:::{list-table} :header-rows: 1 :align: center :widths: 90 10

- ▪ Fondamenti di programmazione

  Canale A-L - Prof. Sterbini

  AA 25-26 - Lezione 19

  ▪ :::{image}attachment:5a837f06-202c-4c6e-9259-dd7a6dc01c97.png :width: 200px ::: :::

RECAP

- i valori di default mutevoli creano effetti collaterali
- Merge Sort
- Alberi binari
- esercizio di scansione di directory

In [3…

```
# utility per tracciare le chiamate ricorsive
from rtrace import traced
import os
# utility che mi permette di aggiungere con %%add_to dei metodi definiti in celle separate alla classe
import jdc
# libreria per generare i grafi in Jupyter
from pygraphviz import AGraph
```

Alberi N-ari (con numero indefinito di figli)

Attributi

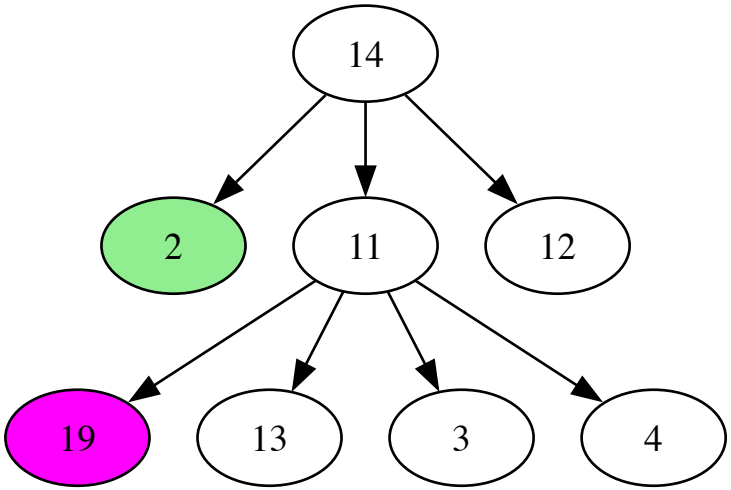
- **value**: valore del nodo

- **sons**: elenco di figli

In [3…
```
G3 = AGraph(directed=True)
G3.add_node(2,style='filled',fillcolor='lightgreen')
G3.add_node(19,style='filled',fillcolor='magenta')
G3.add_edges_from([ [14, 11], [14, 12], [14, 2], [11, 19], [11, 13], [11, 3], [11, 4] ])
G3.layout('dot')
```

In [3…
```
G3
```

Out[3…



In [3…
```
class NodoNario :
    def __init__(self,   V : int,
                         # ATTENTI AL DEFAULT!!! NON USARE []
                          sons : list['NodoNario'] = None):
        self._value = V
        if sons is None:
            self._sons = []      # la lista è creata a RUN-TIME
        else:
            self._sons = sons

    def __repr__(self):
        "stringa usata da pygraphviz"
        return f"{self._value!r}"
    def __str__(self):
        "stringa usata nelle stampe"
```

```
        return f"NodoNario({self._value!r}, {len(self._sons)} sons)"
```

In [3…

```
%%add_to NodoNario

from pygraphviz import AGraph

# metodi che aggiungono all'albero binario come disegnarsi usando la libreria pygraphviz

def _dot(self, rank=0):
    "creo l'elenco di nodi ed archi che Graphviz sa visualizzare"
    nodi = [(self,rank)]           # sicuramente ci va il nodo e la sua altezza
    archi = []
    for son in self._sons:
        archi.append((self, son))  # aggiungo l'arco verso il figlio
        N,A = son._dot(rank+1)     # calcolo i nodi ed archi del sottoalbero del figlio
        nodi  += N                 # e li aggiungo
        archi += A                 # a quelli correnti
    return nodi, archi          # alla fine ritorno l'elenco di (nodo,rank) e quello degli archi

def show(self):
    "creo l'oggetto Digraph per visualizzare il grafo diretto"
    G = AGraph(rankdir='LR', directed=True)          # faccio un grafo orizzontale
    nodi, archi = self._dot()         # calcolo nodi ed archi
    for nodo, rank in nodi:           # aggiungo i nodi
        G.add_node(nodo, rank=rank, label=nodo.__repr__() )   # ciascuno al suo livello
    G.add_edges_from(archi)           # aggiungo anche gli archi che li collegano
    G.layout('dot')                   # calcolo la visualizzazione
    return G                          # torno il grafo, che Jupyter sa mostrare
```

stavolta scriviamo le funzioni come metodi della classe NodoNario

altezza del nodo nell'albero (quanti livelli ha sotto)

- esploro l'albero ricorsivamente
- l'altezza di un nodo è 1 + max altezza dei sottoalberi figli
- l'altezza di una foglia è 1

In [3…

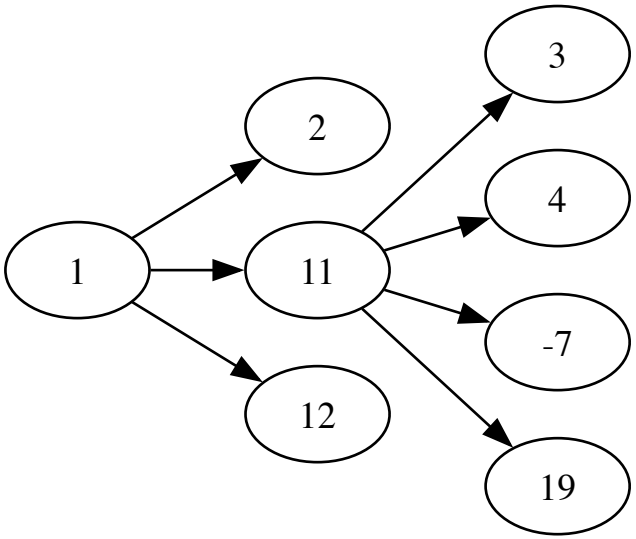
```
%%add_to NodoNario

## metodo per calcolare l'altezza del nodo
@traced()
def altezza(self):
    "l'altezza è 1 + la massima altezza dei sottoalberi"
    # se non ci sono figli si torna 1
    return max( [son.altezza()+1 for son in self._sons], default=1 )
```

In [4…

```
n19 = NodoNario(19)
n2 = NodoNario(2)
n3 = NodoNario(3)
n4 = NodoNario(4)
n13 = NodoNario(-7)
n12 = NodoNario(12)
n11 = NodoNario(11, [n3, n4, n13, n19])
n14 = NodoNario(1, [n2, n11, n12])
n14.show()
```

Out[4…



In [3…

```
n14.altezza()
```

```
entering:      NodoNario(1, 3 sons).altezza( )
|-- entering:   NodoNario(2, 0 sons).altezza( )
|-- exiting:    NodoNario(2, 0 sons).altezza( ) return: 1
|-- entering:   NodoNario(11, 4 sons).altezza( )
|--|-- entering:        NodoNario(3, 0 sons).altezza( )
|--|-- exiting:         NodoNario(3, 0 sons).altezza( ) return: 1
|--|-- entering:        NodoNario(4, 0 sons).altezza( )
|--|-- exiting:         NodoNario(4, 0 sons).altezza( ) return: 1
|--|-- entering:        NodoNario(13, 0 sons).altezza( )
|--|-- exiting:         NodoNario(13, 0 sons).altezza( ) return: 1
|--|-- entering:        NodoNario(19, 0 sons).altezza( )
|--|-- exiting:         NodoNario(19, 0 sons).altezza( ) return: 1
|-- exiting:    NodoNario(11, 4 sons).altezza( ) return: 2
|-- entering:   NodoNario(12, 0 sons).altezza( )
|-- exiting:    NodoNario(12, 0 sons).altezza( ) return: 1
exiting:       NodoNario(1, 3 sons).altezza( ) return: 3
```

Out[3…

```
3
```

stampa (in preordine) di un nodo e dei figli

- aggiungo un argomento livello che incremento ogni volta che scendo per dare l'indentazione giusta
- prima stampo il nodo indentato del livello corrente
- poi stampo ricorsivamente i sottoalberi con livello+1

In [9…

```
%%add_to NodoNario
# metodo per stampare l'albero
def stampa(self, livello=0):
    "stampa ricorsiva dell'albero in PRE ordine"
    indent = '|--'*livello        # indentazione corretta per il mio livello
    print(indent, self)           # PRE ordine
    for son in self._sons:        # per ciascun sottoalbero (se ci sono)
        son.stampa(livello+1)     # lo stampo con indentazione aumentata
```

In [1…

```
n14.stampa()
```

```
NodoNario(1, 3 sons)
|-- NodoNario(2, 0 sons)
|-- NodoNario(11, 4 sons)
|--|-- NodoNario(3, 0 sons)
|--|-- NodoNario(4, 0 sons)
|--|-- NodoNario(13, 0 sons)
|--|-- NodoNario(19, 0 sons)
|-- NodoNario(12, 0 sons)
```

cerchiamo il massimo valore nell'albero

- esploriamo ricorsivamente
- il valore massimo di un sottoalbero è il massimo tra il valore della radice ed i massimi dei sottoalberi

In [3…
```
%%add_to NodoNario

# versione in stile non-funzionale
def massimo(self):
    M = self._value
    for s in self._sons:
        m = s.massimo()
        if M < m:
            M = m
    return M

# versione in stile funzionale
def massimo2(self):
    return max([s.massimo2() for s in self._sons] + [self._value])
```

In [3…
```
n14.massimo2()
```

Out[3… 19

In [4…
```
%%add_to NodoNario
# versione che porta il massimo come argomento
# da aggiornare mano a mano che si esplora l'albero
@traced()
```

```
def massimo3(self, max_corrente=None):
    if max_corrente is None:           # se è il primo nodo che esploro
        max_corrente = self._value     # il massimo è il mio valore
    else:
        max_corrente = max(max_corrente, self._value)   # altrimenti aggiorno il massimo
    for son in self._sons:             # lo passo a ciascun sottoalbero, che mi torna il massimo trovato
        max_corrente = son.massimo3(max_corrente)
    return max_corrente   # ATTENZIONE: DOVETE TORNARE IL NUOVO VALORE!!!
                          # perchè max_corrente è una VARIABILE LOCALE!!!
                          # e tornandola comunicate all'esterno il risultato
                          # (e quindi anche agli altri sottoalberi)
```

In [4…
```
n14.massimo3()
```

```
 entering:      NodoNario(1, 3 sons).massimo3( )
|-- entering:   NodoNario(2, 0 sons).massimo3((1,) )
|-- exiting:    NodoNario(2, 0 sons).massimo3((1,) ) return: 2
|-- entering:   NodoNario(11, 4 sons).massimo3((2,) )
|--|-- entering:        NodoNario(3, 0 sons).massimo3((11,) )
|--|-- exiting:         NodoNario(3, 0 sons).massimo3((11,) ) return: 11
|--|-- entering:        NodoNario(4, 0 sons).massimo3((11,) )
|--|-- exiting:         NodoNario(4, 0 sons).massimo3((11,) ) return: 11
|--|-- entering:        NodoNario(13, 0 sons).massimo3((11,) )
|--|-- exiting:         NodoNario(13, 0 sons).massimo3((11,) ) return: 13
|--|-- entering:        NodoNario(19, 0 sons).massimo3((13,) )
|--|-- exiting:         NodoNario(19, 0 sons).massimo3((13,) ) return: 19
|-- exiting:    NodoNario(11, 4 sons).massimo3((2,) ) return: 19
|-- entering:   NodoNario(12, 0 sons).massimo3((19,) )
|-- exiting:    NodoNario(12, 0 sons).massimo3((19,) ) return: 19
 exiting:       NodoNario(1, 3 sons).massimo3( ) return: 19
```

Out[4…
```
19
```

In [1…
```
n14.massimo(), n14.massimo2(), n14.massimo3()
```

```
entering:      NodoNario(1, 3 sons).massimo3( )
|-- entering:   NodoNario(2, 0 sons).massimo3((1,) )
|-- exiting:    NodoNario(2, 0 sons).massimo3((1,) ) return: 2
|-- entering:   NodoNario(11, 4 sons).massimo3((2,) )
|--|-- entering:        NodoNario(3, 0 sons).massimo3((11,) )
|--|-- exiting:         NodoNario(3, 0 sons).massimo3((11,) ) return: 11
|--|-- entering:        NodoNario(4, 0 sons).massimo3((11,) )
|--|-- exiting:         NodoNario(4, 0 sons).massimo3((11,) ) return: 11
|--|-- entering:        NodoNario(13, 0 sons).massimo3((11,) )
|--|-- exiting:         NodoNario(13, 0 sons).massimo3((11,) ) return: 13
|--|-- entering:        NodoNario(19, 0 sons).massimo3((13,) )
|--|-- exiting:         NodoNario(19, 0 sons).massimo3((13,) ) return: 19
|-- exiting:    NodoNario(11, 4 sons).massimo3((2,) ) return: 19
|-- entering:   NodoNario(12, 0 sons).massimo3((19,) )
|-- exiting:    NodoNario(12, 0 sons).massimo3((19,) ) return: 19
exiting:       NodoNario(1, 3 sons).massimo3( ) return: 19
```

Out[1… (19, 19, 19)

cerchiamo il nodo che contiene il massimo valore

- esploriamo ricorsivamente
- il nodo massimo di un sottoalbero è nodo che contiene il massimo tra il valore della radice ed i massimi dei sottoalberi

In [4…

```
%%add_to NodoNario

def max_node(self):
    M = [self] + [s.max_node() for s in self._sons] # prendo me stesso ed i massimi nodi dei figli
    #print(M)
    return max(M, key=lambda x: x._value)           # a tra tutti cerco il nodo che ha massimo valore

def min_node(self):
    M = [self] + [s.min_node() for s in self._sons] # prendo me stesso ed i massimi nodi dei figli
    #print(M)
    return min(M, key=lambda x: x._value)           # a tra tutti cerco il nodo che ha minimo valore
```

In [4…

```
print(n14.max_node(), n14.min_node())
```

```
NodoNario(19, 0 sons) NodoNario(1, 3 sons)
```

cerchiamo la differenza in altezza tra nodo con valore massimo e nodo con valore minimo

In [4…
```
%%add_to NodoNario
def depth_of_max(self,livello=0):
    # prendo coppie (valore, profondità) di me stesso e di tutti i sottoalberi
    M = [(self._value, livello)] + [ son.depth_of_max(livello+1) for son in self._sons]
    return max(M, key=lambda x: x[0])    # ne torno il (massimo VALORE e sua profondità)

def depth_of_max2(self,livello=0):
    M = self._value
    P = livello
    for son in self._sons:
        m,p = son.depth_of_max2(livello+1)
        if m > M:
            M = m
            P = p
    return M, P

def depth_of_min(self,livello=0):
    M = [(self._value, livello)] + [ son.depth_of_min(livello+1) for son in self._sons]
    return min(M, key=lambda x: x[0])
```

In [4…
```
print(n14.depth_of_max(), n14.depth_of_min())
```

```
(19, 2) (-7, 2)
```

Diametro di un albero binario (lunghezza del percorso più lungo)

Caso 1: la radice ha DUE figli

il diametro è la somma delle due **profondità massime** dei sotto alberi + 2

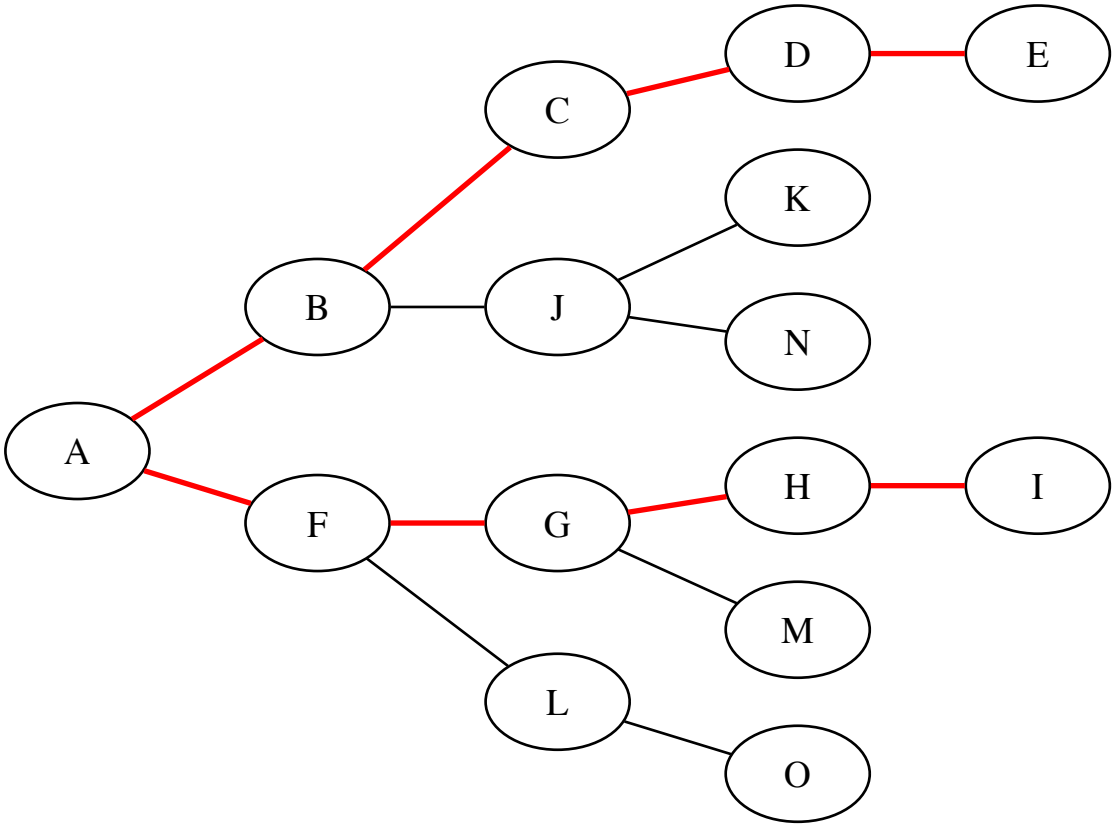
In [5…
```
from pygraphviz import AGraph
G = AGraph(rankdir='LR')
```

```
G.add_edges_from([['A', 'B'],['B', 'C'],['C', 'D'],['D', 'E' ],['A', 'F'],['F', 'G'],['G', 'H'],['H', 'I
                  style='bold', color='red')
G.add_edges_from([['B', 'J'],['J', 'K'],['F', 'L'],['L', 'O' ],['G', 'M'],['J', 'N']])
G.layout('dot')
```

In [5…

```
G
```

Out[5…



DIAMETRO = (3+3)+2 = 8 archi (9 nodi)

Caso 2: la radice ha UN SOLO figlio

il diametro è la **profondità massima** dell'albero

In [5…

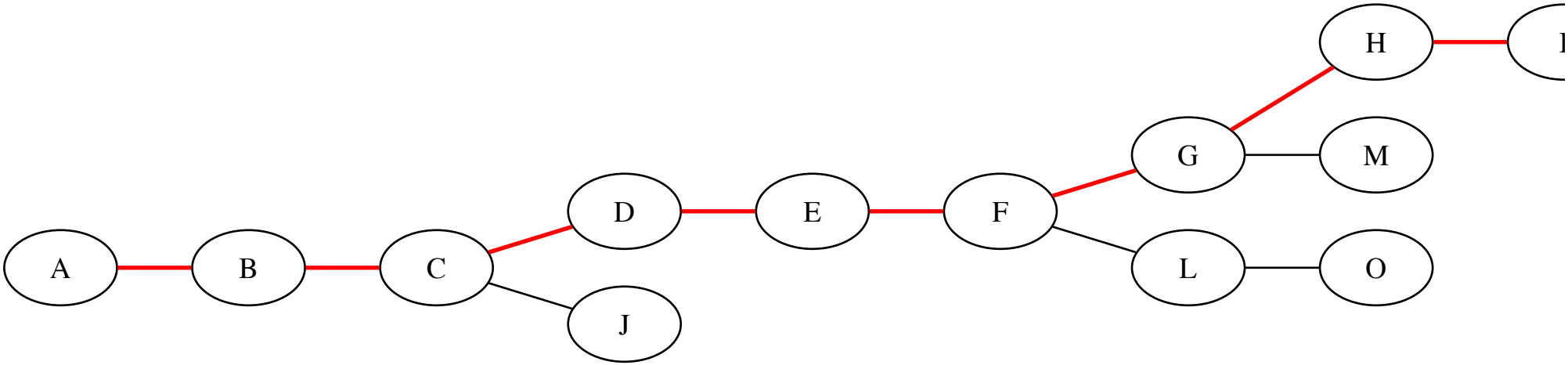
```
from pygraphviz import AGraph
G = AGraph(rankdir='LR')
G.add_edges_from([['A', 'B'],['B', 'C'],['C', 'D'],['D', 'E'],['E','F'],['F', 'G'],['G', 'H'],['H', 'I'
```

```
                  style='bold', color='red')
G.add_edges_from([['C', 'J'],['F', 'L'],['L', 'O'],['G', 'M']])
G.layout('dot')
```

In [5…

```
G
```

Out[5…



DIAMETRO 8 archi (9 nodi)

NOTA: il percorso potrebbe NON passare per la radice

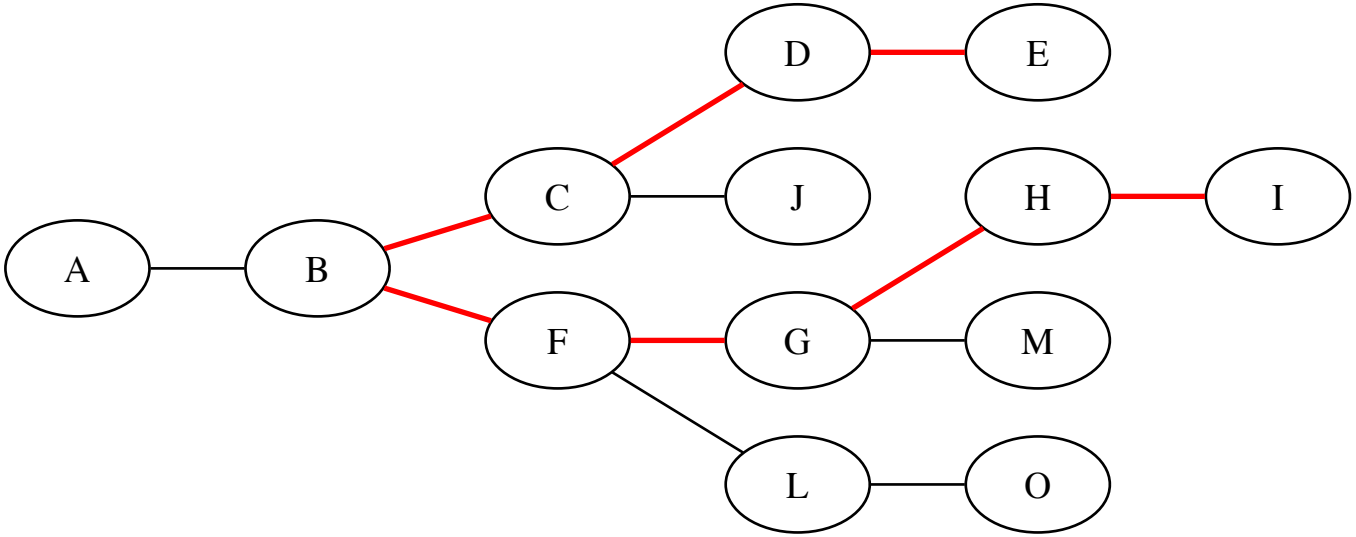
dobbiamo cercare il nodo che fa da radice al sottoalbero col percorso più lungo

il diametro è il più grande valore calcolato per ciascun nodo dell'albero

In [2…

```
from pygraphviz import AGraph
G = AGraph(rankdir='LR')
G.add_edges_from([['B', 'C'],['C', 'D'],['D', 'E'],['B', 'F'],['F', 'G'],['G', 'H'],['H', 'I' ]],
                  style='bold', color='red')
G.add_edges_from([['A', 'B'],['C', 'J'],['F', 'L'],['L', 'O'],['G', 'M']])
G.layout('dot')
G
```

Out [2…



DIAMETRO: 7 archi (8 nodi)

In [5…

```
import jdc

class NodoBinario:
    def __init__(self, V : int, sx : 'NodoBinario' = None, dx : 'NodoBinario' = None):
        self._value = V
        self._sx = sx
        self._dx = dx
    def __repr__(self) -> str :
        "visualizzo il valore del nodo con, se ci sono, altezza e percorso"
        return f"V: {self._value}\nA: {getattr(self,'_altezza','')}\nP: {getattr(self,'_percorso','')}"
```

In [5…

```
%%add_to NodoBinario

from pygraphviz import AGraph

# metodi che aggiungono all'albero binario come disegnarsi usando la libreria pygraphviz

def _dot(self, rank=0):
    "creo l'elenco di nodi ed archi che Graphviz sa visualizzare"
    nodi = [(self,rank)]        # sicuramente ci va il nodo e la sua altezza
    archi = []
```

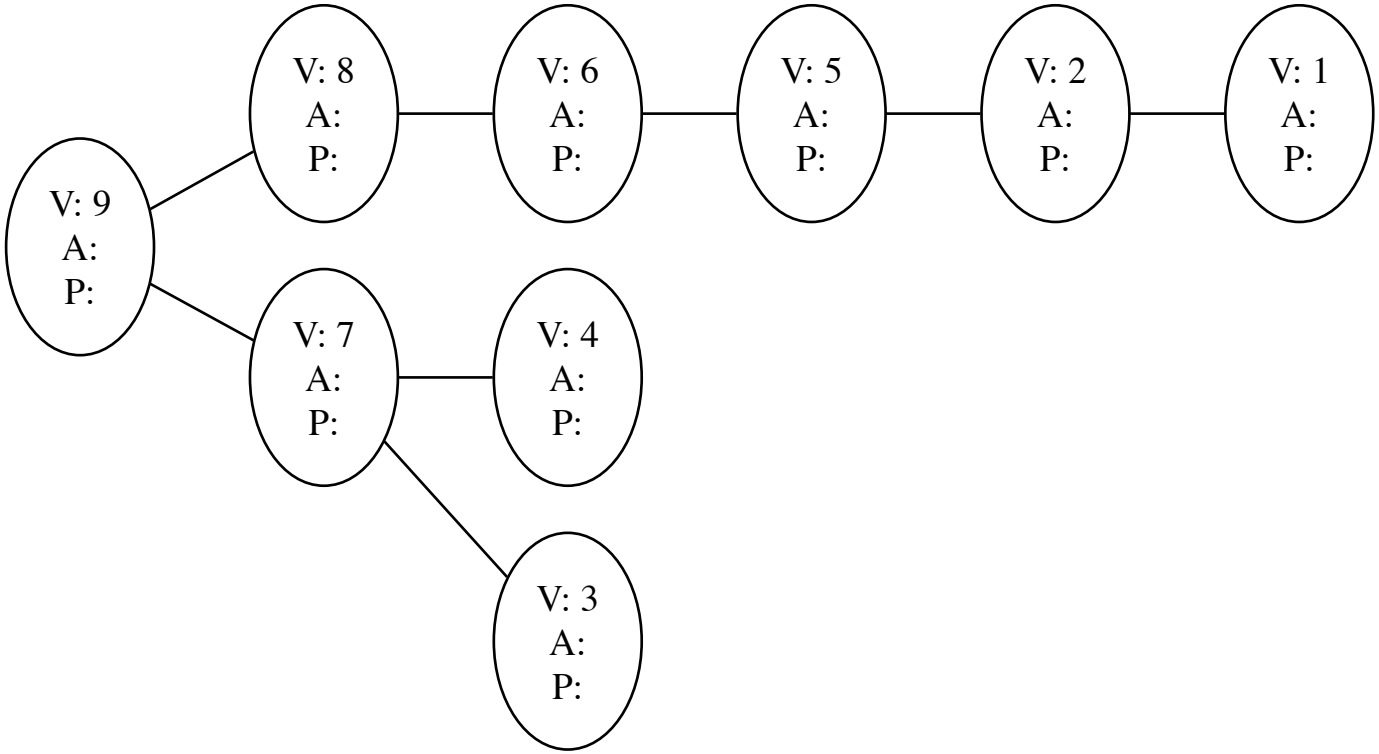
```
    if self._sx:                # se ho un figlio sinistro
        archi.append((self, self._sx))      # aggiungo l'arco sinistro
        NSX,ASX = self._sx._dot(rank+1)     # calcolo i nodi ed archi del sottoalbero sinistro
        nodi  += NSX                        # e li aggiungo
        archi += ASX                        # a quelli correnti
    if self._dx:                # lo stesso a destra
        archi.append((self, self._dx))
        NDX,ADX = self._dx._dot(rank+1)
        nodi += NDX
        archi += ADX
    return nodi, archi          # alla fine ritorno l'elanco di (nodo,rank) e quello degli archi

def show(self):
    "creo l'oggetto Digraph per visualizzare il grafo diretto"
    G = AGraph(rankdir='LR')          # faccio un grafo orizzontale
    nodi, archi = self._dot()         # calcolo nodi ed archi
    for nodo, rank in nodi:           # aggiungo i nodi
        G.add_node(nodo, rank=rank)   # ciascuno al suo livello
    G.add_edges_from(archi)           # aggiungo anche gli archi che li collegano
    G.layout('dot')                   # calcolo la visualizzazione
    return G                          # torno il grafo, che Jupyter sa mostrare
```

In [5…

```
n1= NodoBinario(1)
n2= NodoBinario(2, dx=n1)
n3= NodoBinario(3)
n4= NodoBinario(4)
n5= NodoBinario(5, dx=n2)
n6= NodoBinario(6, dx=n5)
n7= NodoBinario(7, n4, n3)
n8= NodoBinario(8, n6)
r= NodoBinario(9, n8, n7)
r.show()
```

Out [5…



STEP 1: calcolare le altezze di tutti i sottoalberi

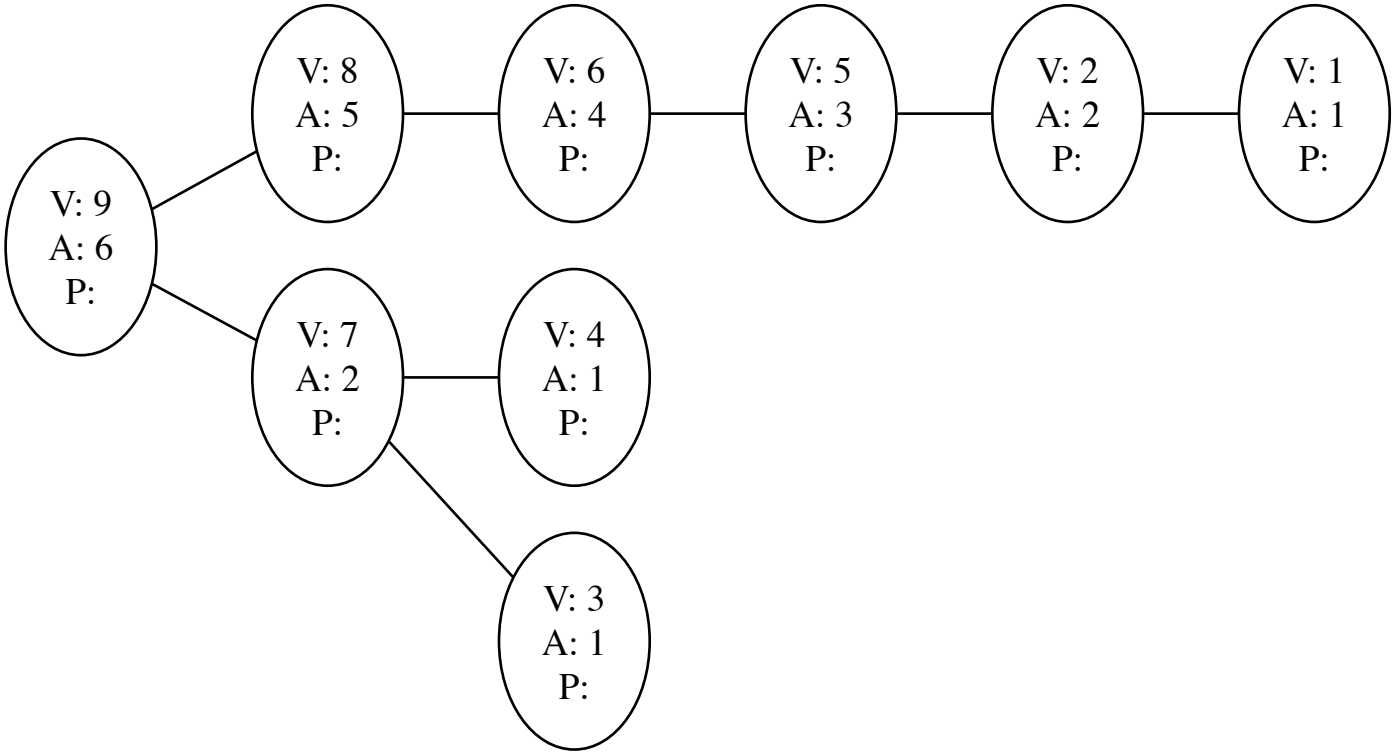
- visita ricorsiva dell'albero
- visto che vogliamo l'**altezza** (dalle foglie) del nodo conviene lavorare **in uscita dalla ricorsione**

In [5…

```
def aggiungi_altezze(root : NodoBinario) -> int :
    if root is None:
        return 0
    A_sx = aggiungi_altezze(root._sx)
    A_dx = aggiungi_altezze(root._dx)
    root._altezza = max(A_sx, A_dx) +1
    return root._altezza

aggiungi_altezze(r)
r.show()
```

Out [5…



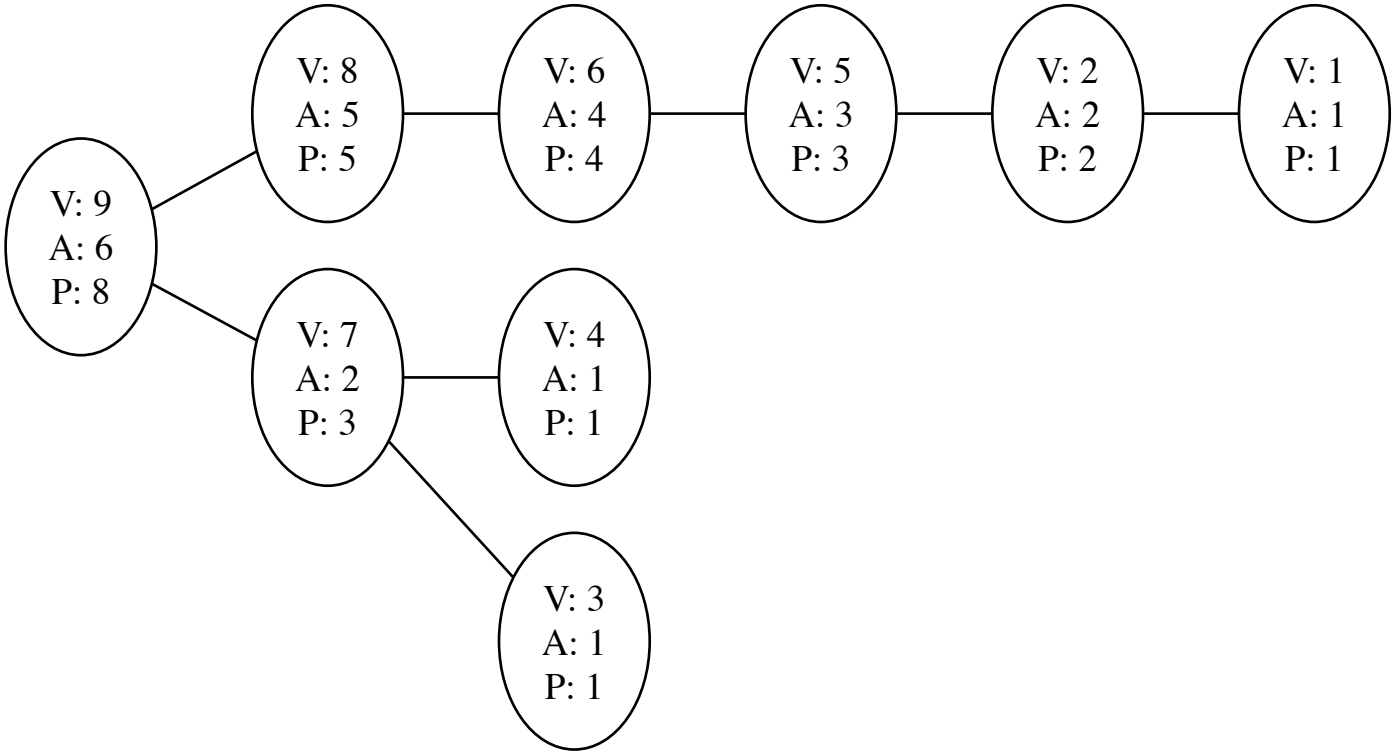
STEP 2: calcolare i percorsi di ciascun sottoalbero

In [5…

```
def aggiungi_percorsi(radice):
    "a ciascun nodo aggiungo l'attributo _percorso di quel sottoalbero"
    if radice is not None:
        A_sx = A_dx = 0
        if radice._sx:
            A_sx = radice._sx._altezza
        if radice._dx:
            A_dx = radice._dx._altezza
        radice._percorso = A_sx + A_dx + 1
        aggiungi_percorsi(radice._sx)
        aggiungi_percorsi(radice._dx)

aggiungi_percorsi(r)
r.show()
```

Out [5…



STEP 3: cercare il nodo col valore massimo di percorso

In [5…

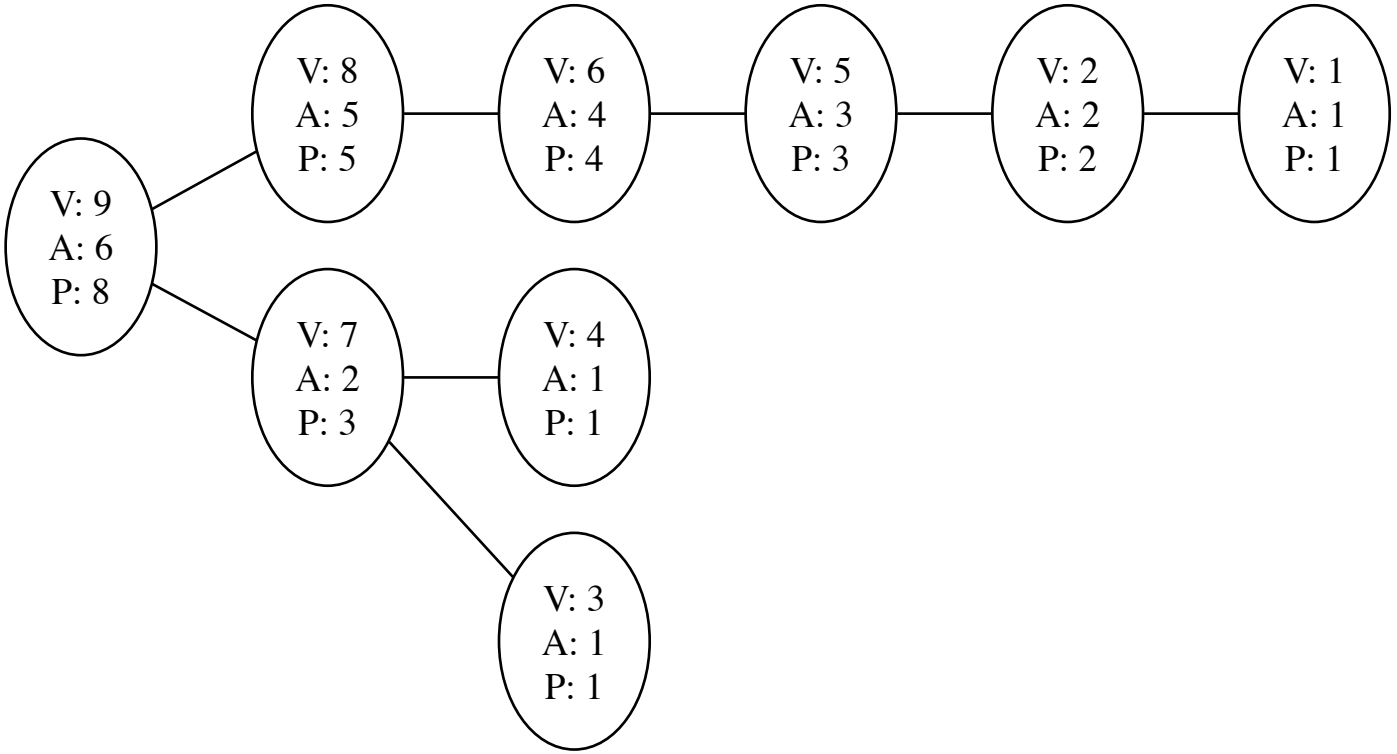
```python
def diametro(radice):
    if radice is None:
        return 0
    D_sx = diametro(radice._sx)    # cerco il percorso più lungo nel sottoalbero sinistro
    D_dx = diametro(radice._dx)    # e poi in quello destro

    # il massimo o sta a sinistra o sta a destra o è il mio percorso
    return max(D_sx, D_dx, radice._percorso)

print('DIAMETRO', diametro(r))
r.show()
```

```
DIAMETRO 8
```

Out [5...



E sugli Alberi N-ari? (per casa)

Un percorso massimo può passare:

- per la radice ed i due sottoalberi più profondi **SE HA ALMENO 2 FIGLI**
- per la radice ed il solo sottoalbero più profondo **SE HA UN SOLO FIGLIO**
- per un nodo interno ...

Soluzione: più o meno come per gli alberi binari

- dobbiamo solo prendere i DUE figli con sottoalbero più profondo