Definitions

> *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)

We focus on *binary trees*.

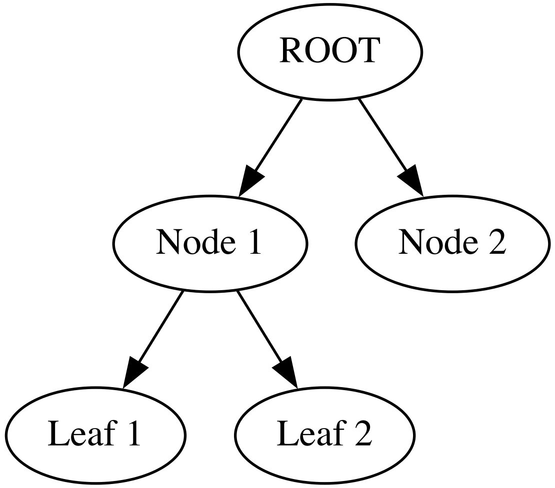
> *Trees that have at most two children*

- Children can be ordered left child and the right child

Binary trees representation

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).

In [278]:
```
draw_binary_tree(binary_tree)
```



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 (dictionnaries and lists)

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

In [199]:

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

Using OOP

In [200]:

```
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 [201]:

```
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).

Get the root of a tree

Return the topmost node in a tree (there can only be one root).

In [254]:
```
def get_root(T):
    if (len(T.keys()) > 0):
        return list(T.keys())[0]
    else:
        return -1
```

In [255]:
```
get_root(T)
```

Out[255]:
```
'A'
```

In [256]:
```
assert get_root({}) == -1
assert get_root({"A": []}) == "A"
assert isinstance(get_root({"A": []}), str) # to make sure there is onl
```

Get the list of nodes

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

Get the list of nodes

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

In [205]:
```
def get_nodes(T):
    return list(T.keys())
```

In [206]:
```
get_nodes(T)
```

Out[206]:
```
['A', 'B', 'C', 'D', 'E']
```

In [208]:
```
assert get_nodes(T) == ['A', 'B', 'C', 'D', 'E']
assert get_nodes({}) == []
```

Get the list of edges

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

Get the list of edges

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

In [209]:
```python
def get_edges(graph):
    edges = []
    for node, neighbors in graph.items():
        for neighbor in neighbors:
            edges.append((node, neighbor))
    return edges
```

In [210]:
```python
get_edges(T)
```

Out[210]:
```
[('A', 'B'), ('A', 'C'), ('B', 'D'), ('B', 'E')]
```

In [211]:
```python
assert get_edges(T) == [('A', 'B'), ('A', 'C'), ('B', 'D'), ('B', 'E')]
assert get_edges({}) == []
```

Get the parent of a node

Return the parent node of a given node (and -1 if the root).

Get the parent of a node

Return the parent node of a given node (and -1 if the root).

In [217]:

```
def get_parent(graph, node_to_find):
    for parent, neighbors in graph.items():
        if node_to_find in neighbors:
            return parent
    return None
```

In [218]:

```
assert get_parent(T, 'D') == 'B'
assert get_parent(T, 'A') is None
assert get_parent({}, '') is None
```

Get the parent of a node

Return the parent node of a given node (and -1 if the root).

In [217]:
```
def get_parent(graph, node_to_find):
    for parent, neighbors in graph.items():
        if node_to_find in neighbors:
            return parent
    return None
```

In [218]:
```
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`.

Get the parent of a node

Return the parent node of a given node (and -1 if the root).

In [217]:
```
def get_parent(graph, node_to_find):
    for parent, neighbors in graph.items():
        if node_to_find in neighbors:
            return parent
    return None
```

In [218]:
```
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`.

In [226]:
```
def is_root(T, node):
    return find_parent(T, node) is None
```

In [227]:
```
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 [228]:

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

In [229]:

```
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 [228]:

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

In [229]:

```
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 [228]:
```
def find_children(graph, parent_node):
    children = graph.get(parent_node, [])
    return children
```

In [229]:
```
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 [233]:
```
def is_leaf(T, node):
    return len(find_children(T, node)) == 0
```

In [234]:
```
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 partent
- Remove a given node

In [267]:
```
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 [268]:
```
U = {"A": []}
add_node(U, "A", 'F')
U
```

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 [295]:
```
T
```

Out[295]:
```
{'A': ['B', 'C'], 'B': ['D', 'E'], 'C': [], 'D': [], 'E': []}
```

In [ ]:
```
v
```

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 [295]:
```
T
```

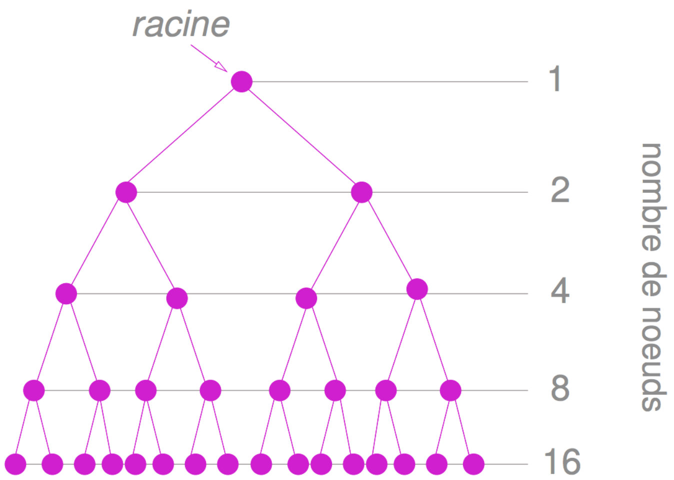
Out[295]:
```
{'A': ['B', 'C'], 'B': ['D', 'E'], 'C': [], 'D': [], 'E': []}
```

In [ ]:
```
v
```

In [291]:
```
def height(T, node):
    if node not in T:
        return 0  # leaf
    children = T[node]
    if not children:
        return 1  # leaf
    list_heights = []
    for child in children:
        list_heights.append(height(T, child))
    return 1 + max(list_heights)
```

In [292]:
```
assert height(T, 'A') == 3
assert height(T, 'B') == 2
assert height(T, 'C') == 1
```

Height of a binary tree



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

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

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

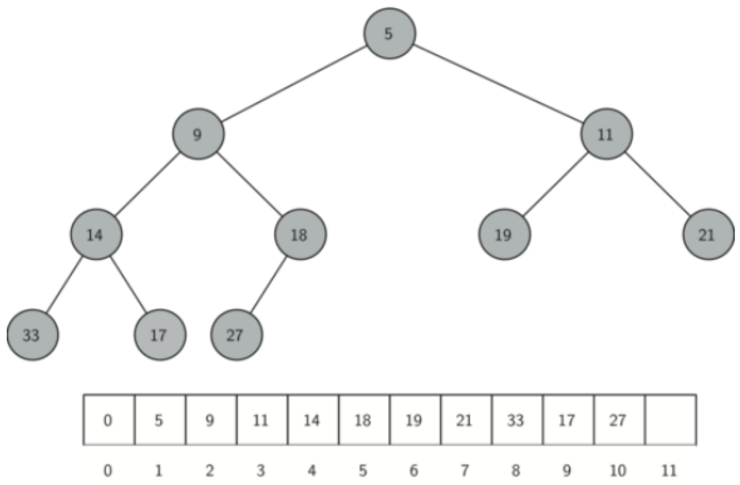
$log(n + 1) = (h + 1)log(2)$

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

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:

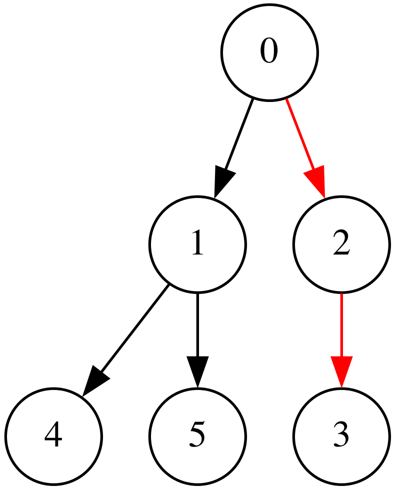
- if the index of a node is equal to i, then the position indicating its left child is at $2i$,
- and the position indicating its right child is at $2i + 1$.

Visualize a tree

In [275]:

```
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 the graph
```

Out[275]:



Visualize a tree

In [277]:

```
from graphviz import Digraph
from IPython.display import display

def draw_binary_tree(tree_dict):
    # Create a new graph
    dot = Digraph(format='png')

    # Recursive function to add nodes and edges
    def add_nodes_and_edges(node, parent_name=None):
        if isinstance(node, dict):
            for key, value in node.items():
                # Add the node
                dot.node(key, key)
                # Add the edge to the parent (if it exists)
                if parent_name:
                    dot.edge(parent_name, key)
                # Recursively call the function for the children
                add_nodes_and_edges(value, key)

    # Call the function to build the tree
    add_nodes_and_edges(tree_dict)

    # Display the graph in the notebook
    display(dot)
```

```
In [ ]:
```