lezione18

November 23, 2023

1 Fondamenti di Programmazione

Andrea Sterbini

lezione 18 - 23 novembre 2023

2 RECAP:

- Ritorsione
- scansione di una lista avanti e indietro, palindroma, esplorazione di una directory
- Alberi binari
- stampa in preordine postordine e inordine

[1]:
```
# decoratore che stampa le chiamate ed uscite di una funzione ricorsiva
from rtrace import trace
```

[2]:
```
## usiamo gli oggetti per rappresentare i nodi dell'albero
class NodoBinario:
    def __init__(self,   V : int, left  : 'NodoBinario' = None, right :␣
 ↪'NodoBinario' = None):
        self._value = V
        self._sx = left
        self._dx = right
    def __repr__(self):
        return f'NodoBinario({self._value})'

n11 = NodoBinario(11)
n10 = NodoBinario(10)
n1 = NodoBinario(1, right=n11)
n0 = NodoBinario(0, left= n10)
r = NodoBinario('radice', n0, n1)
r
```

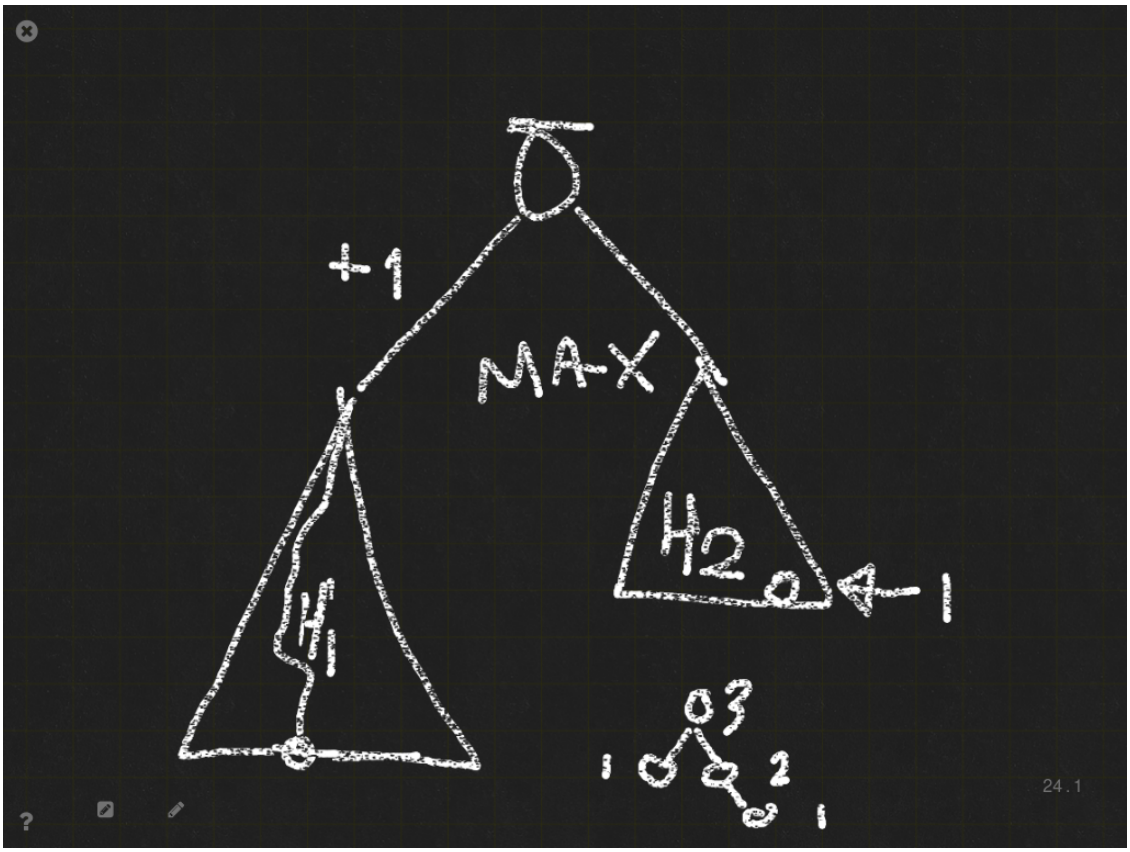
[2]:
```
NodoBinario(radice)
```

2.1 Calcolo della profondità/altezza di un albero (in uscita)

- Una foglia ha altezza 1 (**caso base**)

- un nodo ha altezza 1 + la massima altezza dei sottoalberi (**passo ricorsivo + composizione**)
- **convergenza**: ogni volta che scendo in un sottoalbero i nodi da esaminare diminuiscono almeno di 1 (la radice)

Oppure possiamo considerare come **caso base**: - **None** ha altezza 0



[3]:
```
@trace()
def altezza(radice):
    if radice is None:            # caso base, un albero vuoto ha altezza 0
        return 0
    # altrimenti sono nel caso ricorsivo
    H_sx = altezza(radice._sx)    # calcolo l'altezza a sinistra
    H_dx = altezza(radice._dx)    # calcolo l'altezza a destra
    return 1 + max(H_sx, H_dx)    # e sommo 1 al massimo delle due

altezza.trace(r)
#stampa_PRE(r)
```

```
------------------- Starting recursion -------------------
 entering       altezza(NodoBinario(radice),)
|-- entering    altezza(NodoBinario(0),)
|--|-- entering altezza(NodoBinario(10),)
|--|--|-- entering      altezza(None,)
|--|--|-- exiting       altezza(None,)  returns 0
```

```
|--|--|-- entering      altezza(None,)
|--|--|-- exiting       altezza(None,)  returns 0
|--|-- exiting  altezza(NodoBinario(10),)       returns 1
|--|-- entering altezza(None,)
|--|-- exiting  altezza(None,)  returns 0
|-- exiting     altezza(NodoBinario(0),)        returns 2
|-- entering    altezza(NodoBinario(1),)
|--|-- entering altezza(None,)
|--|-- exiting  altezza(None,)  returns 0
|--|-- entering altezza(NodoBinario(11),)
|--|--|-- entering      altezza(None,)
|--|--|-- exiting       altezza(None,)  returns 0
|--|--|-- entering      altezza(None,)
|--|--|-- exiting       altezza(None,)  returns 0
|--|-- exiting  altezza(NodoBinario(11),)       returns 1
|-- exiting     altezza(NodoBinario(1),)        returns 2
 exiting        altezza(NodoBinario(radice),)   returns 3
-------------------- Ending recursion --------------------
Num calls: 11
```

[3]: 3

2.2 Calcolo della profondità in "andata"

- si fornisce come argomento la profondità massima incontrata finora
- la si aggiorna visitando tutto l'albero
  - (ogni volta che si scende in un sottoalbero si somma 1 alla profondità)
- se il nodo è **None** si torna la profondità corrente
- altrimenti il nodo esiste e la profondità è il massimo delle profondità dei due sottoalberi

[4]:
```python
@trace()
def altezza2(radice, profondità=0):
    if radice is None:
        return profondità
    P_sx = altezza2(radice._sx, profondità+1)
    P_dx = altezza2(radice._dx, profondità+1)
    return max(P_sx, P_dx)

altezza2.trace(r)
```

```
------------------- Starting recursion -------------------
 entering       altezza2(NodoBinario(radice),)
|-- entering    altezza2(NodoBinario(0), 1)
|--|-- entering altezza2(NodoBinario(10), 2)
|--|--|-- entering      altezza2(None, 3)
|--|--|-- exiting       altezza2(None, 3)       returns 3
|--|--|-- entering      altezza2(None, 3)
|--|--|-- exiting       altezza2(None, 3)       returns 3
|--|-- exiting  altezza2(NodoBinario(10), 2)    returns 3
```

```
|--|-- entering altezza2(None, 2)
|--|-- exiting  altezza2(None, 2)       returns 2
|-- exiting     altezza2(NodoBinario(0), 1)     returns 3
|-- entering    altezza2(NodoBinario(1), 1)
|--|-- entering altezza2(None, 2)
|--|-- exiting  altezza2(None, 2)       returns 2
|--|-- entering altezza2(NodoBinario(11), 2)
|--|--|-- entering      altezza2(None, 3)
|--|--|-- exiting       altezza2(None, 3)       returns 3
|--|--|-- entering      altezza2(None, 3)
|--|--|-- exiting       altezza2(None, 3)       returns 3
|--|-- exiting  altezza2(NodoBinario(11), 2)    returns 3
|-- exiting     altezza2(NodoBinario(1), 1)     returns 3
 exiting        altezza2(NodoBinario(radice),)  returns 3
-------------------- Ending recursion --------------------
Num calls: 11
```

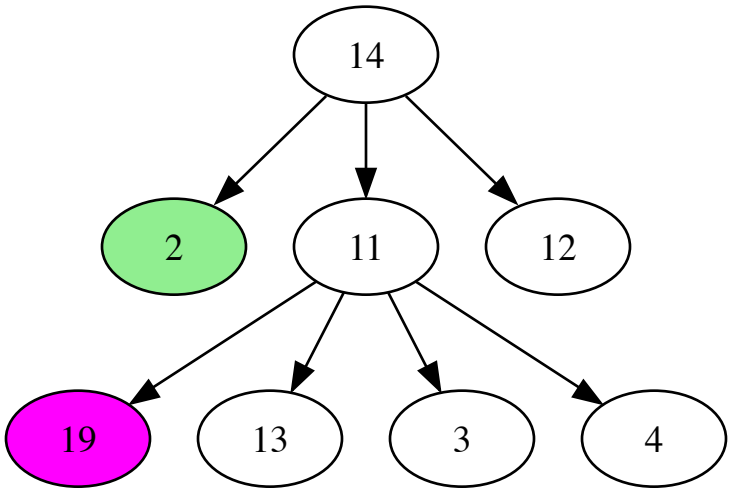
[4]: 3

3 Alberi N-ari (con numero indefinito di figli)

- **value**: valore del nodo
- **sons**: elenco di figli

[5]:
```
from pygraphviz import AGraph
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')
G3
```

[5]:



[6]:
```
# utility che mi permette di aggiungere con %%add_to dei metodi definiti in␣
 ↪celle separate alla classe
import jdc

class NodoNario :
    def __init__(self,   V : int, sons : list['NodoNario'] = None): # ATTENTI␣
 ↪AL DEFAULT!!!
        self._value = V
        if sons is None:
            self._sons = []
        else:
            self._sons = sons

    def __repr__(self):
        return f"NodoNario({self._value}, {len(self._sons)} sons)"
```

3.0.1 stavolta scriviamo le funzioni come metodi della classe NodoNario

3.0.2 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 0

[7]:
```
%%add_to NodoNario

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

[11]:
```
n19_2 = NodoNario(19)
n2 = NodoNario(2)
n19 = NodoNario(19,[n2,n19_2])
n3 = NodoNario(3)
n4 = NodoNario(4)
n13 = NodoNario(13)
n12 = NodoNario(12)
n11 = NodoNario(11, [n3, n4, n13, n19])
n14 = NodoNario(1, [n11, n12])
```

[12]:
```
n14.altezza()
```

[12]: 4

3.0.3 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

[13]:
```
%%add_to NodoNario
# metodo per stampare l'albero
def stampa(self, livello=0):
    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
```

[14]:
```
n14.stampa()
```

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

3.1 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

[15]:
```
%%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])
```

[16]:
```
n14.massimo2()
```

[16]:
```
19
```

[17]:
```
%%add_to NodoNario
# versione che porta il massimo come argomento da aggiornare mano a mano che si␣
 ↪esplora l'albero
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 e che passo al successivo
        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)
```

[18]:
```
n14.massimo3()
```

[18]:
```
19
```

[19]:
```
n14.massimo(), n14.massimo2(), n14.massimo3()
```

[19]:
```
(19, 19, 19)
```

3.2 cerchiamo il nodo col 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

[20]:
```
%%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
```

[21]:
```
n14.max_node(), n14.min_node()
```

[21]:
```
(NodoNario(19, 2 sons), NodoNario(1, 2 sons))
```

[22]:
```
%%add_to NodoNario

def max_nodes(self):
    M = [self] + [ x for s in self._sons
                     for x in s.max_nodes()] # prendo me stesso ed i massimi nodi dei figli
    #print(M)
    massimo = max(M, key=lambda x: x._value)
    massimi = [ m for m in M if m._value == massimo._value ]
    return list(massimi)

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)
```

[23]:
```
n14.max_nodes()
```

[23]:
```
[NodoNario(19, 2 sons), NodoNario(19, 0 sons)]
```

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

[24]:
```
%%add_to NodoNario
def depth_of_max(self,livello=0):
    # prendo coppie (valore, profondità) di me stesso e di tutti i sottoalberi
    M = [ son.depth_of_max(livello+1) for son in self._sons] +[(self._value,␣
 ↪livello)]
    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])
```

[25]:
```
n14.depth_of_max(), n14.depth_of_min(), n14.stampa()
```

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

[25]:
```
((19, 3), (1, 0), None)
```

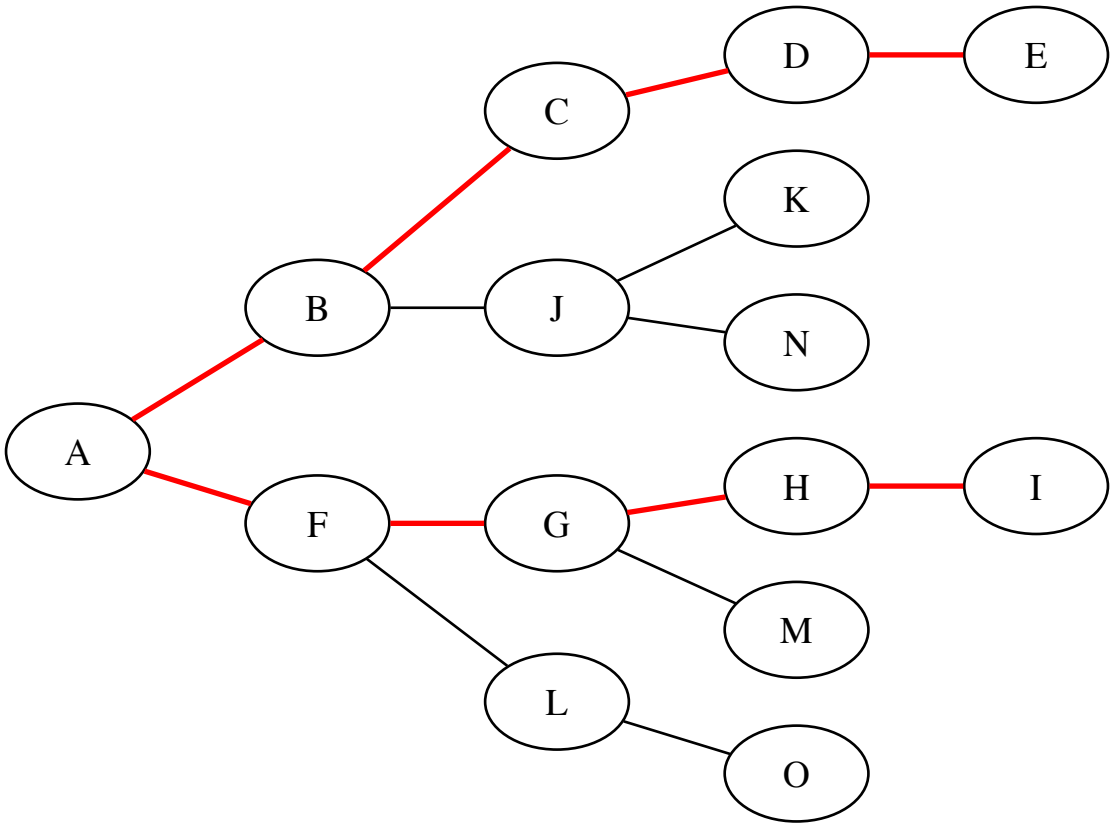
4 Diametro di un albero binario (lunghezza del percorso più lungo)

4.1 Caso 1: il percorso passa per la radice

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

[26]:
```
from pygraphviz import AGraph
G = AGraph(rankdir='LR')
G.add_edges_from([['A', 'B'],['B', 'C'],['C', 'D'],['D', 'E' ]], style='bold',␣
 ↪color='red')
G.add_edges_from([['A', 'F'],['F', 'G'],['G', 'H'],['H', 'I' ]], style='bold',␣
 ↪color='red')
G.add_path(['B', 'J', 'K'])
G.add_path(['F', 'L', 'O'])
G.add_path(['G', 'M'])
G.add_path(['J', 'N'])
G.layout('dot')
G
```

[26]:



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

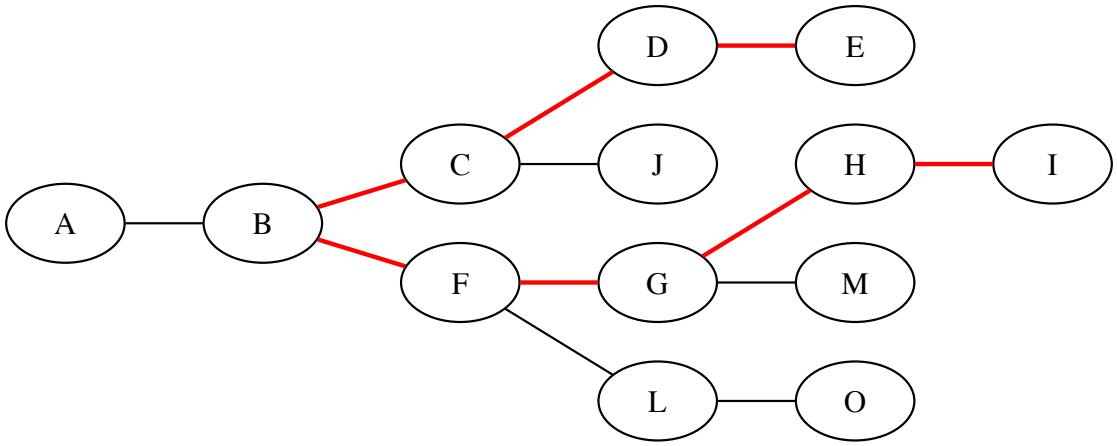
4.2 Caso 2: il percorso NON passa per la radice

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

[27]:
```
from pygraphviz import AGraph
G = AGraph(rankdir='LR')
```

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

[27]:



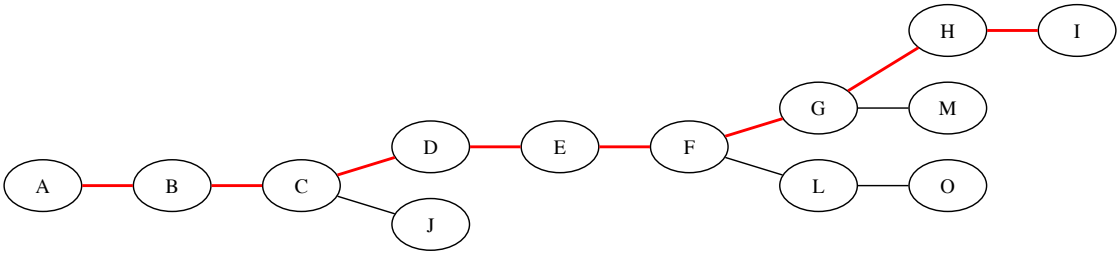
DIAMETRO: 7 archi (8 nodi)

4.3 Caso 3: il percorso NON si biforca

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

[28]:
```
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_path(['C', 'J'])
G.add_path(['F', 'L', 'O'])
G.add_path(['G', 'M'])
G.layout('dot')
G
```

[28]:



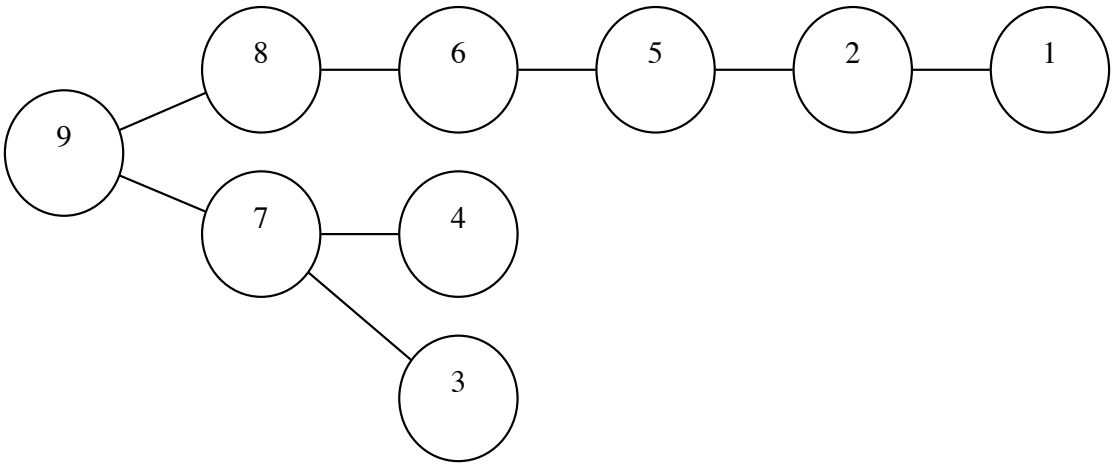
DIAMETRO 8 archi (9 nodi)

[29]:
```
from pygraphviz import AGraph
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"{self.
 ↪_value}\n{getattr(self,'_altezza','')}\n{getattr(self,'_percorso','')}"
    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 sinistr
            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')    # grafo orizzontale
        nodi, archi = self._dot()
        for nodo, rank in nodi:
            G.add_node(nodo, rank=rank)   # con i nodi indicati, ciascuno al␣
 ↪suo livello
        G.add_edges_from(archi)           # ed anche gli archi che li collegano
        G.layout('dot')                   # calcolo la visualizzazione
        return G
```

[30]:
```
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()
```

[30]:



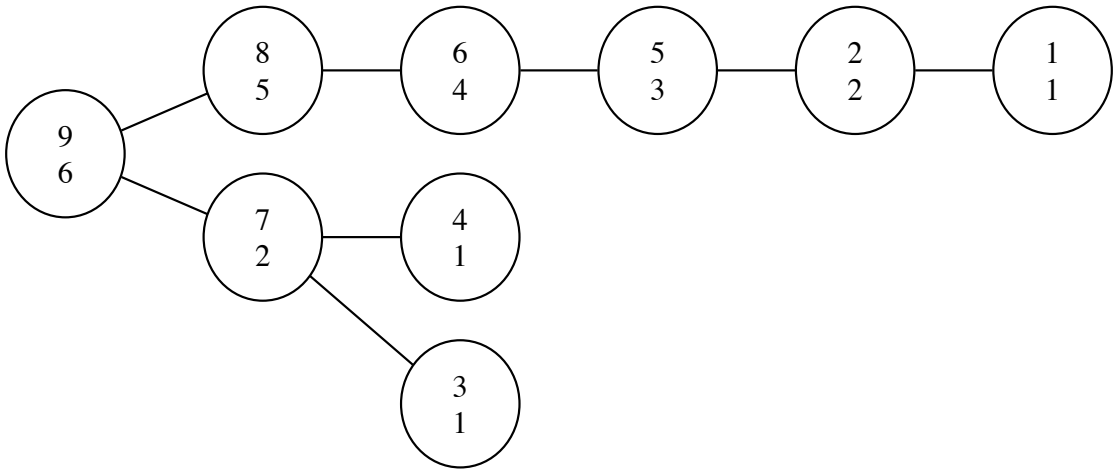
4.4 STEP 1: calcolare le altezze di tutti i sottoalberi

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

[31]:
```
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()
```

[31]:

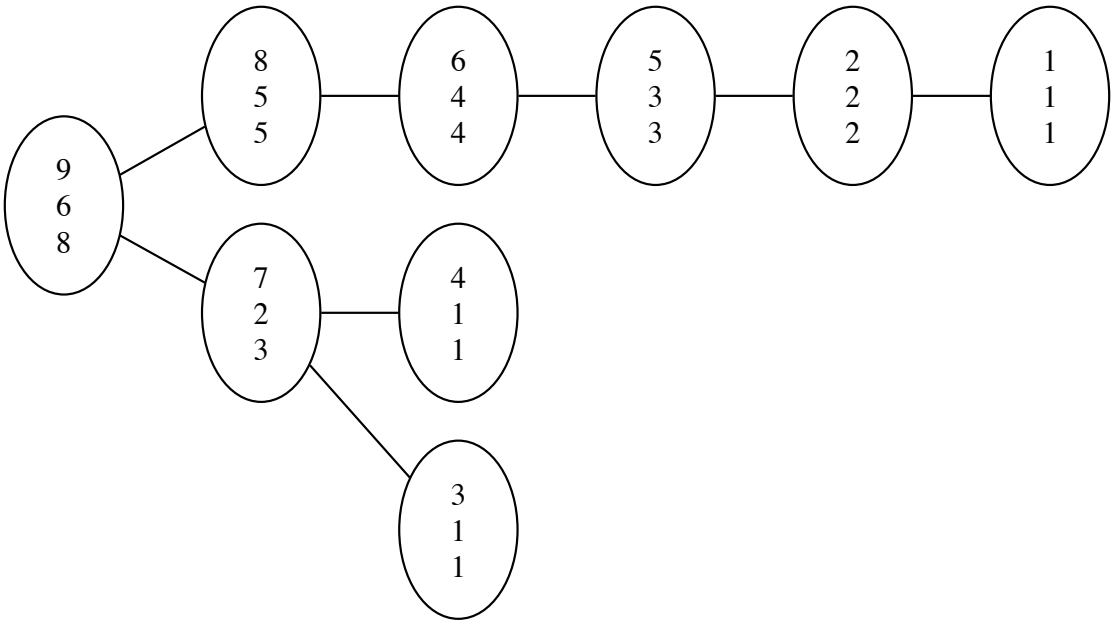


4.5 STEP 2: calcolare i percorsi supponendo che passino per ciascun nodo come radice

[32]:
```
def aggiungi_percorsi(radice):
    "a ciascun nodo aggiungo l'attributo _percorso come se passasse per quel␣
 ↪nodo come radice"
    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()
```

[32]:



4.6 STEP 3: cercare il nodo col valore massimo di percorso o altezza

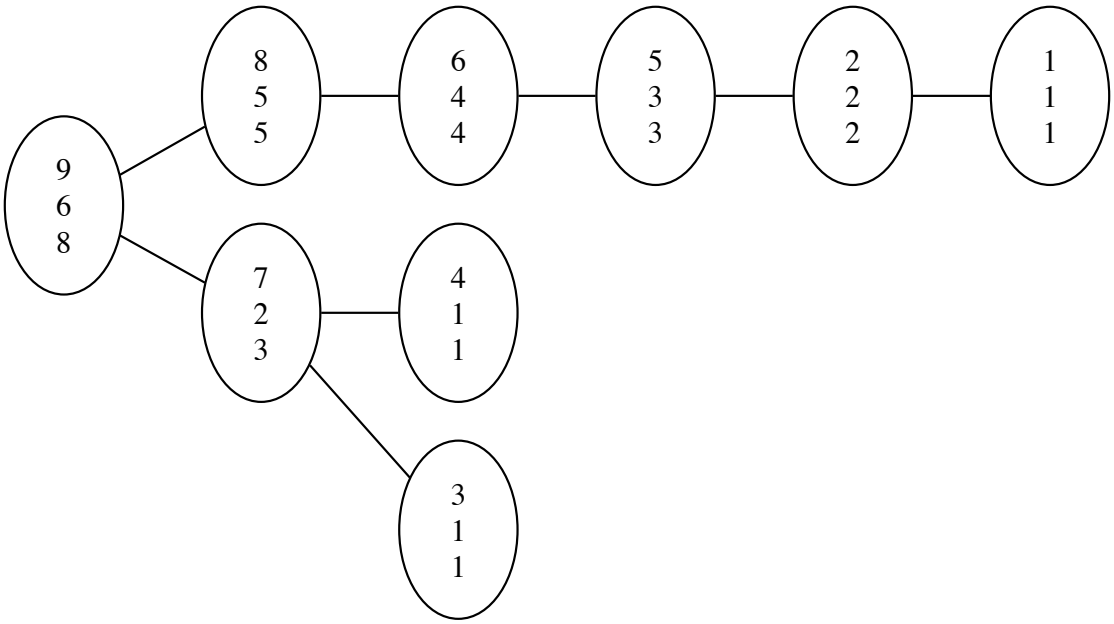
[33]:
```
def diametro(radice):
    if radice is None:
        return 0
    D_sx = diametro(radice._sx)    # cerco il percorso più lungo nel␣
 ↪sottoalbero sinistro
    D_dx = diametro(radice._dx)    # e poi in quello destro

    # il massimo o sta a sinistra o sta a destra o passa per qui oppure è la␣
 ↪mia altezza
    return max(D_sx, D_dx, radice._percorso, radice._altezza)

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

```
DIAMETRO 8
```

[33]:



5 E sugli Alberi N-ari? (TODO)

Un percorso massimo può passare: - per la radice ed i due sottoalberi più profondi **SE ALMENO 2 FIGLI** - per la radice ed il solo sottoalbero più profondo **SE UN SOLO FIGLIO** - per un nodo interno …

Soluzione: come prima

6 MergeSort (ordinamento per fusione)

6.1 osservazione: se due liste sono già ordinate è facile e veloce fonderle in una nuova lista ordinata

- per fondere due liste ordinate (**merge**)
  - il primo elemento della soluzione è uno dei due primi delle due liste (**composizione** della soluzione)
  - il resto della soluzione è la fusione del resto delle due liste (**caso ricorsivo**)
- **convergenza**: almeno un elemento va a posto per ogni chiamata ricorsiva
- **caso base**: una delle due liste è vuota

[34]:
```
@trace(True)
def merge(L1 : list, L2 : list ) -> list:
    if not L1:              # caso base: L1 vuota
        return L2
    if not L2:              # caso base: L2 vuota
        return L1
    if L1[0] <= L2[0]:      # caso ricorsivo con L1[0] minore
```

```
        return [L1[0]] + merge( L1[1:], L2 )
    else:                   # caso ricorsivo con L2[0] minore
        return [L2[0]] + merge( L1,     L2[1:] )

L1 = [1, 3, 5, 7, 9]
L2 = [2, 4, 6, 8]
merge.trace(L1, L2)

# NOTA: per essere più efficienti possiamo preallocare la lista risultato e
 ↪lavorare solo su indici
# NOTA: oppure/inoltre lavorare in modo iterativo
```

```
------------------- Starting recursion -------------------

 entering       merge([1, 3, 5, 7, 9], [2, 4, 6, 8])
|-- entering    merge([3, 5, 7, 9], [2, 4, 6, 8])
|--|-- entering merge([3, 5, 7, 9], [4, 6, 8])
|--|--|-- entering      merge([5, 7, 9], [4, 6, 8])
|--|--|--|-- entering   merge([5, 7, 9], [6, 8])
|--|--|--|--|-- entering        merge([7, 9], [6, 8])
|--|--|--|--|--|-- entering     merge([7, 9], [8])
|--|--|--|--|--|--|-- entering  merge([9], [8])
|--|--|--|--|--|--|--|-- entering       merge([9], [])
|--|--|--|--|--|--|--|-- exiting        merge([9], [])  returns [9]
|--|--|--|--|--|--|-- exiting   merge([9], [8]) returns [8, 9]
|--|--|--|--|--|-- exiting      merge([7, 9], [8])      returns [7, 8, 9]
|--|--|--|--|-- exiting merge([7, 9], [6, 8])   returns [6, 7, 8, 9]
|--|--|--|-- exiting    merge([5, 7, 9], [6, 8])        returns [5, 6, 7, 8, 9]
|--|--|-- exiting       merge([5, 7, 9], [4, 6, 8])     returns [4, 5, 6, 7, 8,
9]
|--|-- exiting  merge([3, 5, 7, 9], [4, 6, 8])  returns [3, 4, 5, 6, 7, 8, 9]
|-- exiting     merge([3, 5, 7, 9], [2, 4, 6, 8])       returns [2, 3, 4, 5, 6,
7, 8, 9]
 exiting        merge([1, 3, 5, 7, 9], [2, 4, 6, 8])    returns [1, 2, 3, 4, 5,
6, 7, 8, 9]

------------------- Ending recursion --------------------
Num calls: 9
```

[34]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

6.2 MA se ho una lista disordinata e la voglio ordinare? (MergeSort)

- la posso spezzare in due liste disordinate (**riduzione**)
- le posso ordinare separatamente (chiamata ricorsiva sui **sottoproblemi**)
- le posso fondere rapidamente (**costruzione della soluzione**) con **merge**
- **convergenza**: a forza di spezzare le liste in 2 si arriverà a liste di 0,1 elementi
- **caso base**: liste di 1 o 0 elementi, sono già ordinate

[35]:
```
## per spezzare in 2 posso usare una slice
L = [1, 5, 2, 9, 4, 6, 1, 90, 23 ]

mid = len(L)//2
L1 = L[:mid]
L2 = L[mid:]

L1, L2
```

[35]:
```
([1, 5, 2, 9], [4, 6, 1, 90, 23])
```

[36]:
```
## oppure una funzione ricorsiva che torna due liste di elementi alternati
@trace()
def splitta(L):
    if not L:
        return [], []
    else:
        primo, *resto = L     # prendo il primo elemento ed il resto
        L1, L2 = splitta(resto)
        L2.append(primo)
        return L2, L1    # aggiungo il valore su una lista e le scambio
splitta.trace(L)
```

```
------------------- Starting recursion -------------------
 entering       splitta([1, 5, 2, 9, 4, 6, 1, 90, 23],)
|-- entering    splitta([5, 2, 9, 4, 6, 1, 90, 23],)
|--|-- entering splitta([2, 9, 4, 6, 1, 90, 23],)
|--|--|-- entering      splitta([9, 4, 6, 1, 90, 23],)
|--|--|--|-- entering   splitta([4, 6, 1, 90, 23],)
|--|--|--|--|-- entering        splitta([6, 1, 90, 23],)
|--|--|--|--|--|-- entering     splitta([1, 90, 23],)
|--|--|--|--|--|--|-- entering  splitta([90, 23],)
|--|--|--|--|--|--|--|-- entering       splitta([23],)
|--|--|--|--|--|--|--|--|-- entering    splitta([],)
|--|--|--|--|--|--|--|--|-- exiting     splitta([],)    returns ([], [])
|--|--|--|--|--|--|--|-- exiting        splitta([23],)  returns ([23], [])
|--|--|--|--|--|--|-- exiting   splitta([90, 23],)      returns ([90], [23])
|--|--|--|--|--|-- exiting      splitta([1, 90, 23],)   returns ([23, 1], [90])
|--|--|--|--|-- exiting         splitta([6, 1, 90, 23],)        returns ([90,
6], [23, 1])
|--|--|--|-- exiting    splitta([4, 6, 1, 90, 23],)     returns ([23, 1, 4],
[90, 6])
|--|--|-- exiting       splitta([9, 4, 6, 1, 90, 23],)  returns ([90, 6, 9],
[23, 1, 4])
|--|-- exiting  splitta([2, 9, 4, 6, 1, 90, 23],)       returns ([23, 1, 4, 2],
[90, 6, 9])
|-- exiting     splitta([5, 2, 9, 4, 6, 1, 90, 23],)    returns ([90, 6, 9, 5],
[23, 1, 4, 2])
```

```
 exiting        splitta([1, 5, 2, 9, 4, 6, 1, 90, 23],) returns ([23, 1, 4, 2,
1], [90, 6, 9, 5])
-------------------- Ending recursion --------------------
Num calls: 10
```

[36]:
```
([23, 1, 4, 2, 1], [90, 6, 9, 5])
```

[37]:
```
## Finalmente realizziamo mergeSort
@trace()
def merge_sort(L : list) -> list:
    if len(L) < 2:                          # [x] e [] sono già ordinate
        return L
    else:
        L1, L2 = splitta(L)                 # 2 sottoliste disordinate
        sorted1 = merge_sort(L1)            # ordino la prima
        sorted2 = merge_sort(L2)            # ordino la seconda
        return merge(sorted1, sorted2)      # le fondo
```

[38]:
```
# TODO: per essere un po' più efficienti si può lavorare con indici in liste␣
 ↪che non vengono modificate

merge_sort.trace(L)
```

```
------------------- Starting recursion -------------------
 entering       merge_sort([1, 5, 2, 9, 4, 6, 1, 90, 23],)
|-- entering    merge_sort([23, 1, 4, 2, 1],)
|--|-- entering merge_sort([1, 4, 23],)
|--|--|-- entering      merge_sort([23, 1],)
|--|--|--|-- entering   merge_sort([23],)
|--|--|--|-- exiting    merge_sort([23],)       returns [23]
|--|--|--|-- entering   merge_sort([1],)
|--|--|--|-- exiting    merge_sort([1],)        returns [1]
|--|--|-- exiting       merge_sort([23, 1],)    returns [1, 23]
|--|--|-- entering      merge_sort([4],)
|--|--|-- exiting       merge_sort([4],)        returns [4]
|--|-- exiting  merge_sort([1, 4, 23],) returns [1, 4, 23]
|--|-- entering merge_sort([2, 1],)
|--|--|-- entering      merge_sort([2],)
|--|--|-- exiting       merge_sort([2],)        returns [2]
|--|--|-- entering      merge_sort([1],)
|--|--|-- exiting       merge_sort([1],)        returns [1]
|--|-- exiting  merge_sort([2, 1],)     returns [1, 2]
|-- exiting     merge_sort([23, 1, 4, 2, 1],)   returns [1, 1, 2, 4, 23]
|-- entering    merge_sort([90, 6, 9, 5],)
|--|-- entering merge_sort([9, 90],)
|--|--|-- entering      merge_sort([9],)
|--|--|-- exiting       merge_sort([9],)        returns [9]
|--|--|-- entering      merge_sort([90],)
```

```
|--|--|-- exiting       merge_sort([90],)       returns [90]
|--|-- exiting  merge_sort([9, 90],)    returns [9, 90]
|--|-- entering merge_sort([5, 6],)
|--|--|-- entering      merge_sort([5],)
|--|--|-- exiting       merge_sort([5],)        returns [5]
|--|--|-- entering      merge_sort([6],)
|--|--|-- exiting       merge_sort([6],)        returns [6]
|--|-- exiting  merge_sort([5, 6],)     returns [5, 6]
|-- exiting     merge_sort([90, 6, 9, 5],)      returns [5, 6, 9, 90]
 exiting        merge_sort([1, 5, 2, 9, 4, 6, 1, 90, 23],)      returns [1, 1,
2, 4, 5, 6, 9, 23, 90]
-------------------- Ending recursion --------------------
Num calls: 17
```

[38]: `[1, 1, 2, 4, 5, 6, 9, 23, 90]`