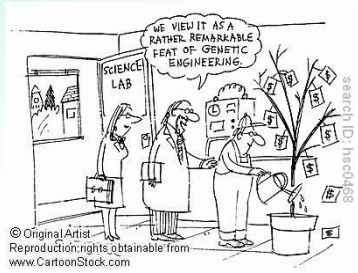
CompSci 105
Lecture 31 and 32 Content

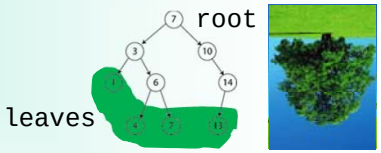
Trees – a non-linear data structure

Textbook: Chapter 6



What is a Tree?

A tree is an abstraction for a hierarchical structure.
It is defined as a set of points called *nodes* and a set of lines called *edges* where an edge connects two distinct nodes.



A tree has three properties:

- One node is distinguished called the root.
- Every node *n* other than the root is connected by an edge to exactly one other node *p* closer to the root.
- A tree is connected in the sense that if we start at any node *n* other than the root and move to the parent of *n*, continue to the parent of the parent of *n*, and so on, we eventually reach the root.

Applications of Trees

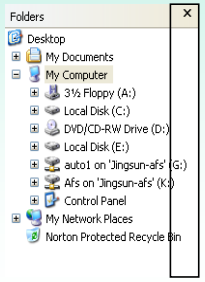
Examples and Applications of trees include:

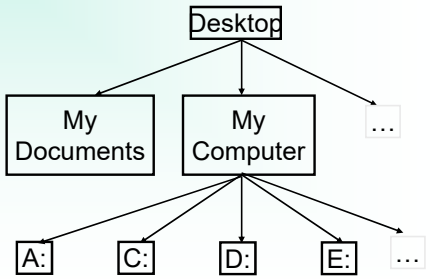
- Family trees
- Directory structures (file system)
- Arithmetic expressions
- Game trees (finding winning positions in a game)
- Binary Space Partitioning (BSP) trees (finding visible objects in a scene)
- Search trees (finding all the paths back out of a maze)
- Quadtrees (for efficient terrain rendering)
- Octree (for fast collision detection, ray tracing etc.)
- Constructed Solid Geometry (CSG) objects
- Joint hierarchies (for skeletal animation of characters)
- etc.

Applications of Trees (Cont'd)

File Systems

A file system can be represented as a tree, with the top-most directory as the root



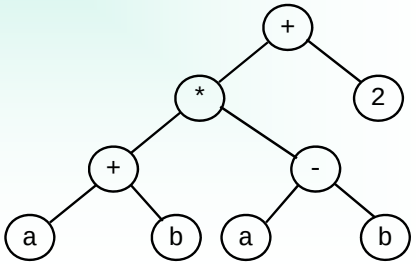


Applications of Trees (Cont'd)

Arithmetic Expressions

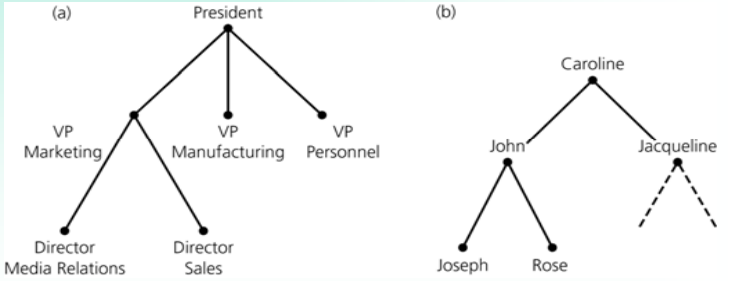
An arithmetic expression can be represented by a tree
- the leave nodes are the variables/values
- the internal nodes are the operations

(a+b) * (a-b) + 2



Applications of Trees (Cont'd)

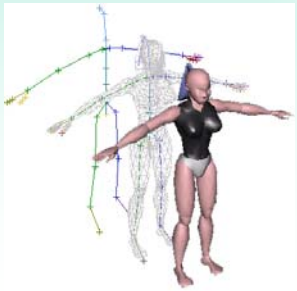
Organisational Charts and Family Trees



Applications of Trees (Cont'd)

Bone Hierarchies for Skeletal Animation

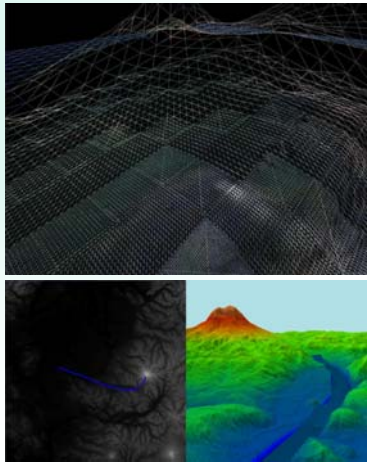
Moving a bone around a joint (e.g. the upper arm around the elbow) moves all child bones at the same time (e.g. the lower arm, hand and fingers)



Applications of Trees (Cont'd)

Quadtrees for Terrain Rendering

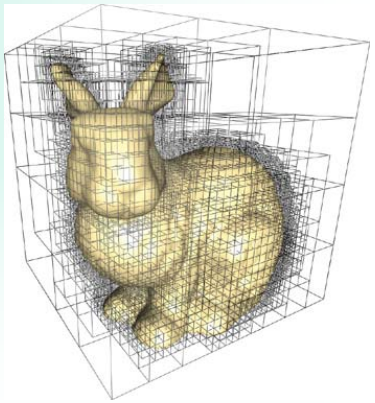
The terrain is divided into squares of different size: large squares for parts of the terrain far away from the viewer, and small squares for parts of the terrain close to the viewer. (Note: squares are divided into triangles in order to achieve continuity and faster rendering)



Applications of Trees (Cont'd)

Octrees for Space Partitioning

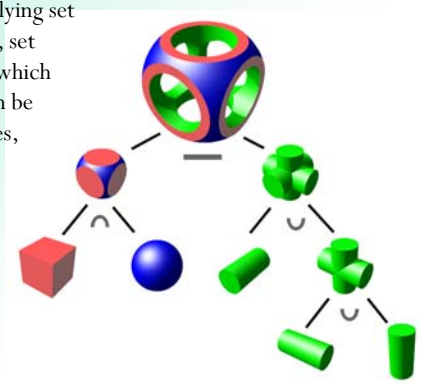
- Divide 3d into cubes.
- Subdivide cube if the object(s) it contains are too complex (e.g. for collision detection, rendering, simulation)



Applications of Trees (Cont'd)

Constructed Solid Geometry (CSG) Objects

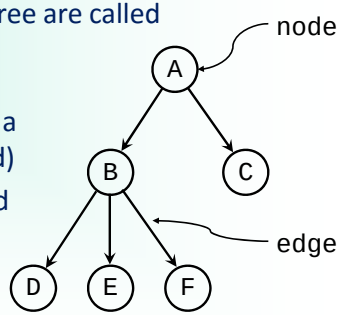
A CSG object is defined by applying set operations (intersection, union, set difference) to simpler objects, which can be again CSG objects or can be primitive objects such as spheres, squares and cylinders.



Terminology

Parts of a tree

- Data objects (the circles) in a tree are called *nodes.*
  In the textbook a node has a *key* (used to identify item) and a *payload* (the actual data stored)
- Links between nodes are called *edges* (always pointing downwards)

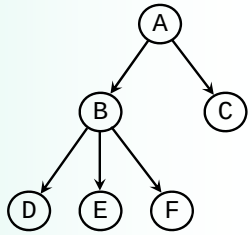


An empty tree is a tree without any nodes.

Terminology (cont'd)

Relationships

- A is a *parent* of B and B is a *child* of A, if an edge points from A to B
- A node can have only one parent
- If two nodes have the same parent they are *siblings*



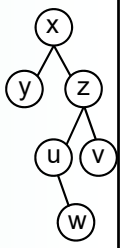
A *k*-node tree is a tree where every node has at most *k* children

Terminology (cont'd)

A node *a* is an *ancestor* of a node *n* if we can reach n by starting from *a* and going to its parent, its parent's parent, and so on. A node is regarded as its own ancestor unless we are talking about the *proper ancestors*.

A node *d* is a descendant of a node *n* if and only if *n* is an ancestor of *d*. A node is regarded as its own descendant unless we are talking about the *proper descendants*.

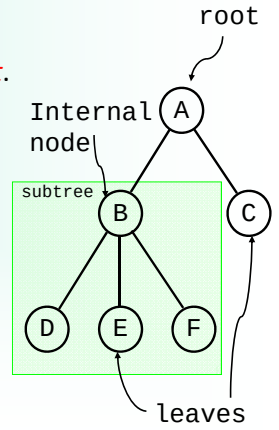
A *path* is a sequence of nodes n_1, n_2,..., n_k such that n_1 is the parent of n_2, n_2 is the parent of n_3, and so on. The length of a path with *k* nodes is *k*-1 (the number of edges forming the path).
A path with a single node (*k* = 1) has a length of 0.



Terminology (cont'd)

Names for different Tree nodes

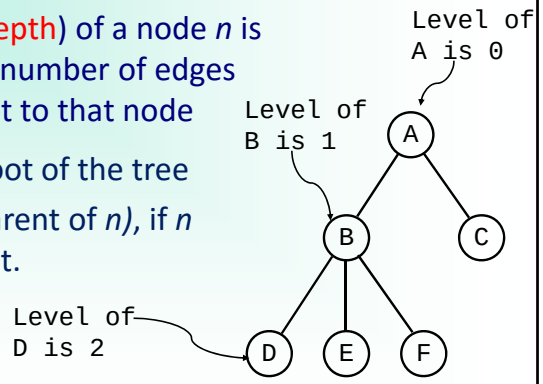
- A node without parent is called *root*.
- A node without child(ren) is called *leaf*.
- The non-leave node are also called *internal nodes*.
- A node and all of its descendants form a *subtree*.
  (i.e. The tree is recursive in nature: a tree is either empty or it is a node whose children are trees)



Terminology (cont'd)

The level (or depth) of a node *n* is equal to the number of edges from the root to that node

- 0 if *n* is the root of the tree
- 1+(level of parent of *n)*, if *n* is not the root.

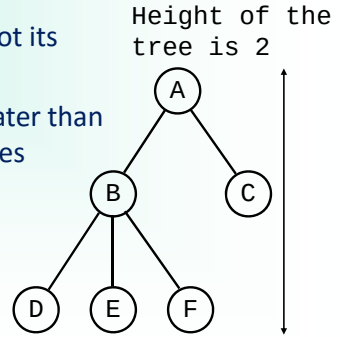


Terminology (cont'd)

The height of a Tree is equal to the maximum level of any node in the tree

- If a tree consist of only the root its height is 0
- Otherwise the height is 1 greater than the height of its tallest subtrees

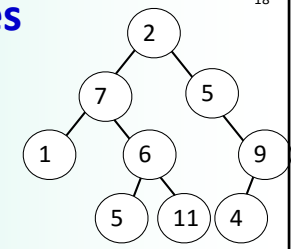
height(Tree) = 1 + max { height($subTree_1$), height($subTree_2$), … }



Fun with trees!! ☺

- Draw as many different trees with exactly four nodes as you can.
- What is the maximum height of a tree with *n* nodes?
  What is its minimum height?
- What tree has only leaf nodes and no internal nodes?
- What is the maximum height of a 2-node tree with 7 nodes? What is its minimum height?
- Construct an Arithmetic Tree for the expression (2+4+7)*(2+8).

Binary trees

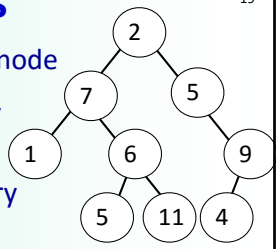


A binary tree is one where the maximum number of children from any node is two. The tree on the right is an example of a binary tree.

What do we want to do with a tree object?

- create a tree object
- insert nodes
- remove nodes
- traverse the tree

Binary trees



BinaryTree() – create a tree with one node

get_left_subtree() – return the binary tree which is the left child.

get_right_subtree() – return the binary tree which is the right child.

get_value() – return the object stored in the current node.

set_value(val) – store the object, val, in the current node.

insert_left(val) – create a new binary tree and insert it as the left child of the current node.

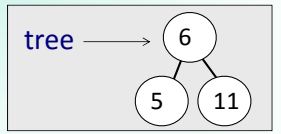
insert_right(val) – create a new binary tree and insert it as the right child of the current node.

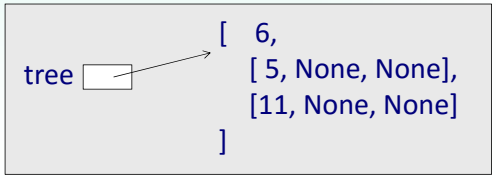
Binary trees – How to implement

Implementation 1: Use a list of lists to store the tree:

- the first element is the value in the node,
- the second element is the list representing the left subtree,
- the third element is the list representing the right subtree

For example,

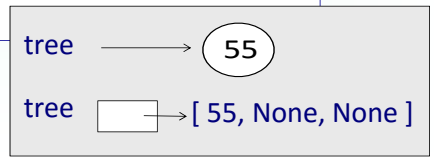




Binary trees – List implementation


```
class ListBinaryTree:
    def __init__(self, value):
        self.node = [value, None, None]
```

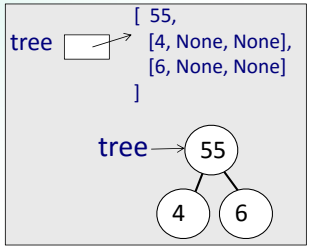
```
from ListBinaryTree import ListBinaryTree
def main():
    tree = ListBinaryTree(55)
main()
```



Binary trees – List implementation


```
class ListBinaryTree:
    def __init__(self, value):
        self.node = [data, None, None]

    def insert_left(self, value):
        new_node = ListBinaryTree(value,
                                  self.node[1], None)
        self.node[1] = new_node

    def insert_right(self, value):
        new_node = ListBinaryTree(value,
                                  None, self.node[2])
        self.node[2] = new_node
```

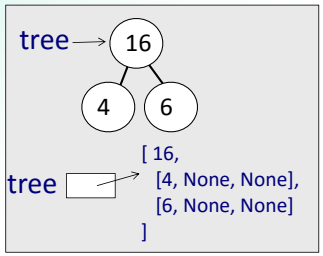
```
def main():
  tree = ListBinaryTree(55)
  tree.insert_left(4)
  tree.insert_right(6)

main()
```



Binary trees – List implementation


```
class ListBinaryTree:
    def __init__(self, value):
        self.node = [value,None, None]

    def insert_left(self, value):
        # see previous slide
    def insert_right(self, value):
        # see previous slide

    def set_value(self, new_value):
        self.node[0] = new_value

    def get_value(self):
        return self.node[0]
```

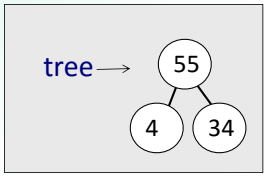
```
def main():
    tree = ListBinaryTree(55)
    tree.insert_left(4)
    tree.insert_right(6)
    tree.set_value(16)
main()
```



Binary trees – List implementation


```
class ListBinaryTree:
    def __init__(self, value):
        self.node = [value, None, None]

    # see previous slides for more

    def get_left_subtree(self):
        return self.node[1]

    def get_right_subtree(self):
        return self.node[2]

    def insert_tree_left(self, tree):
        self.node[1] = tree

    def insert_tree_right(self, tree):
        self.node[2] = tree
```

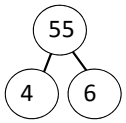
```
def main():
    tree = ListBinaryTree(55)
    tree.insert_left(4)
    right = ListBinaryTree(34)
    tree.insert_tree_right(right)

main()
```



Binary trees – __str__()


```
class ListBinaryTree:
    def __init__(self, value):
        self.node = [value,None, None]

    def insert_left(self, value):#
    def insert_right(self, value):#
    def set_value(self, new_value):#
    def get_value(self):#
    def get_left_subtree(self):#
    def get_right_subtree(self):#


    def __str__(self):
        n = self.node
        return "[" + n[0] + ", " + str(n[1]) + ", " + str(n[2]) + "]"
```

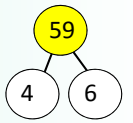
```
def main():
    tree = ListBinaryTree(55)
    tree.insert_left(4)
    tree.insert_right(6)
    print(tree)

main()
```

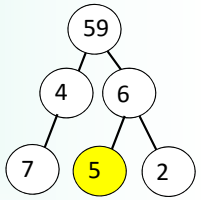



```
[55, [4, None, None],
     [6, None, None] ]
```

Show the tree and the output


```
from ListBinaryTree import ListBinaryTree
def main():
    tree = ListBinaryTree(55)
    tree.insert_left(4)
    tree.insert_right(6)
    tree.set_value(tree.get_value() + 4)
    print("1.", tree.get_value())
    right = tree.get_right_subtree()
    left = tree.get_left_subtree()
    left.insert_left(7)
    right.insert_right(2)
    right.insert_left(5)
    print("2.", tree.get_right_subtree().
                get_left_subtree().get_value())
    print(tree)
main()
```






```
[59, [4, [7, None, None], None],
     [6, [5, None, None], [2, None, None]]]
```

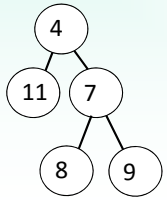
Draw the tree

The output when the following code (just the skeleton is shown here) is executed:

```
tree = ListBinaryTree(…)
…
print(tree)
```

is: `[4, [11, None, None], [7, [8, None, None], [9, None, None]]]`

Draw the tree:



ListBinaryTree – readability


```
class ListBinaryTree:        #many methods are missing
    DATA = 0    #constants forreadability
    LEFT = 1
    RIGHT = 2
    def set_value(self, new_value):
        self.node[self.DATA] = new_value
    def get_left_subtree(self):
        return self.node[self.LEFT]
    def get_right_subtree(self):
        return self.node[self.RIGHT]
    def __str__(self):
        return '['+str(self.node[self.DATA])+', ' + str(self.node[self.LEFT])\
                                    + ', ' + str(self.node[self.RIGHT]) + ']'
```

The constructor, __init__() is not shown here

Traversals of a Binary Tree

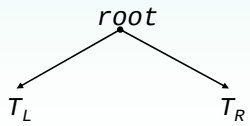
Visit each node in the tree

Recursive structure of a Binary Tree

- root, left subtree, right subtree

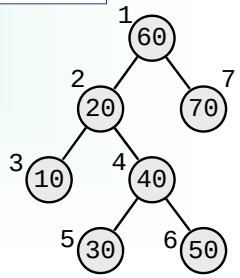
Order of traversal

- Pre-order traversal: root, traverse T_L , traverse T_R
- In-order traversal: traverse T_L , root, traverse T_R
- Post-order traversal: traverse T_L , traverse T_R, root



Pre-order Traversal of Binary Tree

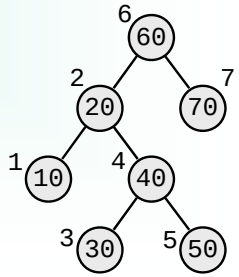

```
def preorder(tree):
    if(tree!=None):
        print(tree.get_value(), end=" ")
        preorder(tree.get_left_subtree())
        preorder(tree.get_right_subtree())
```



Preorder: 60, 20, 10, 40, 30, 50, 70

In-order Traversal of Binary Tree

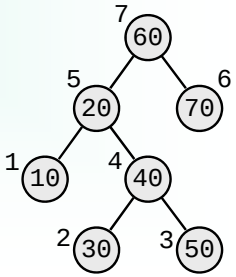

```
def inorder(tree):
    if(tree!=None):
        inorder(tree.get_left_subtree())
        print(tree.get_value(), end=" ")
        inorder(tree.get_right_subtree())
```



Inorder: 10, 20, 30, 40, 50, 60, 70

Post-order Traversal of Binary Tree


```
def postorder(tree):
    if(tree!=None):
        postorder(tree.get_left_subtree())
        postorder(tree.get_right_subtree())
        print(tree.get_value(), end=" ")
```

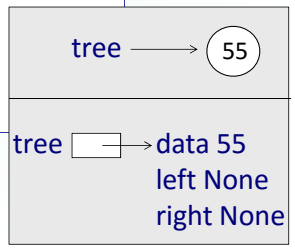


Postorder: 10, 30, 50, 40, 20, 70, 60

Binary trees – Reference implement.


```
class RefBinaryTree:
    def __init__(self, data):
        self.data = data
        self.left = None        # RefBinaryTree, None if empty
        self.right = None       # RefBinaryTree, None if empty
```

```
from RefBinaryTree import RefBinaryTree
def main():
    tree = RefBinaryTree(55)
main()
```



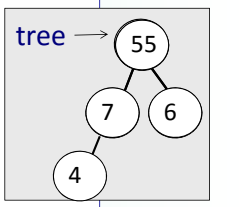
Binary trees – Reference implement.

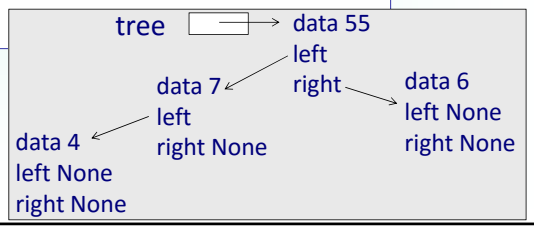

```
class RefBinaryTree:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    def insert_left(self, data):
        t = RefBinaryTree(data)
        if self.left == None:
            self.left = t
        else:
            t.left = self.left
            self.left = t
```

```
    def insert_right(self, data):
        t = RefBinaryTree(data)
        if self.right == None:
            self.right = t
        else:
            t.right = self.right
            self.right = t
```

Binary trees – Reference implement.


```
from RefBinaryTree import RefBinaryTree

def main():
    tree = RefBinaryTree(55)
    tree.insert_left(4)
    tree.insert_right(6)
    tree.insert_left(7)
main()
```



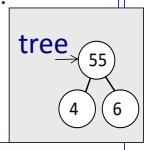


Binary trees – using nodes

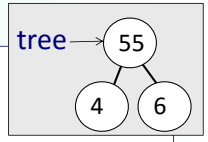

```
class RefBinaryTree:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

    def insert_left(self, data):#
    def insert_right(self, data):#

    def set_value(self, val):
        self.data = val

    def get_value(self):
        return self.data
```

```
    def get_left_subtree(self):
        return self.left

    def get_right_subtree(self):
        return self.right
```

```
def main():
    tree = RefBinaryTree(55)
    tree.insert_left(4)
    tree.insert_right(6)

    r = tree.get_right_subtree()
    print(r.get_value())

main()
```



Binary trees – printing nodes




```
def create_string(self, indent):
    s=str(self.data) + "---+"
    if self.left != None:
        s=s+"\n(l)"+indent+self.left.create_string(indent+"    ")
    if self.right != None:
        s=s+"\n(r)"+indent+self.right.create_string(indent+"    ")
    return s

def __str__(self):
    representation = self.create_string("  ")
    return representation
```

```
55---+
(l)    4---+
(r)    6---+
```

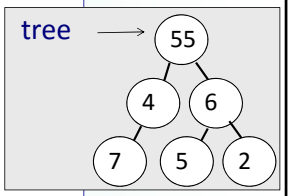
Resulting output

```
"55---+\n(l)    4---+\n(r)    6---+"
```

Returned string

Binary trees – printing nodes


```
from RefBinaryTree import RefBinaryTree
def main():
    tree = RefBinaryTree(55)
    tree.insert_left(4)
    tree.insert_right(6)
    right = tree.get_right_subtree()
    left = tree.get_left_subtree()
    left.insert_left(7)
    right.insert_right(2)
    right.insert_left(5)
    print(tree)
main()
```




```
55---+
(l)    4---+
(l)        7---+
(r)    6---+
(l)        5---+
(r)        2---+
```

Testing if an object is not None


```
def main():
    number = 5
    if number:
        print(1, "Number:", number)
    if 4:
        print(2, "Four")
    value = None
    if value:
        print(3, "True:", value)
    else:
        print(3, "Not True:", value)
main()
```

variable returns True if the object is not None.

```
1 Number: 5
2 Four
3 Not True: None
```

Checking if the child is not None


```
def create_string(self, indent):
    s=str(self.data) + "---+"
    if self.left:
        s=s+"\n(l)"+indent+self.left.create_string(indent+"    ")
    if self.right:
        s=s+"\n(r)"+indent+self.right.create_string(indent+"    ")
    return s
```

Exercise - Draw the tree structure

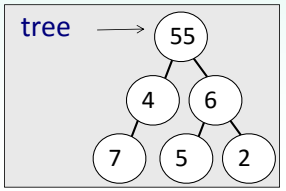

```
from RefBinaryTree import RefBinaryTree
def main():
    tree = RefBinaryTree(5)
    tree.insert_left(8)
    tree.insert_right(2)
    right = tree.get_right_subtree()
    left = tree.get_left_subtree()
    left.insert_left(7)
    value = left.get_value()
    left.set_value(value + 4)
    right.insert_right(3)
    right.insert_left(6)

main()
```

```
5---+
(l)   12---+
(l)        7---+
(r)   2---+
(l)        6---+
(r)        3---+
```

Binary trees – Exercise

TASK: Use the class RefBinaryTree to compute the sum of all values stored in the nodes of the tree. (You can assume that all nodes are real integer values)

```
def get_node_sum(self):
    sum = self.data
    if self.left:
        sum += self.left.get_node_sum()
    if self.right:
        sum += self.right.get_node_sum()
    return sum
```




```
def main():
    tree = RefBinaryTree(55)
    # code to construct tree
    # in picture on the left is omitted
    print(tree.get_node_sum())

main()
```

```
79
```

Named arguments


```
def print_result(num, amt, spaces=3,
                                extra=None):
    message = str(num) + "-" * spaces + str(amt)
    if extra:
        message += " (" + str(extra) + ")"
    print(message)

def main():
    print_result(1, 34)
    print_result(2, 34, 7)
    print_result(3, 34, extra = 6)
    print_result(4, 34, 7, 10)

main()
```

Python allows function arguments to have default values. If the function is called without the argument, the argument gets its default value.

Also, arguments can be specified in any order by using named arguments.

```
1---34
2-------34
3---34 (6)
4-------34 (10)
```

Binary trees – Reference Impl. (v2)


```
class RefBinaryTree:
    def __init__(self, value, l=None, r=None):
        self.data = value
        self.left = l
        self.right = r

    def insert_left(self, value):
        self.left = RefBinaryTree(value, l=self.left)

    def insert_right(self, value):
        self.right = RefBinaryTree(value, r=self.right)

    def set_value(self, val):
        self.data = val
```

Binary trees – Reference Impl. (v2)

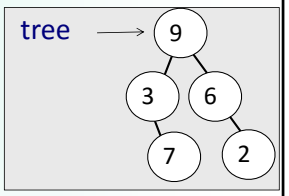

```
class RefBinaryTree:
    #continued from the previous slide

    def get_value(self):
        return self.data

    def get_left_subtree(self):
        return self.left

    def get_right_subtree(self):
        return self.right
```

Exercise


```
class RefBinaryTree:
    def __init__(self, value, l=None, r=None):
        self.data = value
        self.left = l
        self.right = r
    ...
```

```
from RefBinaryTree import RefBinaryTree

def main():
    tree =  ???




    print(tree)

main()
```

Using the RefBinaryTree class, write the code which creates the tree below.




```
9---+
(l)   3---+
(r)       7---+
(r)   6---+
(r)       2---+
```