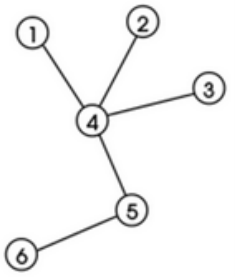
Trees

Trees are connected, undirected graphs with no cycles. Trees represent the simplest graphs in structure from the class of connected graphs, and they are also the most frequently used in practice.



img: http://en.wikipedia.org/wiki/File:Tree_graph.svg

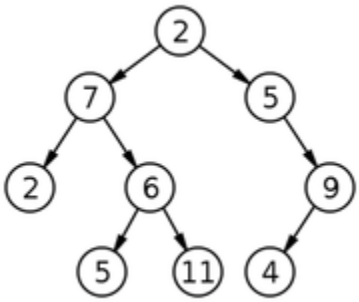
In a tree, nodes are connected by lines called edges or links. A tree with n nodes has n - 1 branches.

Root node

We usually identify a particular node called root and orient the edges in the same direction as the root. Any node other than the root has a single parent. A node can have several children. Nodes without children are called leaf nodes.

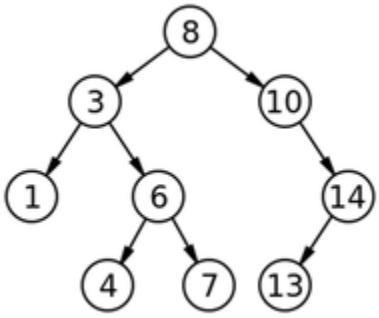
Binary trees

A binary tree is a tree where each node has at most two children, identified as the left child and the right child.



Binary search trees

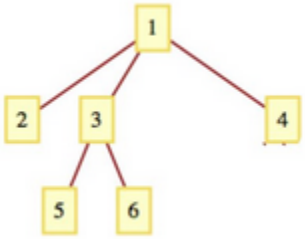
Binary search trees are binary trees that store values sorted in order. For each node: the left subtree has values smaller than the root, and the right subtree has values greater than the root.



Searching the binary search tree is efficient, done in just a few steps. The search is done recursively, by always comparing the sought-after element with the root of the current subtree: if they are equal, we found the element in the tree, if the sought-after element is smaller than the current root, the search continues in the left subtree, and if the sought-after element is greater than the current root, the search continues in the right subtree.

Representing trees in Python

1. Representation of a generic tree – example

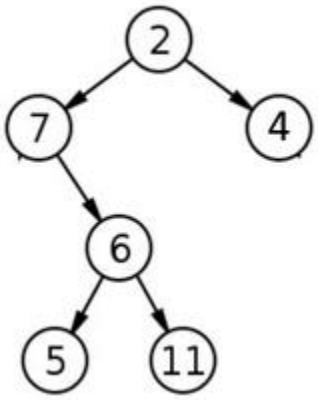



```
generic_tree = { "value": 1, " children":
     [
          { "value": 2, " children": []},
          { "value": 3, " children":
               [
                    { "value": 5, " children": []},
                    { "value": 6, " children": []}
               ]
          },
          { "value": 4, "children": []}
     ]
}
```

2. Representation of a binary tree

We can represent a binary tree recursively as a dictionary with 3 pairs: value, left tree and right tree.

```
tree = {"value": None, "left": None, "right": None}
```




```
binary_tree = { "value" : 2, "left":
        {
                "value": 7, "left": None, "right":
                {
                        "value": 6, "left":
                        {
                                "value": 5, "left ": None, "right": None
                        }, "right":
                        {
                                "value":11, "left": None, "right": None
                        },
                },
        }, "right":
                { "value": 4, "left": None, "right": None
        }
}
```

Traversing binary trees

1. Preorder traversal (root, left subtree, right subtree)

```
def rsd(tree):
     if (tree != None):
          return [tree["value"]] + rsd(tree["left"]) +
               rsd(tree["right"])
     else:
          return []
print(rsd(binary_tree))
```

2. Inorder traversal (left subtree, root, right subtree)

```
def srd(tree):
     if (tree != None):
          return srd(tree["left"]) + [tree["value"]] +
               srd(tree["right"])
     else:
          return []

print(srd(binary_tree))
```

3. Traversal in postorder (left subtree, right subtree, root)

```
def sdr(tree):
     if (tree != None):
          return sdr(tree["left"]) + sdr(tree["right"]) +
               [tree["value"]]
     else:
          return []

print(sdr(binary_tree))
```

Adding a new node to a parent and position (left or right):

```
def add_node_position(parent, new_node, position):
     if (parent[position] == None):
          parent[position] = new_node
     return parent

binary_tree["left"]=add_node_position(binary_tree["left"],
     {"value": 100, "left": None, "right": None}, "left")

print(rsd(binary_tree))
```

Adding a new node to the binary search tree:

```
def add_node(tree, new_node):
     if (tree == None):
          return new_node
     if (new_node["value"] < tree["value"]):
          tree["left"] = add_node(tree["left"] , new_node)
     else:
          tree["right"] = add_node(tree["right"], new_node)
     return tree

print(rsd(add_node(binary_tree,{"value": 1, "left": None,
"right": None})))
```

Deleting a node (or subtree) from a given parent given as a parameter:

```
def delete_node(parent, node_value):
     if (parent["left"]["value"] == node_value):
          parent["left"] = None
     elif(parent["right"]["value"] == node_value ):
          parent["right"] = None

delete_node(binary_tree["left"], 100)
delete_node(binary_tree, 5)

print(rsd(binary_tree))
```