

# UE5 Fundamentals of Algorithms

## Lecture 10: Trees

Ecole Centrale de Lyon, Bachelor of Science in Data Science for Responsible Business

Romain Vuillemot



```
In [5]: import sys
import os
from graphviz import Digraph
from IPython.display import display
from utils import draw_tree_dict
```

# Outline

- Definitions
- Data structures
- Weighted trees

# Trees

*Tree is a hierarchical data structure with nodes connected by edges*

- A non-linear data structures (multiple ways to traverse it)
- Nodes are connected by only one path (a series of edges) so trees have no cycle
- Edges are also called links, they can be traversed in both ways (no orientation)

Example of trees:

- Binary trees, binary search trees, N-ary trees, recursive call trees, etc.
- HOB (Horizontally Ordered Binary), AVL (Adelson-Velskii and Landis, self-balancing trees), ...
- B-trees, forests, lattices, etc.

# Definitions on trees

(similar to the ones for the binary trees)

**Nodes** - a tree is composed of nodes that contain a **value** and **children**.

**Edges** - are the connections between nodes; nodes may contain a value.

**Root** - the topmost node in a tree; there can only be one root.

**Parent and child** - each node has a single parent and up to two children.

**Leaf** - no node below that node.

**Depth** - the number of edges on the path from the root to that node.

**Height** - maximum depth in a tree.

## Definitions on trees (cont.)

**N-ary Tree** - a tree in which each node can have up to  $N$  children. Binary trees is the case where  $N = 2$ .

**Weight** - a quantity is associated to the edges.

**Degree** - the number of child nodes it has. Binary tree is the case where degree is 2.

**Subtree** - a portion of a tree that is itself a tree.

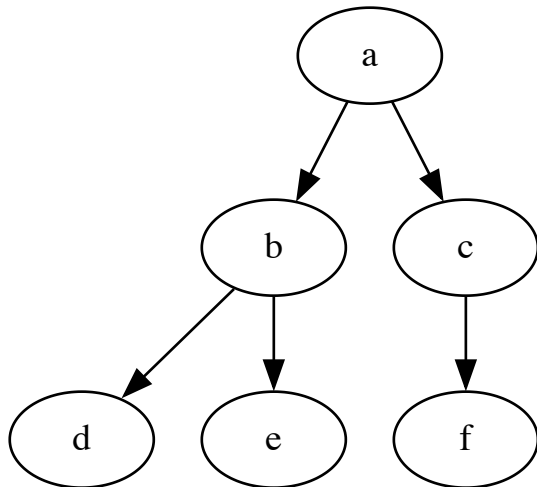
**Forest** - a collection of trees not connected to each other.

# Data structures (dicts + lists)

A simple way is the adjacency list using a dictionary `dict` type.

```
In [26]: tree_dict = {  
    "a": ["b", "c"],  
    "b": ["d", "e"],  
    "c": ["f"],  
    "d": [],  
    "e": [],  
    "f": []  
}
```

```
In [27]: draw_tree_dict(tree_dict)
```



# Data structures (dicts + named lists)

- A variation is to use a named variable for the list.

```
In [28]: tree_dict_name = {  
    "a": {"neighbors": ["b", "c"]},  
    "b": {"neighbors": ["d", "e"]},  
    "c": {"neighbors": ["f"]},  
    "d": {"neighbors": []},  
    "e": {"neighbors": []},  
    "f": {"neighbors": []}  
}
```

```
In [29]: tree_dict_name["a"]["neighbors"]
```

```
Out[29]: ['b', 'c']
```



# Data structures (sets)

- The children are unique and not ordered

```
In [30]: tree_set = {  
    "a": set(["b", "c"]),  
    "b": set(["d", "e"]),  
    "c": set(["f"]),  
    "d": set(),  
    "e": set(),  
    "f": set()  
}
```

# Data structures (lists of lists)

- Each node is an entry in the list
- Children are sub-lists

```
In [31]: tree_list = [  
    ['a', ['b', 'c']],  
    ['b', ['d', 'e']],  
    ['c', ['f', 'g']],  
    ['d', []],  
    ['e', []],  
    ['f', []],  
    ['g', []]  
]
```

# Data structures (tuples)

- Each node is the first tuple
- Children are additional tuple entries
- Warning: tuples are immutable (cannot be changed)

```
In [32]: tree_tuple = ("a", [  
    ("b", []),  
    ("c", [  
        ("d", [  
            ("e", [])  
        ])  
    ])  
])
```

```
In [33]: tree_tuple[0] # cannot be changed
```

```
Out[33]: 'a'
```

# Data structure (class object)

How to create the tree? How to retrieve all nodes? Both iterative and recursive ways.

```
class Node:
    def __init__(self, value, children = []):
        self.value = value
        self.children = children
```

# Data structure (class object)

How to create the tree? How to retrieve all nodes? Both iterative and recursive ways.

```
class Node:
    def __init__(self, value, children = []):
        self.value = value
        self.children = children
```

In [34]:

```
class Node:
    def __init__(self, value, children = []):
        self.value = value
        self.children = children

    def get_all_nodes(self):
        nodes = [self.value]
        for child in self.children:
            nodes += child.get_all_nodes()
        return nodes

    def get_all_nodes_iterative(self):
        nodes = []
        stack = [self]
        while stack:
            current_node = stack.pop()
            nodes.append(current_node.value)
            stack += current_node.children
        return nodes
```

```
In [35]: root = Node("a", [
            Node("b", [
                Node("d"),
                Node("e"),
            ]),
            Node("c", [
                Node("f"),
            ]),
        ])

# or using "root.children"
```

```
In [36]: root.get_all_nodes()
```

```
Out[36]: ['a', 'b', 'd', 'e', 'c', 'f']
```

```
In [37]: root.get_all_nodes_iterative()
```

```
Out[37]: ['a', 'c', 'f', 'b', 'e', 'd']
```

# Weighted trees

*Trees with a quantity associated to the links or the nodes*

- Useful to quantifie both nodes and links
- Storing those values require additionnal data structures

# Data structures for weighted trees (dicts for edges)

- We need to add an extra value to encode values in edges

```
In [38]: tree_w_dict = {'a': [{'b': 0}, {'c': 0}],  
                        'b': [{'d': 0}, {'e': 0}],  
                        'c': [{'f': 0}],  
                        'd': [],  
                        'e': []  
                        }
```

```
In [39]: tree_w_tuple = {  
    'a': [('b', 0), ('c', 0)],  
    'b': [('d', 0), ('e', 0)],  
    'c': [('f', 0)],  
    'd': [],  
    'e': []  
}
```



# Weighted trees as classes

```
In [40]: class Node_weight:
          def __init__(self, data, weight=0):
              self.data = data
              self.children = []
              self.weight = weight

          tree = Node_weight(1)
          child1 = Node_weight(2, weight=5)
          child2 = Node_weight(3, weight=7)
          tree.children = [child1, child2]
```

# Exercise: Calculate the total weight of a tree

*Tip: go through all the nodes and get the edges, then sum their weights.*



# Exercise: Calculate the total weight of a tree

*Tip: go through all the nodes and get the edges, then sum their weights.*

```
In [41]: def get_tree_edges(root):  
    edges = []  
    stack = [(root, None)]  
  
    while stack:  
        node, parent_data = stack.pop()  
  
        for child in node.children:  
            stack.append((child, node.data))  
            edges.append((node.data, child.data, child.weight))  
  
    return edges
```

```
In [42]: tree_w_oo = Node_weight(1)  
child1 = Node_weight(2, weight=5)  
child2 = Node_weight(3, weight=7)  
tree_w_oo.children = [child1, child2]  
get_tree_edges(tree_w_oo)
```

```
Out[42]: [(1, 2, 5), (1, 3, 7)]
```

```
In [43]: sum(tpl[2] for tpl in get_tree_edges(tree_w_oo))
```

Out[43]: 12

# Exercise: Calculate the total weight of a tree

A recursive version:

```
In [44]: def calculate_total_weight(node):  
         total_weight = node.weight  
         for child in node.children:  
             total_weight += calculate_total_weight(child)  
         return total_weight
```

```
In [45]: calculate_total_weight(tree_w_oo)
```

```
Out[45]: 12
```

# An Edge class for edges

- To consider edges as objects
- Can be used as a complement of the nodes (or without the nodes)

```
In [46]: class Edge:
          def __init__(self, source, target):
              self.source = source
              self.target = target

          class Node:
              def __init__(self, label):
                  self.label = label
                  self.children = []

          class Tree:
              def __init__(self, root_label):
                  self.root = Node(root_label)
                  self.edges = []
```

## Main trees properties

- Hierarchical structure
- No cycle
- All nodes connected

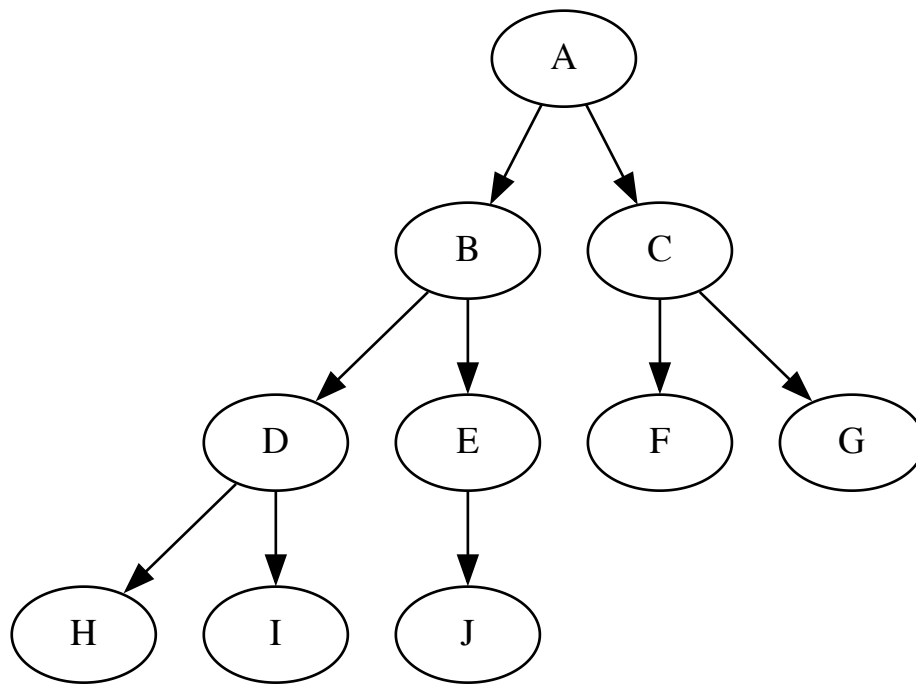
We will mostly use one of the two traversal methods (BFS and DFS) to achieve this.

Also we will using the dictionary-based data structure:

```
In [6]: tree = {  
    "A": ["B", "C"],  
    "B": ["D", "E"],  
    "C": ["F", "G"],  
    "D": ["H", "I"],  
    "E": ["J"],  
    "F": [],  
    "G": [],  
    "H": [],  
    "I": [],  
    "J": []  
}
```

```
In [7]: draw_tree_dict(tree)
```





# Generalized BFS (Breadth-First Search)

```
In [9]: def bfs(tree, start_node):
        queue = [start_node]
        result = []

        while queue:
            node = queue.pop(0)
            result.append(node)
            children = tree.get(node, [])

            for child in children:
                if child is not None:
                    queue.append(child)

        return result
```

```
In [10]: print(bfs(tree, "A"))

['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J']
```

# Generalized DFS (Depth-First Search)

```
In [ ]: def dfs(tree, start_node):  
        stack = [start_node]  
        result = []  
  
        while stack:  
            node = stack.pop()  
            result.append(node)  
            children = tree.get(node, [])  
  
            for child in children:  
                if child is not None:  
                    stack.append(child)  
  
        return result
```

```
In [ ]: print(dfs(tree, "A"))
```

## Tree property: are all nodes connected?

Without having a first node and re-using the dfs

```
In [ ]: def is_tree_connected(tree, start_node):  
        if not tree:  
            return True # An empty tree is considered connected.  
  
        visited = set()  
        stack = []
```

```

stack.append(start_node)

while stack:
    node = stack.pop()
    if node not in visited:
        visited.add(node)
        stack.extend(tree.get(node, []))

return len(visited) == len(tree)

is_tree_connected(tree, "A")

```

```
In [ ]: dfs_check_connected(tree, "A")
```

## Tree property: does the tree have a cycle?

```
In [ ]: def has_cycle_dfs(root):
def dfs(node, parent, visited):
    if node in visited:
        if parent is not None and parent != visited[node]:
            return True
        return False

    visited[node] = parent

    for child in node.children:
        if dfs(child, node, visited):
```

```
return True
```

```
return False
```

```
visited = {}  
return dfs(root, None, visited)
```

What if we add an extra node "K"?

```
tree["F"] = ["A"]
```

Tree property: Check if the tree is an n-ary tree

```
In [ ]: def is_binary_tree(tree, node, n = 2, visited=None):  
        if visited is None:  
            visited = set()  
  
        if node in visited:  
            return True  
  
        visited.add(node)  
        children = tree.get(node, [])  
  
        if len(children) > n:
```

```

        return False

    for child in children:
        if not is_binary_tree(tree, child, n, visited):
            return False

    return True

is_binary_tree(tree, "A", 2)

```

## Get all the edges of a tree

```

In [11]: def generate_edges(graph):
        edges = []
        for node, neighbors in graph.items():
            for neighbor in neighbors:
                edges.append((node, neighbor))
        return edges

```

```

In [12]: generate_edges(tree)

```

```

Out[12]: [('A', 'B'),
          ('A', 'C'),
          ('B', 'D'),
          ('B', 'E'),
          ('C', 'F'),
          ('C', 'G'),
          ('D', 'H'),

```

```
( 'D', 'I'),  
( 'E', 'J')]
```

```
In [13]: def generate_edges_dfs(graph, start_node):  
    edges = []  
    stack = [start_node]  
    visited = []  
  
    while stack:  
        node = stack.pop()  
        visited.append(node)  
        for neighbor in graph[node]:  
            if neighbor not in visited:  
                edges.append((node, neighbor))  
                stack.append(neighbor)  
  
    return edges
```

```
In [15]: generate_edges_dfs(tree, "A")
```

```
Out[15]: {('A', 'B'),  
          ('A', 'C'),  
          ('B', 'D'),  
          ('B', 'E'),  
          ('C', 'F'),  
          ('C', 'G'),  
          ('D', 'H'),  
          ('D', 'I'),  
          ('E', 'J')}
```

