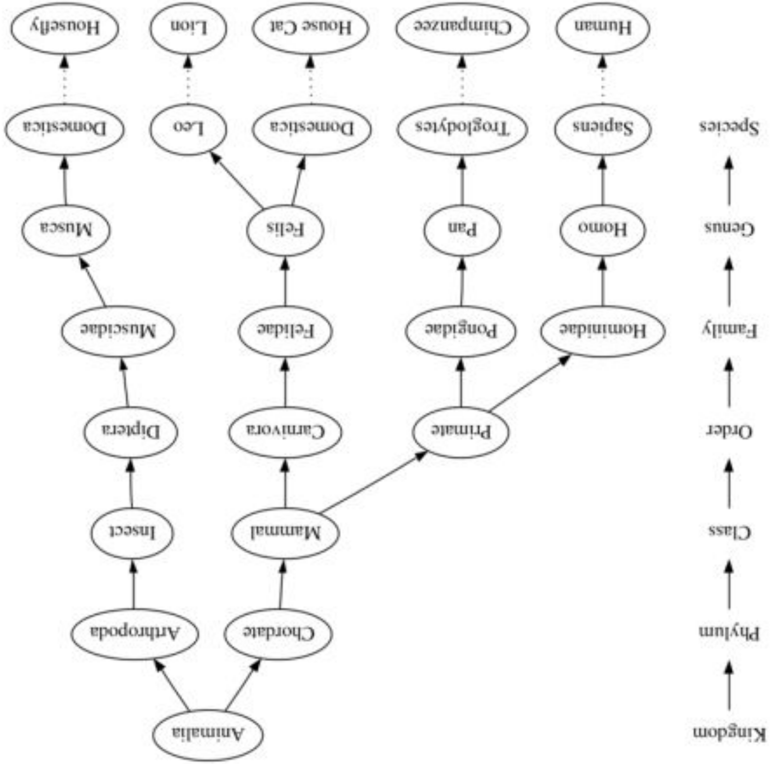
Scientific Programming: Part B

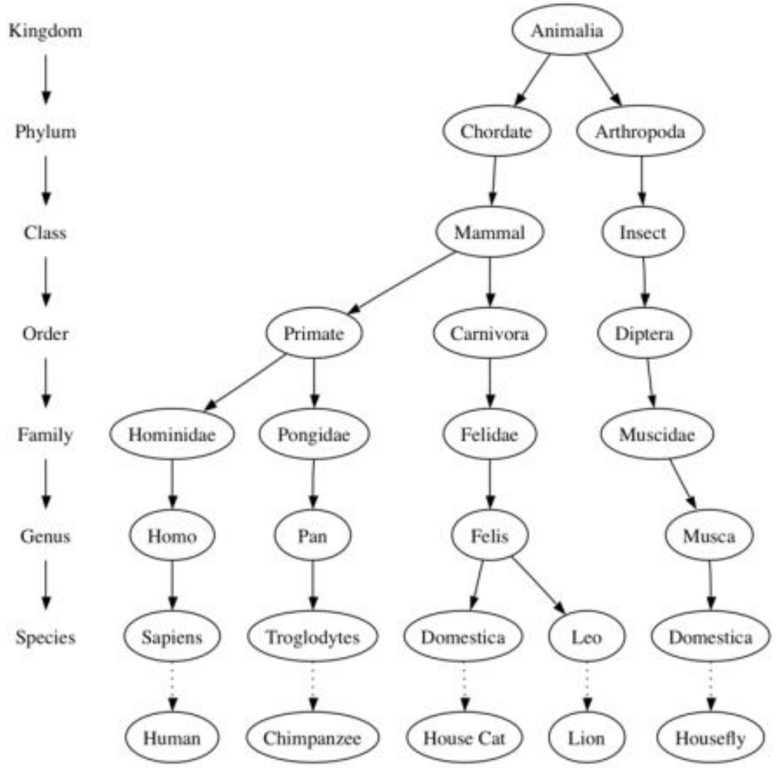
Trees

Luca Bianco - Academic Year 2020-21
luca.bianco@fmach.it
[credits: thanks to Prof. Alberto Montresor]

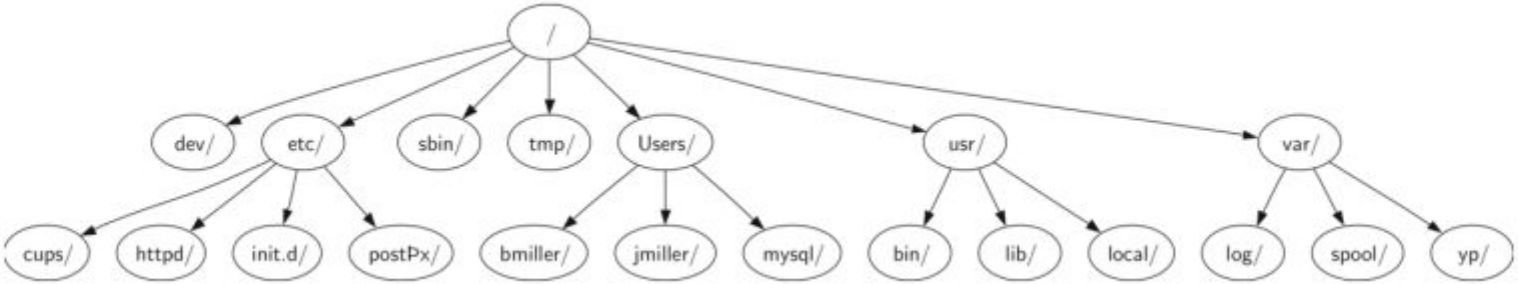
Tree: examples



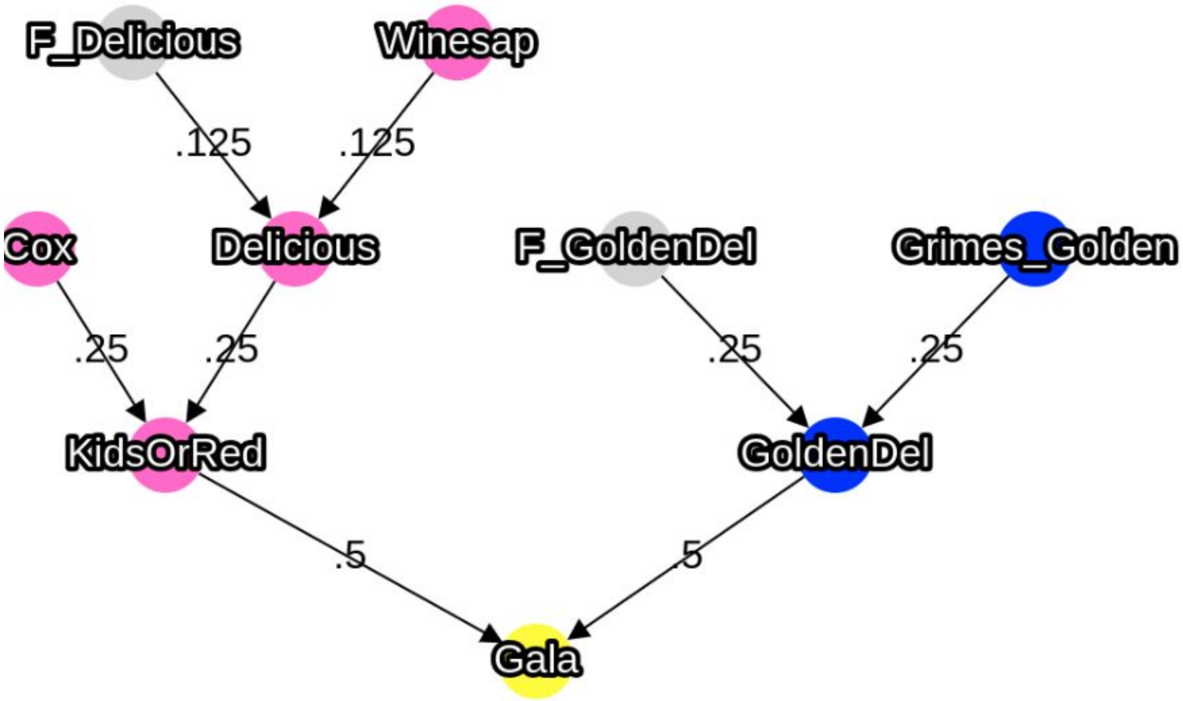
Tree: examples



Tree: examples

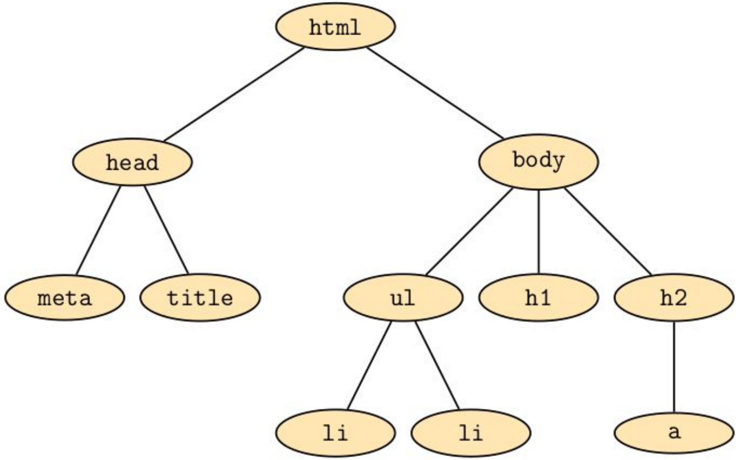


Tree: examples

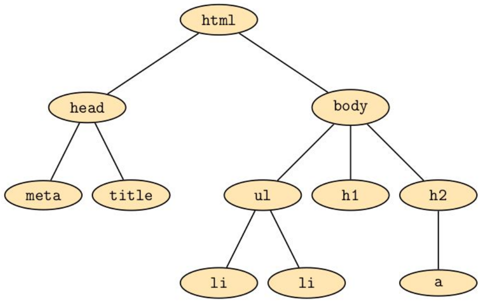


Tree: examples

```
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html" />
    <title>simple</title>
</head>
<body>
<h1>A simple web page</h1>
<ul>
    <li>List item one</li>
    <li>List item two</li>
</ul>
<h2><a href="http://www.cs.luther.edu">Luther CS </a><h2>
</body>
</html>
```



Definitions



Trees are data structures composed of two elements: ***nodes*** and ***edges***.

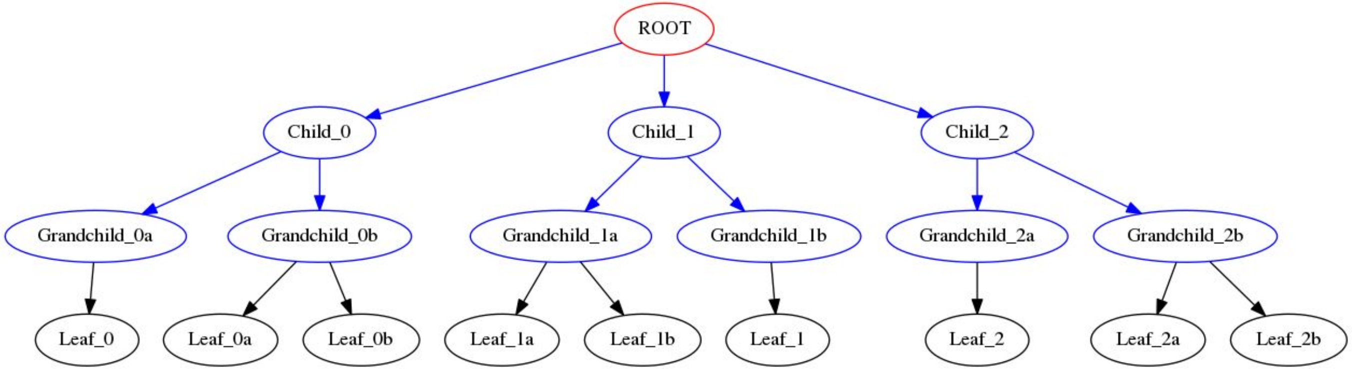
Nodes represent ***things*** and edges represent ***relationships*** (typically non-symmetric) among **two** nodes.

Tree

A tree consists of a set of nodes and a set of edges that connect pairs of nodes, with the following properties:

- One node of the tree is designated as the root node
- Every node n, except the root node, is connected by an edge from exactly one other node p
- A unique path traverses from the root to each node
- The tree is connected

Definitions



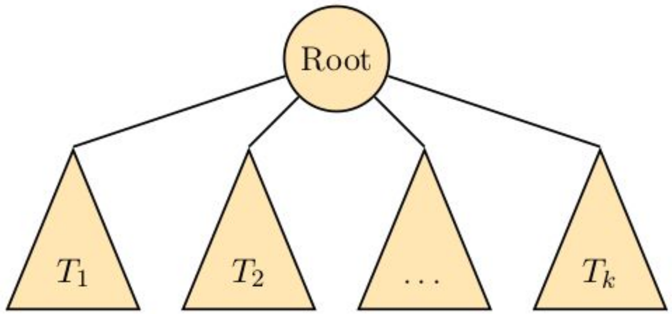
Facts

- One node called the **root** is the top level of the tree and is connected to one or more other nodes;
- If the root is connected to another node by means of one edge, then it is said to be the **parent** of the node (and that node is the **child** of the root);
- Any node can be **parent** of one or more other nodes, the only important thing is that **all nodes have only one parent**;
- The **root is the only exception as it does not have any parent**. Some nodes do not have children and they are called **leaves**;

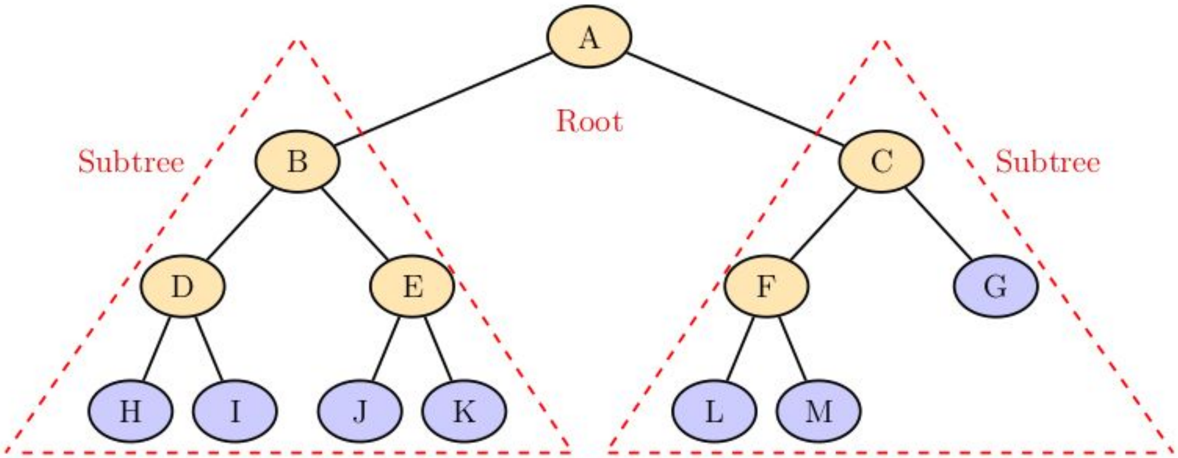
Recursive definition

Tree

A tree is either empty or consists of a root and zero or more subtrees, each of which is also a tree. The root of each subtree is connected to the root of the parent tree by an edge.



Terminology



- A is the tree root
- B, C are roots of their subtrees
- D, E are siblings
- D, E are children of B
- B is the parent of D, E
- Purple nodes are leaves
- The other nodes are internal nodes

Terminology - 2

Depth of a node

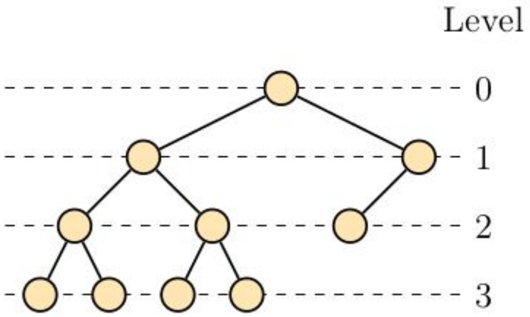
The length of the simple path from the root to the node (measured in number of edges)

Level

The set of nodes having the same depth

Height of the tree

The maximum depth of all its leaves



Height of this tree = 3

Binary tree

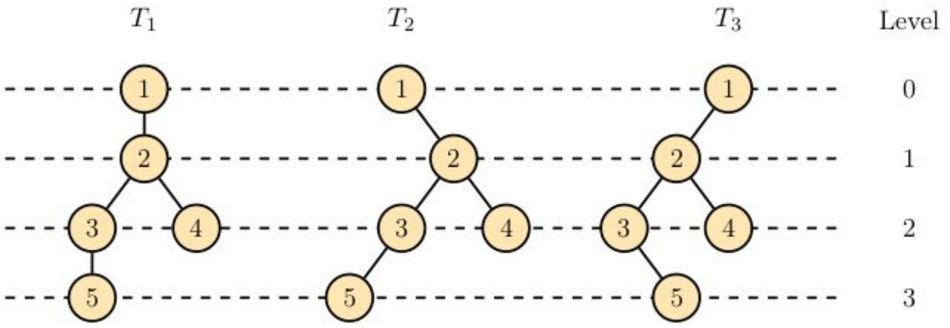
Binary tree

A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child.

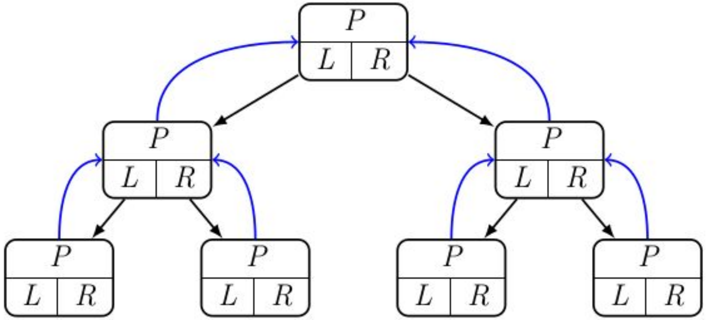
Note: *Two trees T and U having the same nodes, the same children for each node and the same root, are said to be different if a node u is a left child of a node v in T and a right child of the same node in U.*

Three distinct trees.

Note: T1 is not graphically very well represented.



Binary tree: Node



- *parent*: reference to the parent node
- *left*: reference to the left child
- *right*: reference to the left child

When implementing a tree we can define a **node object** and then a **tree object** that stores nodes.

We will use the more compact way which is to **use the recursive definition of a tree**.

Binary tree: ADT

TREE

% Build a new node, initially containing v, with no children or parent
Tree(OBJECT v)

% Read the value stored in this node
OBJECT getValue()

% Write the value stored in this node
setValue(OBJECT v)

% Return the parent, or **none** if this node is the root
TREE getParent()

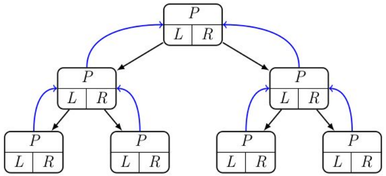
% Return the left (right) child of this node; return **none** if absent
TREE getLeft()
TREE getRight()

% Insert the subtree rooted in t as left (right) child of this node
insertLeft(TREE t)
insertRight(TREE t)

% Delete the subtree rooted on the left (right) child of this node
deleteLeft()
deleteRight()

Binary tree: the code

```python
class BinaryTree:
    #the initializer, set the data
    #all pointers empty
    def __init__(self, value):
        self.__data = value
        self.__right = None
        self.__left = None
        self.__parent = None

    #returns the value
    def getValue(self):
        return self.__data

    #sets the value
    def setValue(self, newval):
        self.__data = newval

    #gets the parent
    def getParent(self):
        return self.__parent

    #sets the parent
    #NOTE: needed because we are using
    #private attributes
    def setParent(self, tree):
        self.__parent = tree

    #gets the right child
    def getRight(self):
        return self.__right

    #gets the left child
    def getLeft(self):
        return self.__left

    #set the right child
    def insertRight(self, tree):
        if self.__right == None:
            self.__right = tree
            tree.setParent(self)

    #sets the left child
    def insertLeft(self, tree):
        if self.__left == None:
            self.__left = tree
            tree.setParent(self)
    #deletes the right subtree
    def deleteRight(self):
        self.__right = None
    #deletes the left subtree
    def deleteLeft(self):
        self.__left = None
```



TREE

% Build a new node, initially containing v, with no children or parent
Tree(OBJECT v)

% Read the value stored in this node
OBJECT getValue()

% Write the value stored in this node
setValue(OBJECT v)

% Return the parent, or **none** if this node is the root
TREE getParent()

% Return the left (right) child of this node; return **none** if absent
TREE getLeft()
TREE getRight()

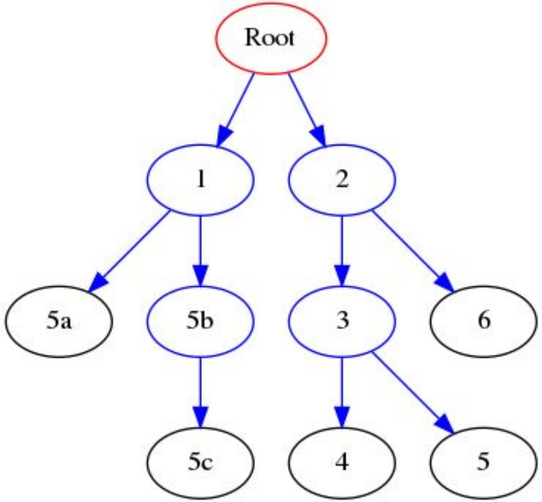
% Insert the subtree rooted in t as left (right) child of this node
insertLeft(TREE t)
insertRight(TREE t)

% Delete the subtree rooted on the left (right) child of this node
deleteLeft()
deleteRight()

Exercise: recursive deleteRight and deleteLeft or just delete

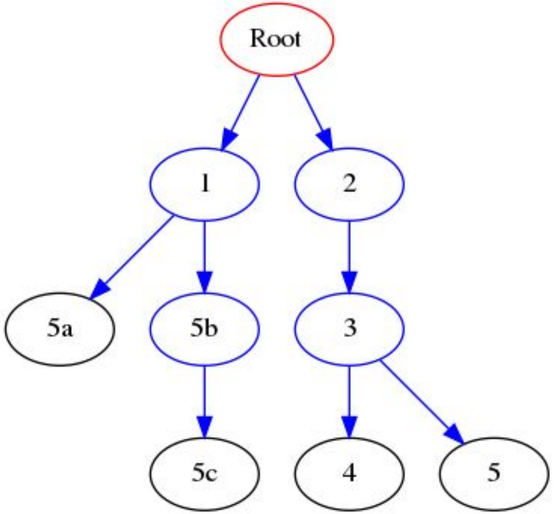
A sample tree...

```python
if __name__ == "__main__":
    BT = BinaryTree("Root")
    bt1 = BinaryTree(1)
    bt2 = BinaryTree(2)
    bt3 = BinaryTree(3)
    bt4 = BinaryTree(4)
    bt5 = BinaryTree(5)
    bt6 = BinaryTree(6)
    bt5a = BinaryTree("5a")
    bt5b = BinaryTree("5b")
    bt5c = BinaryTree("5c")

    BT.insertLeft(bt1)
    BT.insertRight(bt2)
    bt2.insertLeft(bt3)
    bt3.insertLeft(bt4)
    bt3.insertRight(bt5)
    bt2.insertRight(bt6)
    bt1.insertRight(bt5b)
    bt1.insertLeft(bt5a)
    bt5b.insertRight(bt5c)
```



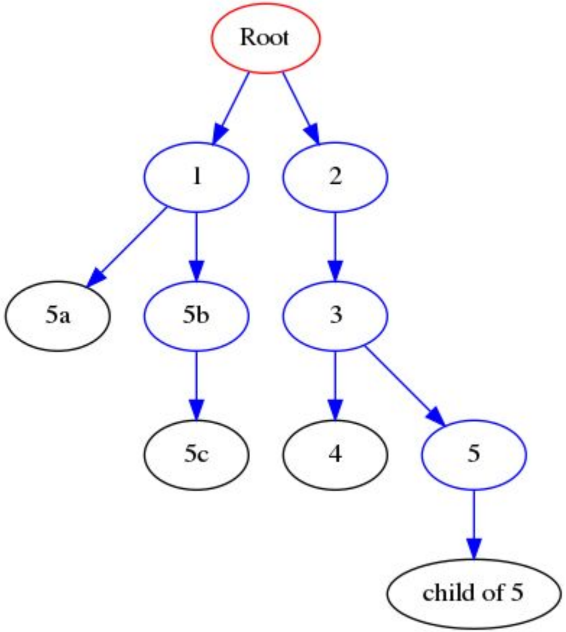
A sample tree...

```python
if __name__ == "__main__":
    BT = BinaryTree("Root")
    bt1 = BinaryTree(1)
    bt2 = BinaryTree(2)
    bt3 = BinaryTree(3)
    bt4 = BinaryTree(4)
    bt5 = BinaryTree(5)
    bt6 = BinaryTree(6)
    bt5a = BinaryTree("5a")
    bt5b = BinaryTree("5b")
    bt5c = BinaryTree("5c")

    BT.insertLeft(bt1)
    BT.insertRight(bt2)
    bt2.insertLeft(bt3)
    bt3.insertLeft(bt4)
    bt3.insertRight(bt5)
    bt2.insertRight(bt6)
    bt1.insertRight(bt5b)
    bt1.insertLeft(bt5a)
    bt5b.insertRight(bt5c)
    print("\nDelete right branch of 2")
    bt2.deleteRight()
```

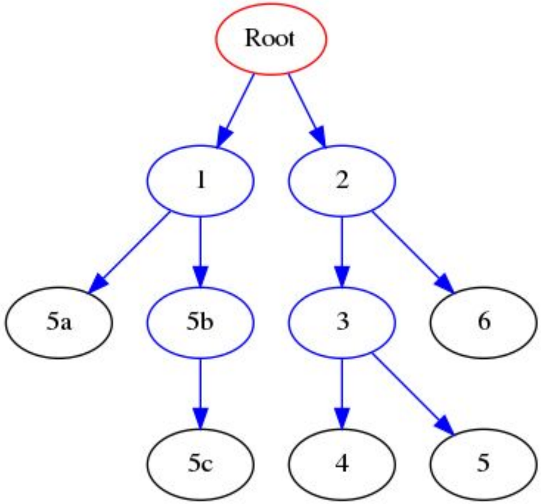


A sample tree...

```python
if __name__ == "__main__":
    BT = BinaryTree("Root")
    bt1 = BinaryTree(1)
    bt2 = BinaryTree(2)
    bt3 = BinaryTree(3)
    bt4 = BinaryTree(4)
    bt5 = BinaryTree(5)
    bt6 = BinaryTree(6)
    bt5a = BinaryTree("5a")
    bt5b = BinaryTree("5b")
    bt5c = BinaryTree("5c")

    BT.insertLeft(bt1)
    BT.insertRight(bt2)
    bt2.insertLeft(bt3)
    bt3.insertLeft(bt4)
    bt3.insertRight(bt5)
    bt2.insertRight(bt6)
    bt1.insertRight(bt5b)
    bt1.insertLeft(bt5a)
    bt5b.insertRight(bt5c)
    print("\nDelete right branch of 2")
    bt2.deleteRight()
```



A sample tree...



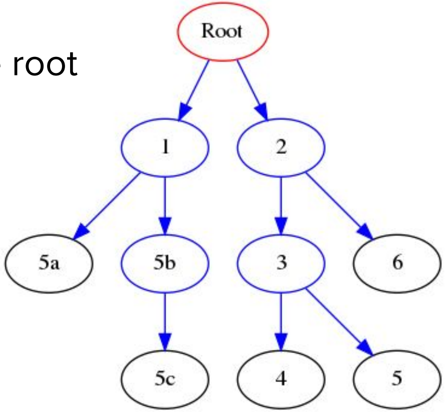

```
if __name__ == "__main__":
    BT = BinaryTree("Root")
    bt1 = BinaryTree(1)
    bt2 = BinaryTree(2)
    bt3 = BinaryTree(3)
    bt4 = BinaryTree(4)
    bt5 = BinaryTree(5)
    bt6 = BinaryTree(6)
    bt5a = BinaryTree("5a")
    bt5b = BinaryTree("5b")
    bt5c = BinaryTree("5c")

    BT.insertLeft(bt1)
    BT.insertRight(bt2)
    bt2.insertLeft(bt3)
    bt3.insertLeft(bt4)
    bt3.insertRight(bt5)
    bt2.insertRight(bt6)
    bt1.insertRight(bt5b)
    bt1.insertLeft(bt5a)
    bt5b.insertRight(bt5c)
```

Exercise. write a print function that gets the root node and prints the tree:

```
Root (r)-> 2
Root (l)-> 1
        1 (r)-> 5b
        1 (l)-> 5a
                5b (r)-> 5c
        2 (r)-> 6
        2 (l)-> 3
                3 (r)-> 5
                3 (l)-> 4
```

A sample tree...

Exercise. write a print function that gets the root node and prints the tree:

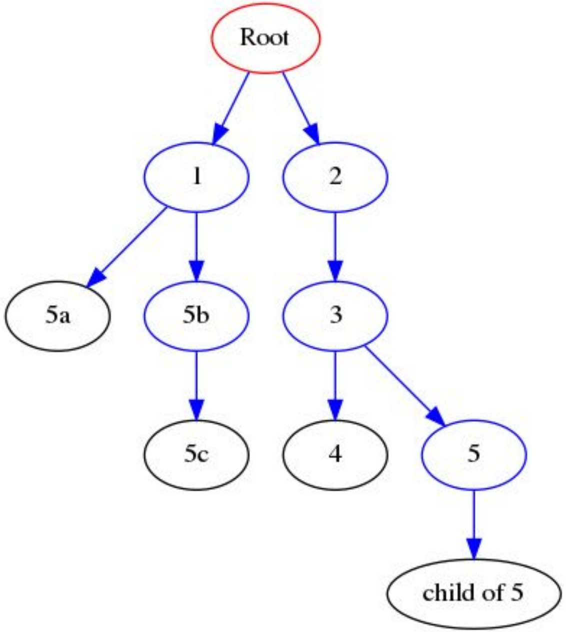


Tabs depend on depth

```
Root (r)-> 2
Root (l)-> 1
	1 (r)-> 5b
	1 (l)-> 5a
		5b (r)-> 5c
	2 (r)-> 6
	2 (l)-> 3
		3 (r)-> 5
		3 (l)-> 4
```

```python
def printTree(root):
    cur = root
    #each element is a node and a depth
    #depth is used to format prints (with tabs)
    nodes = [(cur,0)]
    tabs = ""
    lev = 0
    while len(nodes) >0:
        cur, lev = nodes.pop(-1)
        if cur.getRight() != None:
            print ("{}{} (r)-> {}".format("\t"*lev,
                                          cur.getValue(),
                                          cur.getRight().getValue()))
            nodes.append((cur.getRight(), lev+1))
        if cur.getLeft() != None:
            print ("{}{} (l)-> {}".format("\t"*lev,
                                          cur.getValue(),
                                          cur.getLeft().getValue()))
            nodes.append((cur.getLeft(), lev+1))
```

A sample tree...




```python
def printTree(root):
    cur = root
    #each element is a node and a depth
    #depth is used to format prints (with tabs)
    nodes = [(cur,0)]
    tabs = ""
    lev = 0
    while len(nodes) >0:
        cur, lev = nodes.pop(-1)
        if cur.getRight() != None:
            print ("{}{} (r)-> {}".format("\t"*lev,
                                          cur.getValue(),
                                          cur.getRight().getValue()))
            nodes.append((cur.getRight(), lev+1))
        if cur.getLeft() != None:
            print ("{}{} (l)-> {}".format("\t"*lev,
                                          cur.getValue(),
                                          cur.getLeft().getValue()))
            nodes.append((cur.getLeft(), lev+1))
```

OUTPUT

```
Root (r)-> 2
Root (l)-> 1
	1 (r)-> 5b
	1 (l)-> 5a
		5b (r)-> 5c
	2 (l)-> 3
		3 (r)-> 5
		3 (l)-> 4
			5 (l)-> child of 5
```

Tree traversals

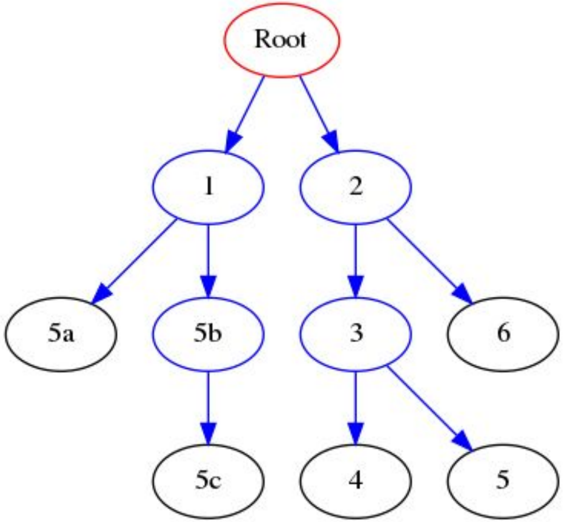
Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

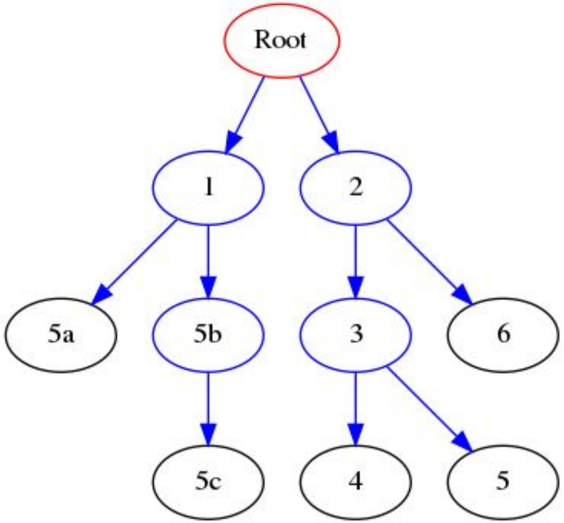
Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack

To store all unfinished calls to DFS(node)



Tree traversals



Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack

Recursively

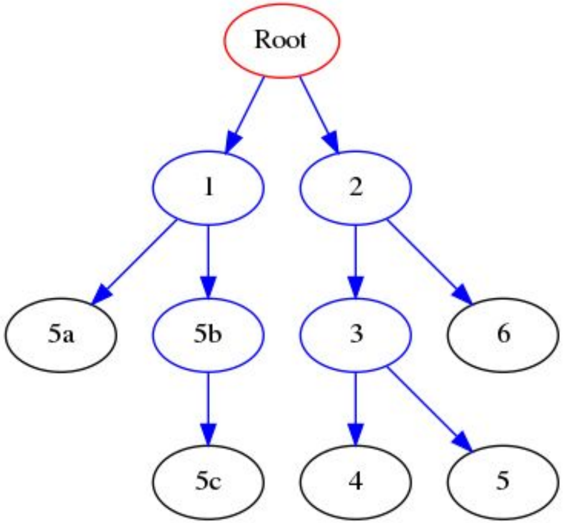
1. visit Root
2. DFS(left)
3. DFS(right)

Preorder:
Root

visit(Root) → print("Root")
call DFS(Root.getLeft())
which is DFS(1) → visit(1)
DFS("1".getLeft())
DFS("1".getRight()
...
call DFS(Root.getRight())
...

To store all unfinished calls to DFS(node)

Tree traversals



Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack

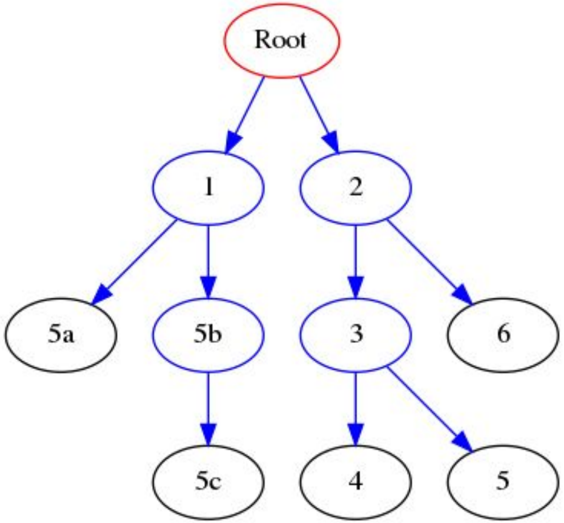
Recursively

1. visit Root
2. DFS(left)
3. DFS(right)

Preorder:
Root
1

Stack: (5c right of 5b!)
1
Root

Tree traversals



Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack

Recursively

1. visit Root
2. DFS(left)
3. DFS(right)

Preorder:
Root
1
5a

Stack: (5c right of 5b!)
5a
1
Root

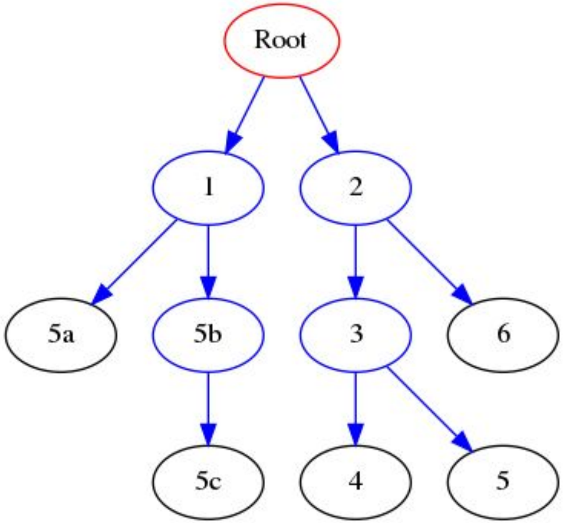
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack



Recursively

1. visit Root
2. DFS(left)
3. DFS(right)

Preorder:
Root
1
5a

Stack: (5c right of 5b!)
1
Root

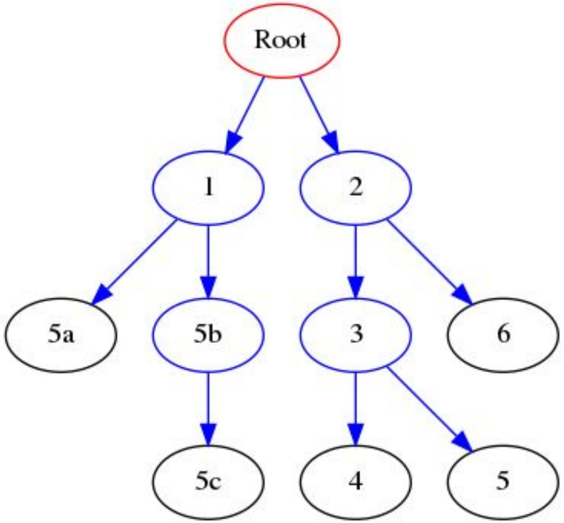
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack



Recursively

1. visit Root
2. DFS(left)
3. DFS(right)

Preorder:
Root
1
5a
5b

Stack: (5c right of 5b!)
5b
1
Root

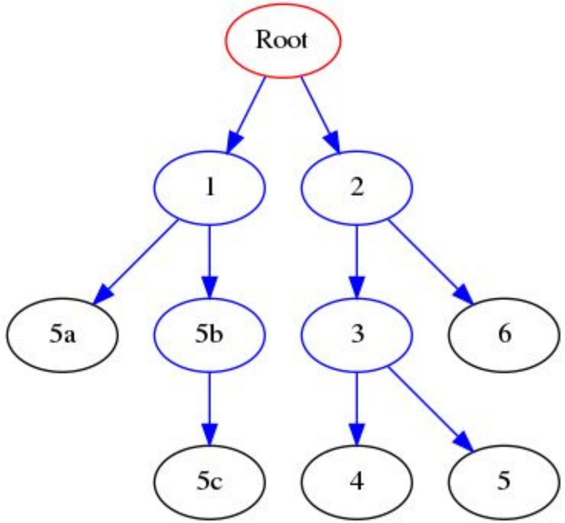
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack



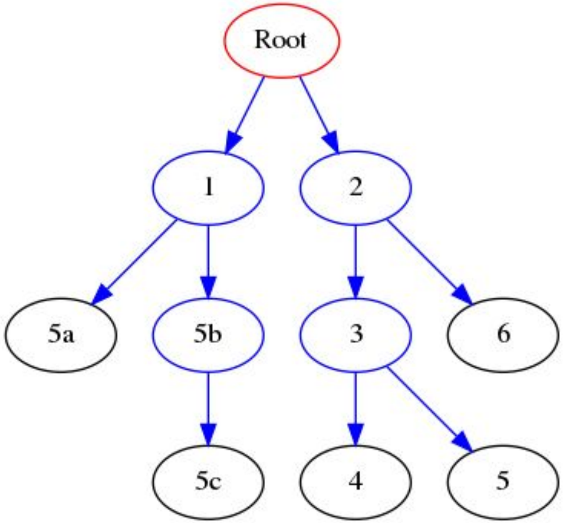
Recursively

1. visit Root
2. DFS(left)
3. DFS(right)

Preorder:
Root
1
5a
5b
5c

Stack: (5c right of 5b!)
5c
5b
1
Root

Tree traversals



Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack

Recursively

1. visit Root
2. DFS(left)
3. DFS(right)

Preorder:
Root
1
5a
5b
5c

Stack: (5c right of 5b!)
5b
1
Root

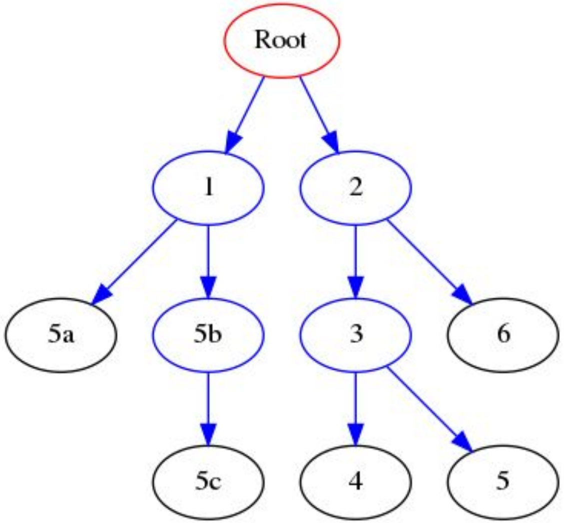
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack



Recursively

1. visit Root
2. DFS(left)
3. DFS(right)

Preorder:
Root
1
5a
5b
5c

Stack: (5c right of 5b!)
1
Root

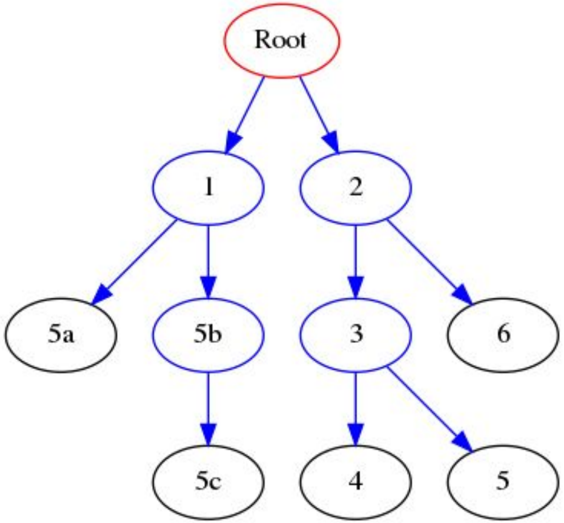
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack



Recursively

1. visit Root
2. DFS(left)
3. DFS(right)

Preorder:
Root
1
5a
5b
5c

Stack: (5c right of 5b!)
Root

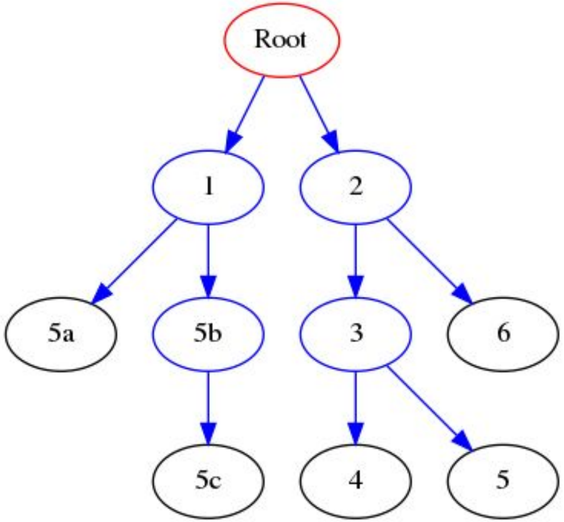
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack



Recursively

1. visit Root
2. DFS(left)
3. DFS(right)

Preorder:
Root
1
5a
5b
5c
2

Stack: (5c right of 5b!)
2
Root

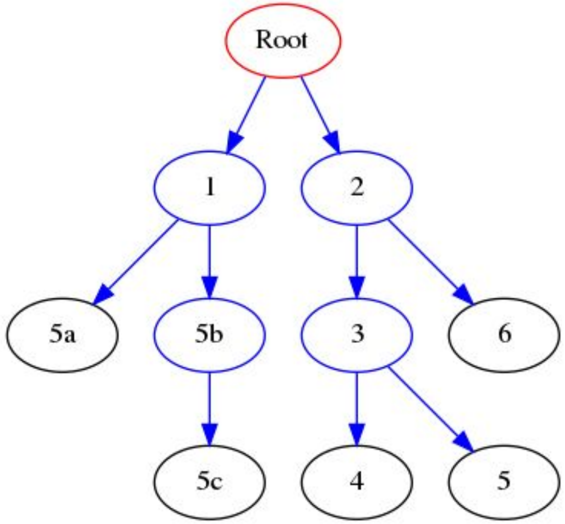
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack



Recursively

1. visit Root
2. DFS(left)
3. DFS(right)

Preorder:
Root
1
5a
5b
5c
2
3

Stack: (5c right of 5b!)
3
2
Root

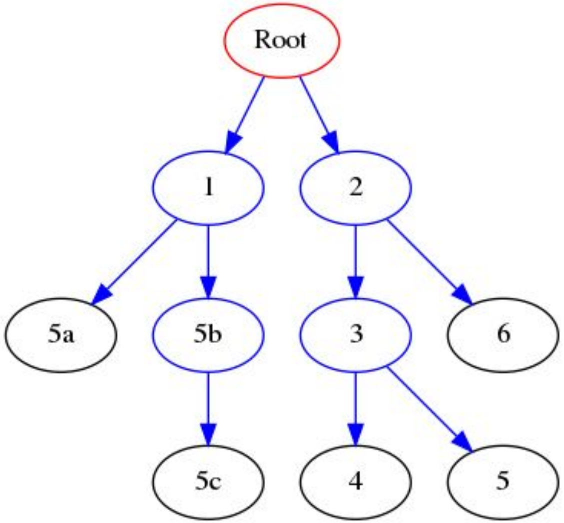
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack



Recursively

1. visit Root
2. DFS(left)
3. DFS(right)

Preorder:
Root
1
5a
5b
5c
2
3
4

Stack: (5c right of 5b!)
4
3
2
Root

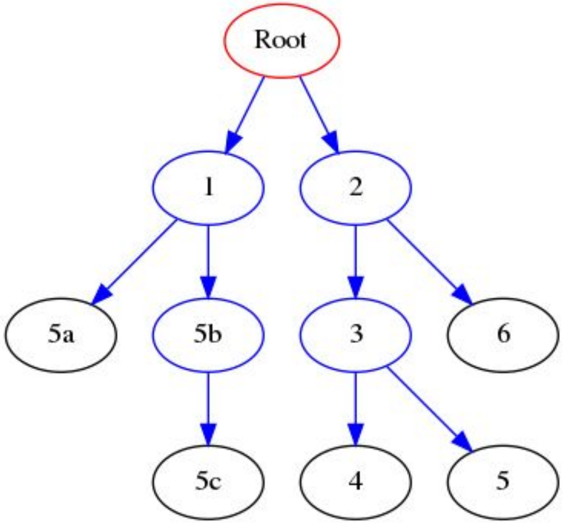
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack



Recursively

1. visit Root
2. DFS(left)
3. DFS(right)

Preorder:
Root
1
5a
5b
5c
2
3
4

Stack: (5c right of 5b!)
3
2
Root

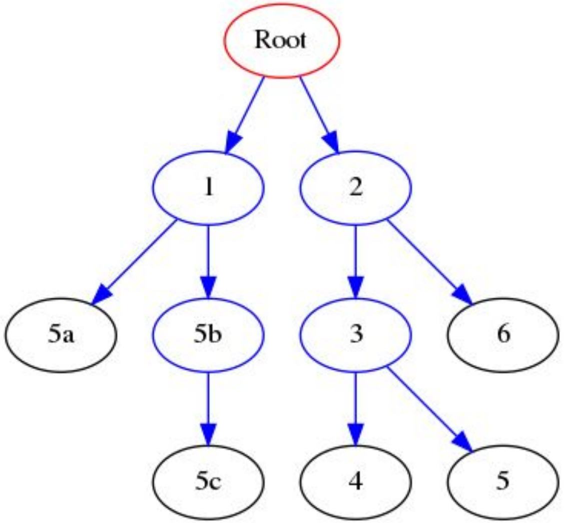
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack



Recursively

1. visit Root
2. DFS(left)
3. DFS(right)

Preorder:
Root
1
5a
5b
5c
2
3
4
5

Stack: (5c right of 5b!)
5
3
2
Root

Tree traversals

Tree traversal / search

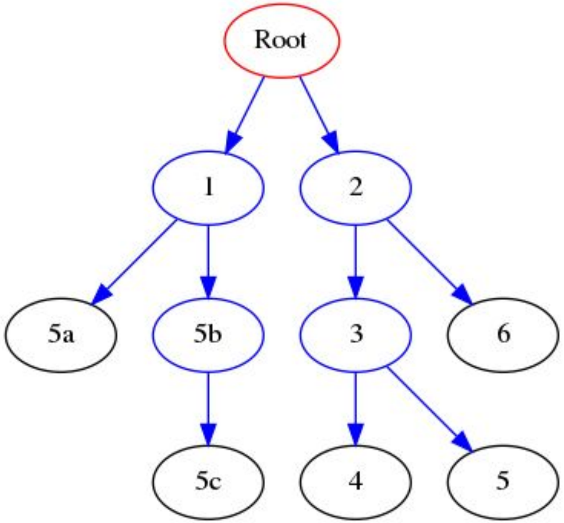
A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack

Recursively

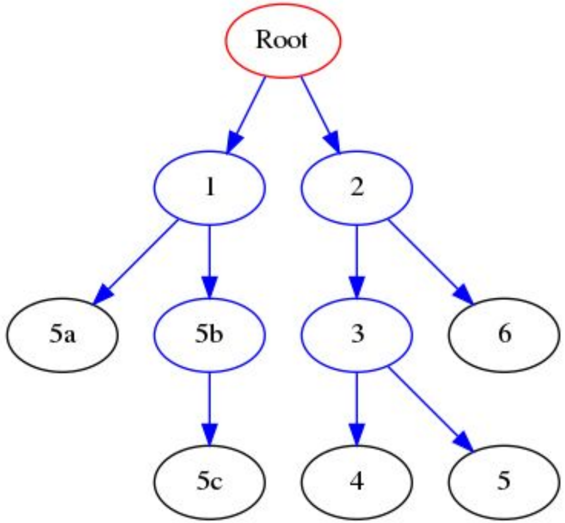
1. visit Root
2. DFS(left)
3. DFS(right)



Preorder:
Root
1
5a
5b
5c
2
3
4
5

Stack: (5c right of 5b!)
3
2
Root

Tree traversals



Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack

Recursively

1. visit Root
2. DFS(left)
3. DFS(right)

Preorder:
Root
1
5a
5b
5c
2
3
4
5

Stack: (5c right of 5b!)
2
Root

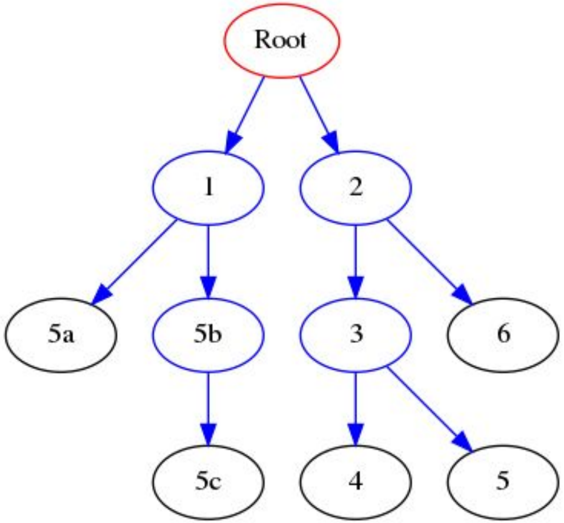
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack



Recursively

1. visit Root
2. DFS(left)
3. DFS(right)

Preorder:
Root
1
5a
5b
5c
2
3
4
5
6

Stack: (5c right of 5b!)
6
2
Root

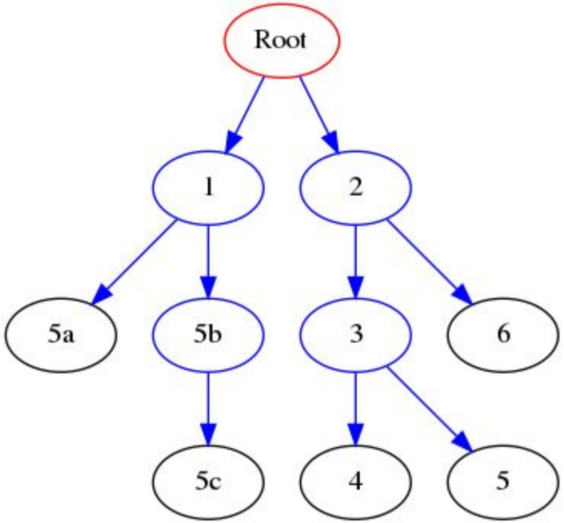
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack



Recursively

1. visit Root
2. DFS(left)
3. DFS(right)

Preorder:
Root
1
5a
5b
5c
2
3
4
5
6

Stack: (5c right of 5b!)
2
Root

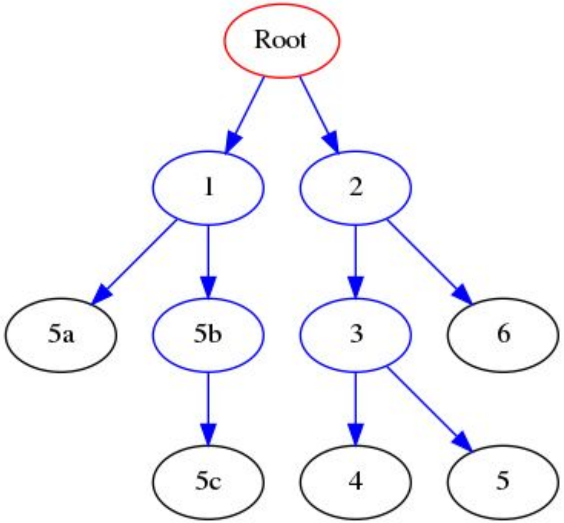
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack



Recursively

1. visit Root
2. DFS(left)
3. DFS(right)

Preorder:
Root
1
5a
5b
5c
2
3
4
5
6

Stack: (5c right of 5b!)
Root

Tree traversals

Tree traversal / search

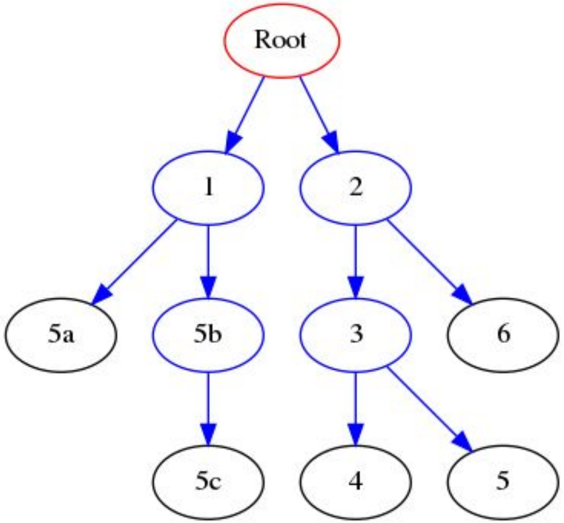
A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack

Recursively

1. visit Root
2. DFS(left)
3. DFS(right)



Preorder:
Root
1
5a
5b
5c
2
3
4
5
6

Stack: (5c right of 5b!)

empty! **Done**

Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

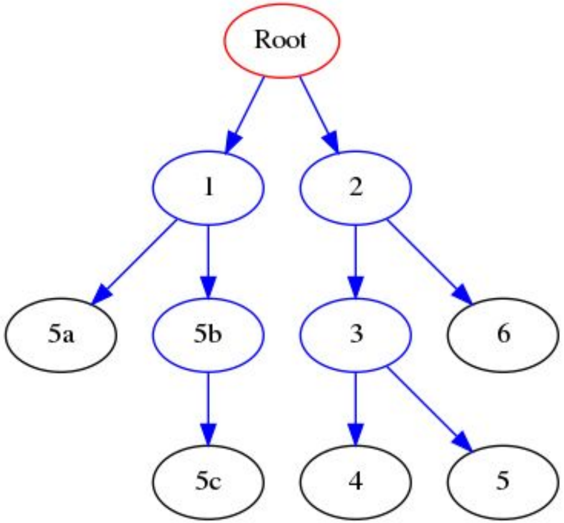
- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack

Recursively

1. DFS(left)
2. visit Root
3. DFS(right)

Inorder:

Stack: (5c right of 5b!)
Root



Tree traversals

Tree traversal / search

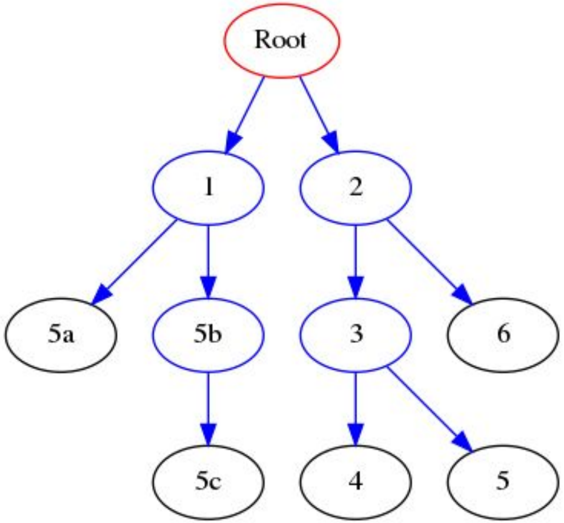
A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack

Recursively

1. DFS(left)
2. visit Root
3. DFS(right)



Inorder:

Stack: (5c right of 5b!)

1

Root

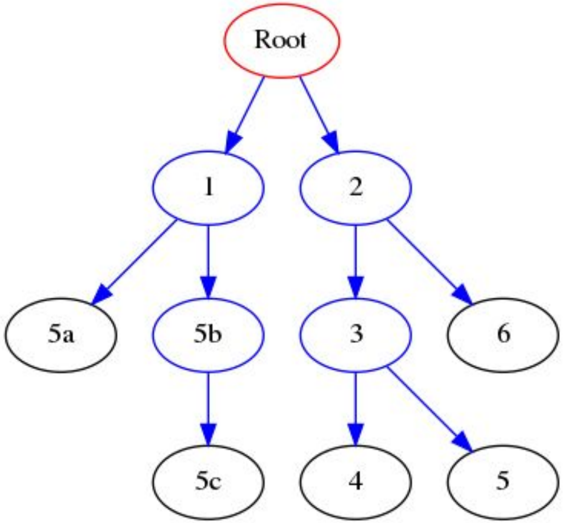
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack



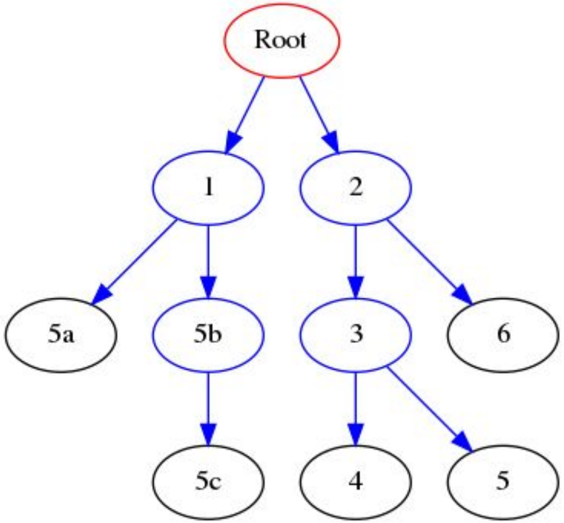
Recursively

1. DFS(left)
2. visit Root
3. DFS(right)

Inorder:
5a

Stack: (5c right of 5b!)
5a
1
Root

Tree traversals



Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack

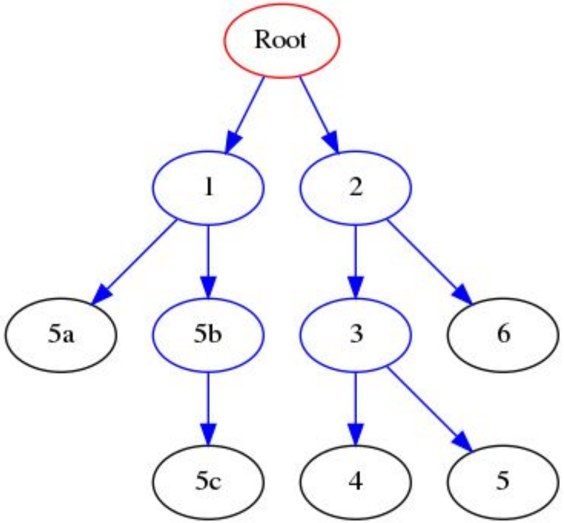
Recursively

1. DFS(left)
2. visit Root
3. DFS(right)

Inorder:
5a

Stack: (5c right of 5b!)
1
Root

Tree traversals



Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack

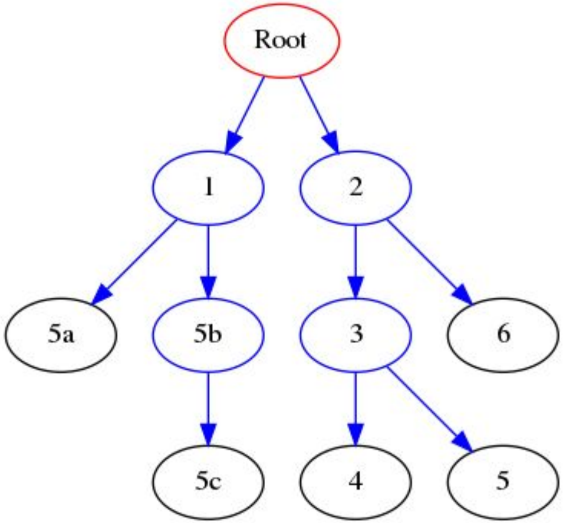
Recursively

1. DFS(left)
2. visit Root
3. DFS(right)

Inorder:
5a
1

Stack: (5c right of 5b!)
1
Root

Tree traversals



Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack

Recursively

1. DFS(left)
2. visit Root
3. DFS(right)

Inorder:
5a
1

Stack: (5c right of 5b!)
5b
1
Root

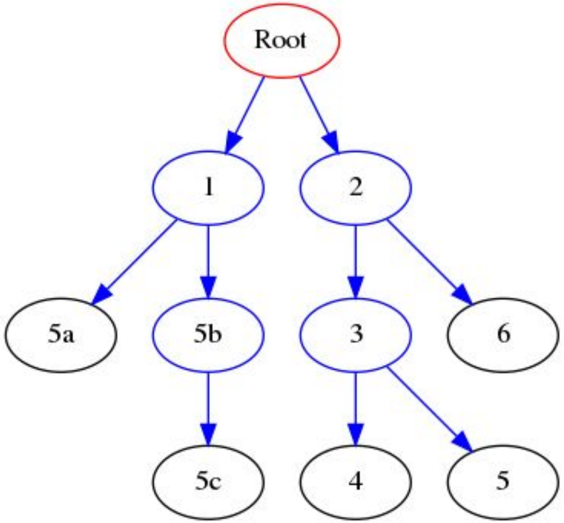
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack



Recursively

1. DFS(left)
2. visit Root
3. DFS(right)

Inorder:
5a
1
5b

Stack: (5c right of 5b!)
5b
1
Root

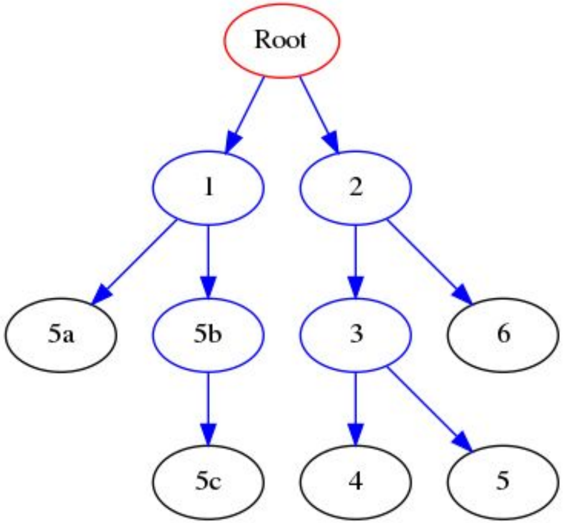
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack



Recursively

1. DFS(left)
2. visit Root
3. DFS(right)

Inorder:
5a
1
5b
5c

Stack: (5c right of 5b!)
5c
5b
1
Root

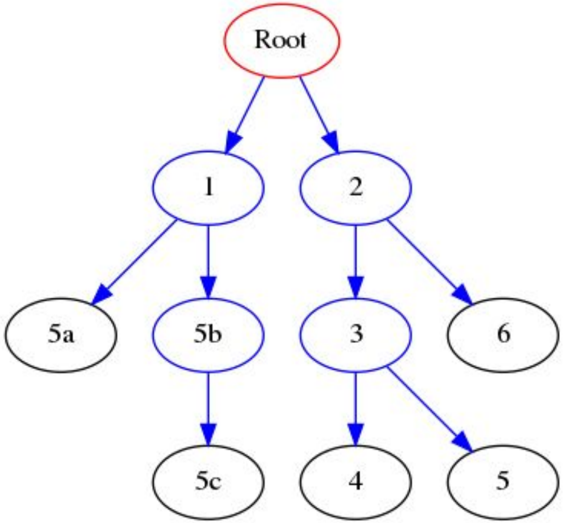
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack



Recursively

1. DFS(left)
2. visit Root
3. DFS(right)

Inorder:
5a
1
5b
5c

Stack: (5c right of 5b!)
5b
1
Root

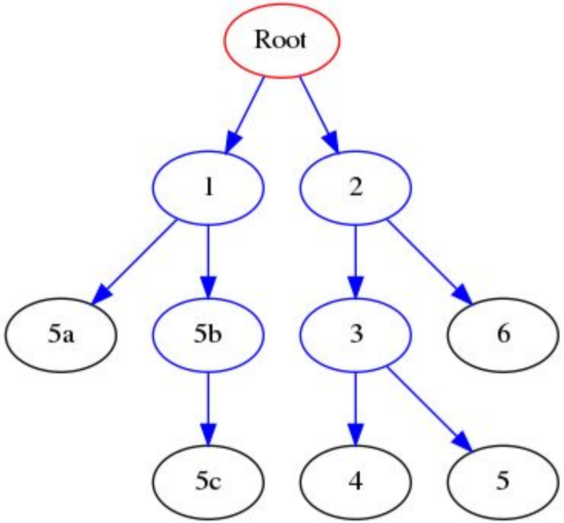
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack



Recursively

1. DFS(left)
2. visit Root
3. DFS(right)

Inorder:
5a
1
5b
5c

Stack: (5c right of 5b!)
1
Root

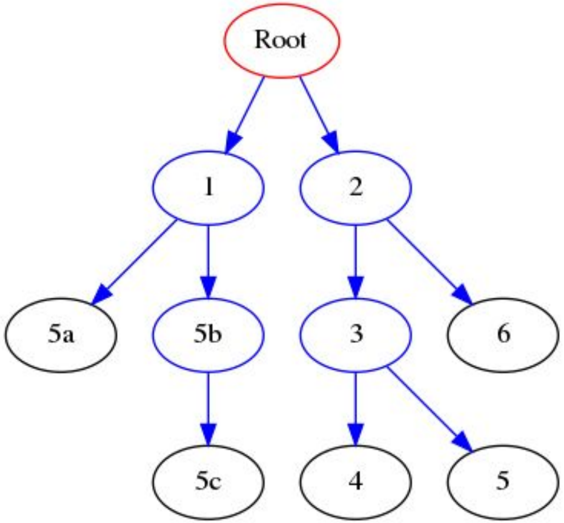
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack



Recursively

1. DFS(left)
2. visit Root
3. DFS(right)

Inorder:
5a
1
5b
5c

Stack: (5c right of 5b!)
Root

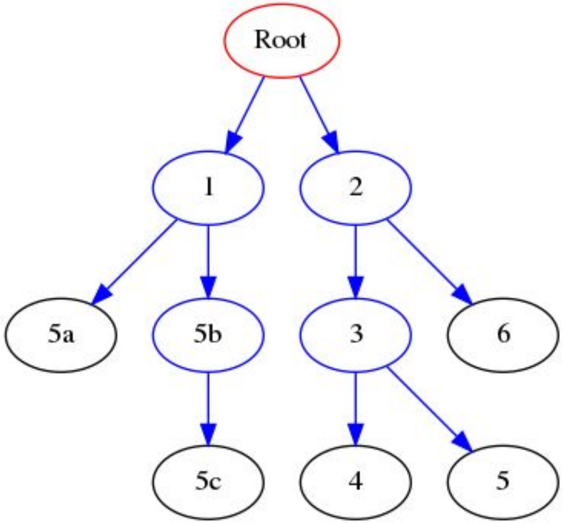
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack



Recursively

1. DFS(left)
2. visit Root
3. DFS(right)

Inorder:
5a
1
5b
5c
Root

Stack: (5c right of 5b!)
Root

Tree traversals

Tree traversal / search

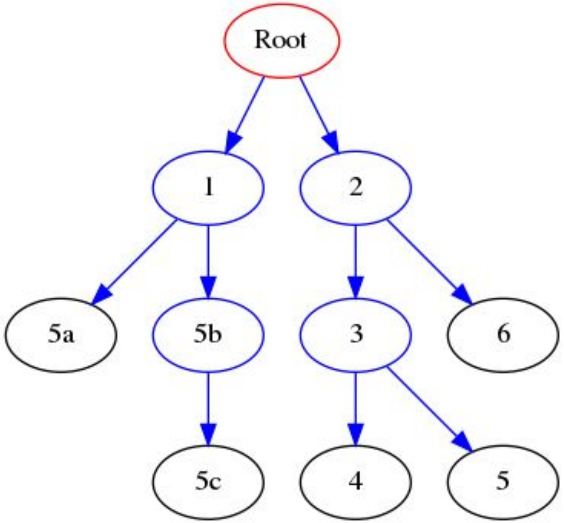
A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack

Recursively

1. DFS(left)
2. visit Root
3. DFS(right)



Inorder:

5a

1

5b

5c

Root

Stack: (5c right of 5b!)

2

Root

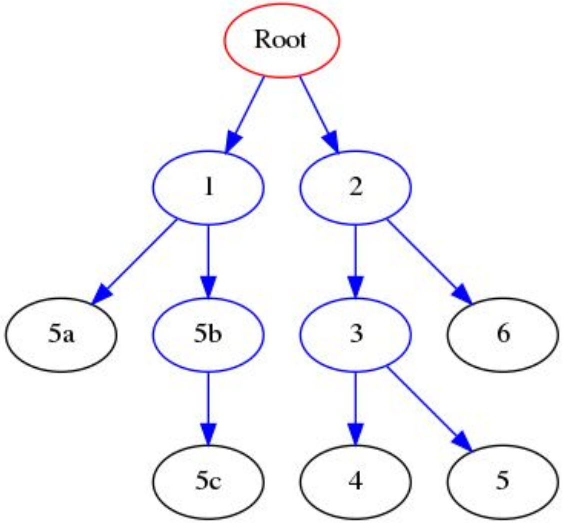
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack



Recursively

1. DFS(left)
2. visit Root
3. DFS(right)

Inorder:
5a
1
5b
5c
Root

Stack: (5c right of 5b!)
3
2
Root

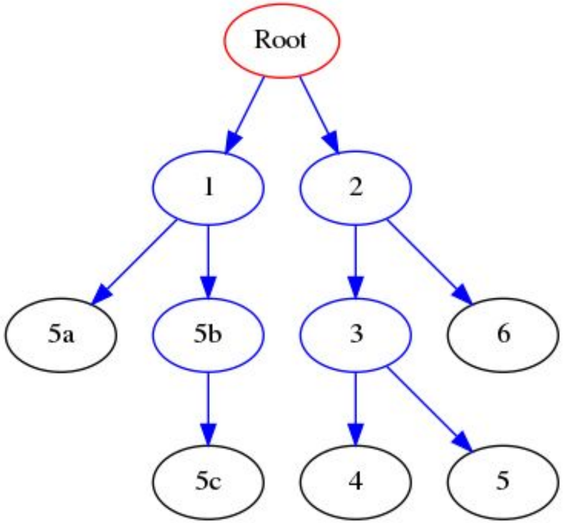
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack



Recursively

1. DFS(left)
2. visit Root
3. DFS(right)

Inorder:
5a
1
5b
5c
Root
4

Stack: (5c right of 5b!)
4
3
2
Root

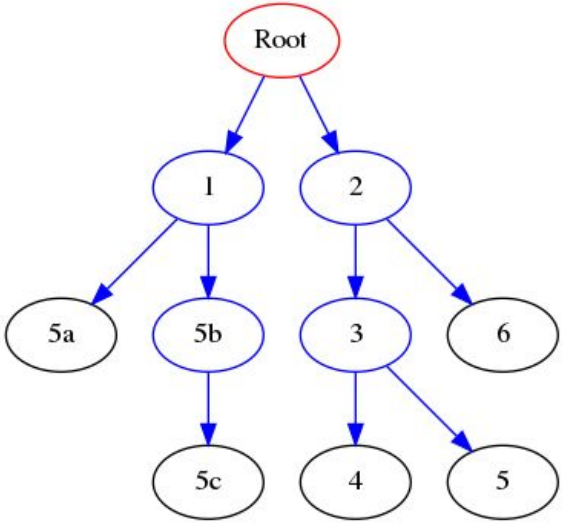
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack



Recursively

1. DFS(left)
2. visit Root
3. DFS(right)

Inorder:
5a
1
5b
5c
Root
4

Stack: (5c right of 5b!)
3
2
Root

Tree traversals

Tree traversal / search

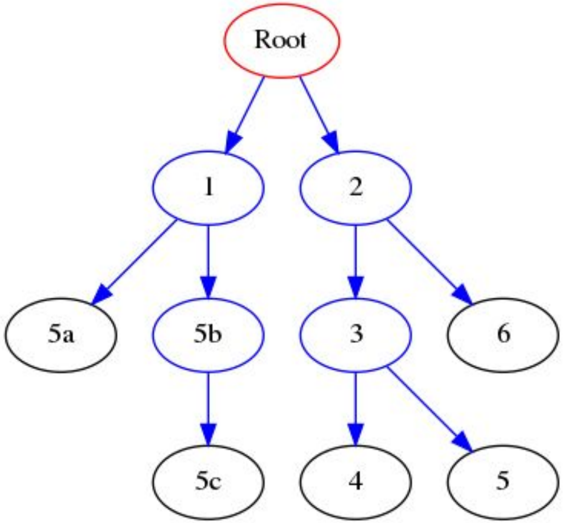
A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack

Recursively

1. DFS(left)
2. visit Root
3. DFS(right)



Inorder:
5a
1
5b
5c
Root
4
3

Stack: (5c right of 5b!)
3
2
Root

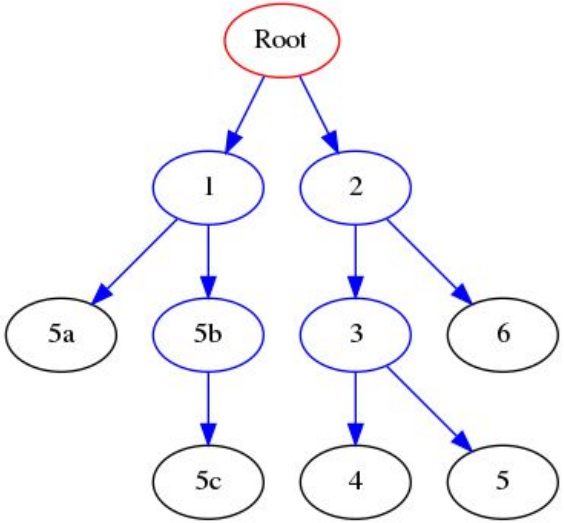
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack



Recursively

1. DFS(left)
2. visit Root
3. DFS(right)

Inorder:
5a
1
5b
5c
Root
4
3
5

Stack: (5c right of 5b!)
5
3
2
Root

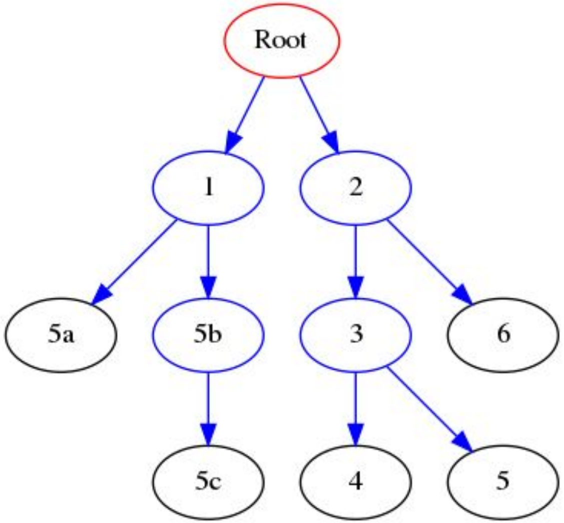
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack



Recursively

1. DFS(left)
2. visit Root
3. DFS(right)

Inorder:
5a
1
5b
5c
Root
4
3
5

Stack: (5c right of 5b!)
3
2
Root

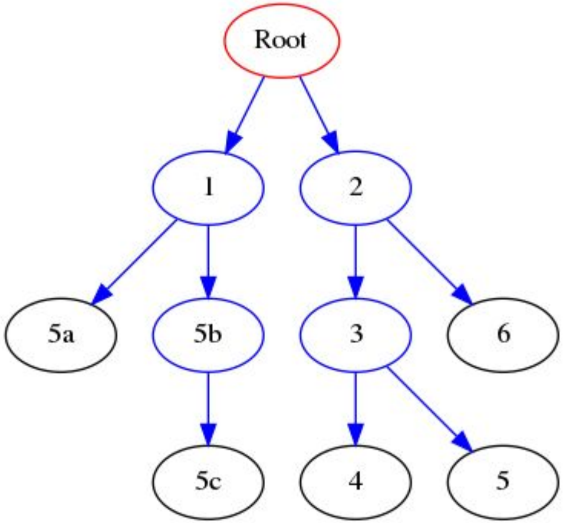
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack



Recursively

1. DFS(left)
2. visit Root
3. DFS(right)

Inorder:
5a
1
5b
5c
Root
4
3
5

Stack: (5c right of 5b!)
2
Root

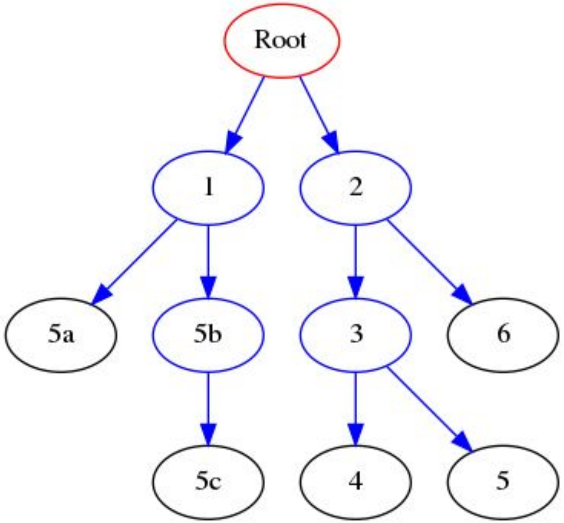
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack



Recursively

1. DFS(left)
2. visit Root
3. DFS(right)

Inorder:
5a
1
5b
5c
Root
4
3
5
2

Stack: (5c right of 5b!)
2
Root

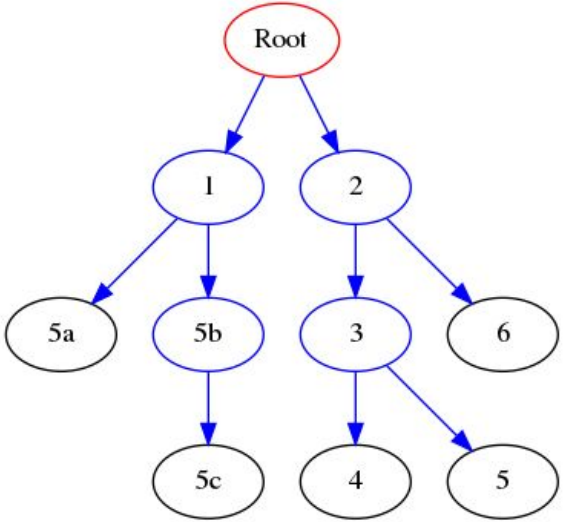
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack



Recursively

1. DFS(left)
2. visit Root
3. DFS(right)

Inorder:
5a
1
5b
5c
Root
4
3
5
2
6

Stack: (5c right of 5b!)
6
2
Root

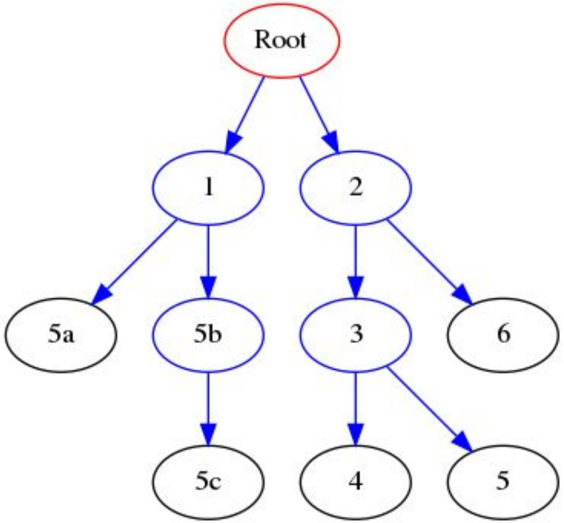
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack



Recursively

1. DFS(left)
2. visit Root
3. DFS(right)

Inorder:

5a
1
5b
5c
Root
4
3
5
2
6

Stack: (5c right of 5b!)

2
Root

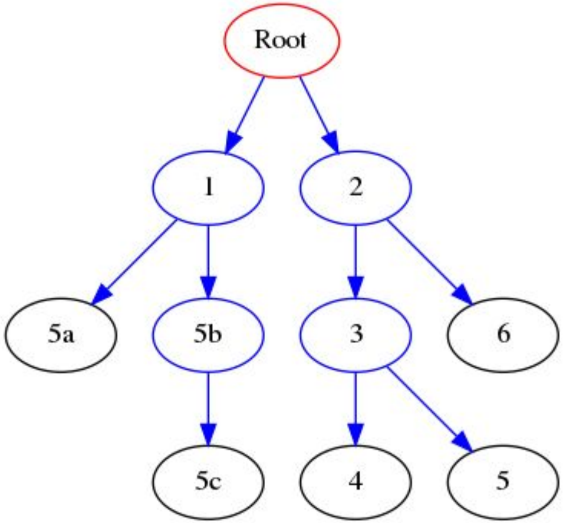
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack



Recursively

1. DFS(left)
2. visit Root
3. DFS(right)

Inorder:

5a
1
5b
5c
Root
4
3
5
2
6

Stack: (5c right of 5b!)

Root

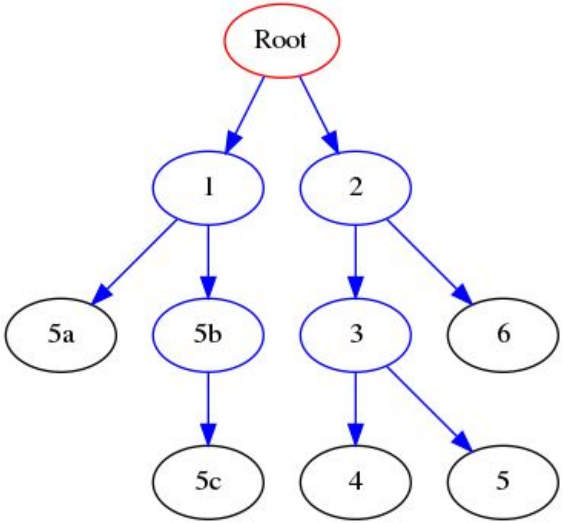
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack



Recursively

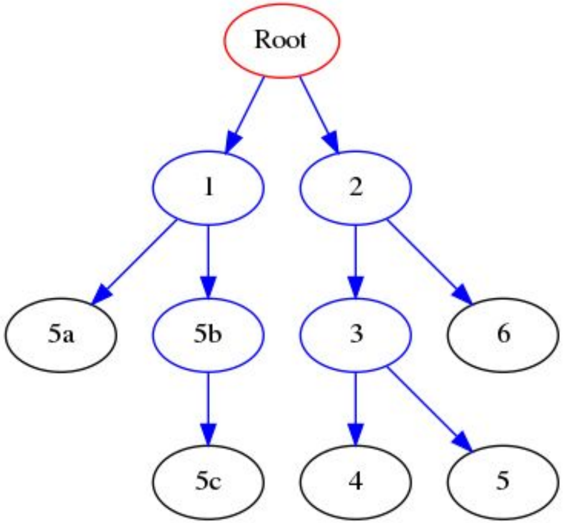
1. DFS(left)
2. visit Root
3. DFS(right)

Inorder:
5a
1
5b
5c
Root
4
3
5
2
6

Stack: (5c right of 5b!)

empty. Done!

Tree traversals



Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack

Recursively

1. DFS(left)
2. DFS(right)
3. visit Root

Postorder:

5a
5c (right of 5b)
5b
1
4
5
3
6
2
Root

Stack: Exercise!

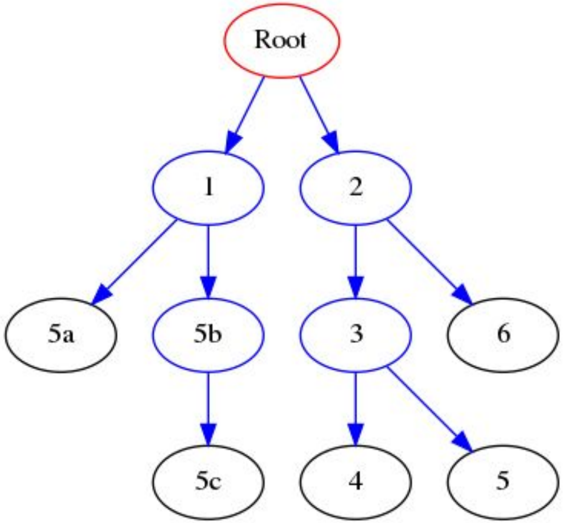
DFS: the code

Notes:

- visit means “print”
- it is a method of the class BinaryTree

```
def DFS(node, kind = "preorder"):
    if node != None:
        if kind == "preorder":
            print("{}".format(node.getValue()))
        DFS(node.getLeft(), kind = kind)
        if kind == "inorder":
            print("{}".format(node.getValue()))
        DFS(node.getRight(), kind = kind)
        if kind == "postorder":
            print("{}".format(node.getValue()))
```

implicit stack



Preorder:	**Inorder:**	**Postorder:**
Root	5a	5a
1	1	5c
5a	5b	5b
5b	5c	1
5c	Root	4
2	4	5
3	3	3
4	5	6
5	2	2
6	6	Root

Tree traversals

Tree traversal / search

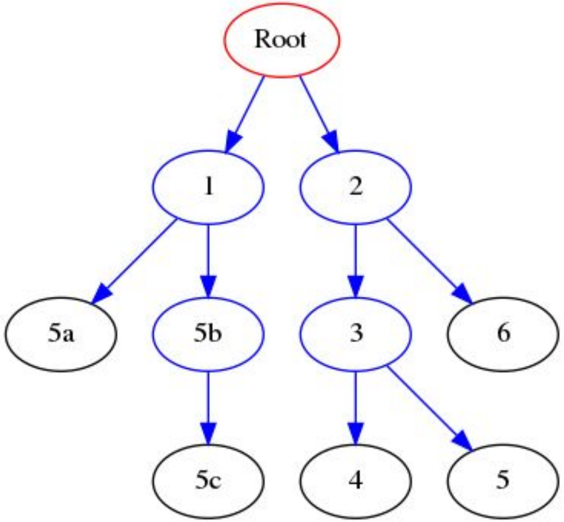
A strategy to pass through (visit) all the nodes of a tree.

Depth-First Search (DFS)

- Each subtree of the tree is visited, one after another
- Three variants (pre/in/post order)
- Requi res a stack

Breadth-First Search (BFS)

- Each level of the tree is visited, one after the other
- Starts from the root
- Requires a queue



Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

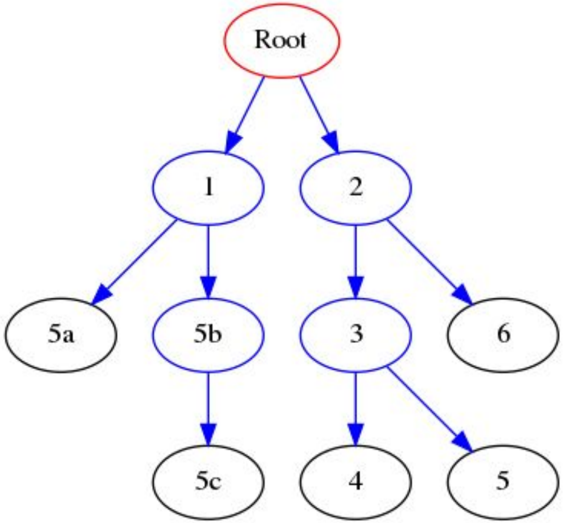
Breadth-First Search (BFS)

- Each level of the tree is visited, one after the other
- Starts from the root
- Requires a queue

0. Add root to the queue Q

Recursively

1. get node from Q
2. visit the node
3. add all children to Q



Visit order

Queue
Root

Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

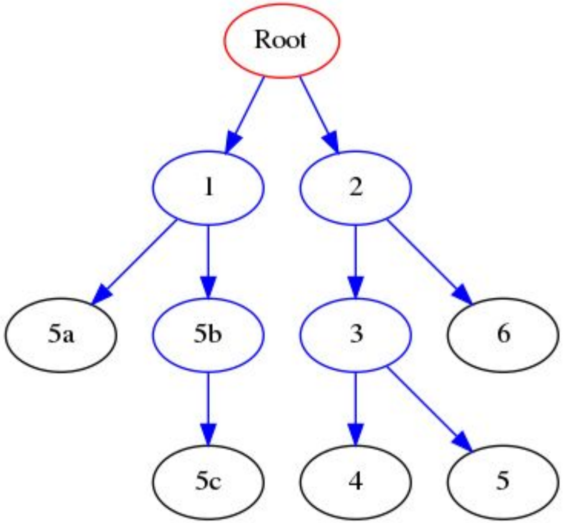
Breadth-First Search (BFS)

- Each level of the tree is visited, one after the other
- Starts from the root
- Requires a queue

0. Add root to the queue Q

Recursively

1. get node from Q
2. visit the node
3. add all children to Q

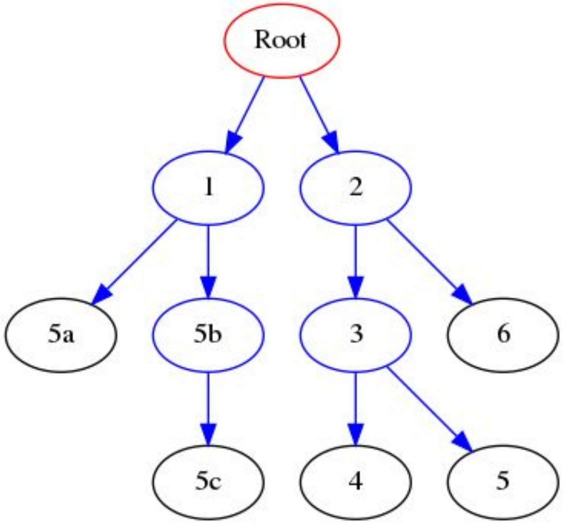


Visit order	**Queue**
Root	1 , 2

Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.



Breadth-First Search (BFS)

- Each level of the tree is visited, one after the other
- Starts from the root
- Requires a queue

0. Add root to the queue Q

Recursively

1. get node from Q
2. visit the node
3. add all children to Q

Visit order	**Queue**
Root	2, 5a, 5b
1	

Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

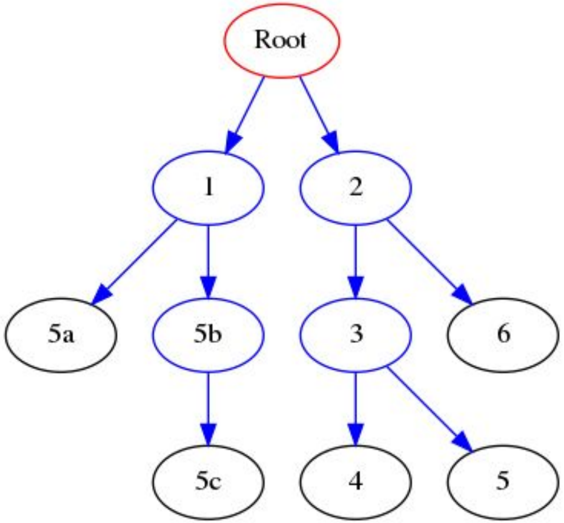
Breadth-First Search (BFS)

- Each level of the tree is visited, one after the other
- Starts from the root
- Requires a queue

0. Add root to the queue Q

Recursively

1. get node from Q
2. visit the node
3. add all children to Q



Visit order
Root
1
2

Queue
5a, 5b, 3, 6

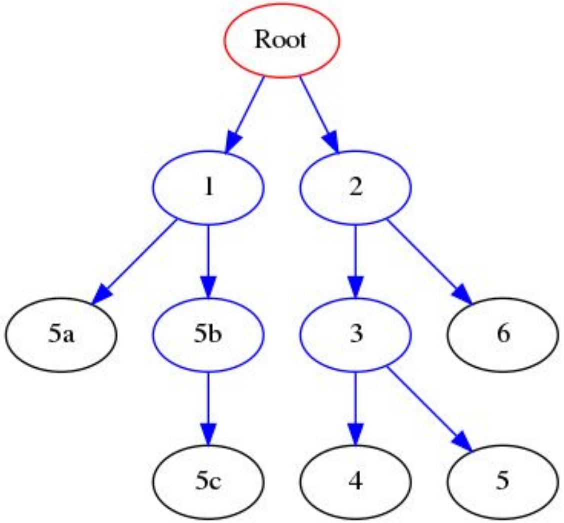
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Breadth-First Search (BFS)

- Each level of the tree is visited, one after the other
- Starts from the root
- Requires a queue



0. Add root to the queue Q

Recursively

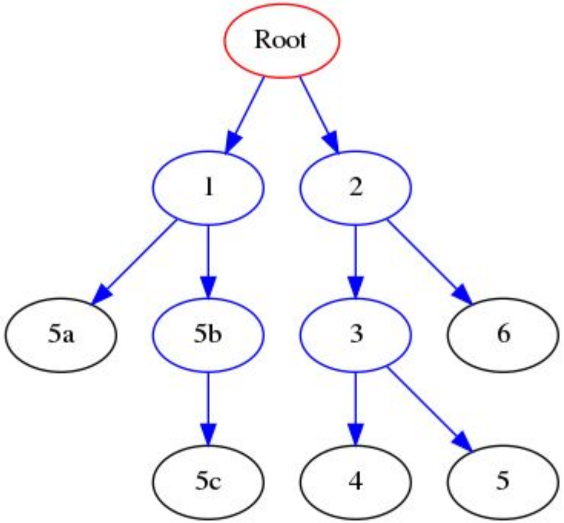
1. get node from Q
2. visit the node
3. add all children to Q

Visit order	**Queue**
Root	5b, 3, 6
1	
2	
5a	

Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.



Breadth-First Search (BFS)

- Each level of the tree is visited, one after the other
- Starts from the root
- Requires a queue

0. Add root to the queue Q

Recursively

1. get node from Q
2. visit the node
3. add all children to Q

Visit order	**Queue**
Root	3, 6, 5c
1	
2	
5a	
5b	

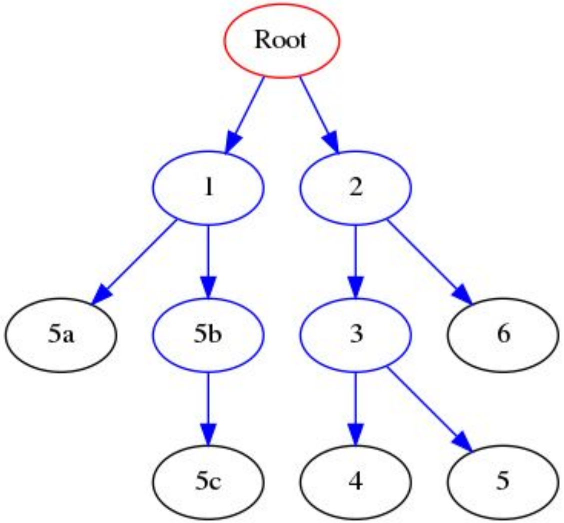
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Breadth-First Search (BFS)

- Each level of the tree is visited, one after the other
- Starts from the root
- Requires a queue



0. Add root to the queue Q

Recursively

1. get node from Q
2. visit the node
3. add all children to Q

Visit order
Root
1
2
5a
5b
3

Queue
6, 5c, 4, 5

Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

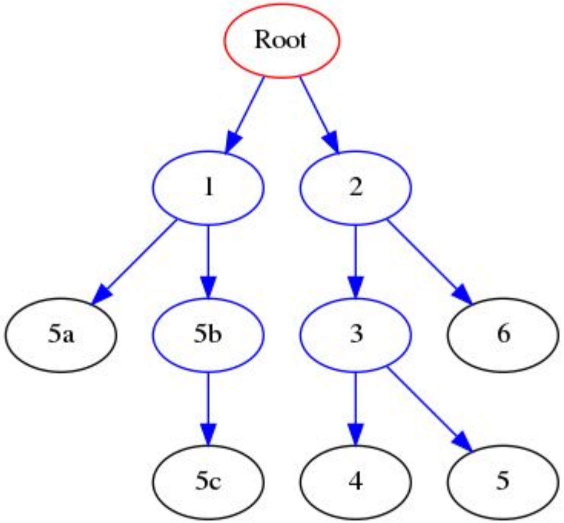
Breadth-First Search (BFS)

- Each level of the tree is visited, one after the other
- Starts from the root
- Requires a queue

0. Add root to the queue Q

Recursively

1. get node from Q
2. visit the node
3. add all children to Q

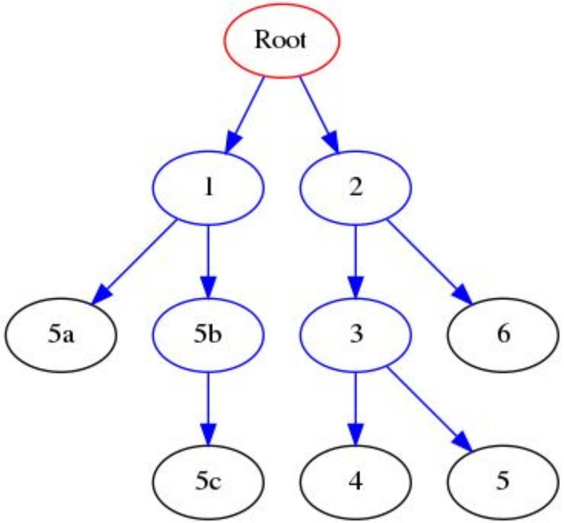


Visit order	Queue
Root	5c, 4, 5
1	
2	
5a	
5b	
3	
6	

Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.



Breadth-First Search (BFS)

- Each level of the tree is visited, one after the other
- Starts from the root
- Requires a queue

0. Add root to the queue Q

Recursively

1. get node from Q
2. visit the node
3. add all children to Q

Visit order	**Queue**
Root	4, 5
1	
2	
5a	
5b	
3	
6	
5c	

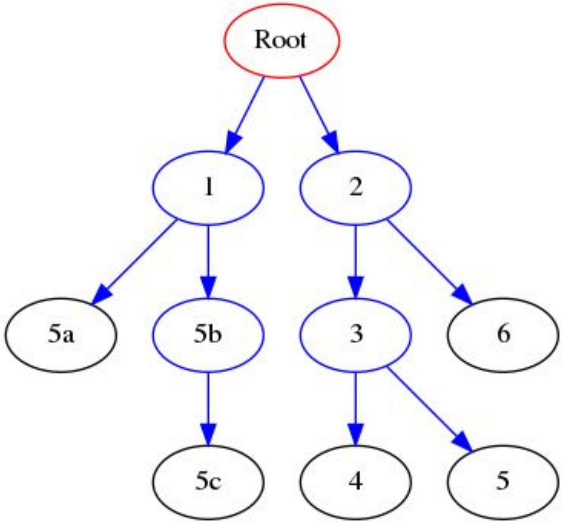
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Breadth-First Search (BFS)

- Each level of the tree is visited, one after the other
- Starts from the root
- Requires a queue



0. Add root to the queue Q

Recursively

1. get node from Q
2. visit the node
3. add all children to Q

Visit order	**Queue**
Root	5
1	
2	
5a	
5b	
3	
6	
5c	
4	

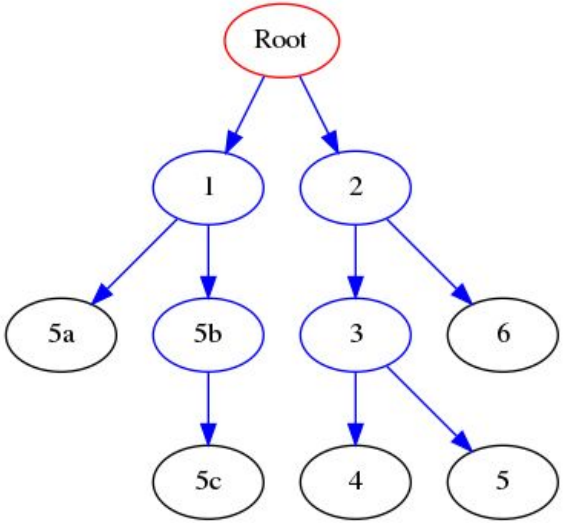
Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.

Breadth-First Search (BFS)

- Each level of the tree is visited, one after the other
- Starts from the root
- Requires a queue



0. Add root to the queue Q

Recursively

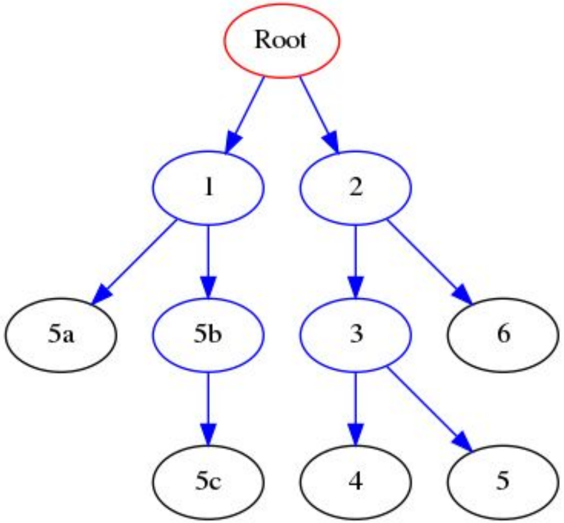
1. get node from Q
2. visit the node
3. add all children to Q

Visit order	**Queue**
Root	
1	Empty. Done
2	
5a	
5b	
3	
6	
5c	
4	
5	

Tree traversals

Tree traversal / search

A strategy to pass through (visit) all the nodes of a tree.



Breadth-First Search (BFS)

- Each level of the tree is visited, one after the other
- Starts from the root
- Requires a queue

0. Add root to the queue Q

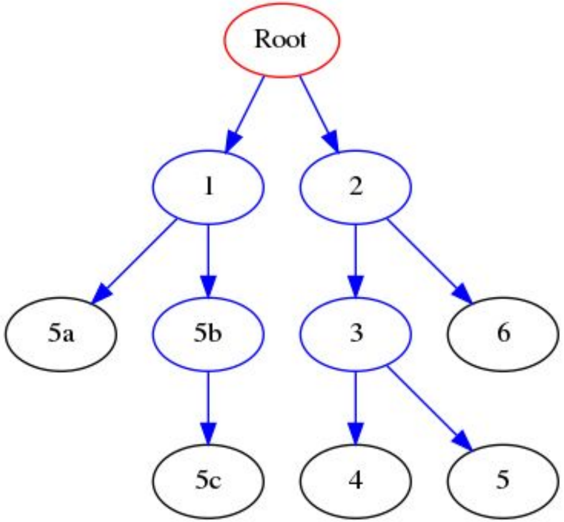
Recursively

1. get node from Q
2. visit the node
3. add all children to Q

Visit order	**Level**
Root	0
1	1
2	1
5a	2
5b	2
3	2
6	2
5c	3
4	3
5	3

Tree traversals: BFS

```
from collections import deque

def BFS(node):
        Q = deque()
        if node != None:
            Q.append(node)

        while len(Q) > 0:
            curNode = Q.popleft()
            if curNode != None:
                print("{}".format(curNode.getValue()))
                Q.append(curNode.getLeft())
                Q.append(curNode.getRight())
```



BFS visit:
Root
1
2
5a
5b
3
6
5c
4
5

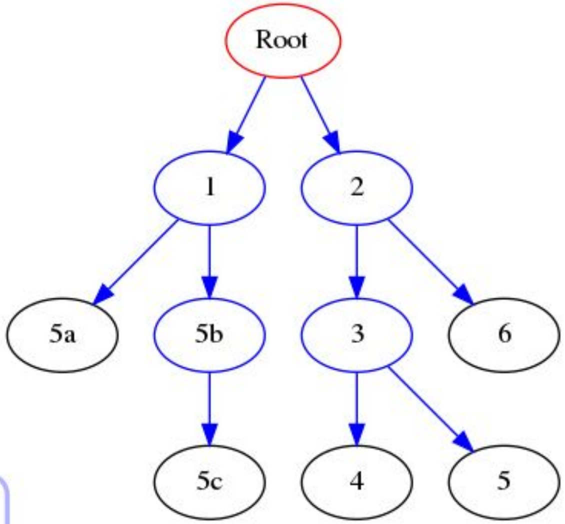
Algorithm

0. Add root to the queue Q

Recursively

1. get node from Q
2. visit the node
3. add all children to Q

Tree traversals: complexity

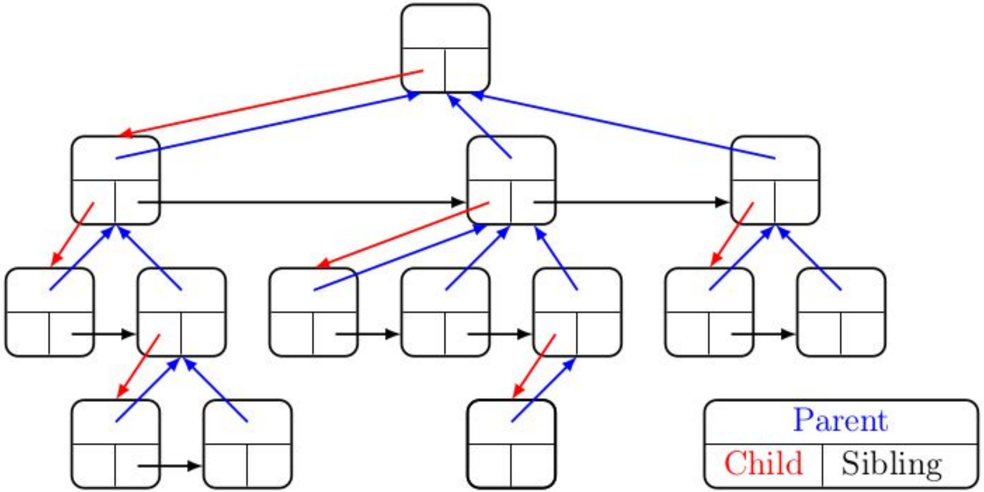


The cost of a visit of a tree containing n nodes is $\Theta(n)$, because each node is visited exactly once.

Generic trees

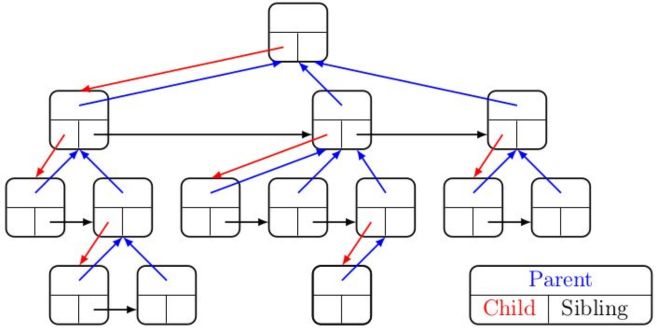
Generic Trees are like binary trees, but **each node can have more than 2 children**. One possible implementation is that each node (that is a subtree in itself) has a **value**, a link to its **parent** and a **list of children**.

Another implementation is that each node has a **value**, a link to its **parent**, a link to its **next sibling** and a link to its **first child**.



Generic trees

TREE
% Build a new node, initially containing v, with no children or parent
Tree(OBJECT v)
% Read the value stored in nodes
OBJECT getValue()
% Write the value stored in nodes
setValue(OBJECT v)
% Returns the parent, or `None` if this node is root
TREE getParent()
% Returns the first child, or `None` if this node is leaf
TREE leftmostChild()
% Returns the next sibling, or `None` if there is none
TREE rightSibling()
% Insert the subtree t as first child of this node
insertChild(TREE t)
% Insert the subtree t as next sibling of this node
insertSibling(TREE t)
% Destroy the subtree rooted in the first child
deleteChild()
% Destroy the subtree rooted in the next sibling
deleteSibling()



Exercise!

Exercise

The visit order of a binary tree containing 9 nodes are the following:

- A, E, B, F, G, C, D, I, H (pre-order) Root-Left-Right
- B, G, C, F, E, H, I, D, A (post-order) Left-Right-Root
- B, E, G, F, C, A, D, H, I (in-order) Left-Root-Right

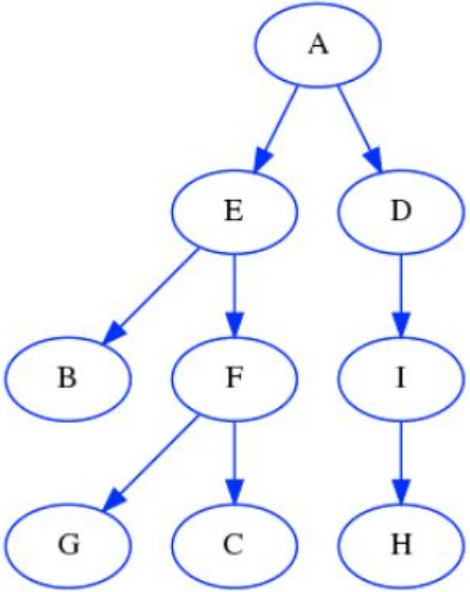
What is the corresponding binary tree? Explain.

Exercise

The visit order of a binary tree containing 9 nodes are the following:

- A, E, B, F, G, C, D, I, H (pre-order)
- B, G, C, F, E, H, I, D, A (post-order)
- B, E, G, F, C, A, D, H, I (in-order)

What is the corresponding binary tree? Explain.

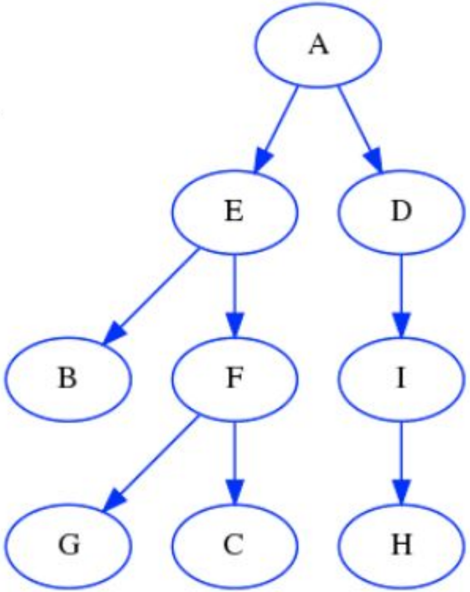


Preorder visit	Postorder visit	Inorder visit
A	B	B
E	G	E
B	C	G
F	F	F
G	E	C
C	H	A
D	I	D
I	D	H
H	A	I

where I is on the right of D and H is on the left of I

Exercises

- The width of a binary tree is the largest number of nodes that belong to the same level. Write a function that given a tree t, returns the width of t.
- The minimal height of a binary tree t is the minimal distance between node v and any of the leaf in its subtree. Write a function that given a tree t, returns the minimal height of t.
- Write a function that given a binary tree t and an integer k, returns the number of nodes at level k



Width: 3
Minimal height: 2
k = 2 → output: 3

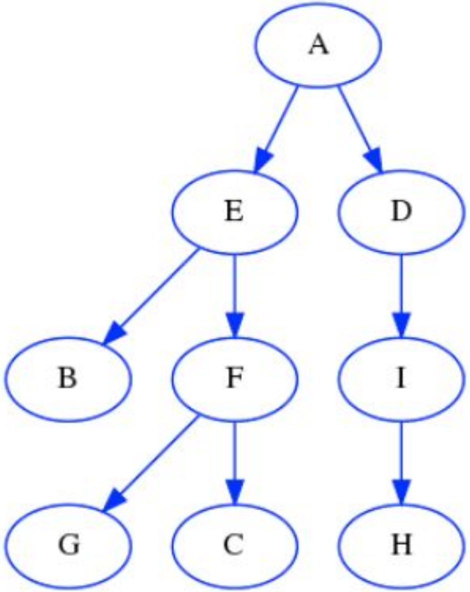
Exercise: width

```python
def getWidth(tree):
    """gets the width of the tree"""
    if tree == None:
        return 0

    level = [tree]
    res = 1
    while len(level) > 0:
        print("Level: {}".format([x.getValue() for x in level]))
        tmp = []
        for t in level:
            r = t.getRight()
            l = t.getLeft()
            if r != None:
                tmp.append(r)
            if l != None:
                tmp.append(l)
        res = max(res,len(tmp))
        level = tmp

    return res
```

```python
print("Width of tree: {}".format(getWidth(exer)))
```

```
Level: ['A']
Level: ['D', 'E']
Level: ['I', 'F', 'B']
Level: ['H', 'C', 'G']
Width of tree: 3
```

similar to BFS but we need to explicitly store the level...

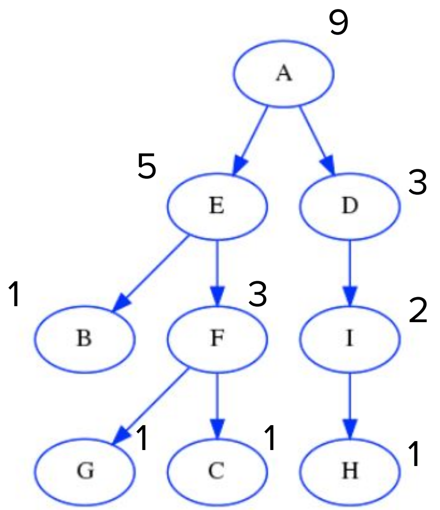


Min Height and nodes at level k are similar...

Exercise: count the nodes of each (sub)tree

How many nodes does a (sub)tree have?

IDEA: similar to DFS postorder-visit (summing the counts). Remember to add 1 for the root.

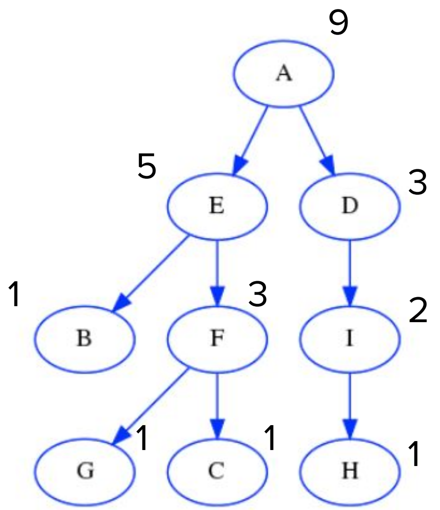


Exercise: count the nodes of each (sub)tree

How many nodes does a (sub)tree have?

IDEA: similar to DFS postorder-visit (summing the counts). Remember to add 1 for the root.

```python
def count_nodes(tree):
    """counts the nodes of each (sub)tree rooted at 'tree'"""
    if tree == None:
        return 0
    else:
        l = count_nodes(tree.getLeft())
        r = count_nodes(tree.getRight())
        return l + r + 1 #the count of the right, that of the left + the root
```



The tree rooted at 'A' has 9 nodes
The tree rooted at 'E' has 5 nodes
The tree rooted at 'D' has 3 nodes
The tree rooted at 'B' has 1 nodes
The tree rooted at 'F' has 3 nodes
The tree rooted at 'I' has 2 nodes
The tree rooted at 'G' has 1 nodes
The tree rooted at 'C' has 1 nodes
The tree rooted at 'H' has 1 nodes