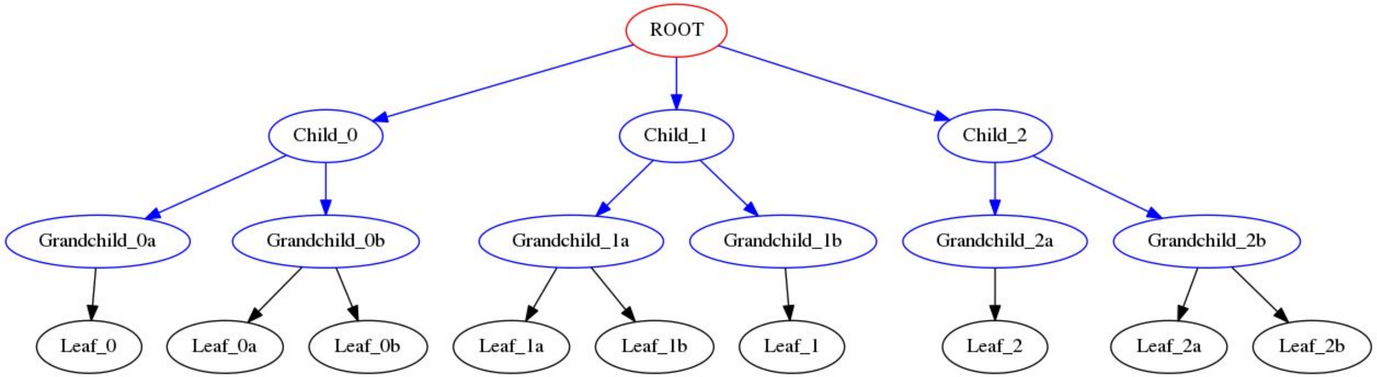
Scientific Programming Practical 17

Introduction

Luca Bianco - Academic Year 2018-19
luca.bianco@fmach.it

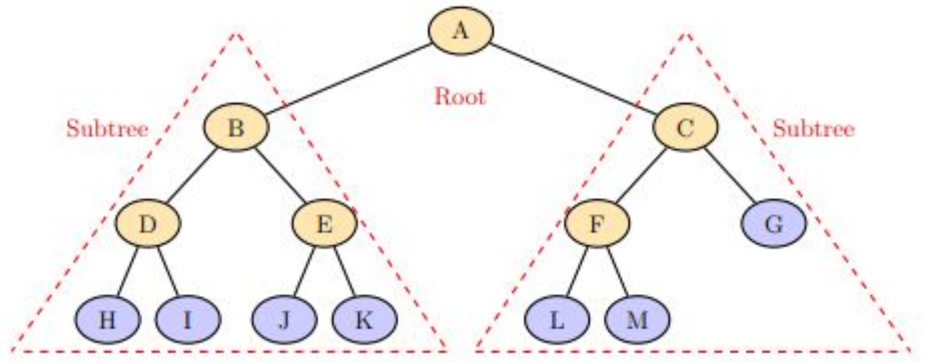
Trees

Trees are data structures composed of two elements *nodes* and *edges*. Nodes represent *things* and edges represent *relationships* among **two** nodes.
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**.



Tree: recursive definition

*A tree is **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.*

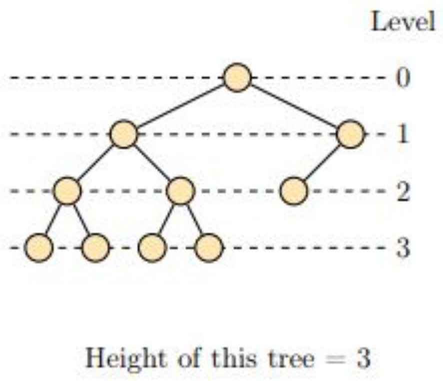


- A is the tree root
- B, C are roots of their subtrees
- D, E are siblings
- D, E are children of B
- B is the parent of D, E
- Purple nodes are leaves
- The other nodes are internal nodes

Tree: some properties

Some properties of **nodes** and **trees** are:

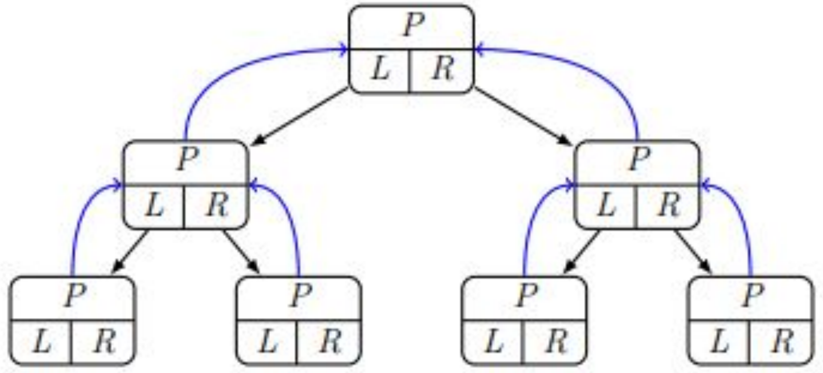
1. The **depth of a node** N: the **length of the pat**h connecting the root to N counted as number of edges in the path.
2. The **level (n)** of a tree: **set of nodes at the same depth** (n).
3. **Height** : is the maximum depth of all leaves.
4. **Width** : the maximum number of nodes in each level.



Width of this tree is = 4

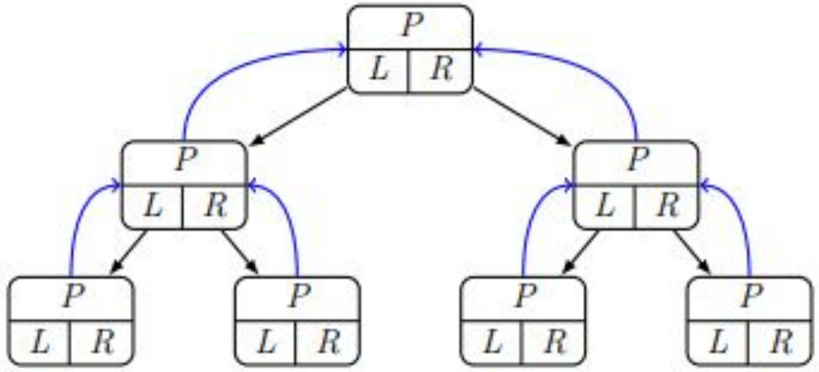
Binary Trees

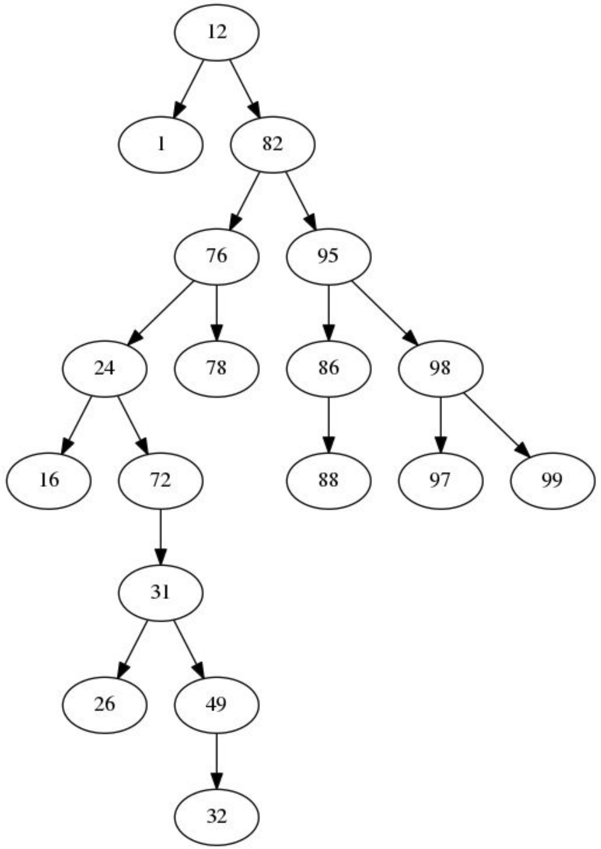
Binary trees are **trees** where each node has at most two children: the **right child** and the **left child**.



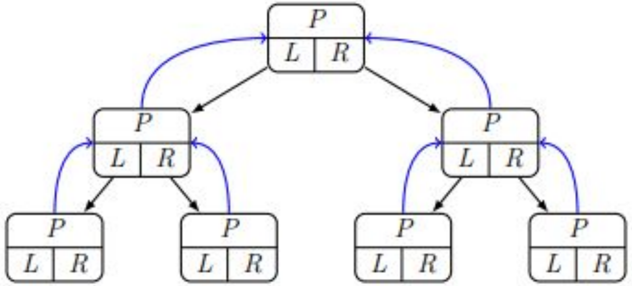
Binary Trees

Binary trees are **trees** where each node has at most two children: the **right child** and the **left child**.

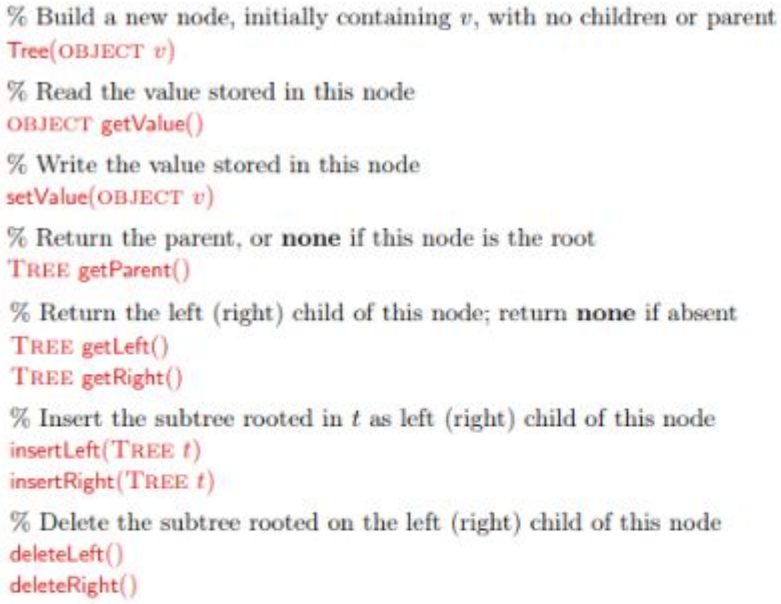




Binary Trees: ADT



Note: if we want to keep attributes as private, we need to add another method to the previously seen methods: **myTree.setParent(tree)** that sets the parent of a tree **myTree** to **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()

```python
class BinaryTree:
    def __init__(self, value):
        self.__data = value
        self.__right = None
        self.__left = None
        self.__parent = None

    def getValue(self):
        return self.__data

    def setValue(self, newval):
        self.__data = newval

    def getParent(self):
        return self.__parent
    #needed because we are using private attributes
    def setParent(self, tree):
        self.__parent = tree

    def getRight(self):
        return self.__right

    def getLeft(self):
        return self.__left
```

```python
    def insertRight(self, tree):
        if self.__right == None:
            self.__right = tree
            tree.setParent(self)

    def insertLeft(self, tree):
        if self.__left == None:
            self.__left = tree
            tree.setParent(self)

    def deleteRight(self):
        self.__right = None

    def deleteLeft(self):
        self.__left = None



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)
        #print("{}{}".format("\t"*lev, cur.getValue()))
        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))
```

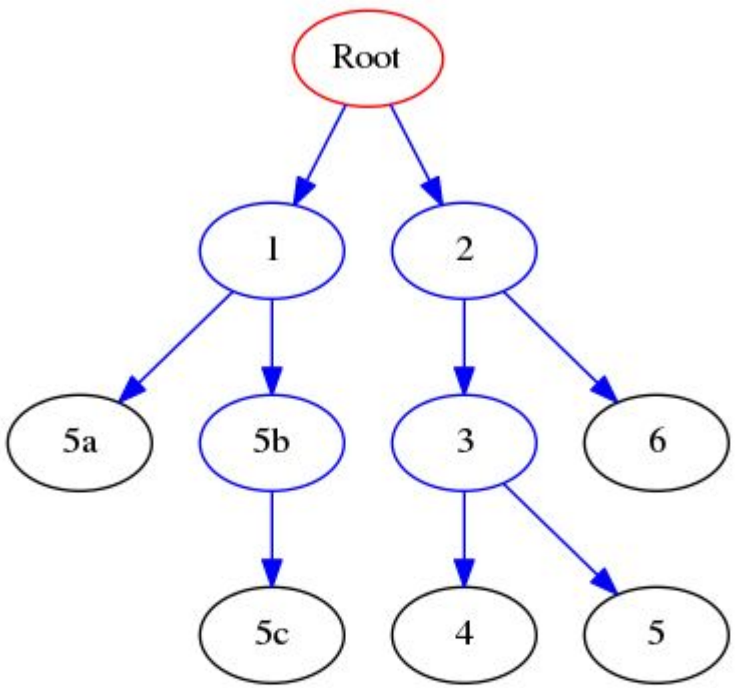
```python
if __name__ == "__main__":
    BT = BinaryTree("Root")
    bt1 = BinaryTree(1)
    bt2 = BinaryTree(2)
    bt3 = BinaryTree(3)
    bt4 = BinaryTree(4)
    bt5 = BinaryTree(5)
    bt6 = BinaryTree(6)
    bt5a = BinaryTree("5a")
    bt5b = BinaryTree("5b")
    bt5c = BinaryTree("5c")

    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)
    printTree(BT)
    print("\nDelete right branch of 2")
    bt2.deleteRight()
    printTree(BT)

    print("\nInsert left branch of 5")
    newN = BinaryTree("child of 5")
    bt5.insertLeft(newN)
    printTree(BT)
```

```
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

Delete right branch of 2
Root (r)-> 2
Root (l)-> 1
        1 (r)-> 5b
        1 (l)-> 5a
                5b (r)-> 5c
        2 (l)-> 3
                3 (r)-> 5
                3 (l)-> 4

Insert left branch of 5
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
```

Visiting the nodes

Viting a tree, means going through the nodes following the connections.

There are two different types of visits: **depth first search** and **breadth first search**.

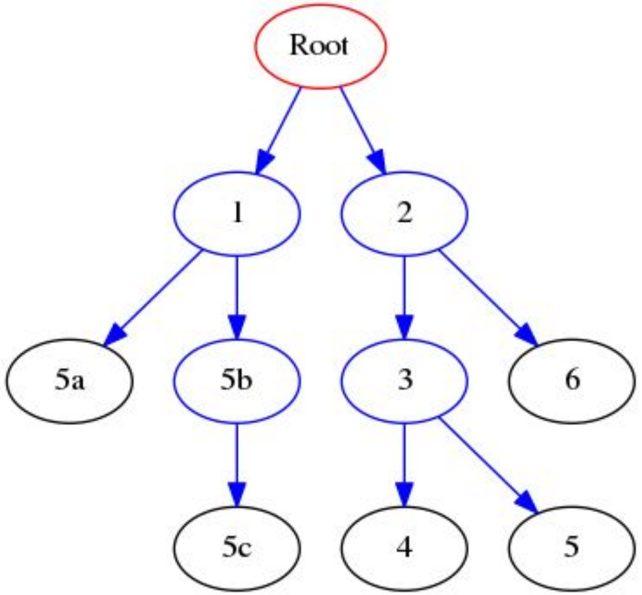


Depth First Search (DFS)

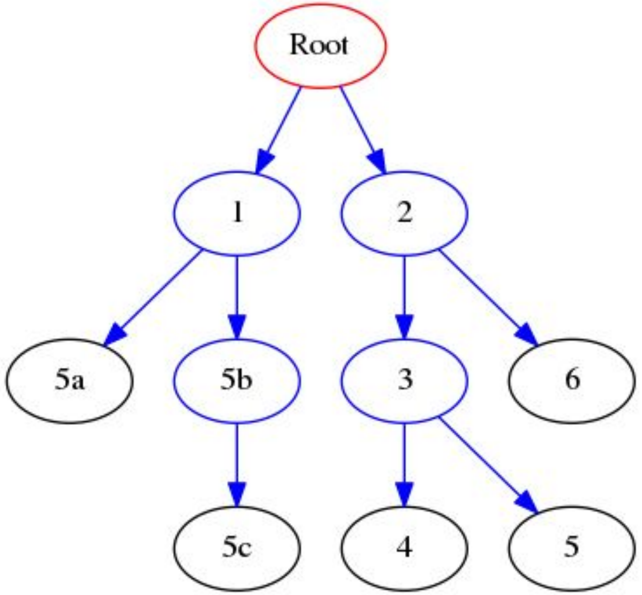
Given a tree T, depth first search (DFS) **visits all the subtrees of T going as deep as it can before going back and down another branch** until all the tree is visited.

3 different types of visits:

1. the root is visited before the visiting the subtree: pre-order;
2. the root is visited after the left subtree but before the right subtree: in-order;
3. the root is visited after the left and right subtrees: post-order



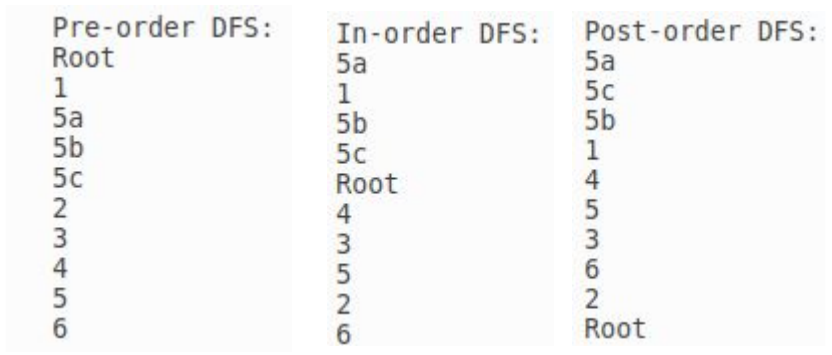
Depth First Search (DFS)



Given a tree T, depth first search (DFS) **visits all the subtrees of T going as deep as it can before going back and down another branch** until all the tree is visited.

3 different types of visits:

1. the root is visited before the visiting the subtree: pre-order;
2. the root is visited after the left subtree but before the right subtree: in-order;
3. the root is visited after the left and right subtrees: post-order



```
Pre-order DFS:
Root
1
5a
5b
5c
2
3
4
5
6
```

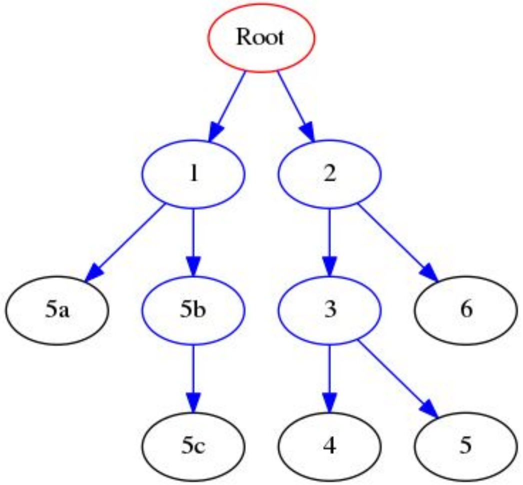
```
In-order DFS:
5a
1
5b
5c
Root
4
3
5
2
6
```

```
Post-order DFS:
5a
5c
5b
1
4
5
3
6
2
Root
```

Depth First Search (DFS)

```
def preOrderDFS(self):
    if self != None:
        r = self.getRight()
        l = self.getLeft()
        print(self.getValue())
        if l != None:
            l.preOrderDFS()
        if r != None:
            r.preOrderDFS()

def inOrderDFS(self):
    if self != None:
        r = self.getRight()
        l = self.getLeft()
        if l != None:
            l.inOrderDFS()
        print(self.getValue())

        if r != None:
            r.inOrderDFS()
```

```
def postOrderDFS(self):
    if self != None:
        r = self.getRight()
        l = self.getLeft()
        if l != None:
            l.postOrderDFS()
        if r != None:
            r.postOrderDFS()

        print(self.getValue())
```

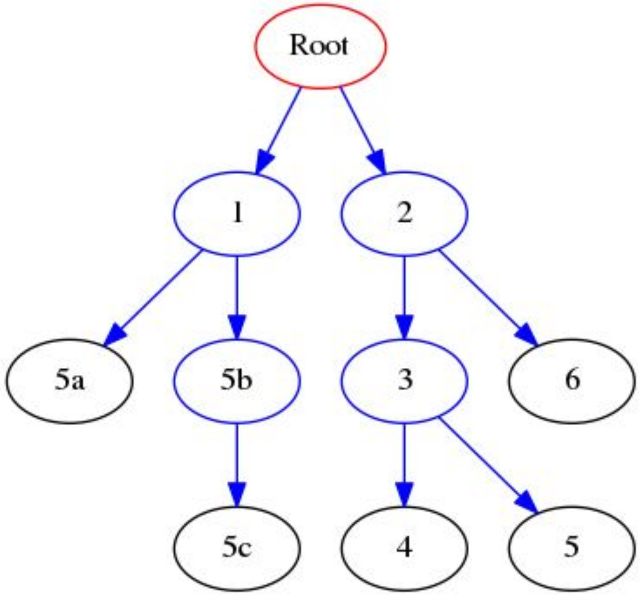


```
Pre-order DFS:    In-order DFS:    Post-order DFS:
Root              5a               5a
1                 1                5c
5a                5b               5b
5b                5c               1
5c                Root             4
2                 4                5
3                 3                3
4                 5                6
5                 2                2
6                 6                Root
```

Breadth First Search (BFS)

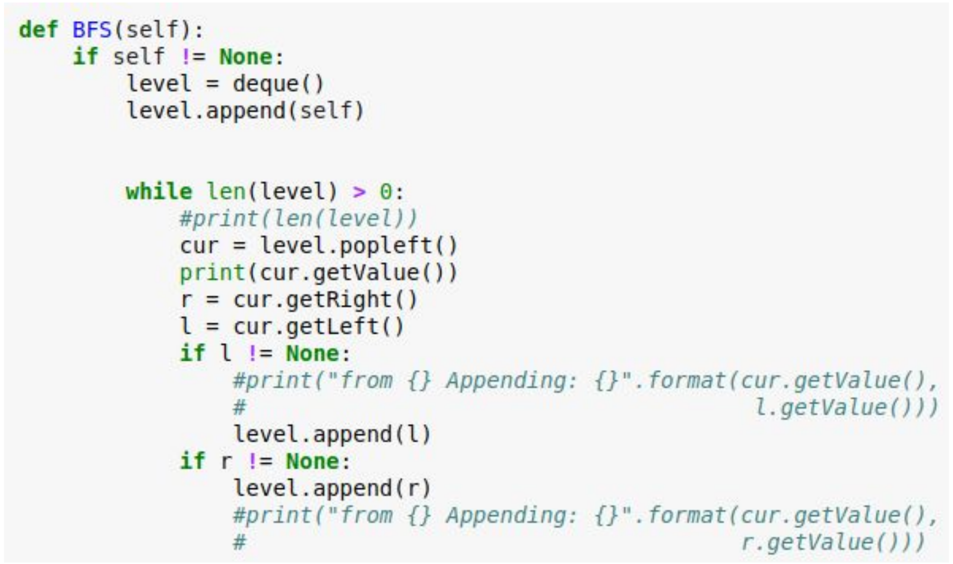
Breadth first search visits **all the higher levels of a tree before going down**. Basically, this search goes **as wide as it can before going deep**.

This visit **makes use of a queue**: for each tree, it adds the left and right subtrees to the queue and recursively visits them in the order they have been pushed into the queue.

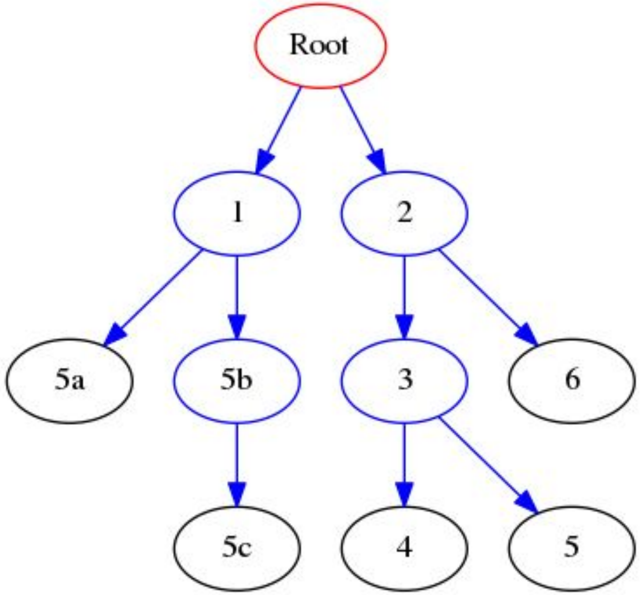


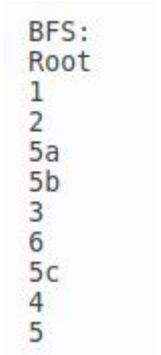
```
BFS:
Root
1
2
5a
5b
3
6
5c
4
5
```

Breadth First Search (BFS)



```
def BFS(self):
    if self != None:
        level = deque()
        level.append(self)


        while len(level) > 0:
            #print(len(level))
            cur = level.popleft()
            print(cur.getValue())
            r = cur.getRight()
            l = cur.getLeft()
            if l != None:
                #print("from {} Appending: {}".format(cur.getValue(),
                #                                     l.getValue()))
                level.append(l)
            if r != None:
                level.append(r)
                #print("from {} Appending: {}".format(cur.getValue(),
                #                                   r.getValue()))
```

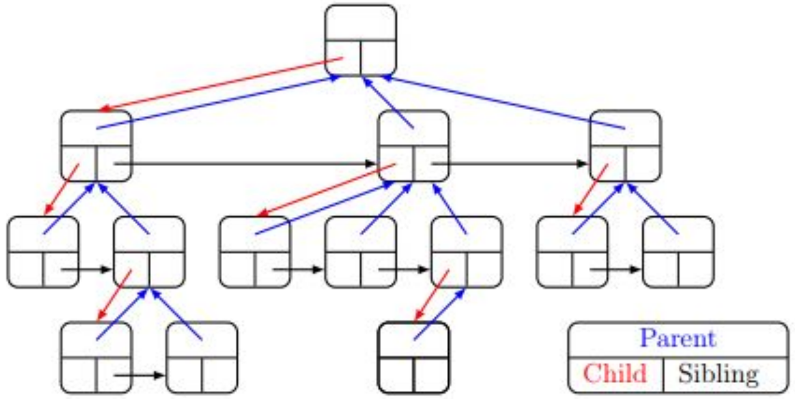




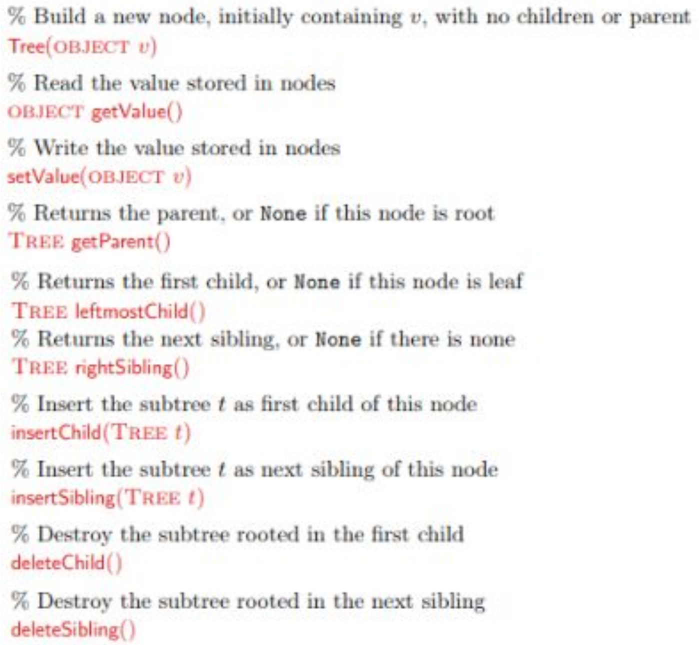
```
BFS:
Root
1
2
5a
5b
3
6
5c
4
5
```

Generic Trees

Generic Trees are like binary trees, but **each node can have more than 2 children**. In a possible implementation, 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: ADT



% 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()

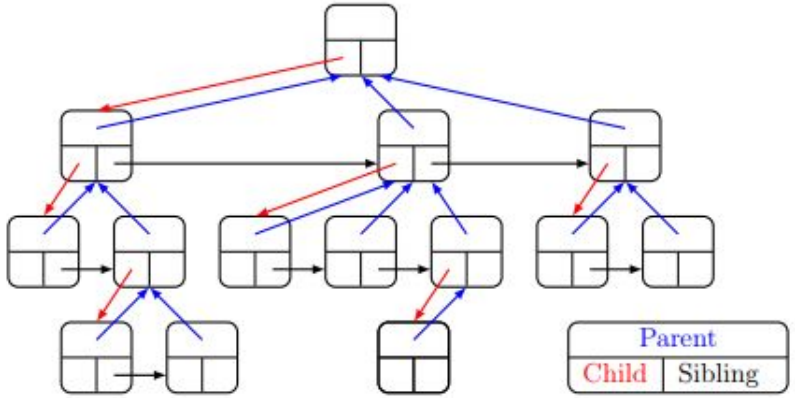
% Returns the first child, or `None` if this node is leaf
TREE leftmostChild()
% Returns the next sibling, or `None` if there is none
TREE rightSibling()

% 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()



Generic Trees: ADT

```python
class GenericTree:
    #This actually allows for forests
    #unless we force the root to have no siblings.
    def __init__(self, value):
        self.__data = value
        self.__parent = None
        self.__child = None
        self.__sibling = None

    def getValue(self):
        return self.__data

    def setValue(self,newvalue):
        self.__data = newvalue

    def getParent(self):
        return self.__parent
```

```python
def insertChild(self,child):
    if type(child) != GenericTree:
        raise TypeError("parameter child is not a GenericTree")
    else:
        nextC = None
        print("from {} adding child --> {}".format(self.getValue(),
                                                   child.getValue()))
        if self.__child != None:
            nextC = self.__child
        child.setParent(self)
        child.setSibling(nextC)
        self.__child = child
```

% 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()

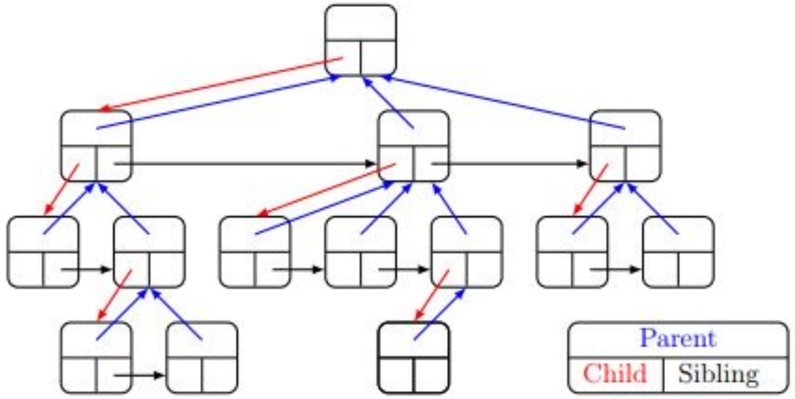
% Returns the next sibling, or **None** if there is none
TREE rightSibling()

% Insert the subtree t as first child of this node
insertChild(TREE t)

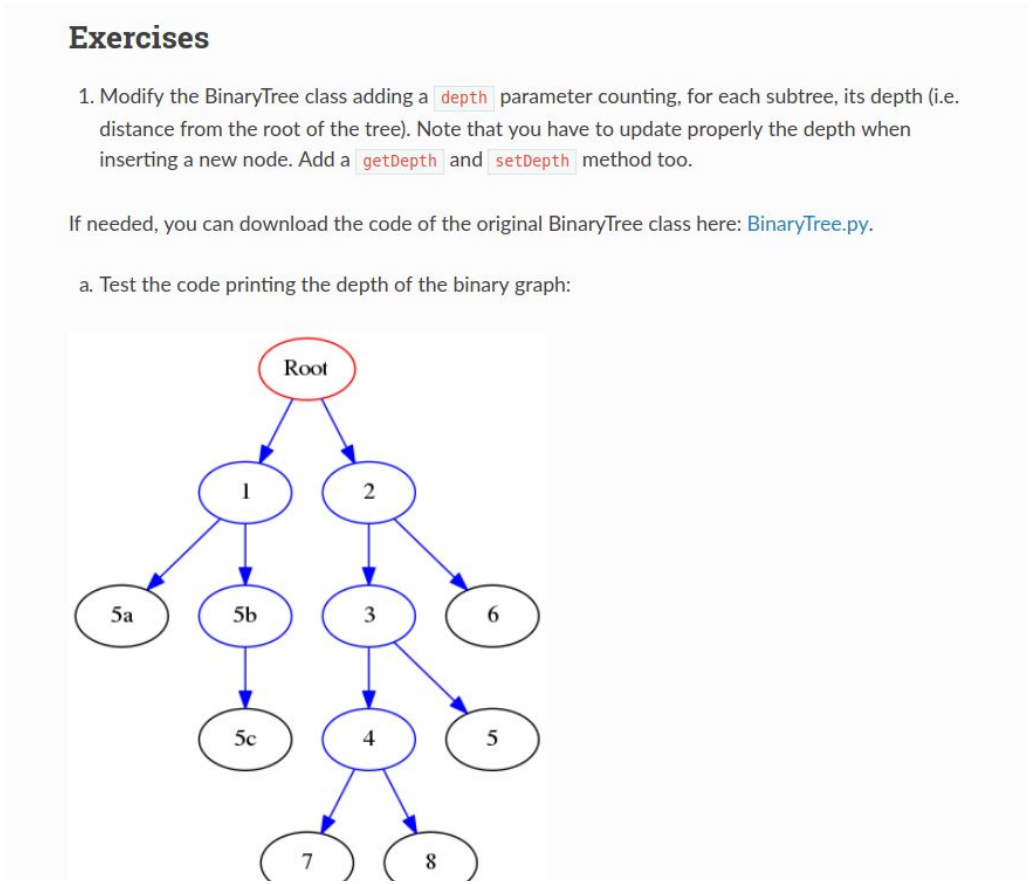
% Insert the subtree t as next sibling of this node
insertSibling(TREE t)

% Destroy the subtree rooted in the first child
deleteChild()

% Destroy the subtree rooted in the next sibling
deleteSibling()



http://qcbprolab.readthedocs.io/en/latest/practical17.html

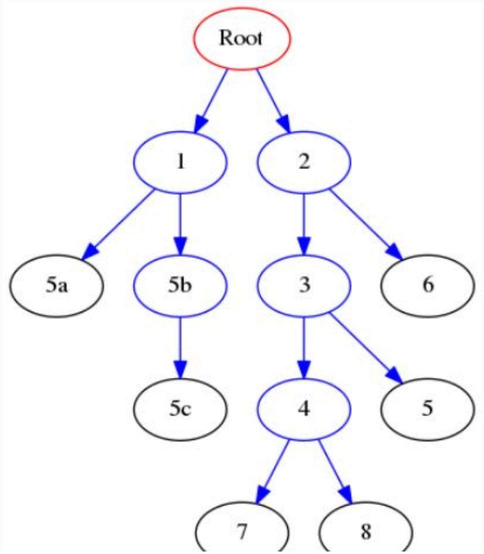


Exercises

1. Modify the BinaryTree class adding a `depth` parameter counting, for each subtree, its depth (i.e. distance from the root of the tree). Note that you have to update properly the depth when inserting a new node. Add a `getDepth` and `setDepth` method too.

If needed, you can download the code of the original BinaryTree class here: BinaryTree.py.

a. Test the code printing the depth of the binary graph:


Root
1
2
5a
5b
3
6
5c
4
5
7
8