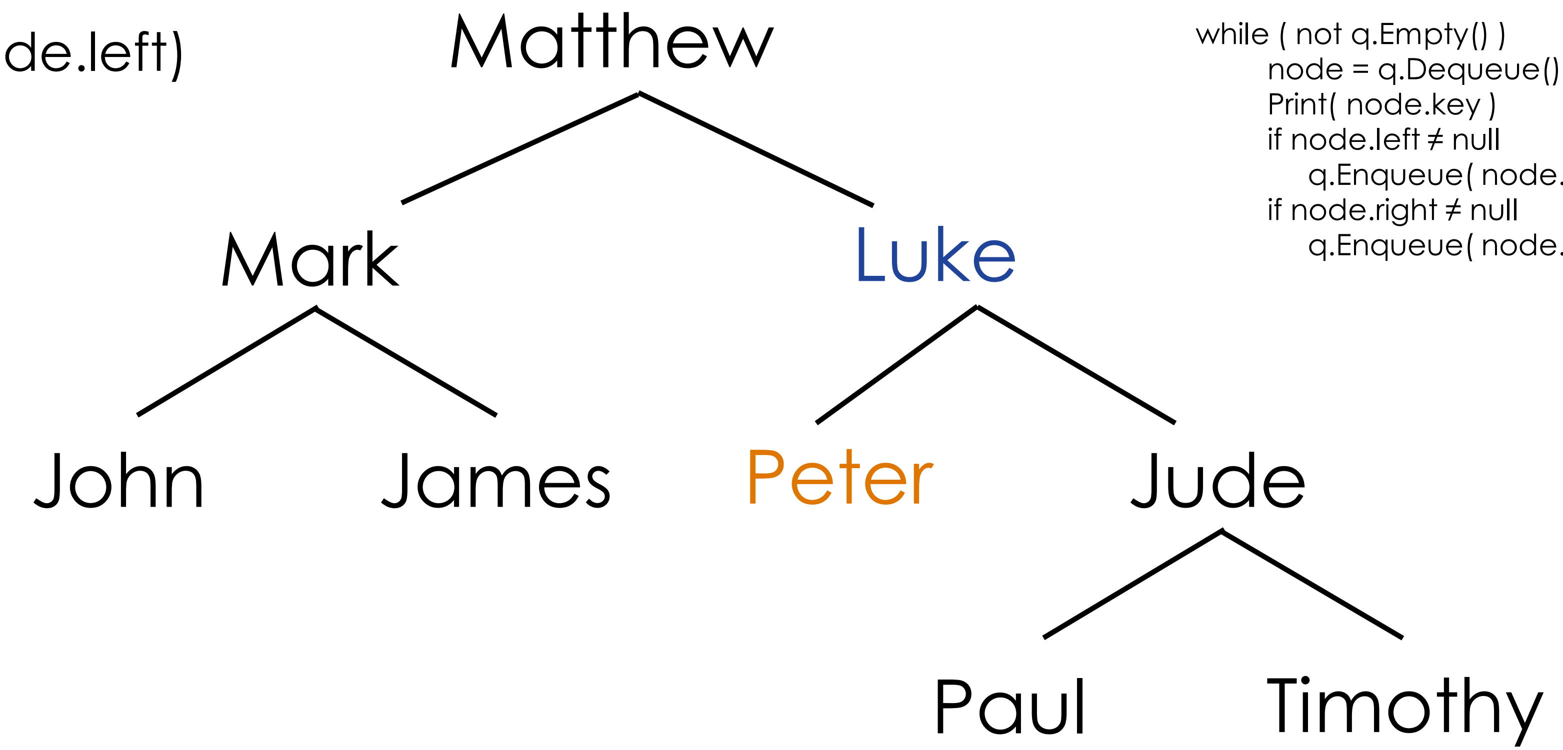
Output: Matthew, Mark, Luke

node

Queue: John, James

BFT using Q

Enqueue (node.left)




```
while ( not q.Empty() )
    node = q.Dequeue()
    Print( node.key )
    if node.left ≠ null
        q.Enqueue( node.left )
    if node.right ≠ null
        q.Enqueue( node.right )
```

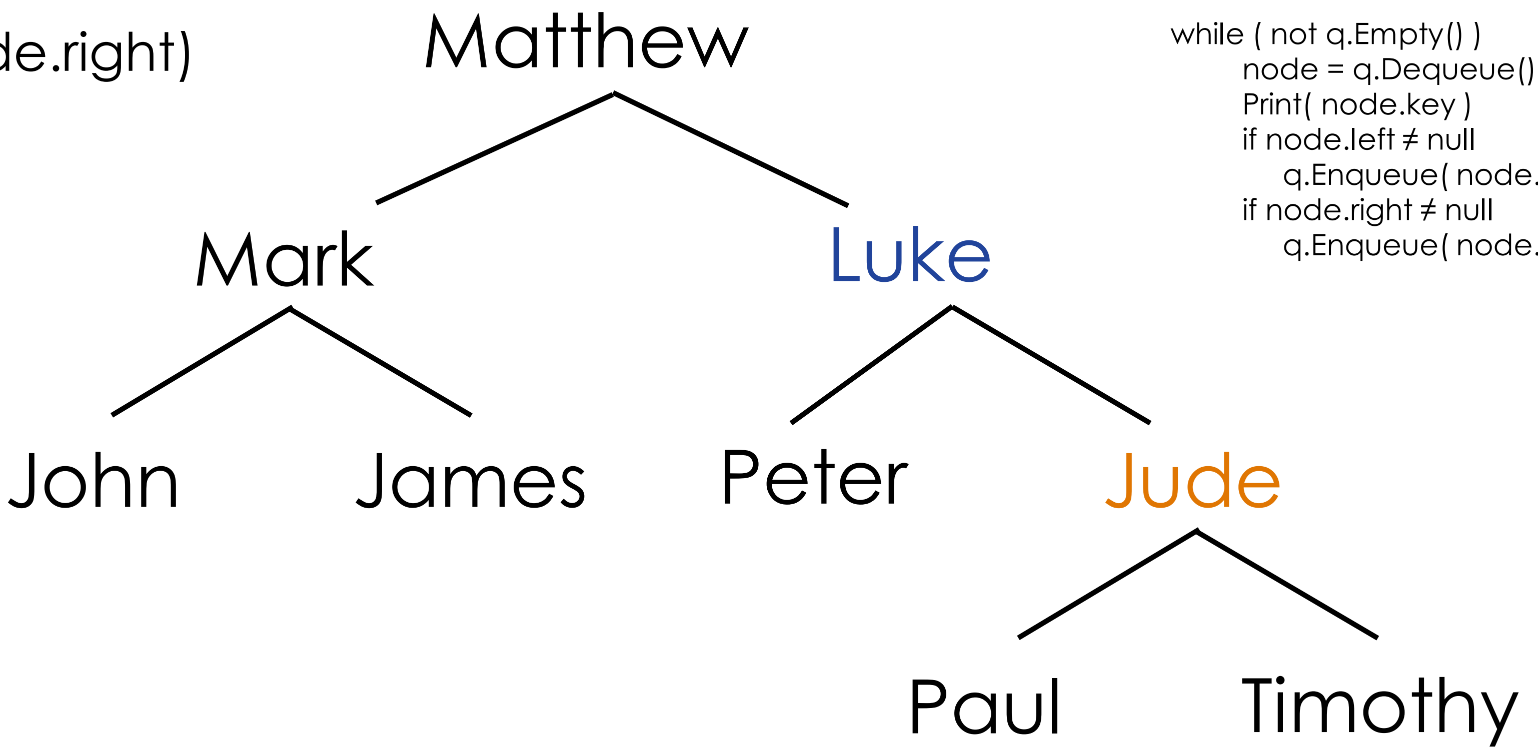
Output: Matthew, Mark, Luke

node

Queue: John, James, Peter

BFT using Q

Enqueue (node.right)




```
while ( not q.Empty() )
    node = q.Dequeue()
    Print( node.key )
    if node.left ≠ null
        q.Enqueue( node.left )
    if node.right ≠ null
        q.Enqueue( node.right )
```

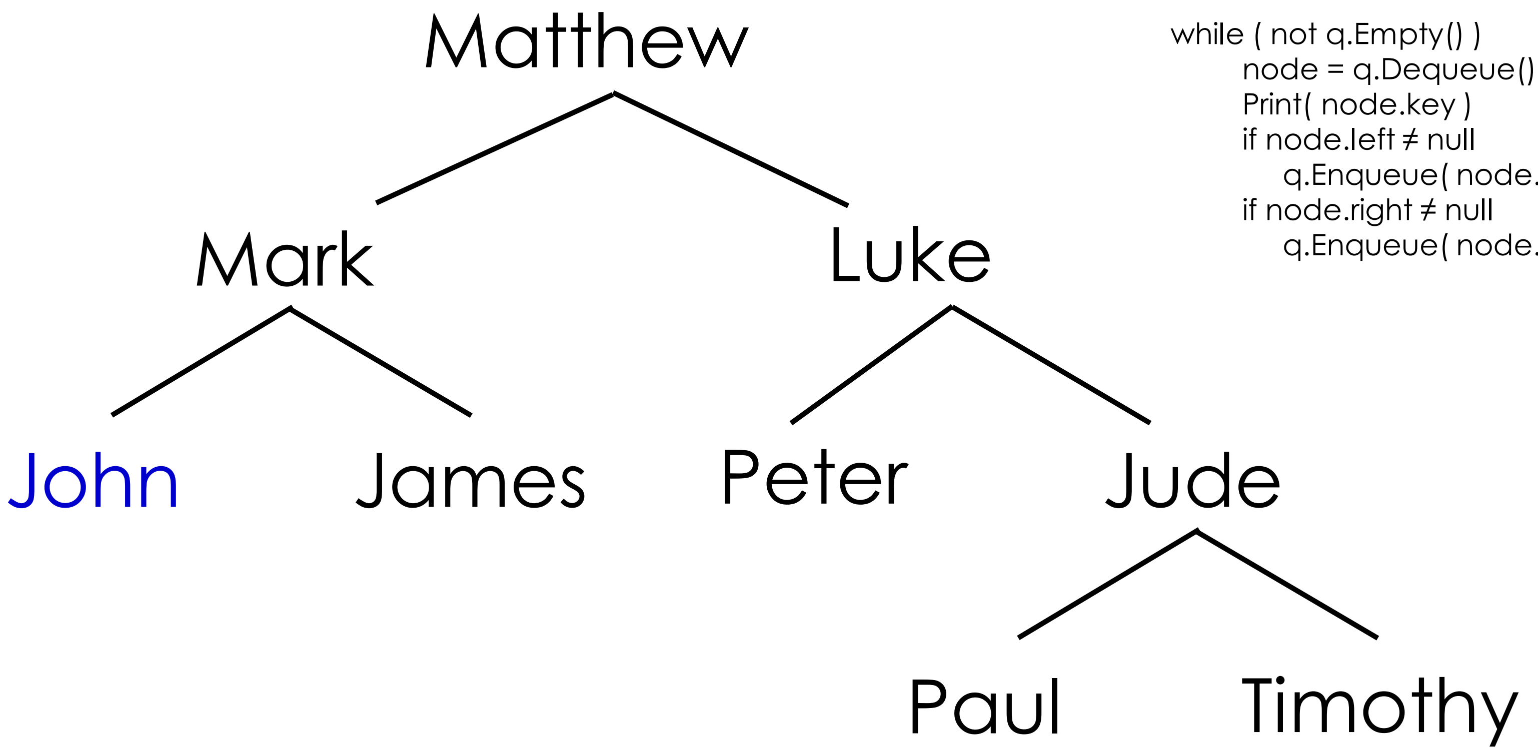
Output: Matthew, Mark, Luke

node

Queue: John, James, Peter, Jude

BFT using Q

Dequeue ()




```
while ( not q.Empty() )
    node = q.Dequeue()
    Print( node.key )
    if node.left ≠ null
        q.Enqueue( node.left )
    if node.right ≠ null
        q.Enqueue( node.right )
```

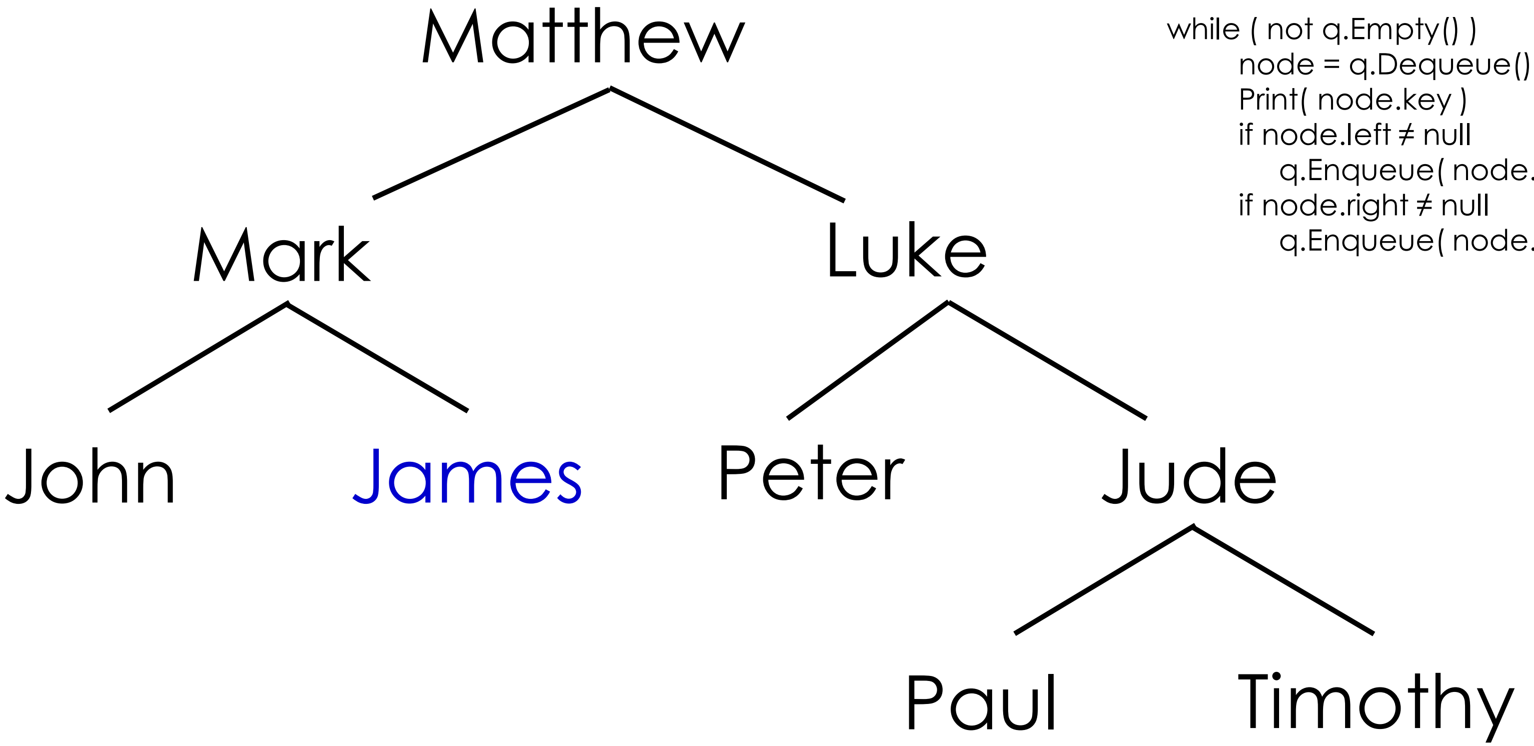
Output: Matthew, Mark, Luke, John

node

Queue: James, Peter, Jude

BFT using Q

Dequeue ()




```
while ( not q.Empty() )
    node = q.Dequeue()
    Print( node.key )
    if node.left ≠ null
        q.Enqueue( node.left )
    if node.right ≠ null
        q.Enqueue( node.right )
```

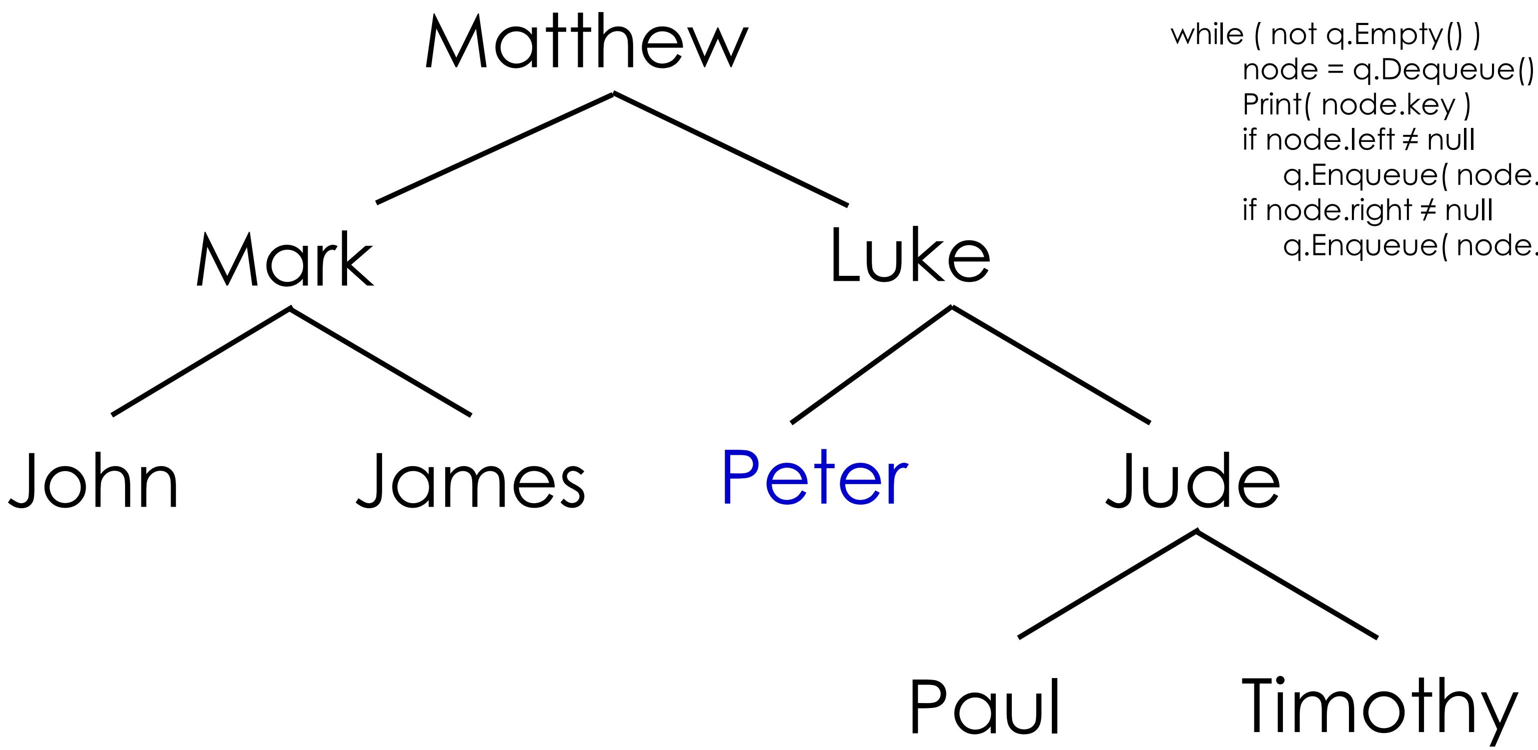
Output: Matthew, Mark, Luke, John, James

node

Queue: Peter, Jude

BFT using Q

Dequeue ()




```
while ( not q.Empty() )
        node = q.Dequeue()
        Print( node.key )
        if node.left ≠ null
            q.Enqueue( node.left )
        if node.right ≠ null
            q.Enqueue( node.right )
```

Output: Matthew, Mark, Luke, John, James, Peter

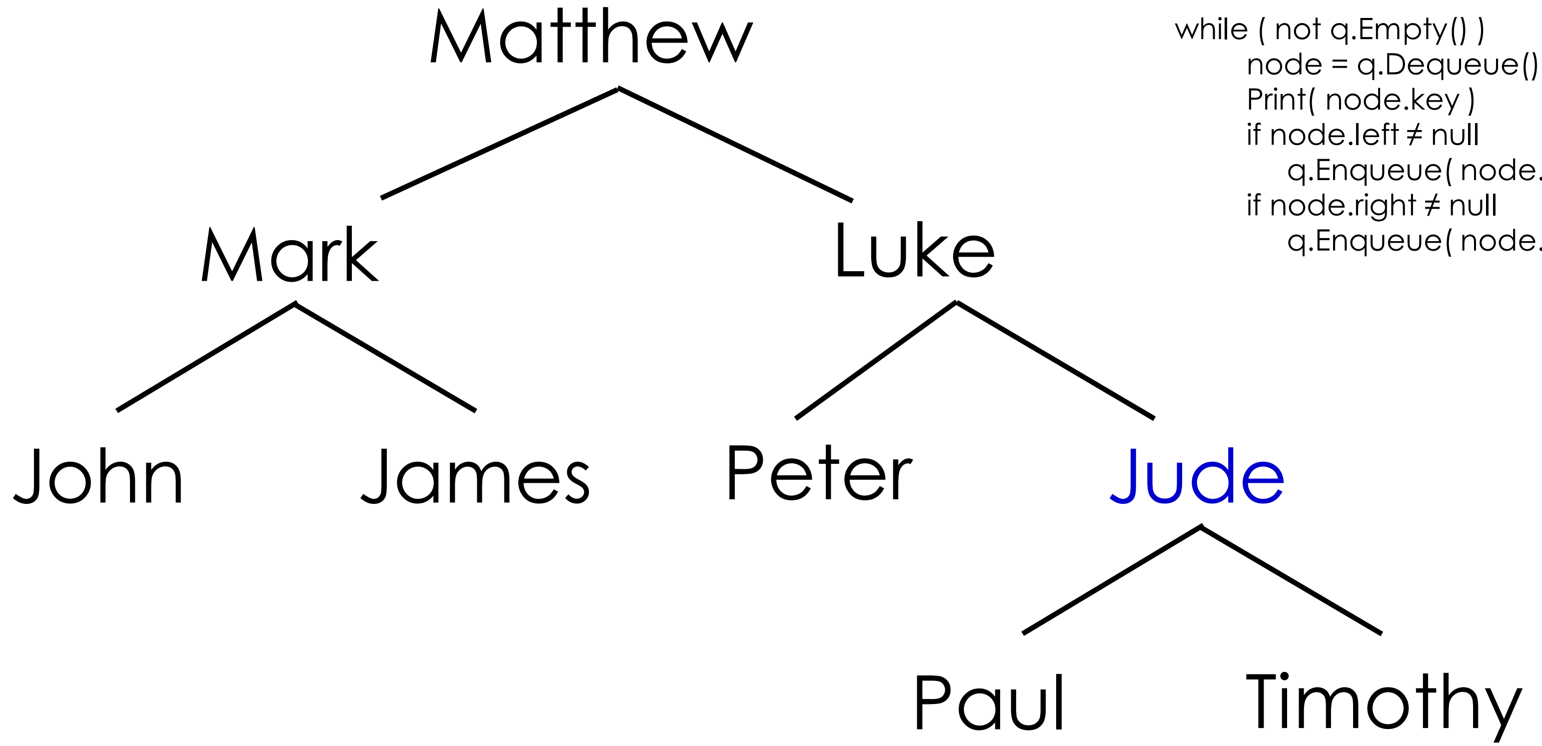
node

Queue: Jude

BFT using Q

Deque (

)




```
while ( not q.Empty() )
    node = q.Dequeue()
    Print( node.key )
    if node.left ≠ null
        q.Enqueue( node.left )
    if node.right ≠ null
        q.Enqueue( node.right )
```

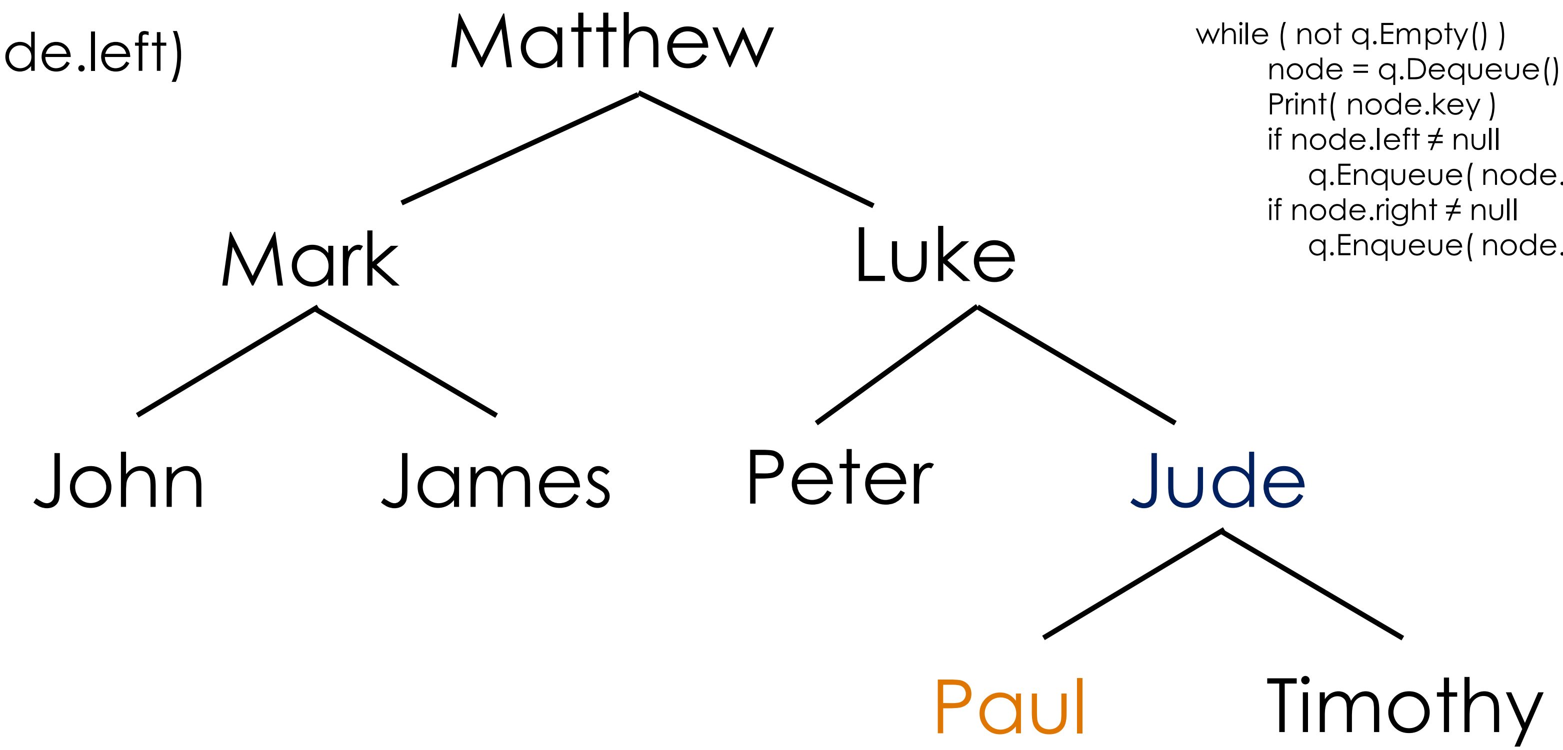
Output: Matthew, Mark, Luke, John, James, Peter, Jude

node

Queue:

BFT using Q

Enqueue (node.left)




```
while ( not q.Empty() )
        node = q.Dequeue()
        Print( node.key )
        if node.left ≠ null
             q.Enqueue( node.left )
        if node.right ≠ null
             q.Enqueue( node.right )
```

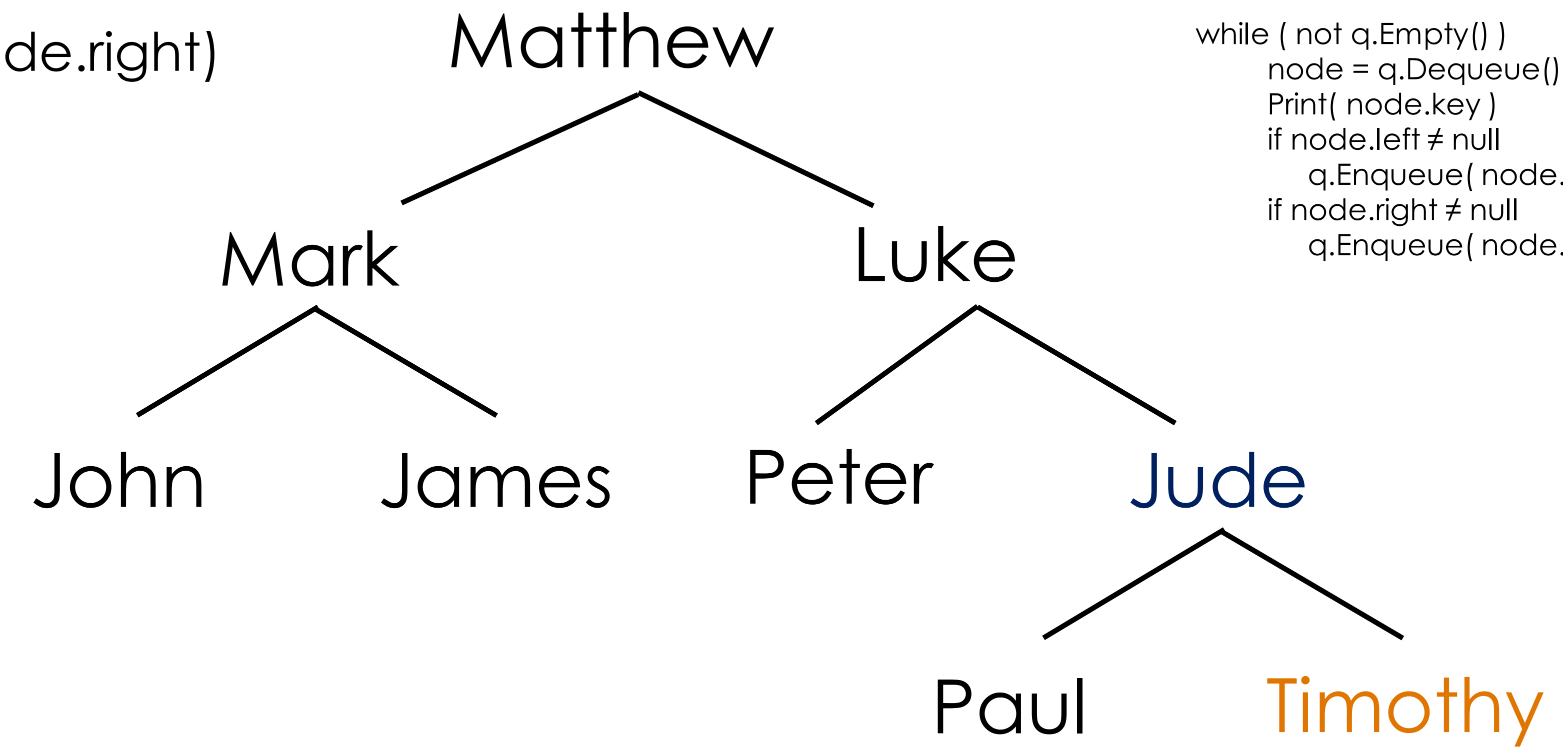
Output: Matthew, Mark, Luke, John, James, Peter, Jude

node

Queue: Paul

BFT using Q

Enqueue (node.right)




```
while ( not q.Empty() )
    node = q.Dequeue()
    Print( node.key )
    if node.left ≠ null
        q.Enqueue( node.left )
    if node.right ≠ null
        q.Enqueue( node.right )
```

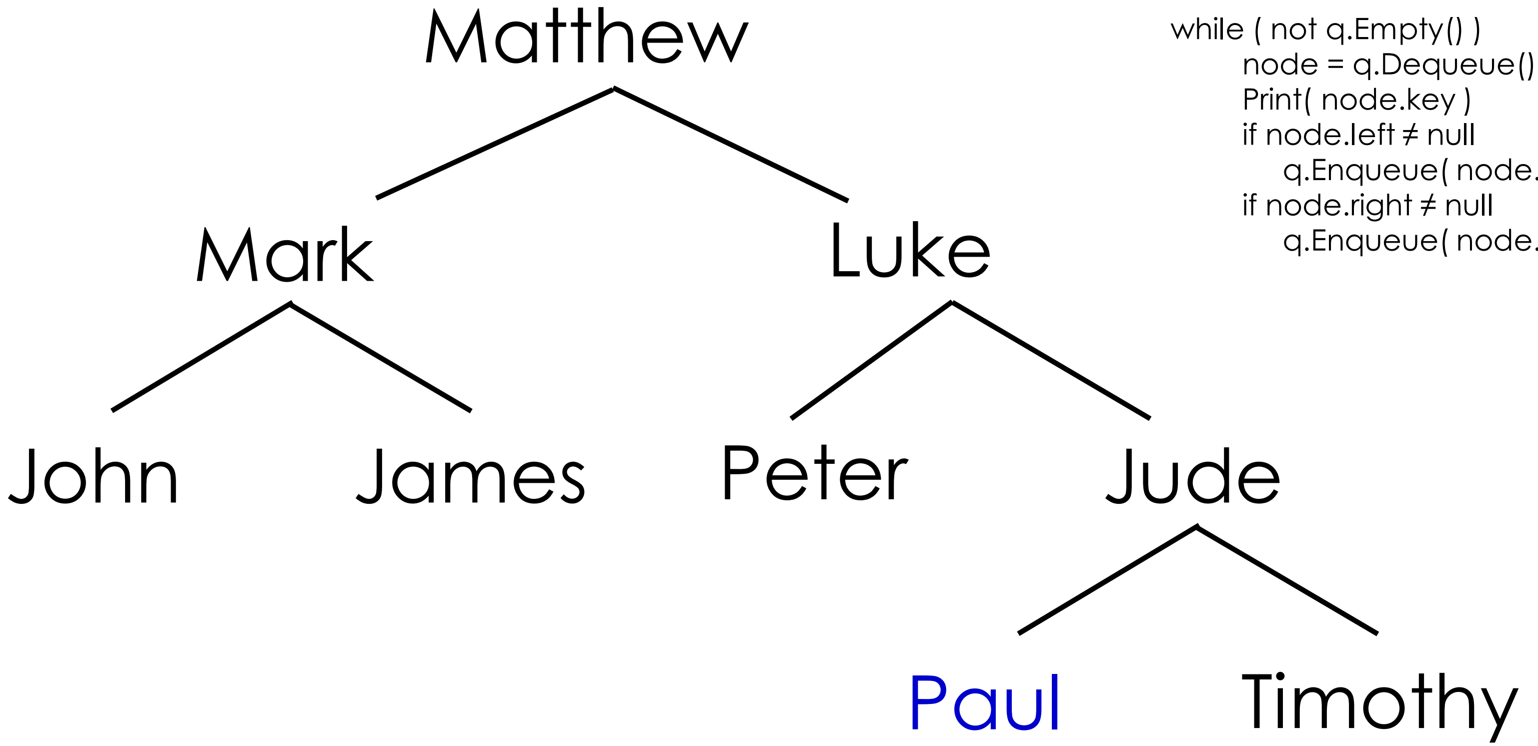
Output: Matthew, Mark, Luke, John, James, Peter, Jude

node

Queue: Paul, Timothy

BFT using Q

Dequeue ()




```
while ( not q.Empty() )
    node = q.Dequeue()
    Print( node.key )
    if node.left ≠ null
        q.Enqueue( node.left )
    if node.right ≠ null
        q.Enqueue( node.right )
```

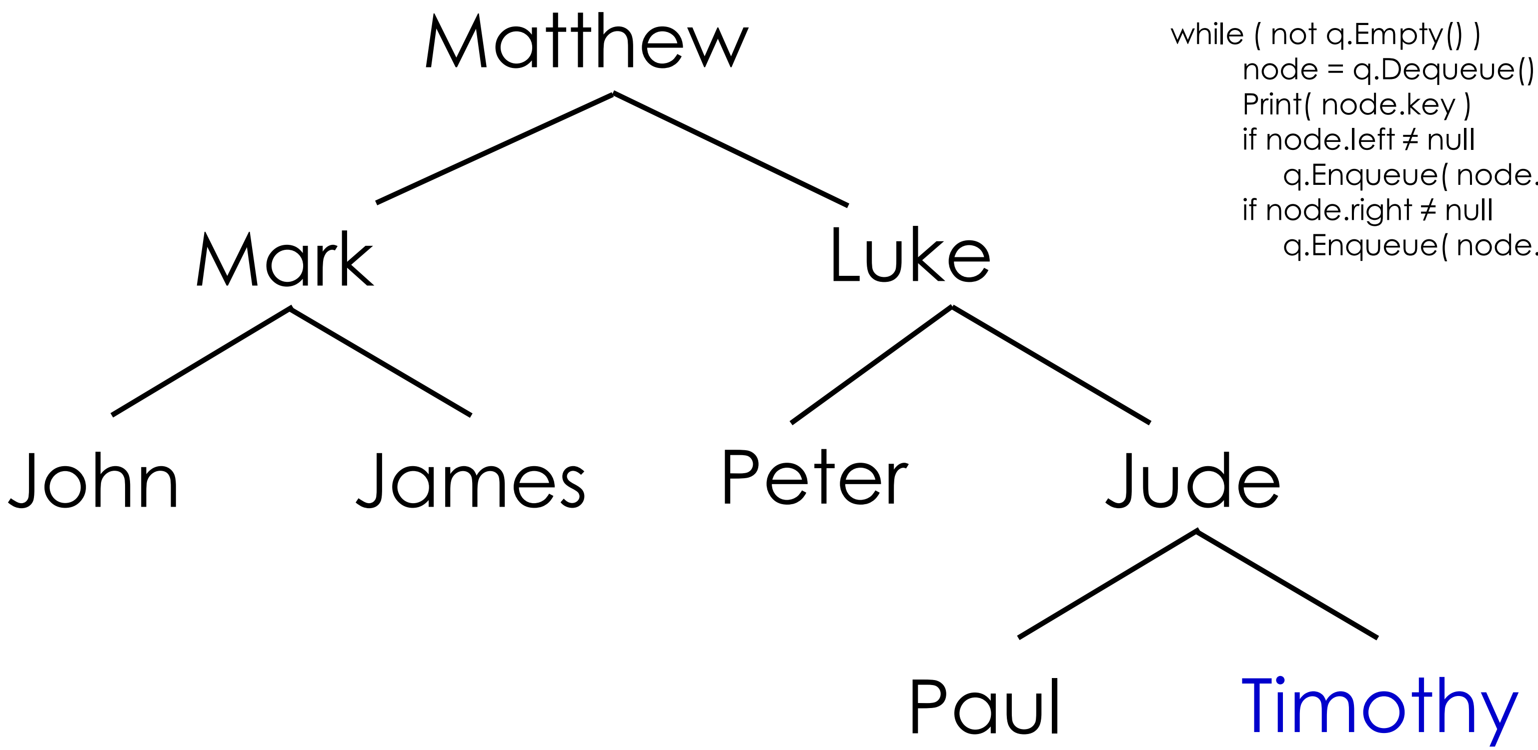
Output: Matthew, Mark, Luke, John, James, Peter, Jude, Paul

node

Queue: Timothy

BFT using Q

Deque (): Dequeue ()




```
while ( not q.Empty() )
    node = q.Dequeue()
    Print( node.key )
    if node.left ≠ null
        q.Enqueue( node.left )
    if node.right ≠ null
        q.Enqueue( node.right )
```

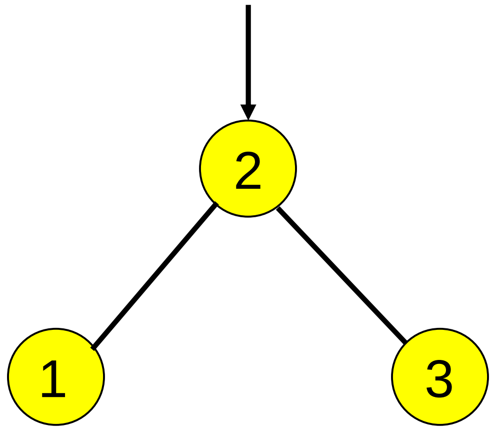
Output: Matthew, Mark, Luke, John, James, Peter, Jude, Paul, Timothy

node

Queue:

Depth-first Traversal Implementation using Recursion

- We completely traverse one sub-tree to before exploring a sibling sub-tree
- We can implement DFT using
  - Stack
  - Recursion
- Three modes of traversal
  - PreOrder, InOrder, PostOrder



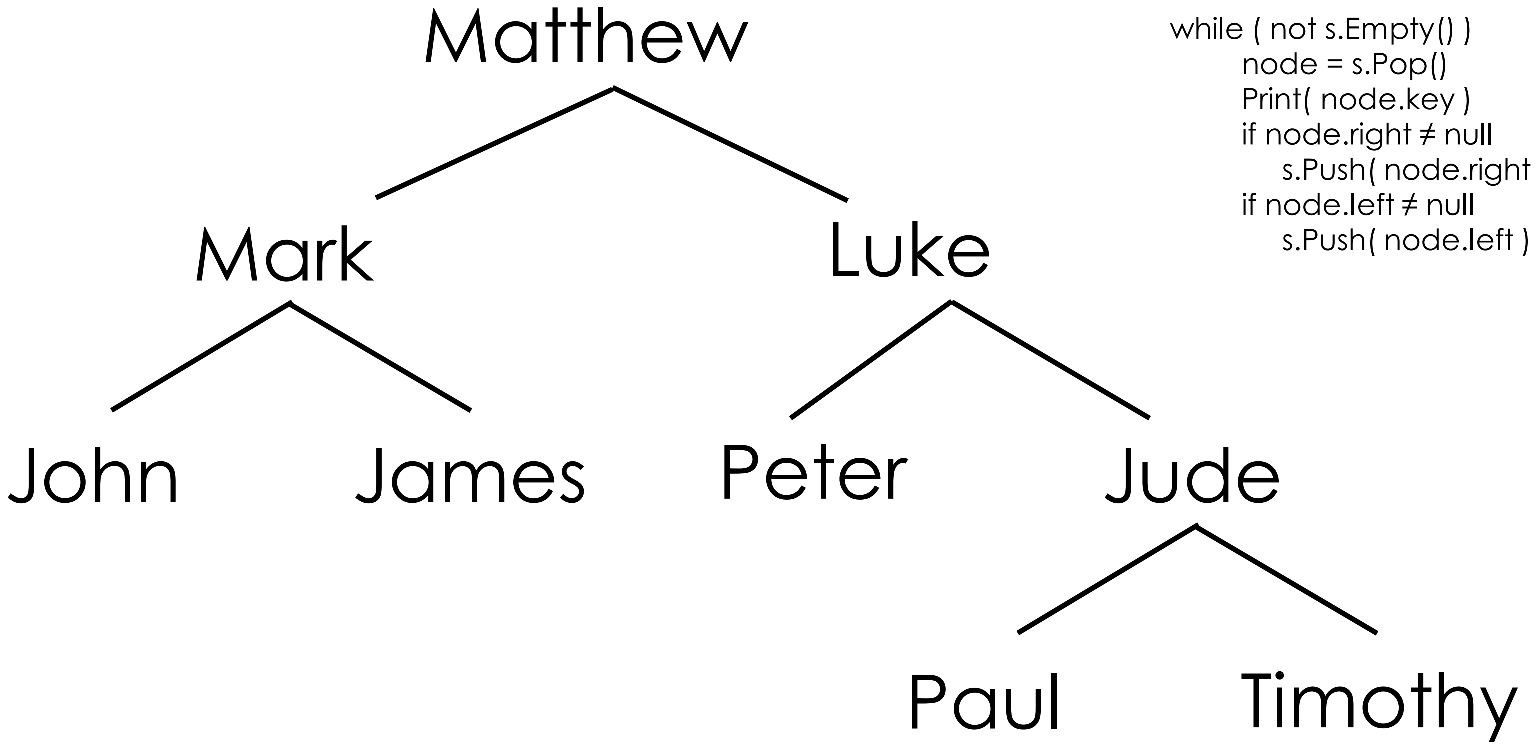
PreOrder: 2, 1, 3
InOrder: 1, 2, 3
PostOrder: 1, 3, 2

Non-recursive Implementation of Depth-first Traversal

nonRecursiveDFT (Node node)

```
if node == null
    return
else
    Stack s
    s.push( node )
    while ( not s.Empty() )
        node = s.Pop()
        Print( node.key ) // Do something to the node
        if node.right ≠ null
            s.push( node.right )
        if node.left ≠ null
            s.push( node.left )
```

DFT using Stack



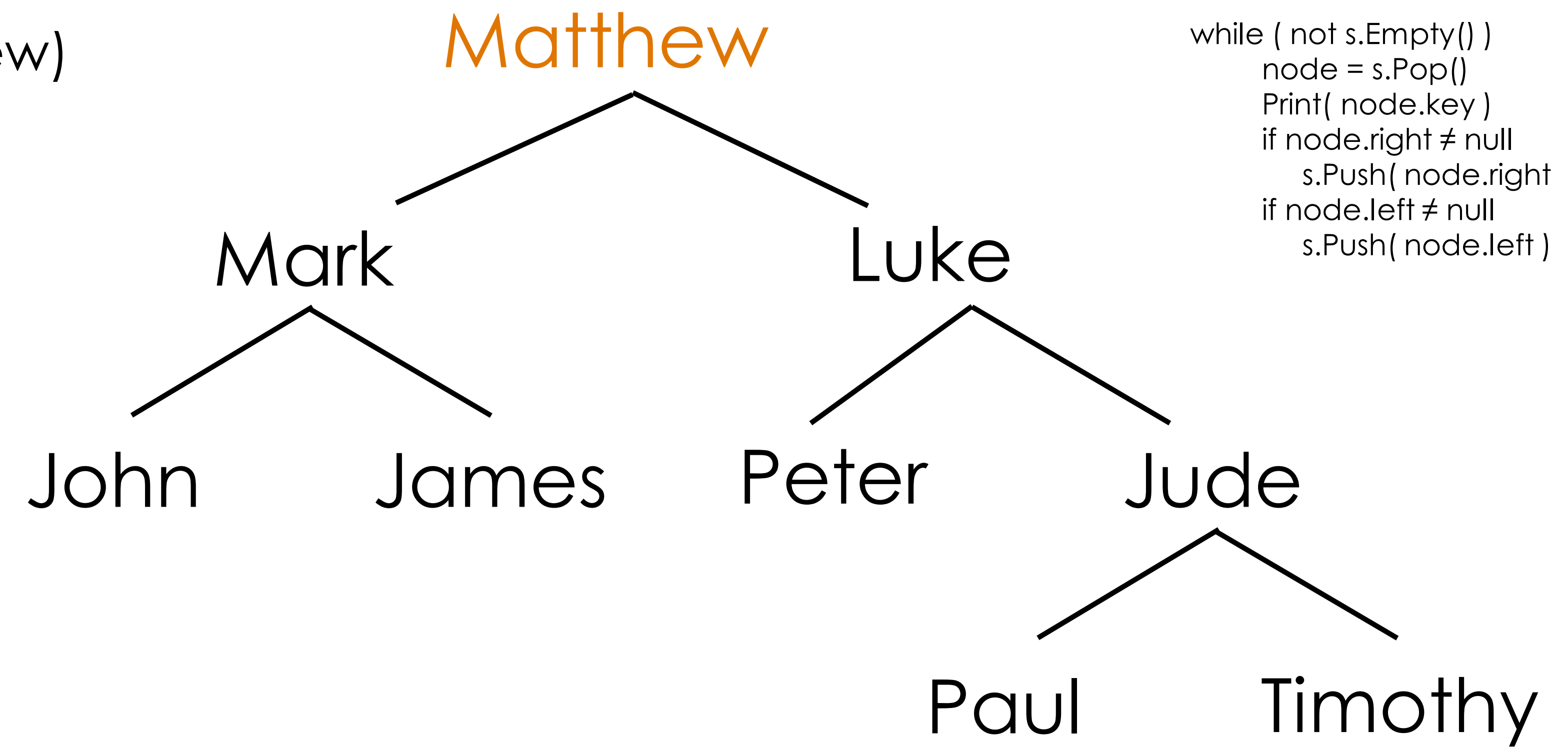

```
while ( not s.Empty() )
        node = s.Pop()
        Print( node.key )
        if node.right ≠ null
             s.Push( node.right )
        if node.left ≠ null
             s.Push( node.left )
```

Output:

Stack:

DFT using Stack

Push (Matthew)



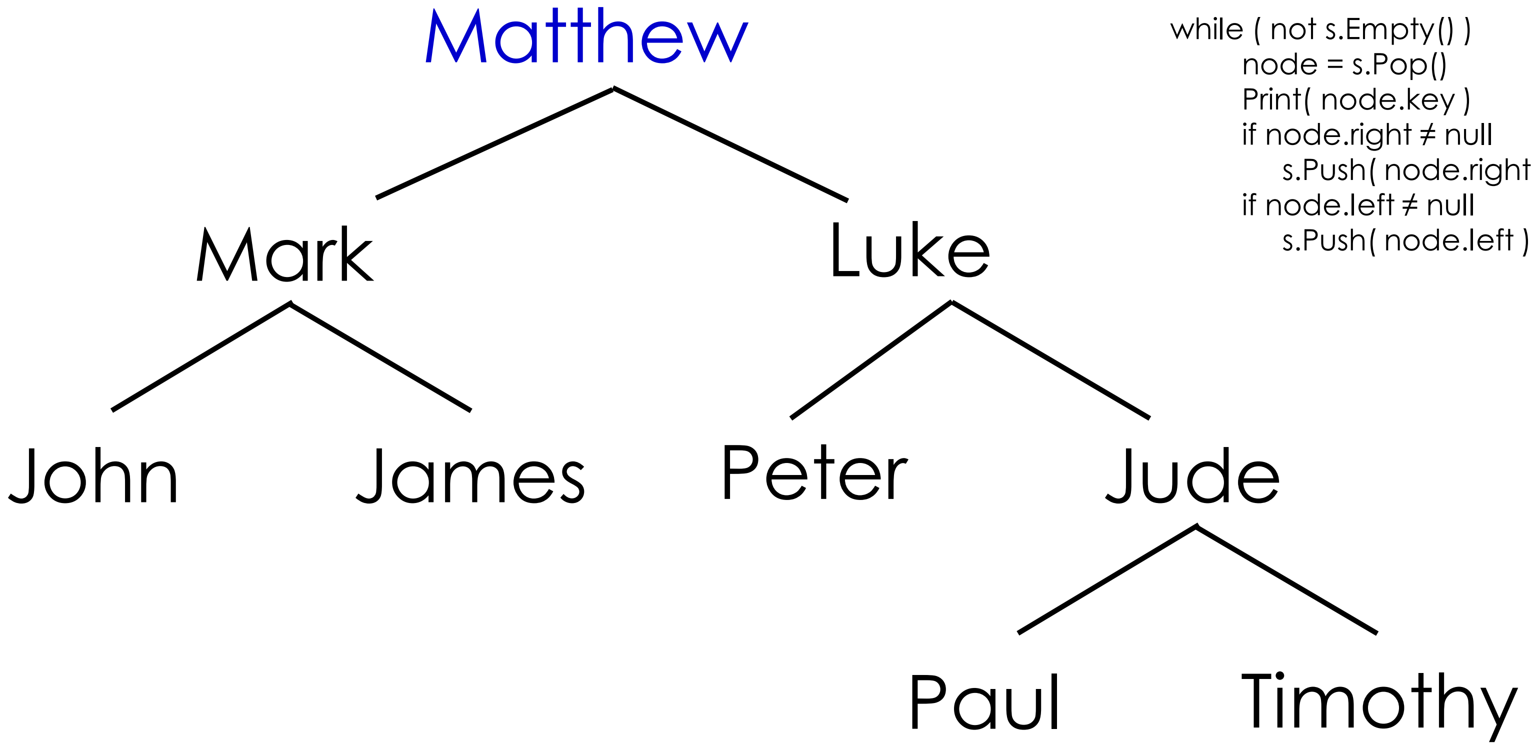

```
while ( not s.Empty() )
    node = s.Pop()
    Print( node.key )
    if node.right ≠ null
        s.Push( node.right )
    if node.left ≠ null
        s.Push( node.left )
```

Output:

Stack: Matthew

DFT using Stack

Pop ()



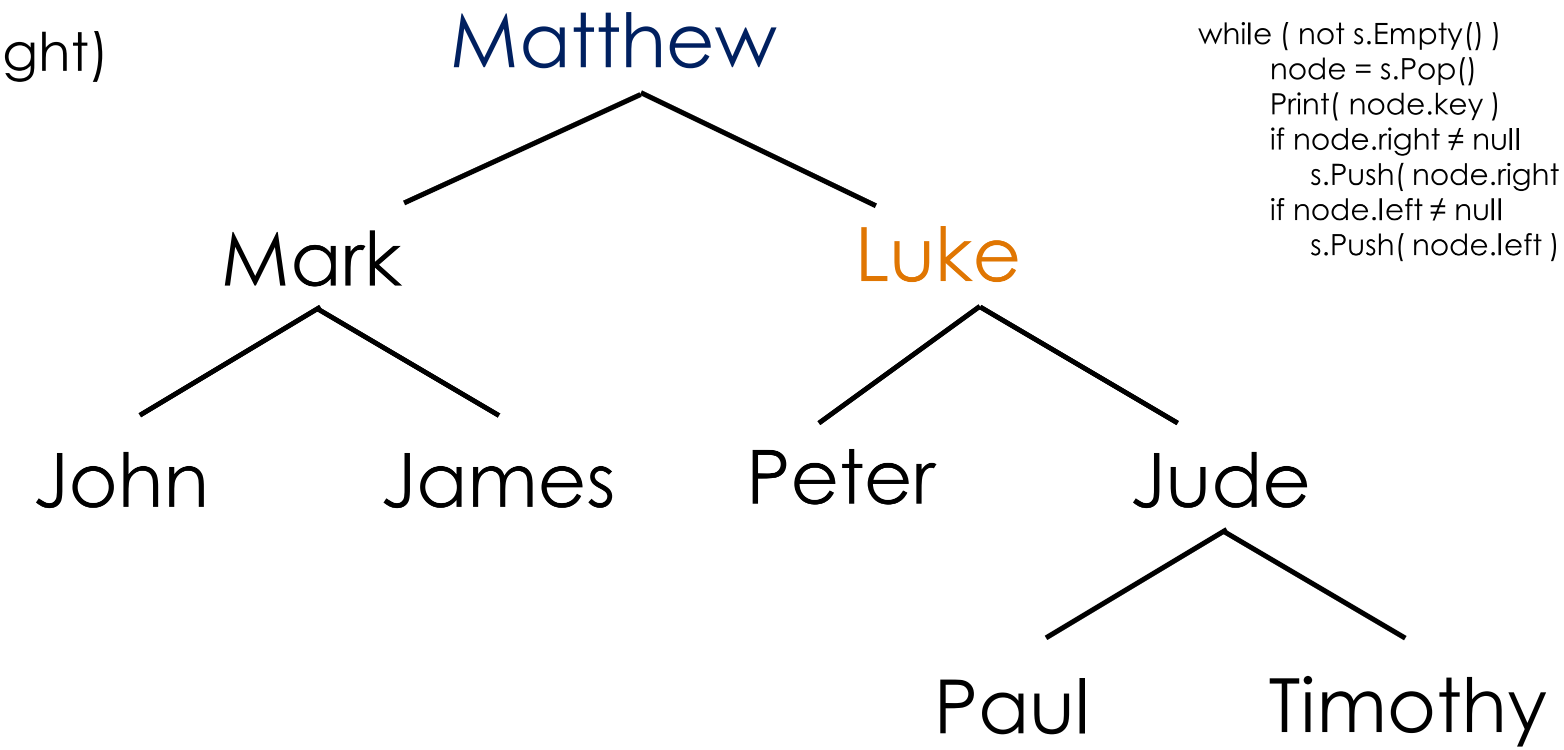

```
while ( not s.Empty() )
    node = s.Pop()
    Print( node.key )
    if node.right ≠ null
        s.Push( node.right )
    if node.left ≠ null
        s.Push( node.left )
```

Output: Matthew
node

Stack:

DFT using Stack

Push (node.right)




```
while ( not s.Empty() )
    node = s.Pop()
    Print( node.key )
    if node.right ≠ null
        s.Push( node.right )
    if node.left ≠ null
        s.Push( node.left )
```

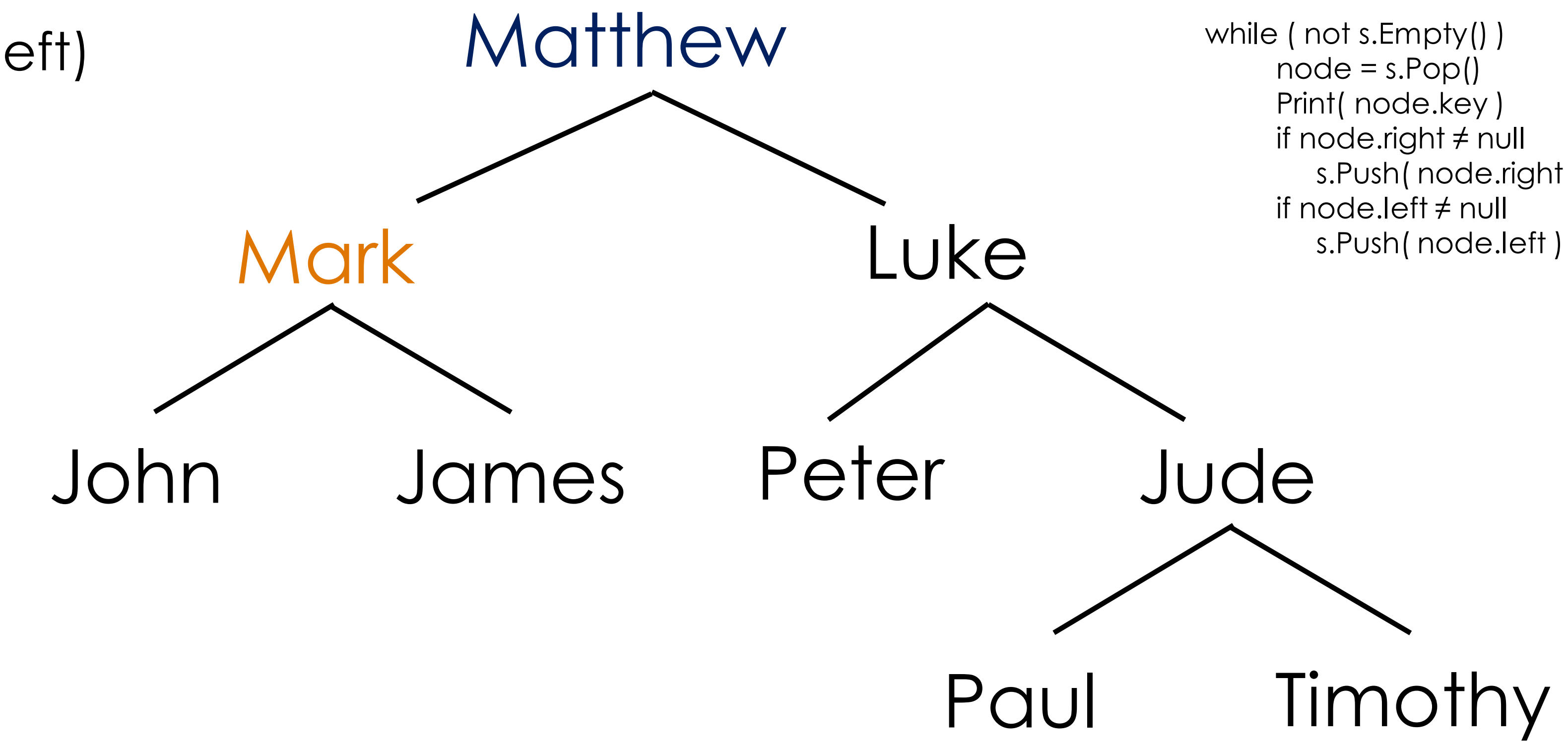
Output: Matthew

node

Stack: Luke

DFT using Stack

Push (node.left)




```
while ( not s.Empty() )
    node = s.Pop()
    Print( node.key )
    if node.right ≠ null
        s.Push( node.right )
    if node.left ≠ null
        s.Push( node.left )
```

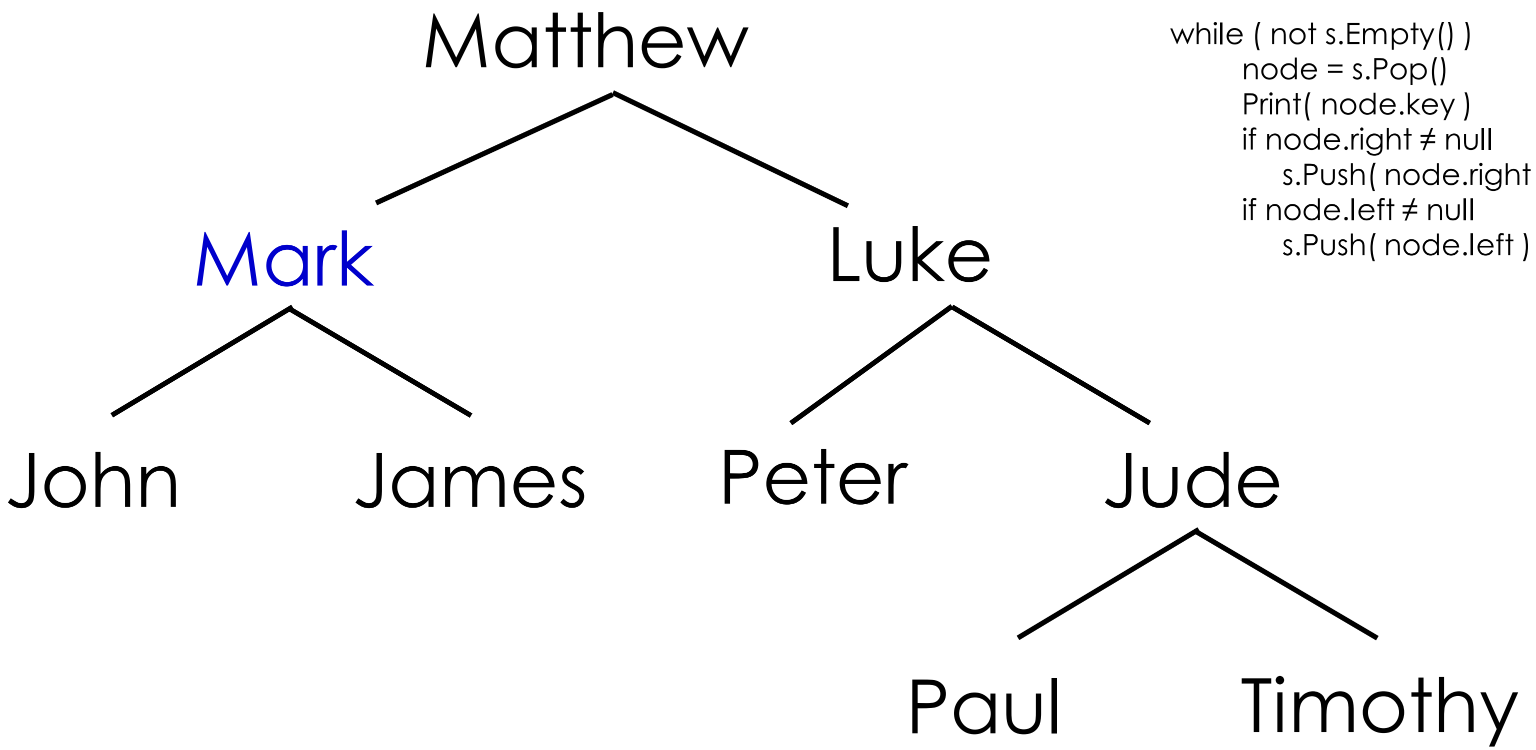
Output: Matthew

node

Stack: Luke, Mark

DFT using Stack

Pop ()




```
while ( not s.Empty() )
    node = s.Pop()
    Print( node.key )
    if node.right ≠ null
        s.Push( node.right )
    if node.left ≠ null
        s.Push( node.left )
```

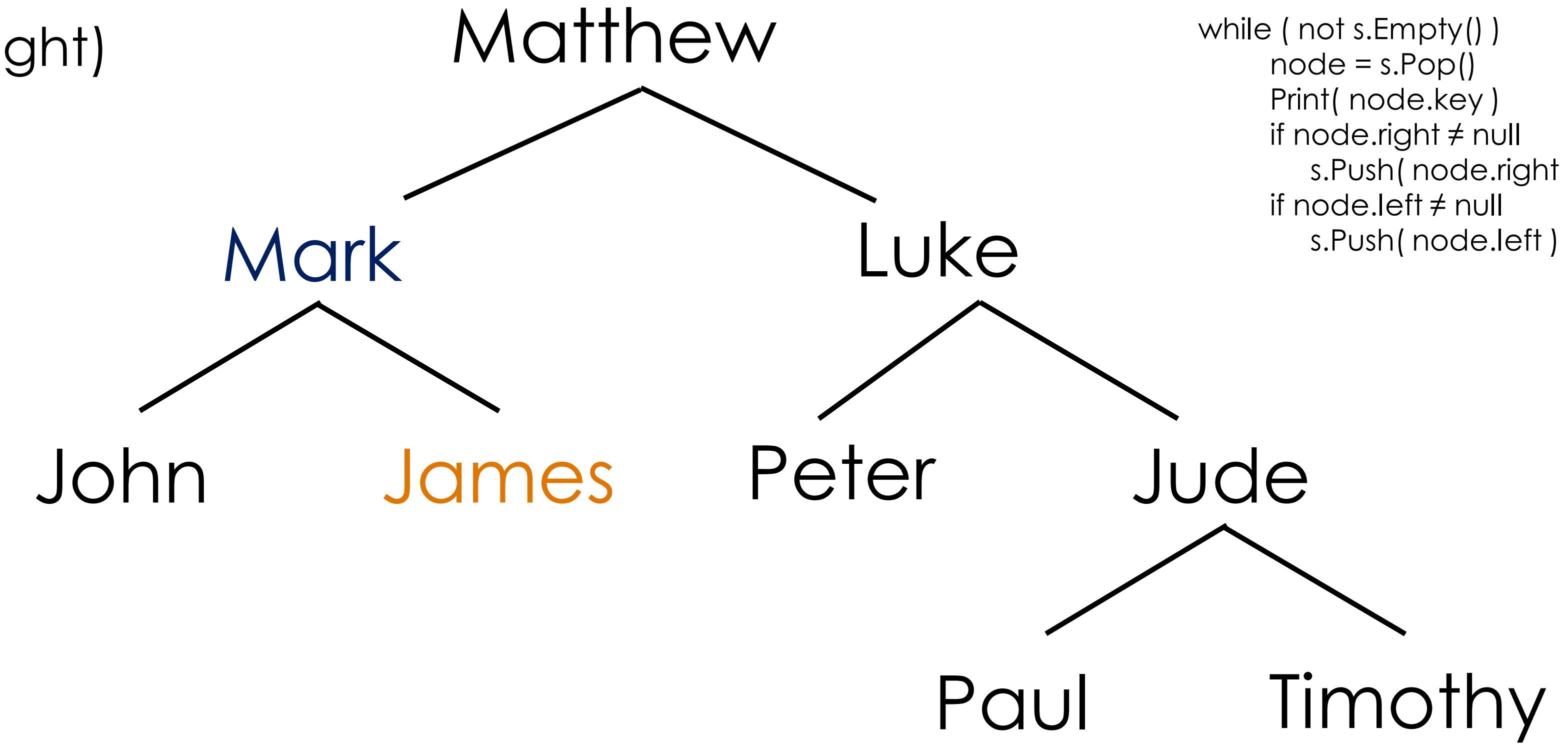
Output: Matthew, Mark

node

Stack: Luke

DFT using Stack

Push (node.right)




```
while ( not s.Empty() )
        node = s.Pop()
        Print( node.key )
        if node.right ≠ null
            s.Push( node.right )
        if node.left ≠ null
            s.Push( node.left )
```

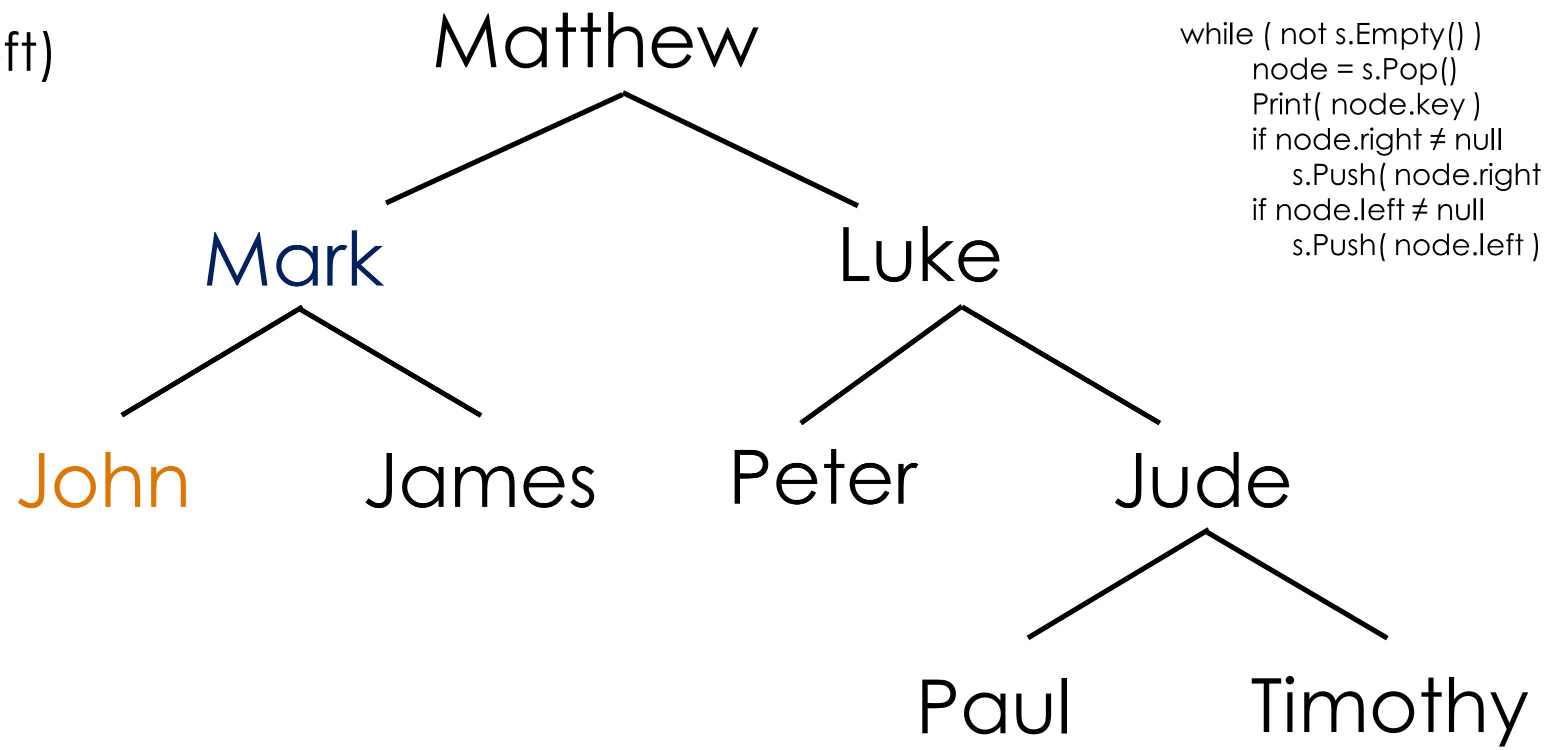
Output: Matthew, Mark

node

Stack: Luke, James

DFT using Stack

Push (node.left)




```
while ( not s.Empty() )
    node = s.Pop()
    Print( node.key )
    if node.right ≠ null
        s.Push( node.right )
    if node.left ≠ null
        s.Push( node.left )
```

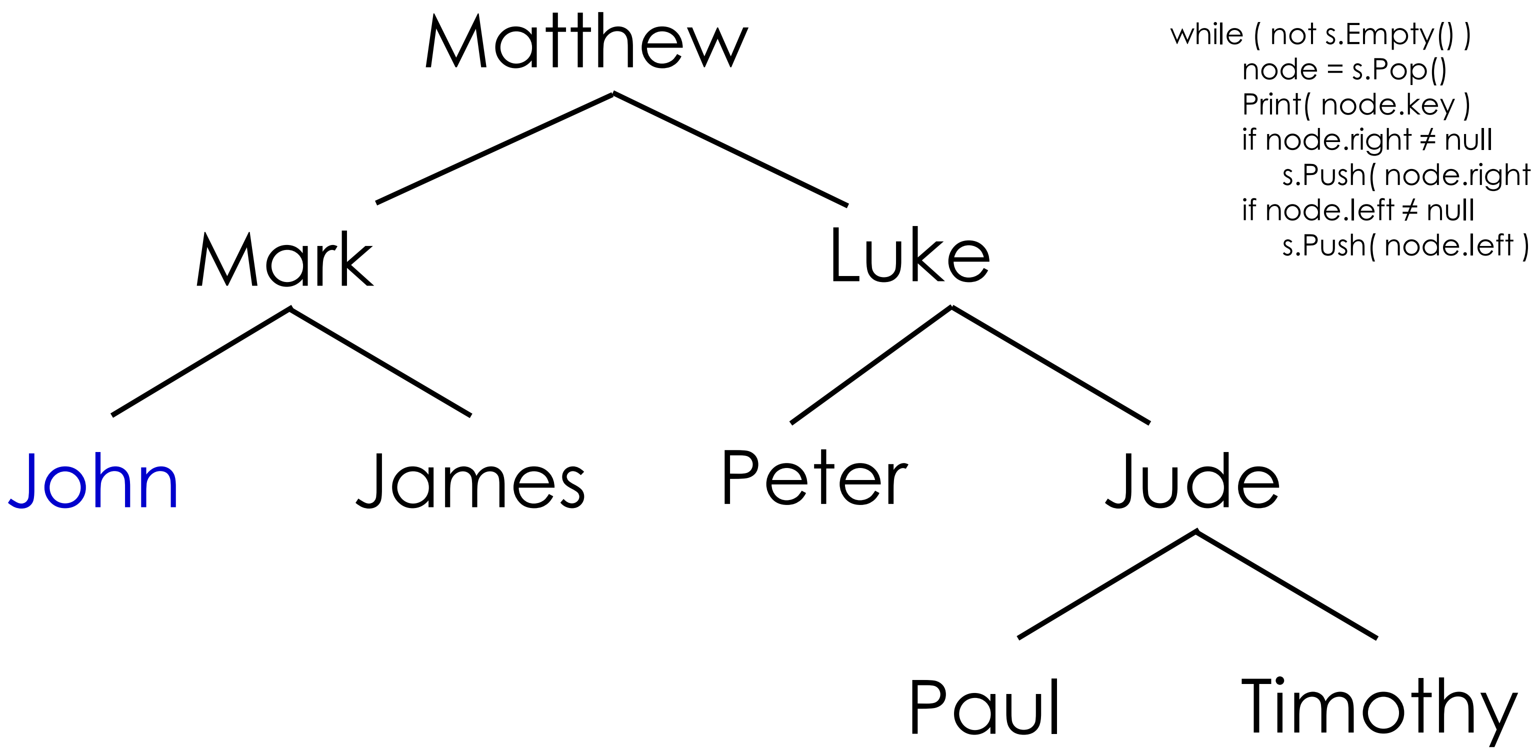
Output: Matthew, Mark

node

Stack: Luke, James, John

DFT using Stack

Pop ()



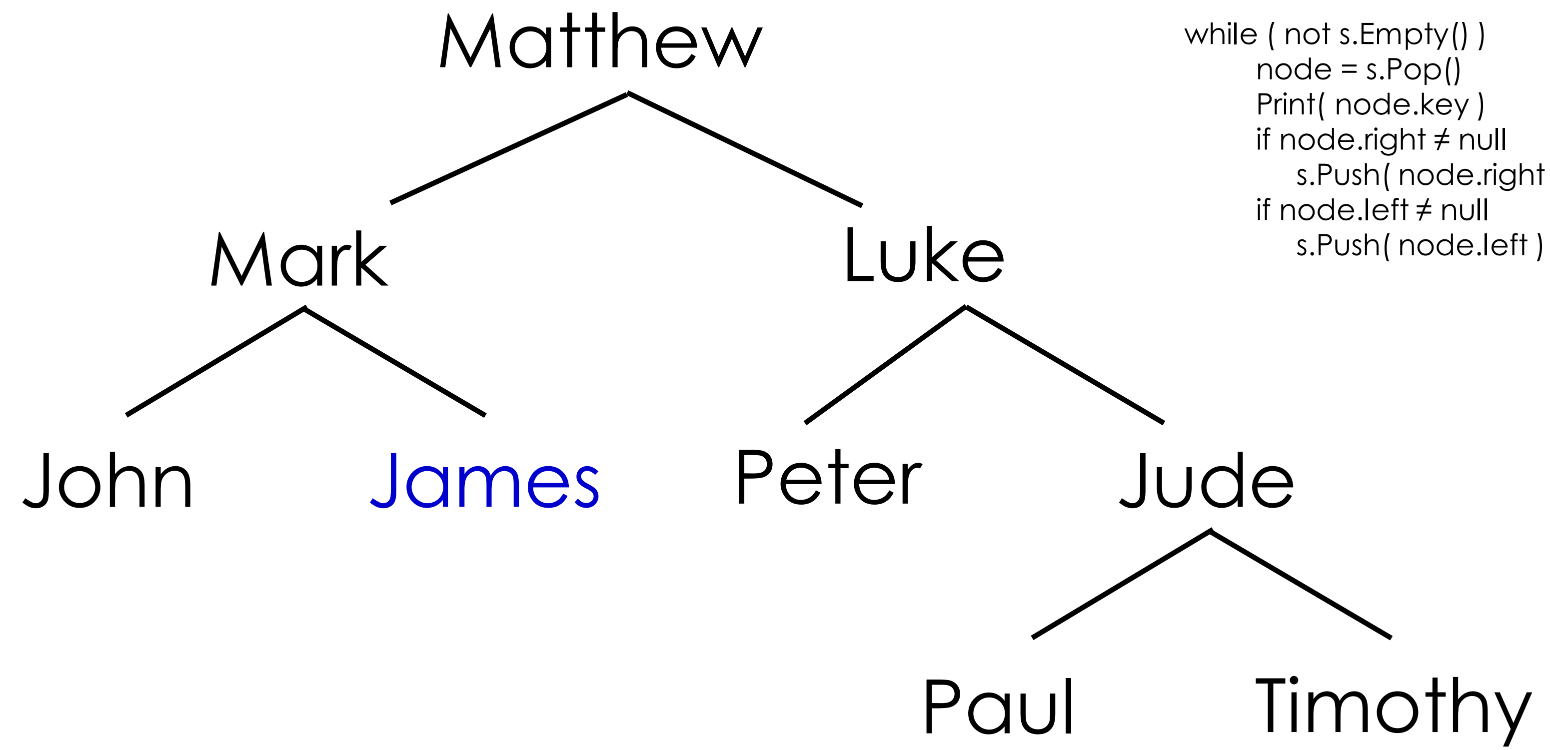

```
while ( not s.Empty() )
        node = s.Pop()
        Print( node.key )
        if node.right ≠ null
            s.Push( node.right )
        if node.left ≠ null
            s.Push( node.left )
```

Output: Matthew, Mark, John
node

Stack: Luke, James

DFT using Stack

Pop ()




```
while ( not s.Empty() )
        node = s.Pop()
        Print( node.key )
        if node.right ≠ null
            s.Push( node.right )
        if node.left ≠ null
            s.Push( node.left )
```

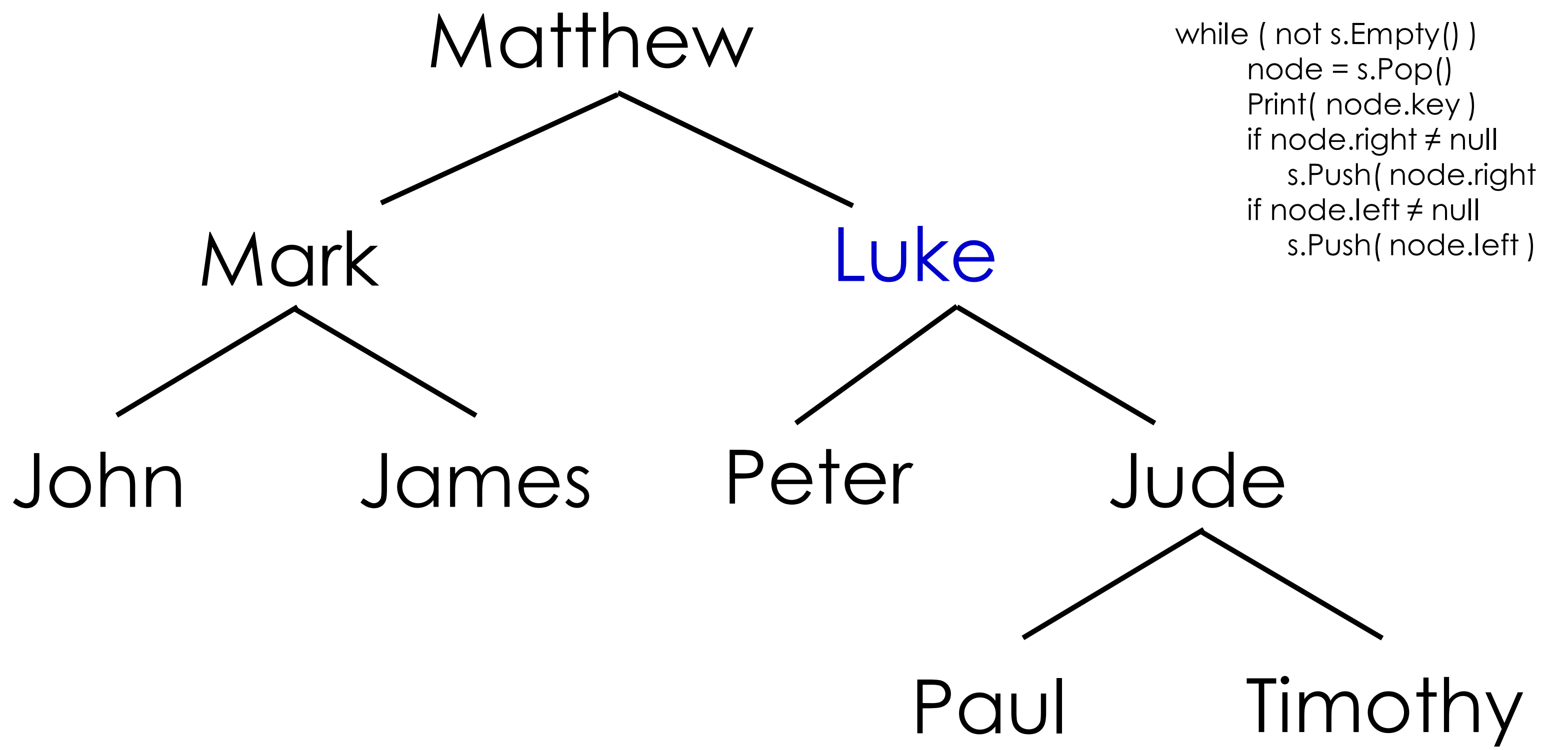
Output: Matthew, Mark, John, James

node

Stack: Luke

DFT using Stack

Pop ()




```
while ( not s.Empty() )
    node = s.Pop()
    Print( node.key )
    if node.right ≠ null
        s.Push( node.right )
    if node.left ≠ null
        s.Push( node.left )
```

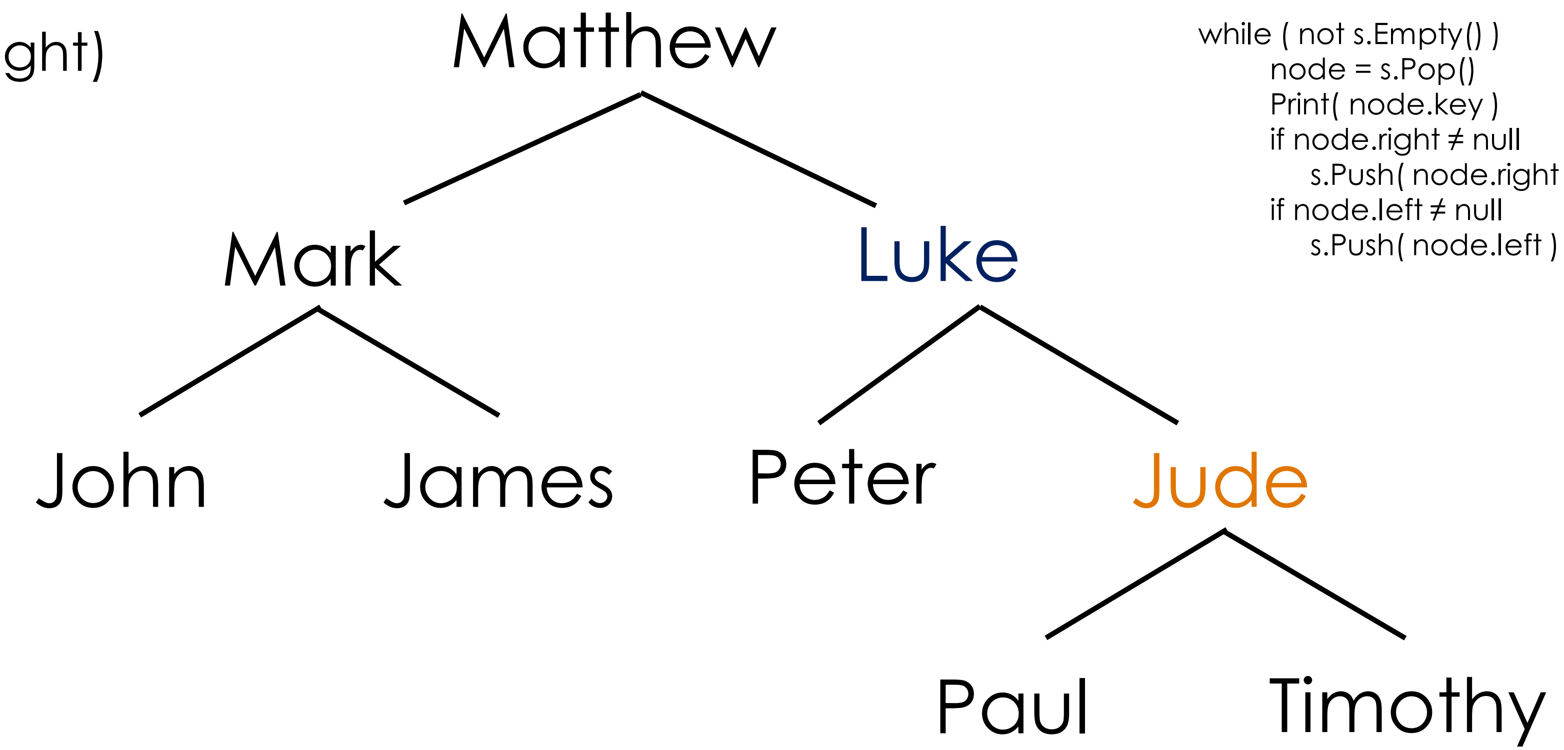
Output: Matthew, Mark, John, James, Luke

node

Stack:

DFT using Stack

Push (node.right)




```
while ( not s.Empty() )
    node = s.Pop()
    Print( node.key )
    if node.right ≠ null
        s.Push( node.right )
    if node.left ≠ null
        s.Push( node.left )
```

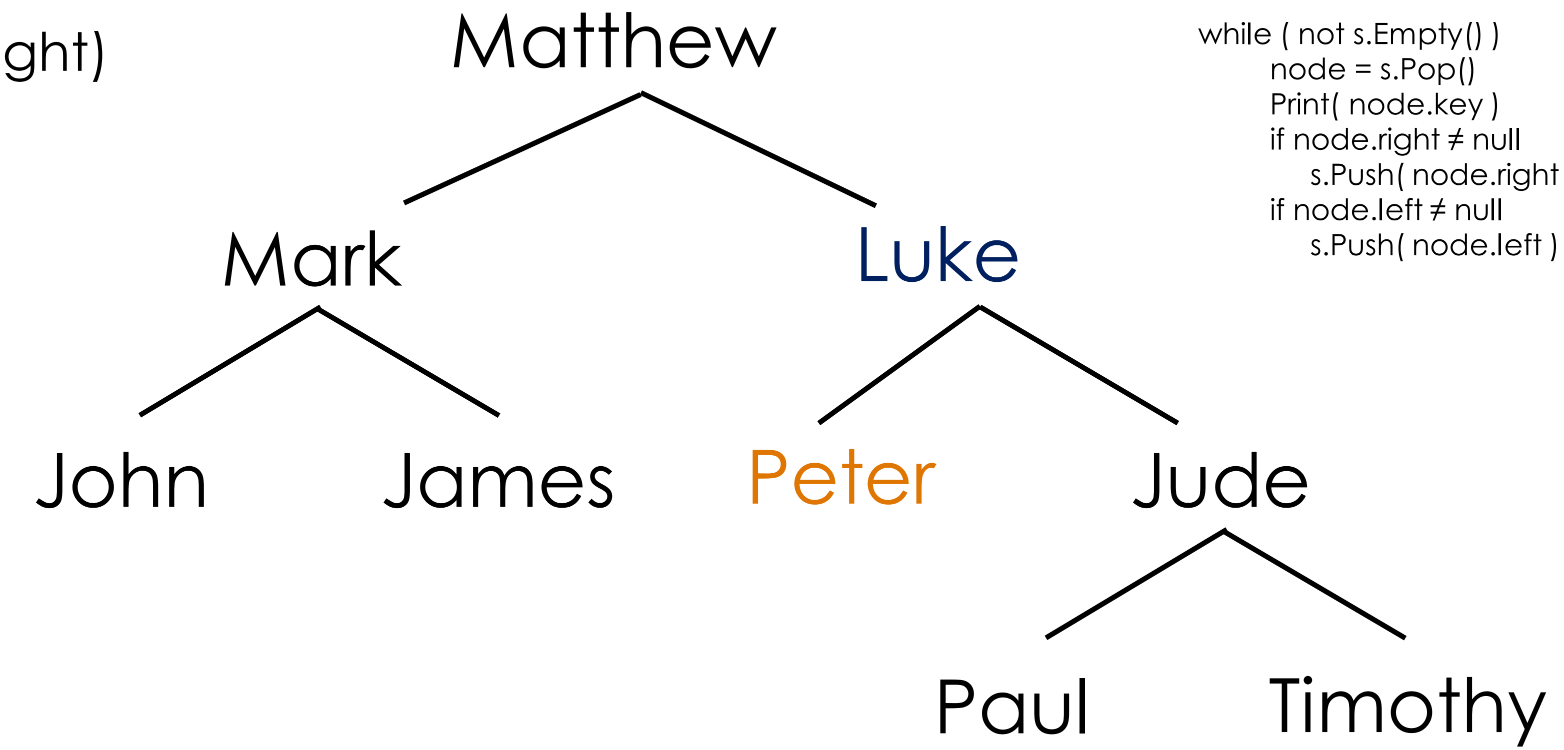
Output: Matthew, Mark, John, James, Luke

node

Stack: Jude

DFT using Stack

Push (node.right)




```
while ( not s.Empty() )
    node = s.Pop()
    Print( node.key )
    if node.right ≠ null
        s.Push( node.right )
    if node.left ≠ null
        s.Push( node.left )
```

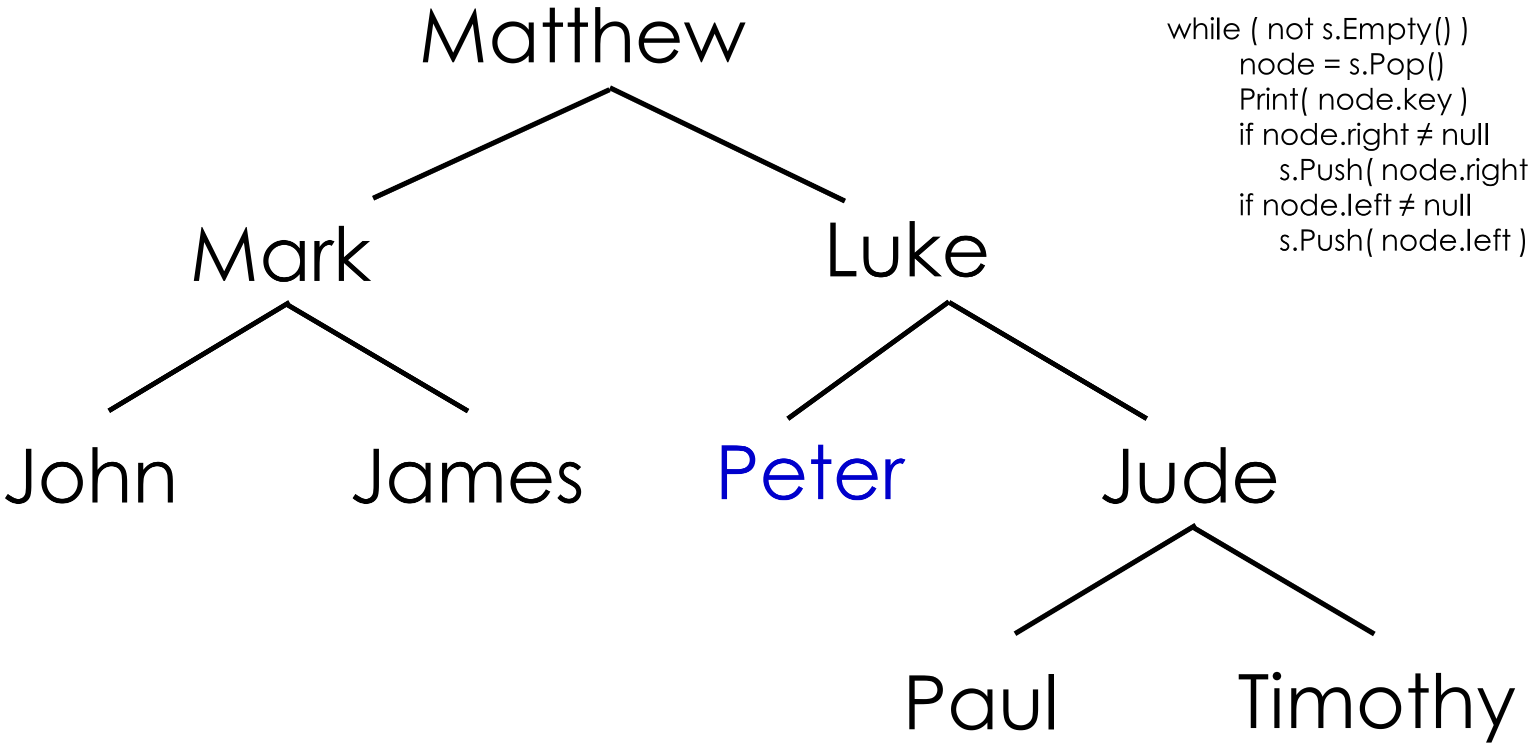
Output: Matthew, Mark, John, James, Luke

node

Stack: Jude, Peter

DFT using Stack

Pop ()




```
while ( not s.Empty() )
        node = s.Pop()
        Print( node.key )
        if node.right ≠ null
            s.Push( node.right )
        if node.left ≠ null
            s.Push( node.left )
```

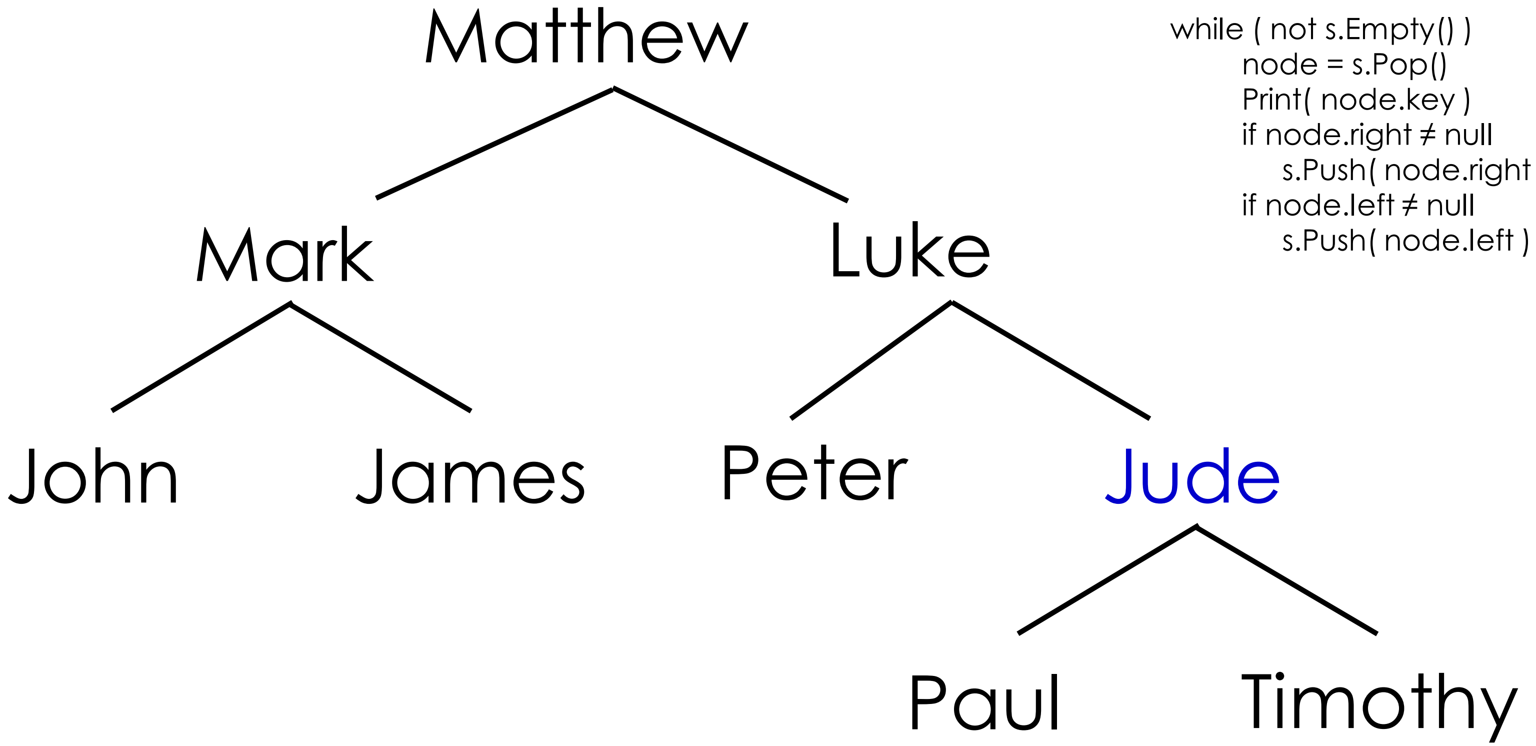
Output: Matthew, Mark, John, James, Luke, Peter

node

Stack: Jude

DFT using Stack

Pop ()




```
while ( not s.Empty() )
        node = s.Pop()
        Print( node.key )
        if node.right ≠ null
            s.Push( node.right )
        if node.left ≠ null
            s.Push( node.left )
```

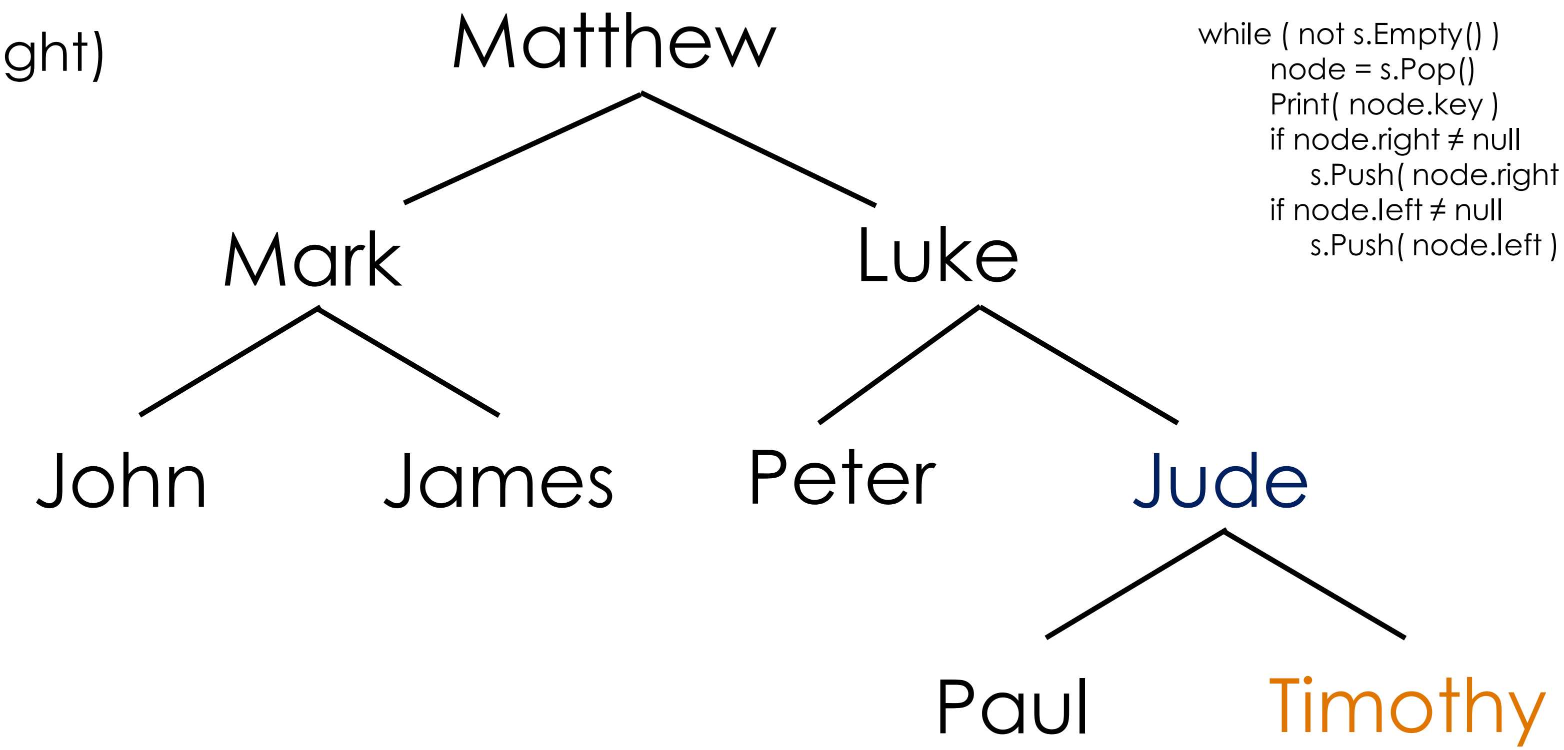
Output: Matthew, Mark, John, James, Luke, Peter, Jude

node

Stack:

DFT using Stack

Push (node.right)




```
while ( not s.Empty() )
    node = s.Pop()
    Print( node.key )
    if node.right ≠ null
        s.Push( node.right )
    if node.left ≠ null
        s.Push( node.left )
```

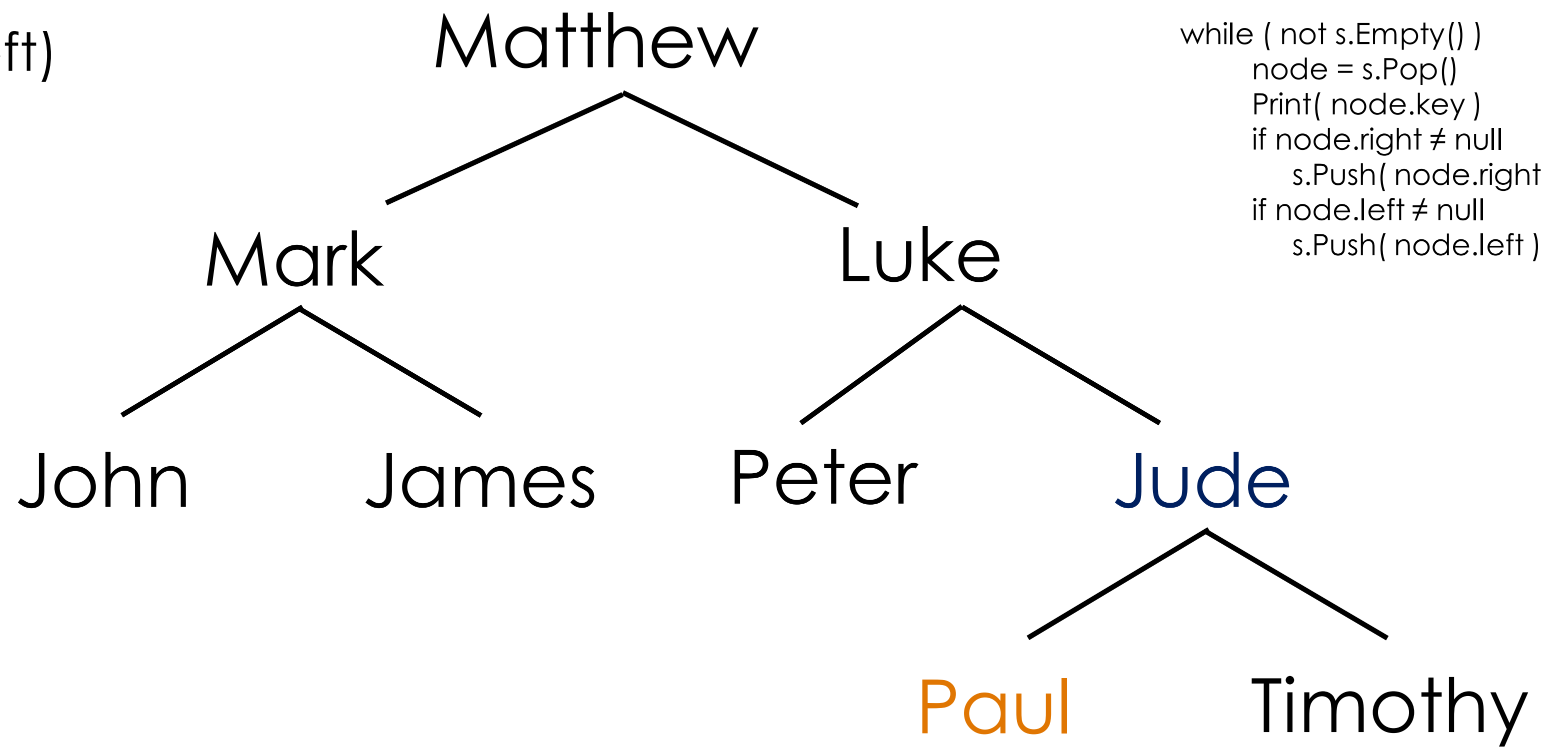
Output: Matthew, Mark, John, James, Luke, Peter, Jude

node

Stack: Timothy

DFT using Stack

Push (node.left)




```
while ( not s.Empty() )
    node = s.Pop()
    Print( node.key )
    if node.right ≠ null
        s.Push( node.right )
    if node.left ≠ null
        s.Push( node.left )
```

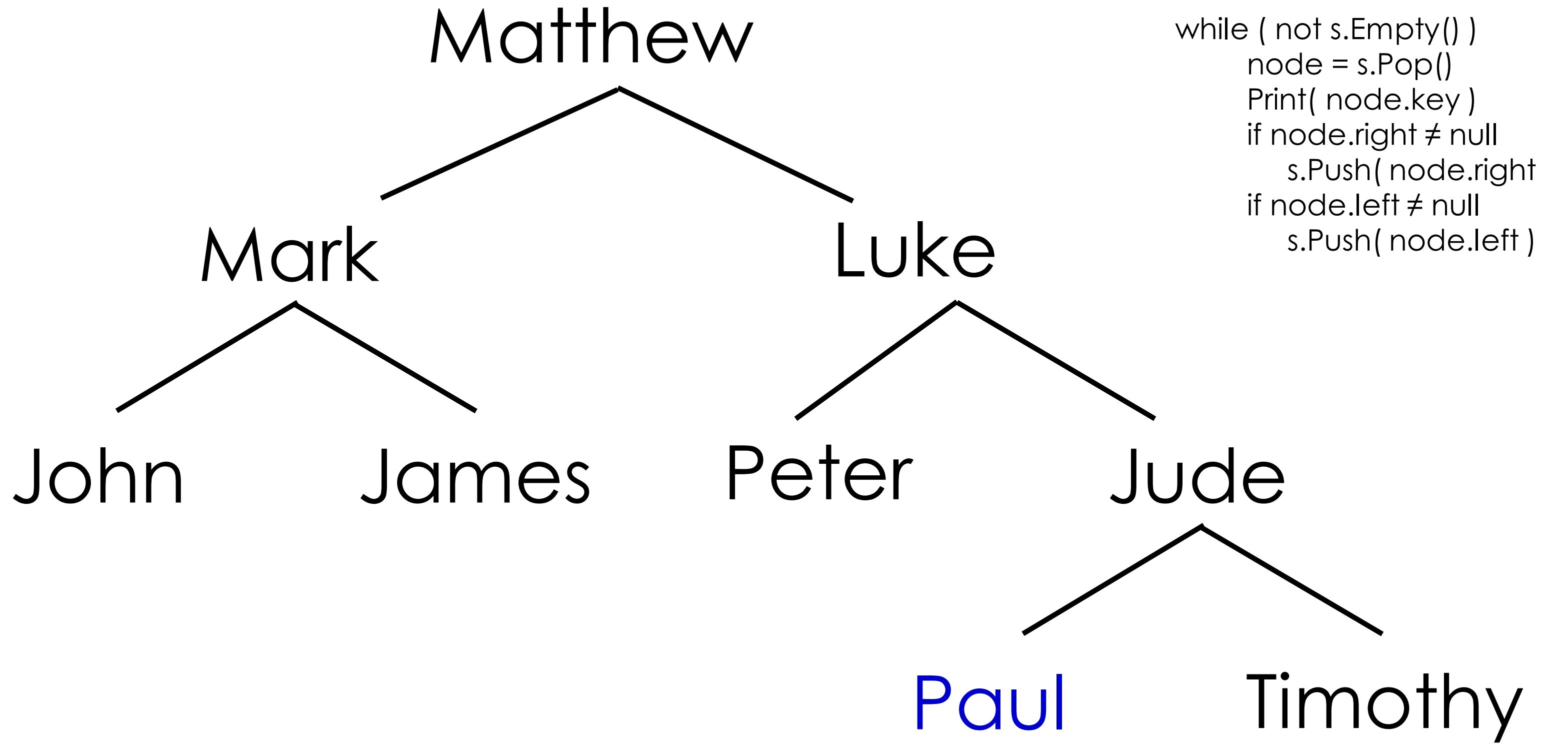
Output: Matthew, Mark, John, James, Luke, Peter, Jude

node

Stack: Timothy, Paul

DFT using Stack

Pop ()




```
while ( not s.Empty() )
      node = s.Pop()
      Print( node.key )
      if node.right ≠ null
          s.Push( node.right )
      if node.left ≠ null
          s.Push( node.left )
```

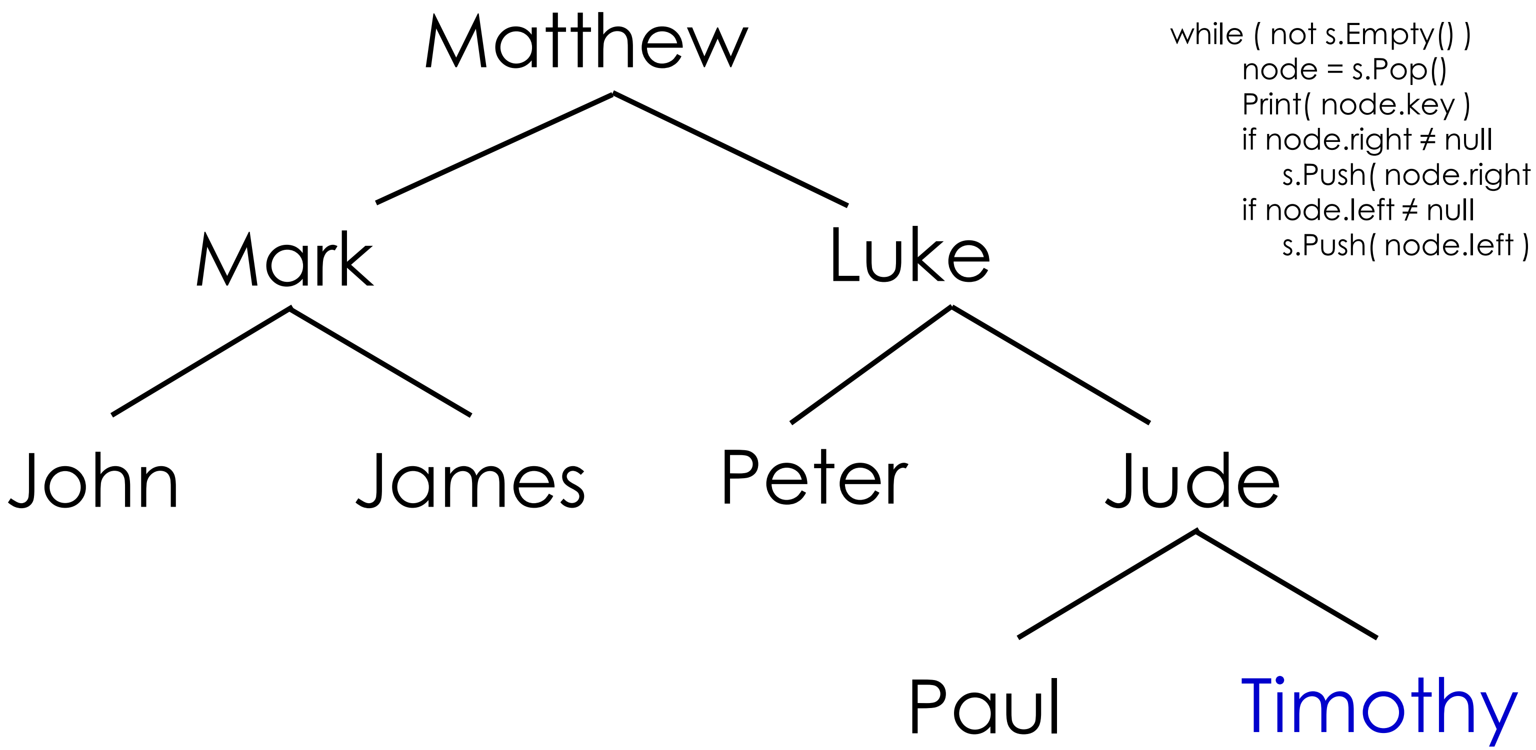
Output: Matthew, Mark, John, James, Luke, Peter, Jude, Paul

Stack: node

Timothy

DFT using Stack

Pop ()



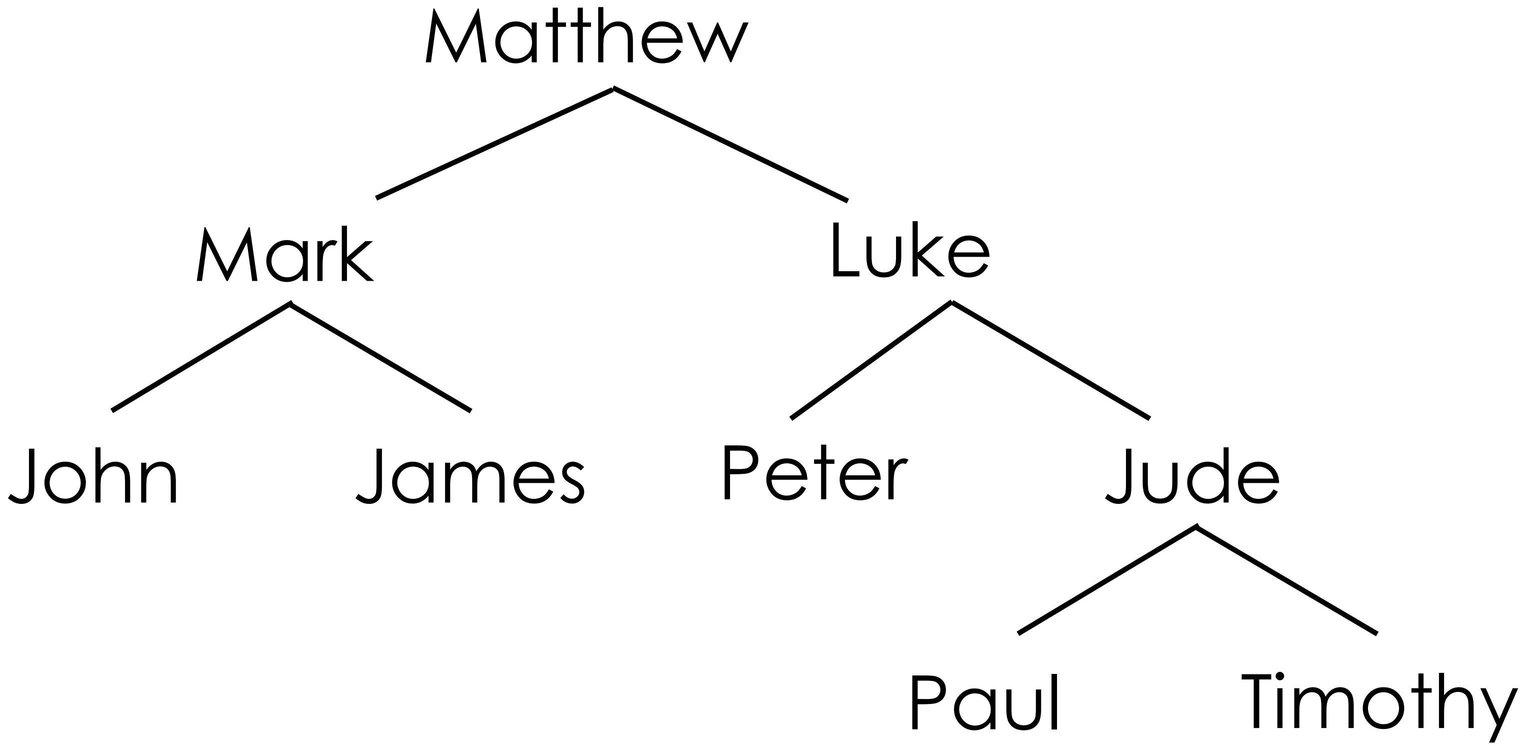

```
while ( not s.Empty() )
        node = s.Pop()
        Print( node.key )
        if node.right ≠ null
            s.Push( node.right )
        if node.left ≠ null
            s.Push( node.left )
```

Output: Matthew, Mark, John, James, Luke, Peter, Jude, Paul, Timothy

Stack:

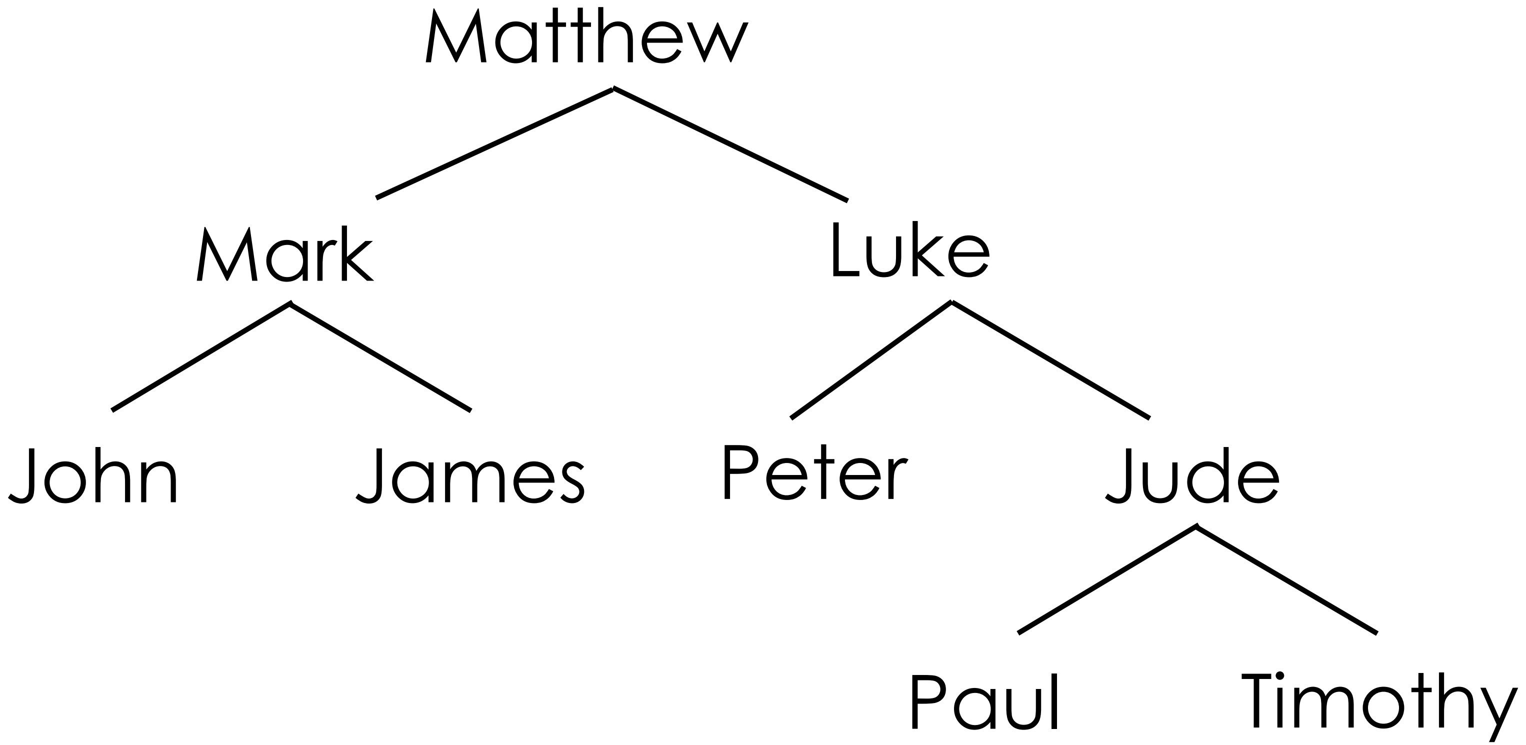
node

Depth-first Traversal (PreOrder) Implementation using **Recursion**



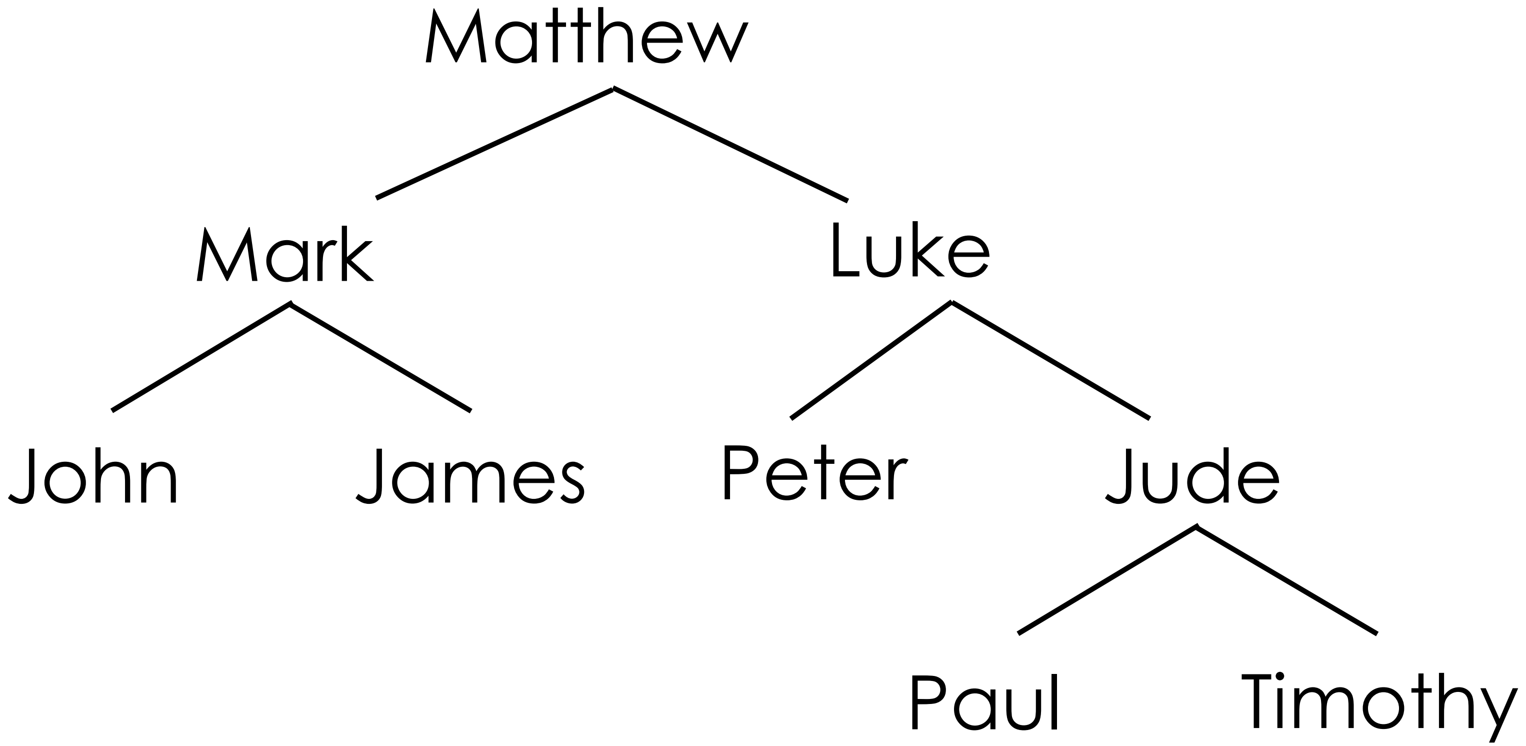
Output:

Depth-first Traversal (PreOrder)



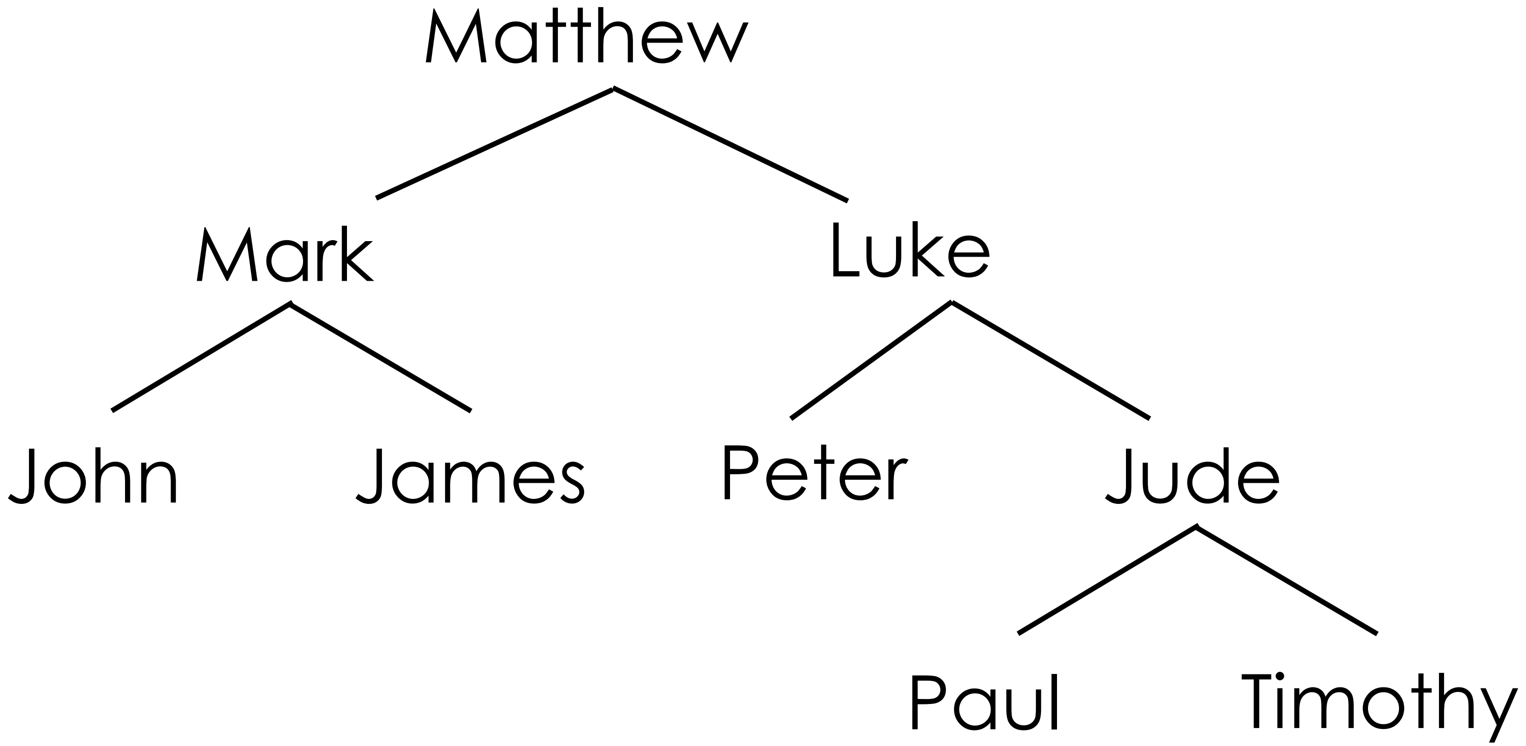
Output: Matthew, Mark, John, James, Luke, Peter, Jude, Paul, Timothy

Depth-first Traversal (InOrder)



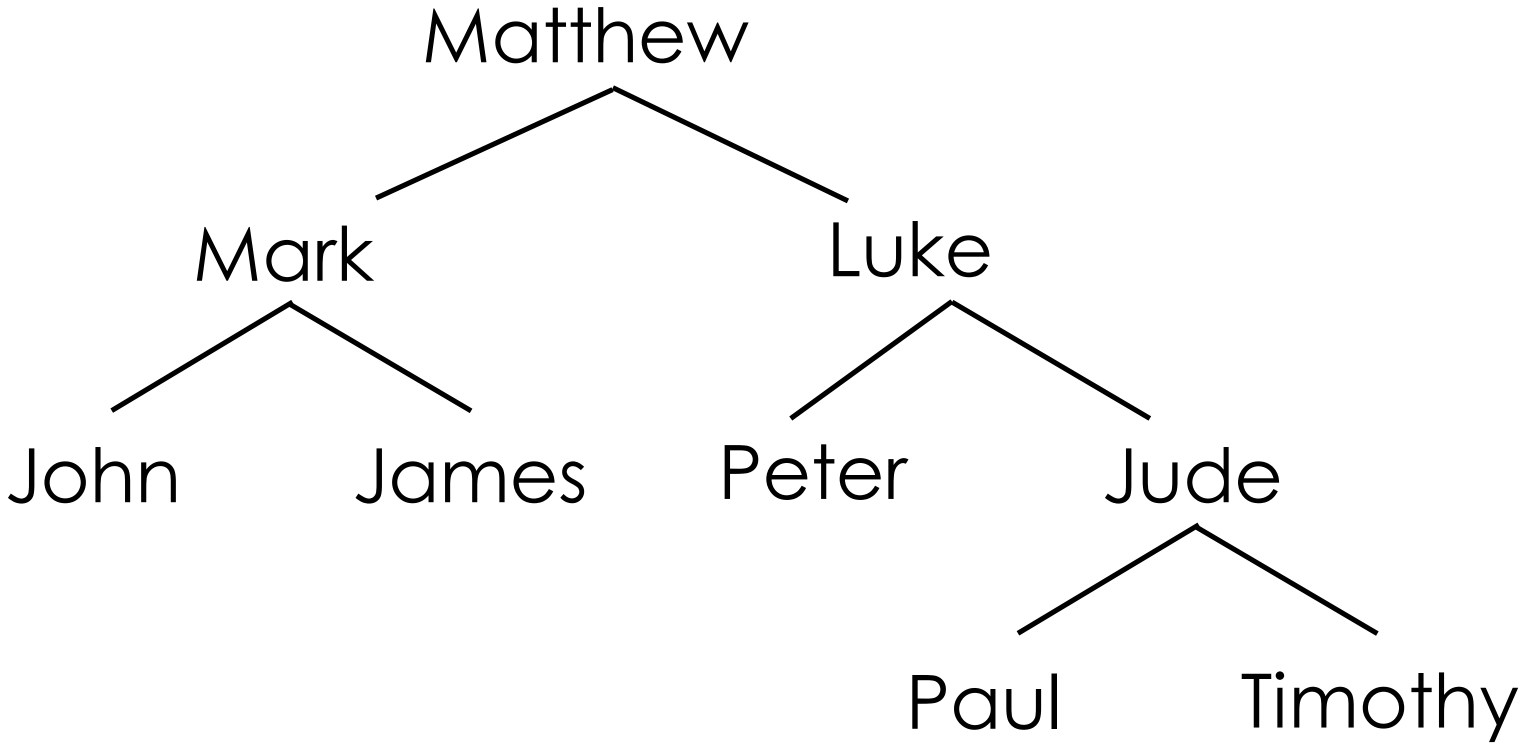
Output:

Depth-first Traversal (InOrder)



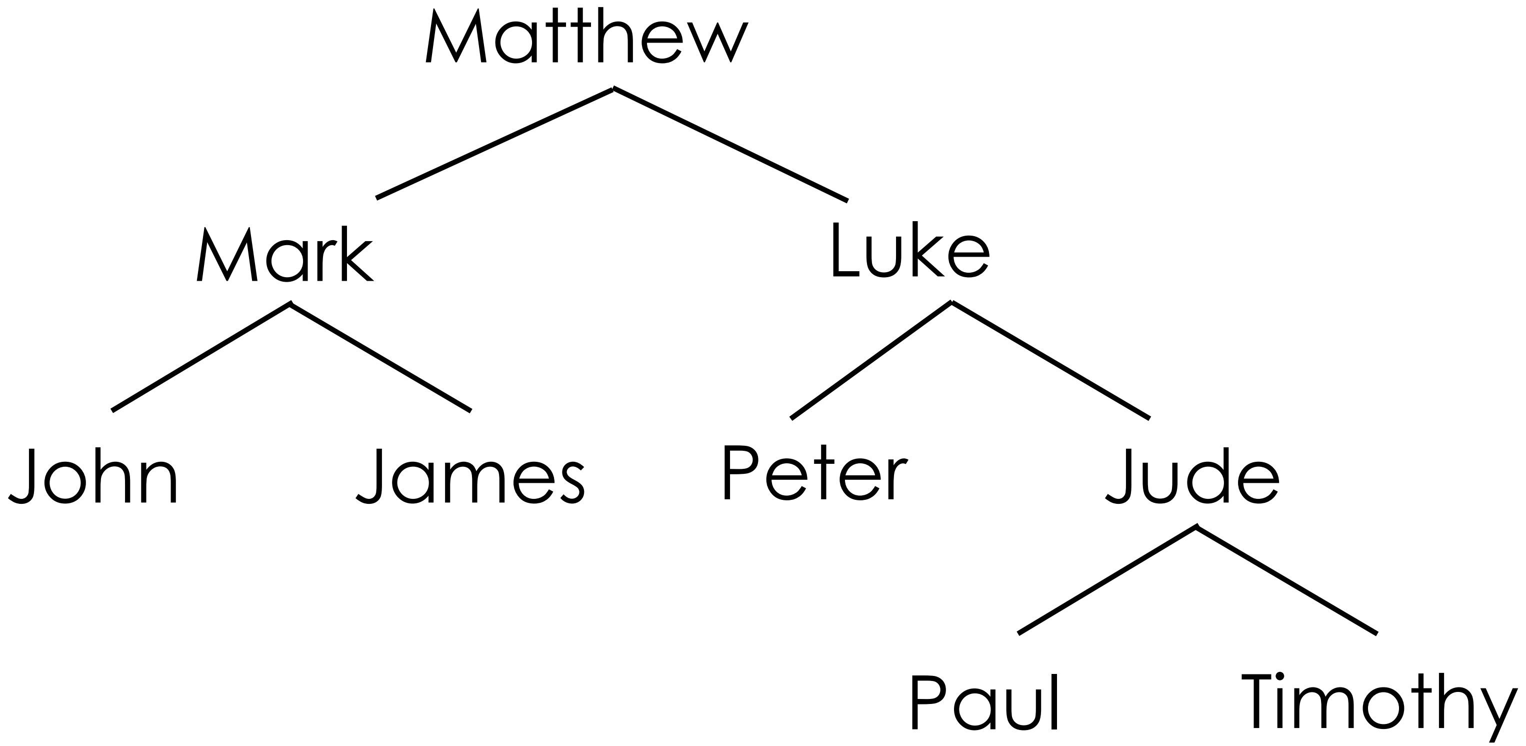
Output: John, Mark, James, Matthew, Peter, Luke, Paul, Jude, Timothy

Depth-first Traversal (PostOrder)



Output:

Depth-first Traversal (PostOrder)



Output: John, James, Mark, Peter, Paul, Timothy, Jude, Luke, Matthew

Depth-first Traversal (InOrder)

InOrderTraversal (Node node)

```
if node == null
    return
else
    InOrderTraversal (node.left)
    Print (node.key)
    InOrderTraversal (node.right)
```

Depth-first Traversal (PreOrder)

```
PreOrderTraversal ( Node node )
if node == null
    return
else
    Print (node.key)
    PreOrderTraversal (node.left)
    PreOrderTraversal (node.right)
```

Depth-first Traversal (PostOrder)

PostOrderTraversal (Node node)

```
if node == null
    return
else
    PostOrderTraversal (node.left)
    PostOrderTraversal (node.right)
    Print (node.key)
```

Unordered Tree Summary

- Tree is Data Structure with a node that can point to multiple nodes
- Searching a key in unordered tree takes O(n)
- Degenerate trees look like Linked list which takes Θ(n) to build
- If the target key tends to be in the earlier nodes, Breadth-first search is the best strategy
- If the target key tends to be in the deeper nodes or leaf nodes, Depth-first search is the best strategy
- Traversal can be: PreOrder, InOrder, or PostOrder
- Traversal implementation can be either recursive or non-recursive
- Non-recursive implementation of BFT is to use Q whileas DFT is to use Stack