

INF TC1 - Lecture 3: Binary Trees

Ecole Centrale de Lyon

romain.vuillemot@ec-lyon.fr

Outline

- Definitions
- Data structures
- Basic operations
- Properties

Definitions

A **Tree** is a hierarchical data structure with nodes (vertex) connected by links (edge)

- 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)
- Trees are most commonly represented as a node-lin diagram, with the root at the top and the leaves (nodes without children) at the bottom).

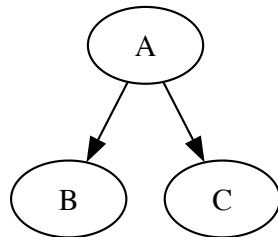
Binary trees

We focus on *binary trees*.

Trees that have at most two children

- Children are ordered (left and right)

```
In [5]: T = {  
        'A': ['B', 'C'],  
        }  
  
draw_directed_graph(T)
```



Binary trees data structures

Binary trees can be stored in multiple ways

- The first element is the value of the node.
- The second element is the left subtree.
- The third element is the right subtree.

Here are examples:

- Adjacency list `T = {'A': ['B', 'C']}`
- Arrays `["A", "B"]`
- Class / Object-oriented programming `Class Node()`

Other are possible: using linked list, modules, etc.

Adjacency lists are the most common ways and can be achieved in multiple fashions.

Binary trees data structures (dict + lists)

Binary trees using dictionnaires where nodes are keys and edges are Lists.

```
In [6]: T = {  
    'A' : ['B', 'C'],  
    'B' : ['D', 'E'],  
    'C' : [],  
    'D' : [],  
    'E' : []  
}
```

Using OOP

```
In [7]: class Node:
        def __init__(self, value):
            self.value = value
            self.left = None
            self.right = None

        def get_value(self):
            return self.value

        def set_value(self, v = None):
            self.value = v
```

```
In [8]: root = Node(4)
        root.left = Node(2)
        root.right = Node(5)
        root.left.left = Node(1)
        root.left.right = Node(3)
```

Definitions on 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.

Basic operations

Get the root of a tree

Return the topmost node in a tree (there can only be one root). Assuming the tree is stored in an ordered dict().

Get the root of a tree

Return the topmost node in a tree (there can only be one root). Assuming the tree is stored in an ordered dict().

```
In [9]: def get_root(T):  
        if (len(T.keys()) > 0):  
            return list(T.keys())[0]  
        else:  
            return -1
```

```
In [10]: get_root(T)
```

```
Out[10]: 'A'
```

```
In [11]: assert get_root({}) == -1  
         assert get_root({"A": []}) == "A"  
         assert isinstance(get_root({"A": []}), str) # to make sure there is onl
```

Get all nodes from a Tree

Return all the nodes in the tree (as a list of node names).

Get all nodes from a Tree

Return all the nodes in the tree (as a list of node names).

```
In [15]: def get_nodes(T):  
         return list(T.keys())
```

```
In [16]: get_nodes(T)
```

```
Out[16]: ['A', 'B', 'C', 'D', 'E']
```

```
In [17]: assert get_nodes(T) == ['A', 'B', 'C', 'D', 'E']  
         assert get_nodes({}) == []
```

Get all links from a Tree

Return all the links as a list of pairs as `Tuple`.

Get all links from a Tree

Return all the links as a list of pairs as `Tuple`.

```
In [18]: def get_links(tree):  
         links = []  
         for node, neighbors in tree.items():  
             for neighbor in neighbors:  
                 links.append((node, neighbor))  
         return links
```

```
In [19]: get_links(T)
```

```
Out[19]: [('A', 'B'), ('A', 'C'), ('B', 'D'), ('B', 'E')]
```

```
In [20]: assert get_links(T) == [('A', 'B'), ('A', 'C'), ('B', 'D'), ('B', 'E')]  
         assert get_links({}) == []
```

Get the parent of a node

Return the parent node of a given node (and `None` if the root).

Get the parent of a node

Return the parent node of a given node (and `None` if the root).

```
In [17]: def get_parent(tree, node_to_find):  
         for parent, neighbors in tree.items():  
             if node_to_find in neighbors:  
                 return parent  
         return None
```

```
In [18]: assert get_parent(T, 'D') == 'B'  
         assert get_parent(T, 'A') is None  
         assert get_parent({}, '') is None
```

Check if the node is the root

Return True if the root not, else `None`.

Check if the node is the root

Return True if the root not, else `None`.

```
In [19]: def is_root(T, node):  
         return get_parent(T, node) is None
```

```
In [20]: assert is_root(T, 'A') == True
```

Get the children of a node

Given a node, return all its children as a `List`.

Get the children of a node

Given a node, return all its children as a `List`.

```
In [21]: def find_children(graph, parent_node):  
         children = graph.get(parent_node, [])  
         return children
```

```
In [22]: assert find_children(T, 'A') == ['B', 'C']  
         assert find_children(T, 'B') == ['D', 'E']  
         assert find_children(T, 'C') == []
```

Get the children of a node

Given a node, return all its children as a `List`.

```
In [21]: def find_children(graph, parent_node):  
        children = graph.get(parent_node, [])  
        return children
```

```
In [22]: assert find_children(T, 'A') == ['B', 'C']  
        assert find_children(T, 'B') == ['D', 'E']  
        assert find_children(T, 'C') == []
```

Check if the node is a leaf

Return `True` if the node has no children.

Get the children of a node

Given a node, return all its children as a `List`.

```
In [21]: def find_children(graph, parent_node):  
        children = graph.get(parent_node, [])  
        return children
```

```
In [22]: assert find_children(T, 'A') == ['B', 'C']  
        assert find_children(T, 'B') == ['D', 'E']  
        assert find_children(T, 'C') == []
```

Check if the node is a leaf

Return `True` if the node has no children.

```
In [23]: def is_leaf(T, node):  
        return len(find_children(T, node)) == 0
```

```
In [24]: assert is_leaf(T, 'C')  
        assert not is_leaf(T, 'A')
```

Add/Delete a node

Given a tree as input.

- Add a node to given a current partent
- Remove a given node

Add/Delete a node

Given a tree as input.

- Add a node to given a current parent
- Remove a given node

```
In [25]: def add_node(graph, parent, new_node):  
        if parent in graph:  
            graph[parent].append(new_node)  
        else:  
            graph[parent] = [new_node]  
  
        def delete_node(graph, node_to_delete):  
            for parent, children in graph.items():  
                if node_to_delete in children:  
                    children.remove(node_to_delete)  
                    if not children:  
                        del graph[parent]
```

```
In [26]: U = {"A": []}  
        add_node(U, "A", 'F')  
        U
```

```
Out[26]: {'A': ['F']}
```

Height of a tree

Calculate the longest path from the root to leaves. Tip: use a recursive approach

- if the node is a leaf, return 1
- for a current node, the height is the max height of its children + 1

Height of a tree

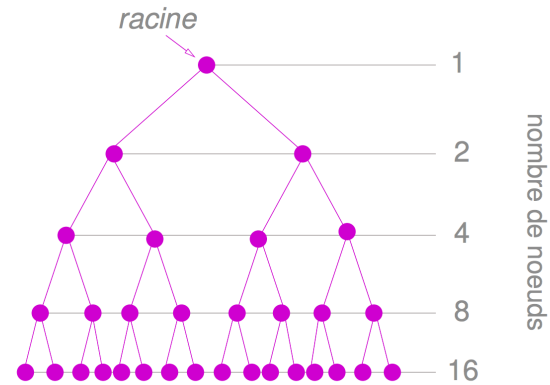
Calculate the longest path from the root to leaves. Tip: use a recursive approach

- if the node is a leaf, return 1
- for a current node, the height is the max height of its children + 1

```
In [38]: def tree_height(tree, node):  
         if node not in tree or not tree[node]:  
             return 1  
         left_height = tree_height(tree, tree[node][0]) if tree[node][0] else 0  
         right_height = tree_height(tree, tree[node][1]) if tree[node][1] else 0  
         return 1 + max(left_height, right_height)
```

```
In [39]: assert tree_height(T, 'A') == 3  
         assert tree_height(T, 'B') == 2  
         assert tree_height(T, 'C') == 1
```

Height of a binary tree



$$n = 2^{(h+1)} - 1$$

$$n + 1 = 2^{(h+1)}$$

$$\log(n + 1) = \log(2^{(h+1)})$$

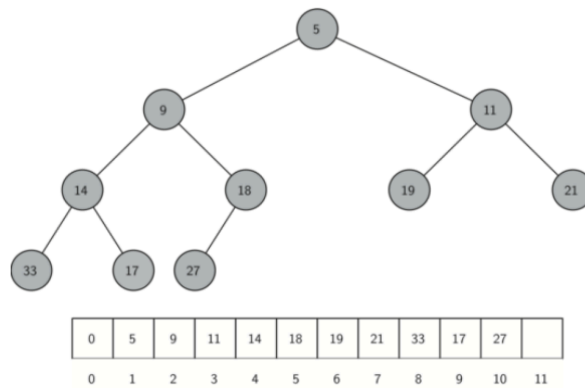
$$\log(n + 1) = (h + 1)\log(2)$$

$$\log(n + 1)/\log(2) = h + 1$$

$$\text{so } h = \log(n + 1)/\log(2) - 1$$

h is equivalent to $\log(n)$

Binary trees (using Arrays)



In a complete or balanced binary tree:

- if the index of a node is equal to i , then the position indicating its left child is at $2i$,
- and the position indicating its right child is at $2i + 1$.

Also works for ternary trees, etc.

Visualize a tree

```
In [37]: from graphviz import Digraph

dot = Digraph()

dot.node_attr['shape'] = 'circle'

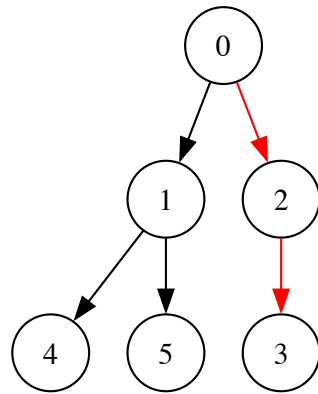
dot.node('0', label='0') # Root
dot.node('1')
dot.node('2')
dot.node('3')
dot.node('4')
dot.node('5')

dot.edge('0', '1')
dot.edge('1', '4')
dot.edge('1', '5')

dot.edge('0', '2', color='red')
dot.edge('2', '3', color='red')

dot # render
```

Out [37]:



In []:

