# Table of Contents



# Fondamenti di Programmazione


**Andrea Sterbini**


lezione 17 - 2 dicembre 2022

# RECAP:

- Analisi OOP
- Ereditarietà
- Esempio di gerarchia di figure disegnate con Turtle
- Ricorsione e sue proprietà
- Esempi: fattoriale, fibonacci, GCD di Eulero

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

In [93]:
```
# mi costruisco una lista di valori casuali
import random
lista = random.choices(range(-10000,10001), k=10)
```

## Somma ricorsiva di una lista (svolta in uscita)

- **caso base**: se la lista è vuota la somma è 0
- **se la lista non è vuota**: sommiamo il primo elemento alla somma del resto della lista

In [94]:

```python
@trace()
def somma_ricorsiva(L):
    if L:           # se caso ricorsivo
        primo, *resto = L
        return primo + somma_ricorsiva(resto)
    else:
        return 0    # altrimenti caso base

somma = somma_ricorsiva.trace(lista)
somma2 = sum(lista)
somma, somma2
```

```
------------------- Starting recursion -------------------
 entering       somma_ricorsiva([3936, 2638, 5718, 6266, 7819, -8920, -3804, -9131, 9984,
-1098],)
|-- entering    somma_ricorsiva([2638, 5718, 6266, 7819, -8920, -3804, -9131, 9984, -109
8],)
|--|-- entering somma_ricorsiva([5718, 6266, 7819, -8920, -3804, -9131, 9984, -1098],)
|--|--|-- entering      somma_ricorsiva([6266, 7819, -8920, -3804, -9131, 9984, -1098],)
|--|--|--|-- entering   somma_ricorsiva([7819, -8920, -3804, -9131, 9984, -1098],)
|--|--|--|--|-- entering        somma_ricorsiva([-8920, -3804, -9131, 9984, -1098],)
|--|--|--|--|--|-- entering     somma_ricorsiva([-3804, -9131, 9984, -1098],)
|--|--|--|--|--|--|-- entering  somma_ricorsiva([-9131, 9984, -1098],)
|--|--|--|--|--|--|--|-- entering       somma_ricorsiva([9984, -1098],)
|--|--|--|--|--|--|--|--|-- entering    somma_ricorsiva([-1098],)
|--|--|--|--|--|--|--|--|--|-- entering somma_ricorsiva([],)
|--|--|--|--|--|--|--|--|--|-- exiting  somma_ricorsiva([],)    returns 0
|--|--|--|--|--|--|--|--|-- exiting     somma_ricorsiva([-1098],)       returns -1098
|--|--|--|--|--|--|--|-- exiting        somma_ricorsiva([9984, -1098],) returns 8886
|--|--|--|--|--|--|-- exiting   somma_ricorsiva([-9131, 9984, -1098],)  returns -245
|--|--|--|--|--|-- exiting      somma_ricorsiva([-3804, -9131, 9984, -1098],)   returns -
```

In [95]:

```
@trace()
def somma_ricorsiva_distruttiva(L):
    if L:           # se caso ricorsivo
        ultimo = L.pop()
        return ultimo + somma_ricorsiva_distruttiva(L)
    else:
        return 0    # altrimenti caso base

somma3 = somma_ricorsiva_distruttiva.trace(lista.copy())
# controlliamo che venga il risultato giusto
somma2, somma3
```

```
------------------- Starting recursion ------------------
 entering       somma_ricorsiva_distruttiva([3936, 2638, 5718, 6266, 7819, -8920, -3804,
-9131, 9984, -1098],)
|-- entering    somma_ricorsiva_distruttiva([3936, 2638, 5718, 6266, 7819, -8920, -3804,
-9131, 9984],)
|--|-- entering somma_ricorsiva_distruttiva([3936, 2638, 5718, 6266, 7819, -8920, -3804,
-9131],)
|--|--|-- entering      somma_ricorsiva_distruttiva([3936, 2638, 5718, 6266, 7819, -8920,
-3804],)
|--|--|--|-- entering   somma_ricorsiva_distruttiva([3936, 2638, 5718, 6266, 7819, -892
0],)
|--|--|--|--|-- entering        somma_ricorsiva_distruttiva([3936, 2638, 5718, 6266, 781
9],)
|--|--|--|--|--|-- entering     somma_ricorsiva_distruttiva([3936, 2638, 5718, 6266],)
|--|--|--|--|--|--|-- entering  somma_ricorsiva_distruttiva([3936, 2638, 5718],)
|--|--|--|--|--|--|--|-- entering       somma_ricorsiva_distruttiva([3936, 2638],)
|--|--|--|--|--|--|--|--|-- entering    somma_ricorsiva_distruttiva([3936],)
|--|--|--|--|--|--|--|--|--|-- entering somma_ricorsiva_distruttiva([],)
|--|--|--|--|--|--|--|--|--|-- exiting  somma_ricorsiva_distruttiva([],)       returns 0
```

## Somma iterativa di una lista

- accumuliamo la somma con un ciclo

In [96]:
```python
def somma_iter(L : list[int]) -> int:
    somma = 0
    N = len(L)
    for i in range(N):
        somma += L[i]
    return somma

somma_iter(lista), sum(lista)
```

Out[96]: (13408, 13408)

## Somma ricorsiva in avanti (simulando il ciclo)

- le variabili di stato, **somma**, **i** ed **N**
  - diventano argomenti della funzione
  - ad ogni step le aggiorniamo nella chiamata ricorsiva

In [97]:
```python
@trace()
def _somma_ric_avanti(L : list[int], i : int,
                      N : int, somma : int) -> int :
    if i==N:
        return somma
    else:
        return _somma_ric_avanti(L, i+1, N, somma + L[i]) # AGGIORNAMENTO

def somma_ric_avanti(L):
    return _somma_ric_avanti(L, 0, len(L), 0)

somma_ric_avanti(lista)

```

Out[97]: 13408

## Stampa di una lista

- **in avanti** (prima del passo ricorsivo)
- **indietro** (dopo il passo ricorsivo)
- **avanti poi indietro** (prima e dopo)

Analisi:

- la lista vuota non stampa nulla (caso base)
- una volta stampato il primo elemento va stampato il resto (passo ricorsivo)

In [100]:
```python
# per scandire e stampare una lista in avanti
def stampa_in_avanti(L):
    if L:
        primo, *resto = L
        print(primo, end=' ')
        stampa_in_avanti(resto)

stampa_in_avanti(lista)
print()
print(lista)
```

```
3936 2638 5718 6266 7819 -8920 -3804 -9131 9984 -1098
[3936, 2638, 5718, 6266, 7819, -8920, -3804, -9131, 9984, -1098]
```

In [102]:
```python
# per scandire e stampare una lista a rovescio
def stampa_dalla_fine(L):
    if L:
        primo, *resto = L
        stampa_dalla_fine(resto)
        print(primo, end=' ')

stampa_dalla_fine(lista)
print()
print(lista)
# TODO per scandire una lista sia in avanti che a rovescio
#      basta stampare sia prima della ricorsione che dopo
```

```
-1098 9984 -9131 -3804 -8920 7819 6266 5718 2638 3936
[3936, 2638, 5718, 6266, 7819, -8920, -3804, -9131, 9984, -1098]
```

# MergeSort (ordinamento per fusione)

### osservazione: se due liste sono 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
  - il resto della soluzione è la fusione del resto delle due liste

- **convergenza**: almeno un elemento va a posto per ogni chiamata ricorsiva
- **caso base**: una delle due liste è vuota

In [103]:

```python
@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
# 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]
```

## E se ho una lista <u>disordinata</u> e la voglio ordinare? (MergeSort)

- la posso spezzare in due liste <u>disordinate</u> (**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

In [104]:
```
## per spezzare in 2 posso usare una slice
L = [1, 5, 2, 9, 4, 6, 1, 90 ]

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

L1, L2
```

Out[104]:
```
([1, 5, 2, 9], [4, 6, 1, 90])
```

In [105]:

```python
## 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],)
|-- entering    splitta([5, 2, 9, 4, 6, 1, 90],)
|--|-- entering splitta([2, 9, 4, 6, 1, 90],)
|--|--|-- entering      splitta([9, 4, 6, 1, 90],)
|--|--|--|-- entering   splitta([4, 6, 1, 90],)
|--|--|--|--|-- entering        splitta([6, 1, 90],)
|--|--|--|--|--|-- entering     splitta([1, 90],)
|--|--|--|--|--|--|-- entering  splitta([90],)
|--|--|--|--|--|--|--|-- entering       splitta([],)
|--|--|--|--|--|--|--|-- exiting        splitta([],)    returns ([], [])
|--|--|--|--|--|--|-- exiting   splitta([90],)  returns ([90], [])
|--|--|--|--|--|-- exiting      splitta([1, 90],)       returns ([1], [90])
|--|--|--|--|-- exiting         splitta([6, 1, 90],)    returns ([90, 6], [1])
|--|--|--|-- exiting    splitta([4, 6, 1, 90],) returns ([1, 4], [90, 6])
|--|--|-- exiting       splitta([9, 4, 6, 1, 90],)      returns ([90, 6, 9], [1, 4])
|--|-- exiting  splitta([2, 9, 4, 6, 1, 90],)   returns ([1, 4, 2], [90, 6, 9])
|-- exiting     splitta([5, 2, 9, 4, 6, 1, 90],)        returns ([90, 6, 9, 5], [1, 4,
2])
```

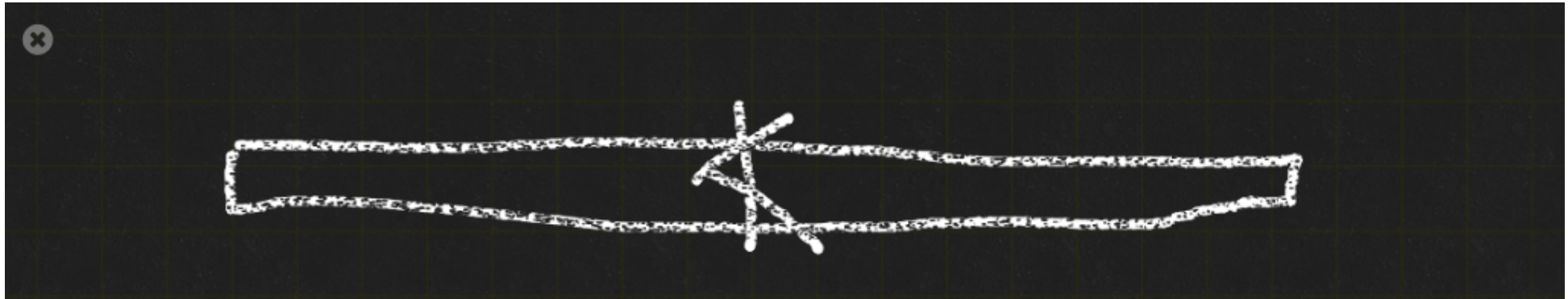
In [106]:

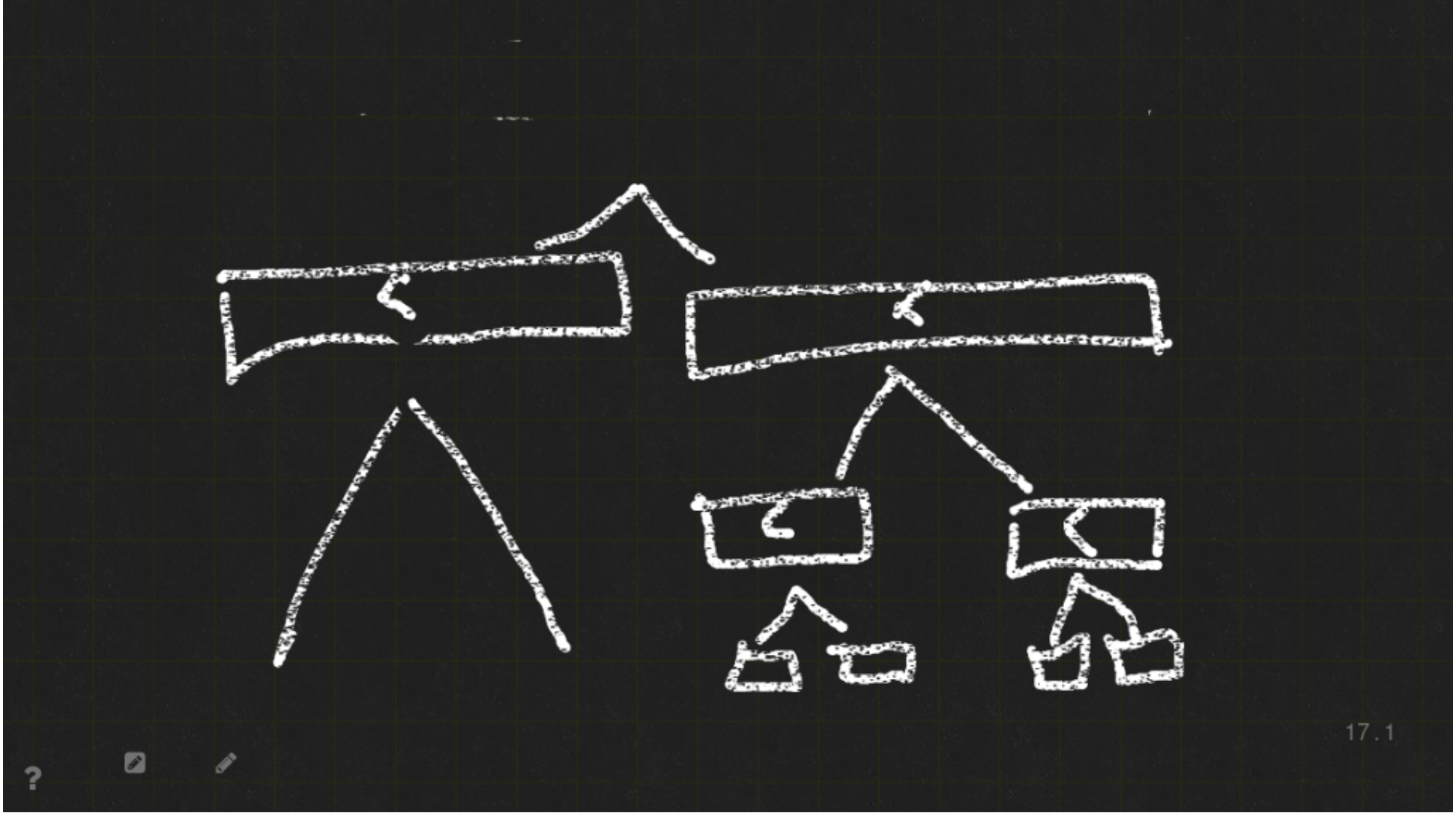
```python
## Finalmente realizziamo mergeSort
@trace()
def merge_sort(L : list) -> list:
    if len(L) < 2:              # [x] e [] sono già ordinate
        return L
    else:
        L1, L2 = splitta(L)             # 2 sottoliste disordinate
        sorted1 = merge_sort(L1)        # ordino la prima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

# TODO: per essere un po' più efficienti si può lavorare con indici in liste che non

merge_sort.trace(L)
```

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






## Alberi binari

- si parte da un nodo "radice" che non ha "padri"
- ogni nodo piò avere fino a 2 "figli"
- i nodi senza figli sono le "foglie"
- ogni nodo ha un solo "padre"

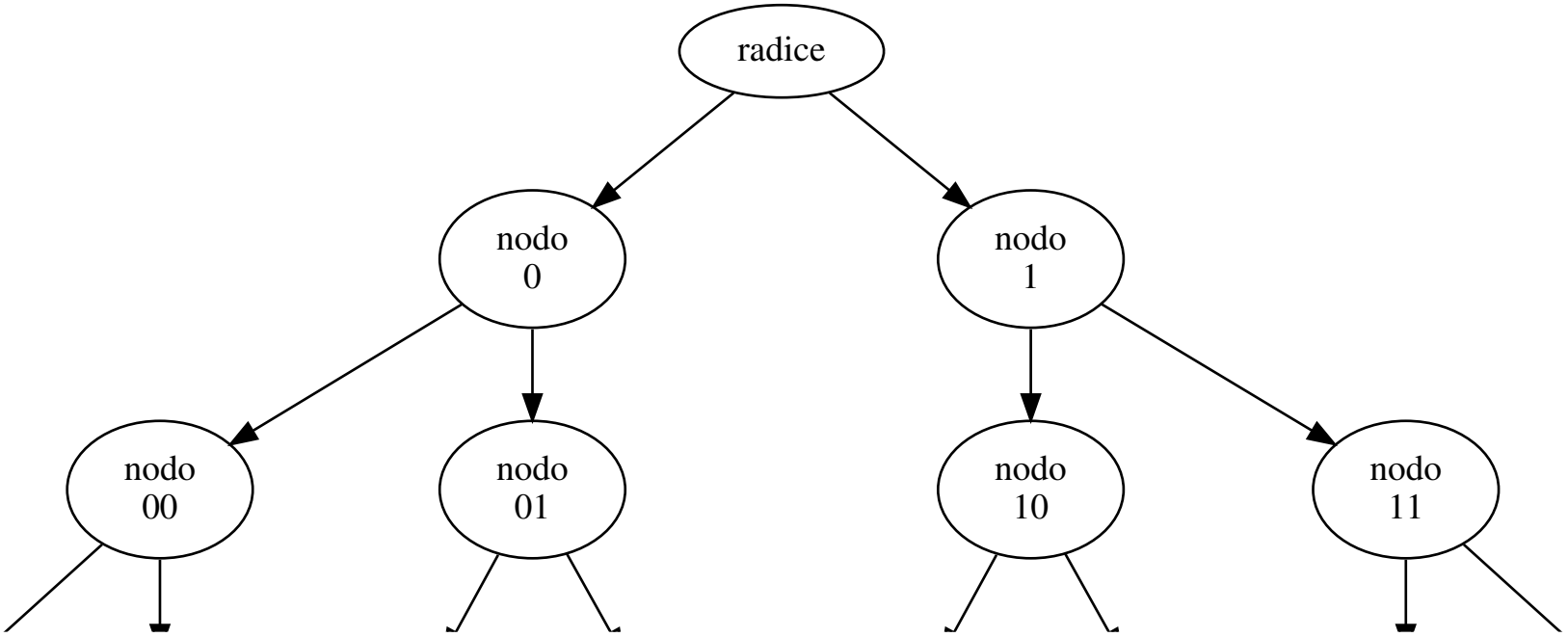
In [13]:

```
import graphviz
T = graphviz.Digraph()
T.body.append('''
    radice -> "nodo\n0" -> "nodo\n00" -> "foglia\n000"
    radice -> "nodo\n1" -> "nodo\n10" -> "foglia\n100"
    "nodo\n0" -> "nodo\n01" -> "foglia\n010"
    "nodo\n1" -> "nodo\n11" -> "foglia\n110"
    "nodo\n00" -> "foglia\n001"
    "nodo\n10" -> "foglia\n101"
    "nodo\n01" -> "foglia\n011"
    "nodo\n11" -> "foglia\n111"
''')
```

In [14]:

```
T
```

Out[14]:



In [107]:
```python
# 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(100, n0, n1)
r
```

Out[107]: NodoBinario(100)

## Stampa di un albero

**con visita in PREordine (la radice prima dei sottoalberi)**

In [108]:

```python
# radice è un nodo oppure None
# per stampare indentato passo un argomento 'livello'
# e lo incremento ogni volta che scendo in un sottoalbero
def stampa_PRE(radice : NodoBinario, livello : int =0) -> None :
    print('|--'*livello, radice)
    if radice:
        stampa_PRE(radice._sx, livello+1)
        stampa_PRE(radice._dx, livello+1)

stampa_PRE(r)
```

```
 NodoBinario(100)
|-- NodoBinario(0)
|--|-- NodoBinario(10)
|--|--|-- None
|--|--|-- None
|--|-- None
|-- NodoBinario(1)
|--|-- None
|--|-- NodoBinario(11)
|--|--|-- None
|--|--|-- None
```

**con visita in POSTordine (la radice DOPO i sottoalberi)**

In [109]:
```
def stampa_POST(radice, livello=0):
    if radice:
        stampa_POST(radice._sx, livello+1)
        stampa_POST(radice._dx, livello+1)
    print('|--'*livello, radice)

stampa_POST(r)
```

```
|--|--|-- None
|--|--|-- None
|--|-- NodoBinario(10)
|--|-- None
|-- NodoBinario(0)
|--|-- None
|--|--|-- None
|--|--|-- None
|--|-- NodoBinario(11)
|-- NodoBinario(1)
NodoBinario(100)
```

**con visita INordine (la radice TRA i sottoalberi)**

In [70]:
```
def stampa_IN(radice, livello=0):
    if radice:
        stampa_IN(radice._sx, livello+1)
    print('|--'*livello, radice)
    if radice:
        stampa_IN(radice._dx, livello+1)

stampa_IN(r)
```

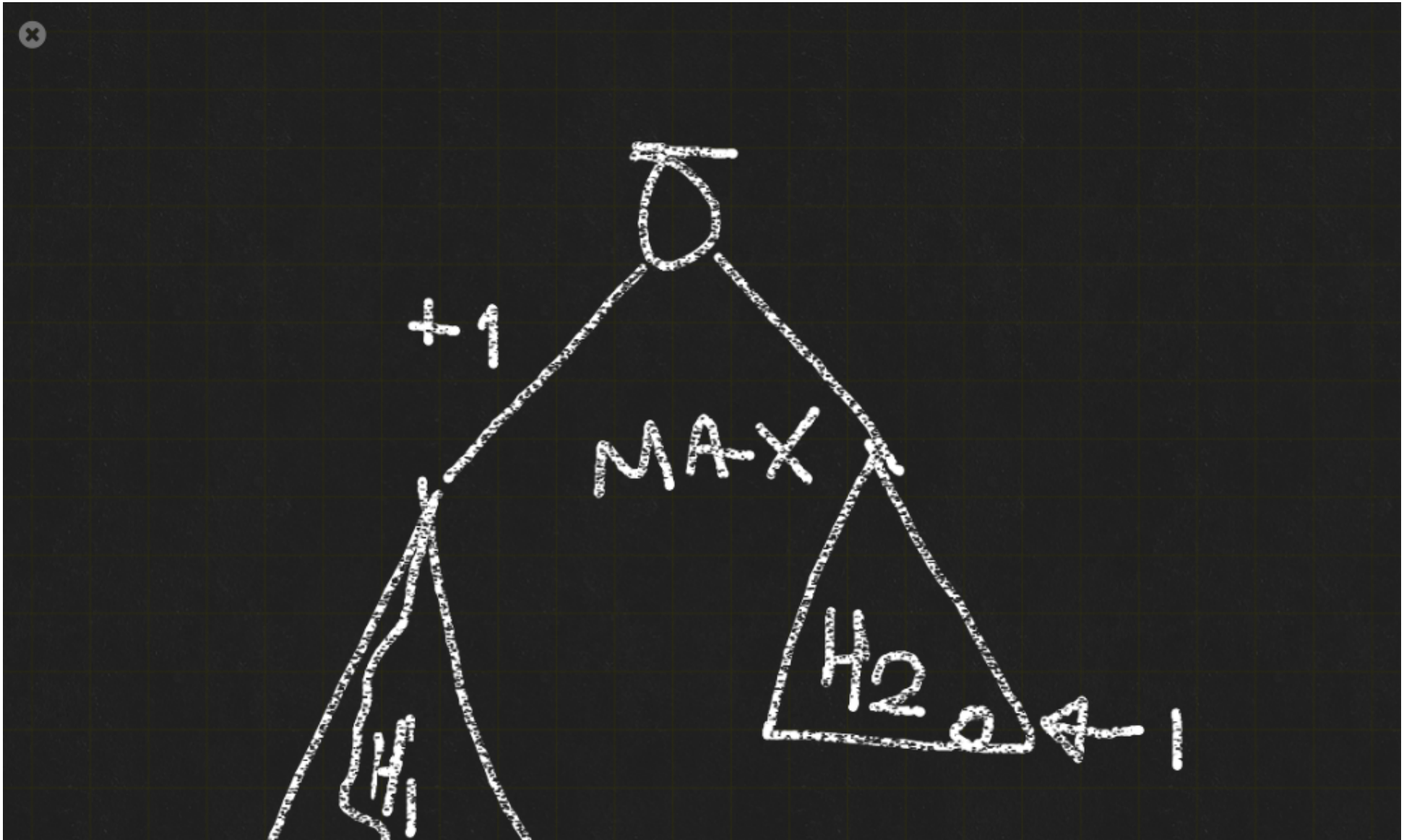
```
|--|--|-- None
|--|-- NodoBinario(10)
|--|--|-- None
|-- NodoBinario(0)
|--|-- None
NodoBinario(100)
|--|-- None
|-- NodoBinario(1)
|--|--|-- None
|--|-- NodoBinario(11)
|--|--|-- None
```

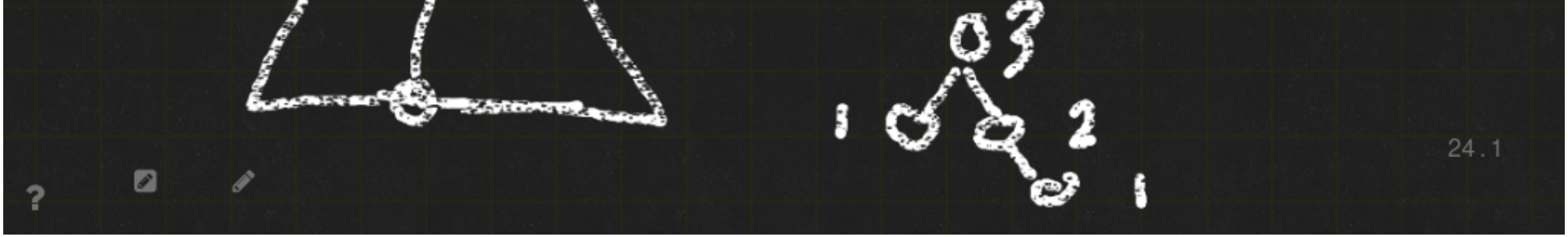
## Calcolo della profondità/altezza di un albero (in uscita)

- Una foglia ha altezza 1 (caso base)
- un nodo ha altezza 1 + la massima altezza dei sottoalberi (passo ricorsivo + composizione)

Oppure:

- **None** ha altezza 0




24.1

In [112]:

```python
@trace()
def altezza(radice):
    if radice is None:
        return 0
    H_sx = altezza(radice._sx)
    H_dx = altezza(radice._dx)
    return 1 + max(H_sx, H_dx)

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

```
------------------- Starting recursion -------------------
 entering       altezza(NodoBinario(100),)
|-- 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(100),)      returns 3
-------------------- Ending recursion --------------------
Num calls: 11
 NodoBinario(100)
|-- NodoBinario(0)
|--|-- NodoBinario(10)
|--|--|-- None
|--|--|-- None
|--|-- None
|-- NodoBinario(1)
|--|-- None
|--|-- NodoBinario(11)
|--|--|-- None
```

```
|--|--|-- None
```

## 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

In [113]:

```
@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(100),)
|-- 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)
```

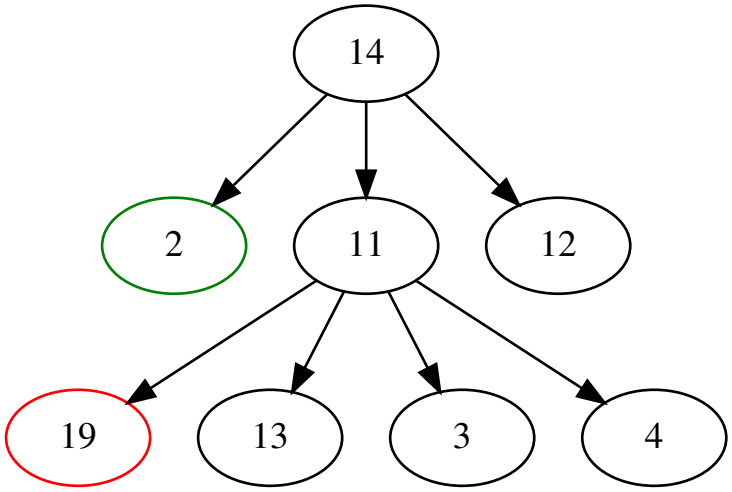
## Alberi N-ari (con numero indefinito di figli)

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

In [117]:
```
G3 = graphviz.Digraph()
G3.body.append('''
    2 [color=green]
    19 [color=red]
    14 -> 11
    14 -> 12
    14 -> 2
    11 -> 19
    11 -> 13
    11 -> 3
    11 -> 4
''')
```

In [118]:
```
G3
```

Out[118]:



In [114]:
```python
# utility che mi permette di aggiungere con %%add_to dei metodi definiti in celle sep
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)"
```

**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 0

In [118]:
```python
%%add_to NodoNario

## metodo per calcolare l'altezza del nodo
def altezza(self):
    # se non ci sono figli si torna 0
    return max( [son.altezza()+1 for son in self._sons], default=1 )
```

In [116]:
```python
n19 = NodoNario(19)
n2 = NodoNario(2)
n3 = NodoNario(3)
n4 = NodoNario(4)
n13 = NodoNario(13)
n12 = NodoNario(12)
n11 = NodoNario(11, [n3, n4, n13, n19])
n14 = NodoNario(14, [n2, n11, n12])
```

In [119]:
```
1 n14.altezza()
```

Out[119]: 3

### stampa (in preordine) di un nodo e dei figli

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

In [120]:
```
1 %%add_to NodoNario
2 # metodo per stampare l'albero
3 def stampa(self, livello=0):
4     indent = '|--'*livello
5     print(indent, self)
6     for son in self._sons:
7         son.stampa(livello+1)
```

In [121]:
```
1 n14.stampa()
```

```
 NodoNario(14, 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 valori dei sottoalberi

In [125]:
```
%%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 [129]:
```
n14.massimo2()
```

Out[129]: 19

In [130]:
```
%%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:
        max_corrente = self._value
    else:
        max_corrente = max(max_corrente, self._value)
    for son in self._sons:
        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
```

In [132]:
```
n14.massimo3()
```

Out[132]: 19

In [88]:
```
n14.massimo(), n14.massimo2(), n14.massimo3()
```

Out[88]: (19, 19, 19)

## 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 valori dei sottoalberi

In [137]:
```
%%add_to NodoNario

def max_node(self):
    M = [self] + [s.max_node() for s in self._sons]
    #print(M)
    return max(M, key=lambda x: x._value)
def min_node(self):
    M = [self] + [s.min_node() for s in self._sons]
    #print(M)
    return min(M, key=lambda x: x._value)


```

In [138]:
```
n14.max_node(), n14.min_node()
```

Out[138]:
```
(NodoNario(19, 0 sons), NodoNario(2, 0 sons))
```

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

In [141]:
```
%%add_to NodoNario
def depth_of_max(self,livello=0):
    M = [(self._value, livello)] + [ son.depth_of_max(livello+1)
                                     for son in self._sons]
    return max(M, key=lambda x: x[0])

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._son
    return min(M, key=lambda x: x[0])
```

In [140]:
```
1 n14.depth_of_max(), n14.depth_of_min()
```

Out[140]:
```
((19, 2), (2, 1))
```