λ Programació i Algorísmia 2

Jordi Delgado, Dept. CS, UPC
jdelgado@cs.upc.edu

(basat en **cs61**, de l'U.Berkeley)

S'agraeix en Jordi Cortadella l'autorització per fer servir material de les seves transparències d'AP2

De què va PA2?

- Aquest és un curs sobre *Programació*, que hom podria dir que és l'*art* I la *ciència* de construir artefactes (*programes*) que realitzen computacions i/o interactuen amb el món físic
- Cal aprendre un *Llenguatge de Programació* (LP), Python en el nostre cas, però per programar cal aprendre moltes més coses:
  - Disseny del que fan els programes
  - Anàlisi de l'eficiència dels programes
  - Verificació del funcionament correcte
  - Gestió de la seva complexitat
- Aquest curs, però, és sobre les ***idees generals*** en programació. Així, podreu aplicar el que apreneu a pràcticament qualsevol llenguatge de programació

Construint Abstraccions amb Dades

Filosofia

- Antigament, hom describia els programes com a jerarquies d'accions: *descomposició procedimental* (*procedural decomposition*).
- Des dels anys 70, però, l'èmfasi va passar a les *dades* sobre les que operaven les funcions.
- Un *Tipus Abstracte de Dades* (TAD) representa un determinat conjunt d'entitats i, sobre tot, les *operacions* que hom pot realitzar sobre aquestes entitats.
- Podem organitzar un programa al voltant dels TADs que fa servir.
- Per a cada TAD es defineix una *interfície* que descriu quines operacions estan disponibles pels usuaris (*clients*) d'aquell TAD (l'*API*, *Application Programmer's Interface*).
- Tipicament, la interfície consisteix en funcions.
- La col·lecció d'especificacions (sintàctiques i semàntiques) d'aquestes funcions constitueix l'*especificació del TAD*.
- ELs TADs s'anomenen *abstractes* perquè *no cal conèixer la seva implementació per fer-los servir*. Només cal conèixer la interfície.

Classificació

- Com a exemple de TAD ja vam veure a PA1 el tipus `parell`, implementat amb funcions.
- Les funcions que formen part de la interfície d'un TAD s'acostumen a classificar en:
  - Funcions *constructores*: Creen noves entitats del TAD corresponent. Per exemple, la funció `parell`.
  - Funcions d'*accés* o *consultores* (*getters*) Retornen propietats de les entitats del TAD. Per exemple, `dret` i `esquerre`.
  - Funcions *modificadores* (*setters*): modifiquen les entitats del TAD. Per exemple, `canvia_esquerre` i `canvia_dret`.

Nombres Racionals

- En el llibre *Composing Programs* trobem els nombres racionals com a exemple de TAD:

```
def construeix_racional(n,d):                   # Constructora
    """ Retorna el nombre racional n/d, suposant n i d són enters i d != 0 """

def numerador(r):                               # Consultora
    """ Retorna el numerador del nombre racional r """

def denominador(r):                             # Consultora
    """ Retorna el denominador del nombre racional r """
```

- Les dues darreres definicions pretenen que r és realment un nombre racional.
- Però, des d'aquest punt de vista, les definicions de `numerador` i `denominador` són problemàtiques. Per què?

Una especificació millor

- El problema és que el *numerador* (o *denominador*) d'un nombre racional no està ben definit.
- Si `construeix_racional` produís nombres racionals, el retornat per `construeix_racional(1,2)` i per `construeix_racional(2,4)` haurien de ser idèntics. Igual que `construeix_racional(-1,1)` i per `construeix_racional(1,-1)`.
- Així doncs, una millor especificació seria:

```
def numerador(r):                                    # Consultora
    """ Retorna el numerador del nombre racional r amb valor
        absolut més petit """

def denominador(r):                                  # Consultora
    """ Retorna el denominador del nombre racional r amb valor > 0
        més petit """
```

Operacions addicionals

- Els racionals, sent nombres, haurien de disposar d'operacions numèriques, i d'altres:

```
def suma_racionals(p,q):
    """ Retorna la suma dels nombres racionals p i q """

def multiplica_racionals(p,q):
    """ Retorna el producte dels nombres racionals p i q """

def str_racionals(r):
    """ Return R com a string expressant una fracció racional.
        >>> str_racional(construeix_racional(2, 4))
        1/2
        >>> str_racional(construeix_racional(3, 1))
        3       """

def igual_racionals(p,q):
    """ Retorna True sii els nombres racionals p i q són iguals"""
```

Utilitzant els racionals

- Ara podem escriure funcions que (a banda de la sintaxi) manipulen els racionals com podrien manipular qualsevol altra mena de nombres:

```
def nombre_harmonic_aprox(n):
    """ Retorna una aproximació a 1 + 1/2 + 1/3 + ... + 1/n """
    s = 0.0
    for k in range(1,n+1):
        s = s + 1/k
    return s

def nombre_harmonic_exacte(n):
    """ Retorna 1 + 1/2 + 1/3 + ... + 1/n com a nombre racional """
    s = construeix_racional(0,1)
    for k in range(1,n+1):
        s = suma_racionals(s,construeix_racional(1,k))
    return s
```

- Més endavant veurem com "*arreglar*" la sintaxi.

Representant els racionals

- Podem representar els nombres racionals fent servir els parells que hem vist més enrere, però també llistes o tuples de Python.

```
from math import gcd

def construeix_racional(n,d):
    """ Retorna el nombre racional n/d, suposant n i d són enters i d != 0 """
    g = gcd(n, d)
    n //= g; d //= g
    return (n, d)      # Fem servir tuples

def numerador(r):
    """ Retorna el numerador del nombre racional r amb valor absolut més petit """
    return r[0]

def denominador(r):
    """ Retorna el denominador del nombre racional r amb valor > 0 més petit """
    return r[1]
```

Implementant operacions addicionals...

- Ara podem implementar les operacions addicionals, per exemple la suma:

```
from math import gcd

def construeix_racional(n,d):
    """ Retorna el nombre racional n/d, suposant n i d són enters i d != 0 """
    g = gcd(n, d)
    n //= g; d //= g
    return (n, d)
...
```

```
def suma_racionals(p,q):
    """ Retorna la suma dels nombres racionals p i q """
    n0, d0 = p
    n1, d1 = q
    n = n0 * d1 + n1 * d0
    d = d0 * d1
    g = gcd(n, d)
    n //= g; d //= g
    return (n, d)
```

- Cap comentari?

Violació de l'abstracció i DRY

- Ja que hem creat una abstracció (`construeix_racional`, `numerador`, `denominador`) cal fer-la servir:
  - Canvis posteriors en la implementació no afectaran a altres funcions
  - El codi queda més clar, ja que noms ben escollits a l'API fan clara la intenció del programador.
- Millors implementacions de les operacions addicionals serien:

```
def suma_racionals(p,q):
    n1 = numerador(p) * denominador(q)
    n2 = numerador(q) * denominador(p)
    d  = denominador(p) * denominador(q)
    return construeix_racional(n1 + n2, d)
```

```
def producte_racionals(p,q):
    n = numerador(p) * numerador(q)
    d = denominador(p) * denominador(q)
    return construeix_racional(n, d)
```

Violació de l'abstracció i DRY

- Ja que hem creat una abstracció (`construeix_racional`, `numerador`, `denominador`) cal fer-la servir:
  - Canvis posteriors en la implementació no afectaran a altres funcions
  - El codi queda més clar, ja que noms ben escollits a l'API fan clara la intenció del programador.
- Millors implementacions de les operacions addicionals serien:

```python
def str_racionals(r):
    n = numerador(r)
    d = denominador(r)
    return str(n) if d == 1 else f"{n}/{d}"

def igual_racionals(p,q):
    n1 =   numerador(p) * denominador(q)
    n2 = denominador(p) *   numerador(q)
    return (n1 == n2)
```

Capes d'Abstracció

- Així doncs, podem dividir les operacions sobre racionals d'aquesta manera:

Primitives	`( . . . )`	`...[. . .]`
Representació	`construeix_racional`	`numerador` `denominador`
Operacions derivades	`suma_racionals` `igual_racionals`	`producte_racionals` `str_racionals`
Programa usuari	`nombre_harmonic_exacte`	

- Les linies representen *barreres d'abstracció*.
- Capes per sobre d'una barrera no utilitzen res que estigui per sota.
- Capes per sota d'una barrera només fan servir les operacions de la capa immediatament superior. Direm que són *exportades*.

Violació de l'abstracció

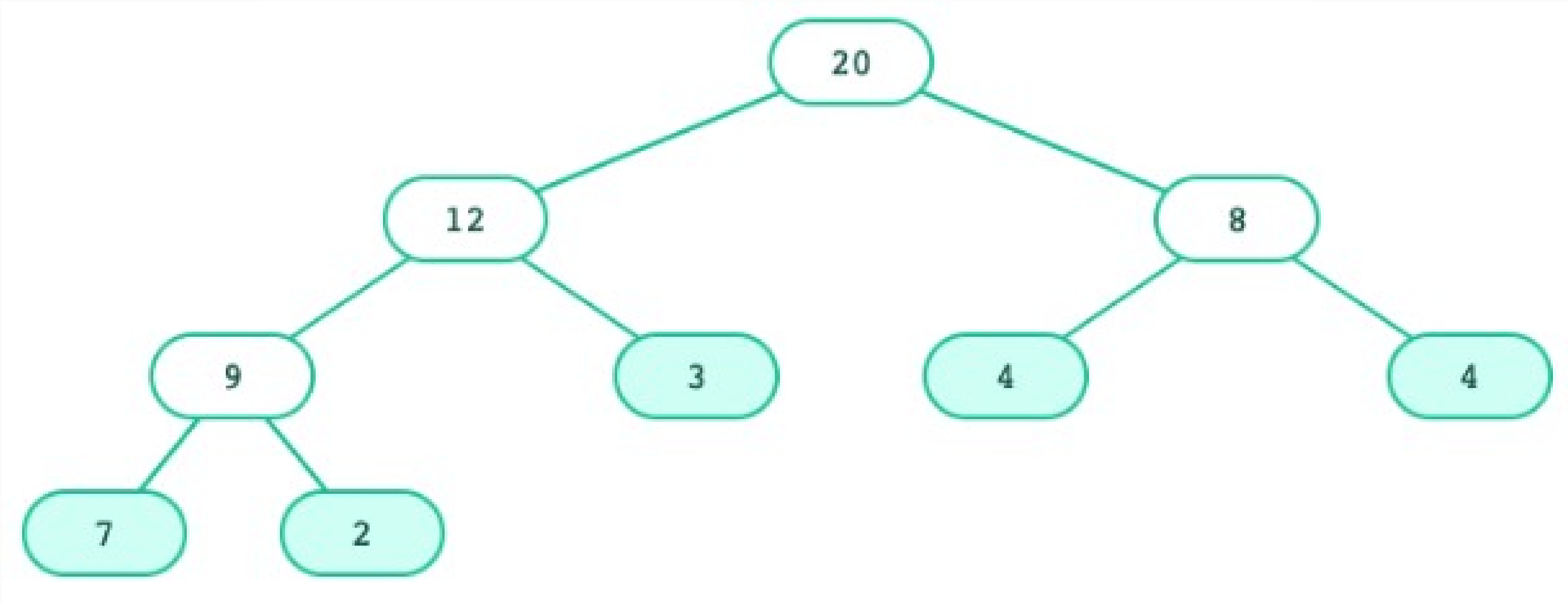
- Així, a més de violar el principi de DRY, la implementació:

```
def suma_racionals(p,q):
    """ Retorna la suma dels nombres racionals p i q """
    n0, d0 = p        # LLEIG!!
    n1, d1 = q        # LLEIG!!
    n = n0 * d1 + n1 * d0
    d = d0 * d1
    g = gcd(n, d)
    n //= g; d //= g
    return (n, d)     # LLEIG!!
```

viola la barrera d'abstracció per sobre de `suma_racionals`, fent servir detalls de la implementació en lloc d'utilitzar les operacions exportades.

Arbres

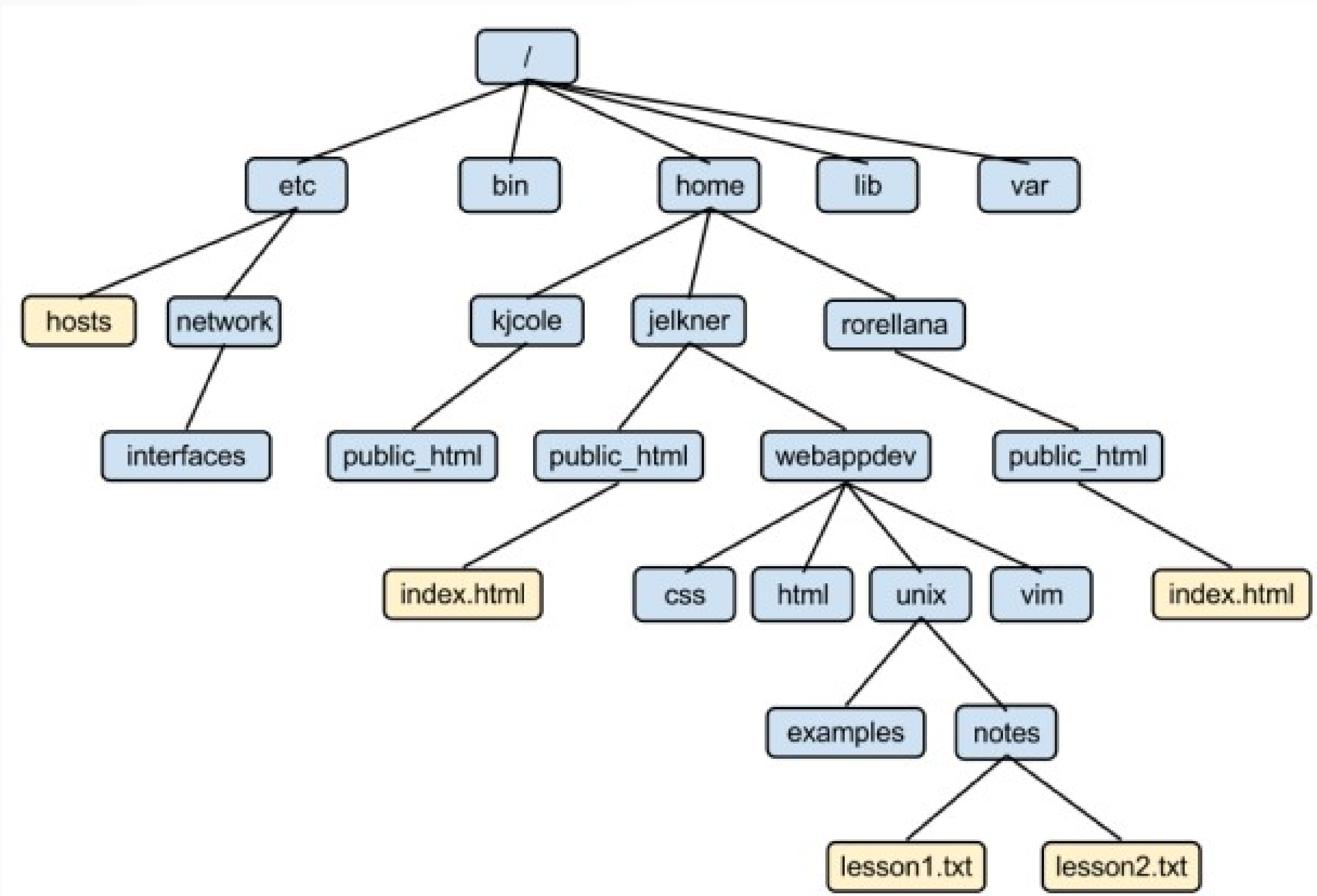
Arbres

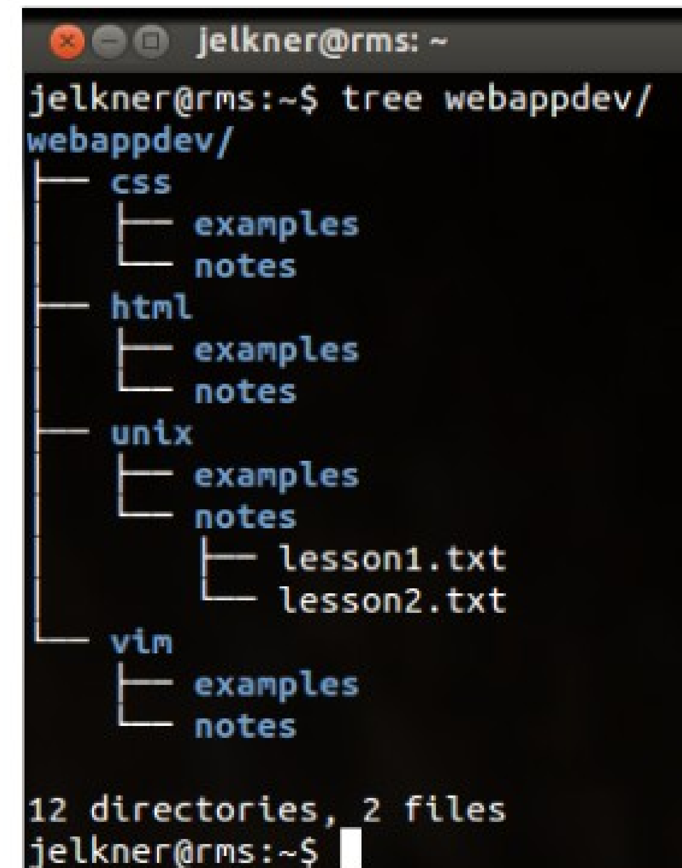


- Cada oval s'anomena ***node*.**
- El node de dalt de tot s'anomena ***arrel***.
- Cada node és l'arrel d'un altre arbre (anomenat ***subarbre***). Els nodes immediatament per sota són ***fills***. De vegades també s'anomena ***fills*** als subarbres que tenen com a arrel els nodes immediatament per sota.
- Els nodes sense fills s'anomenen ***fulles***. La resta són nodes ***interiors***.
- Generalment, cada node té una ***etiqueta***.

Per a què serveixen els arbres? Són a tot arreu!

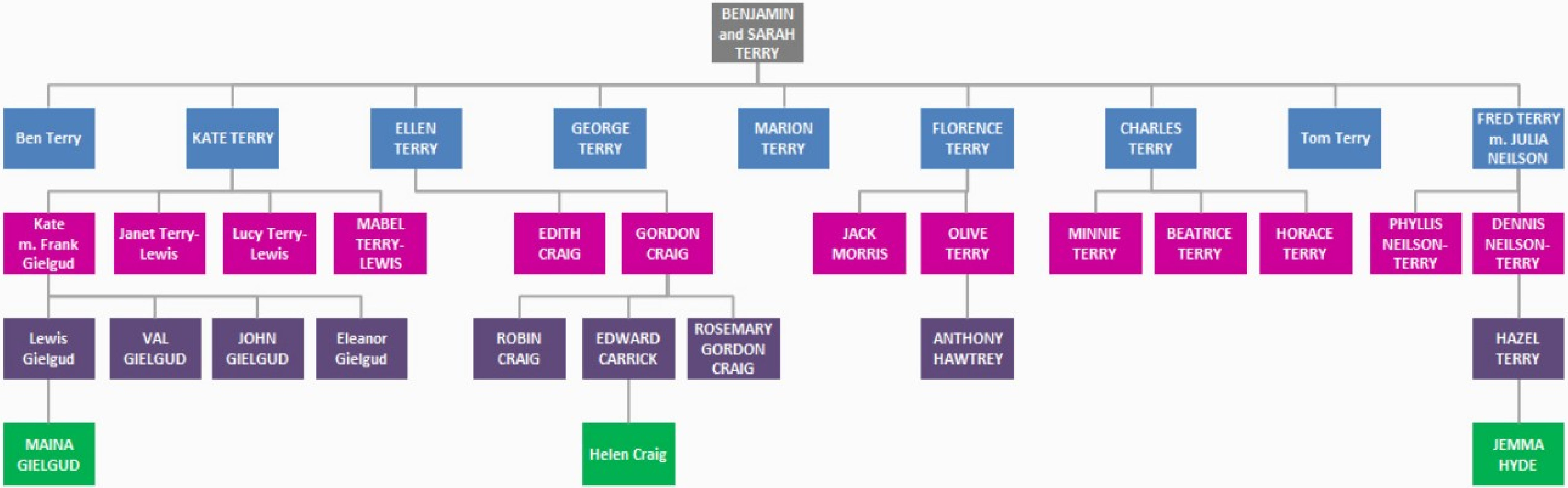
- Sistemes de fitxers…





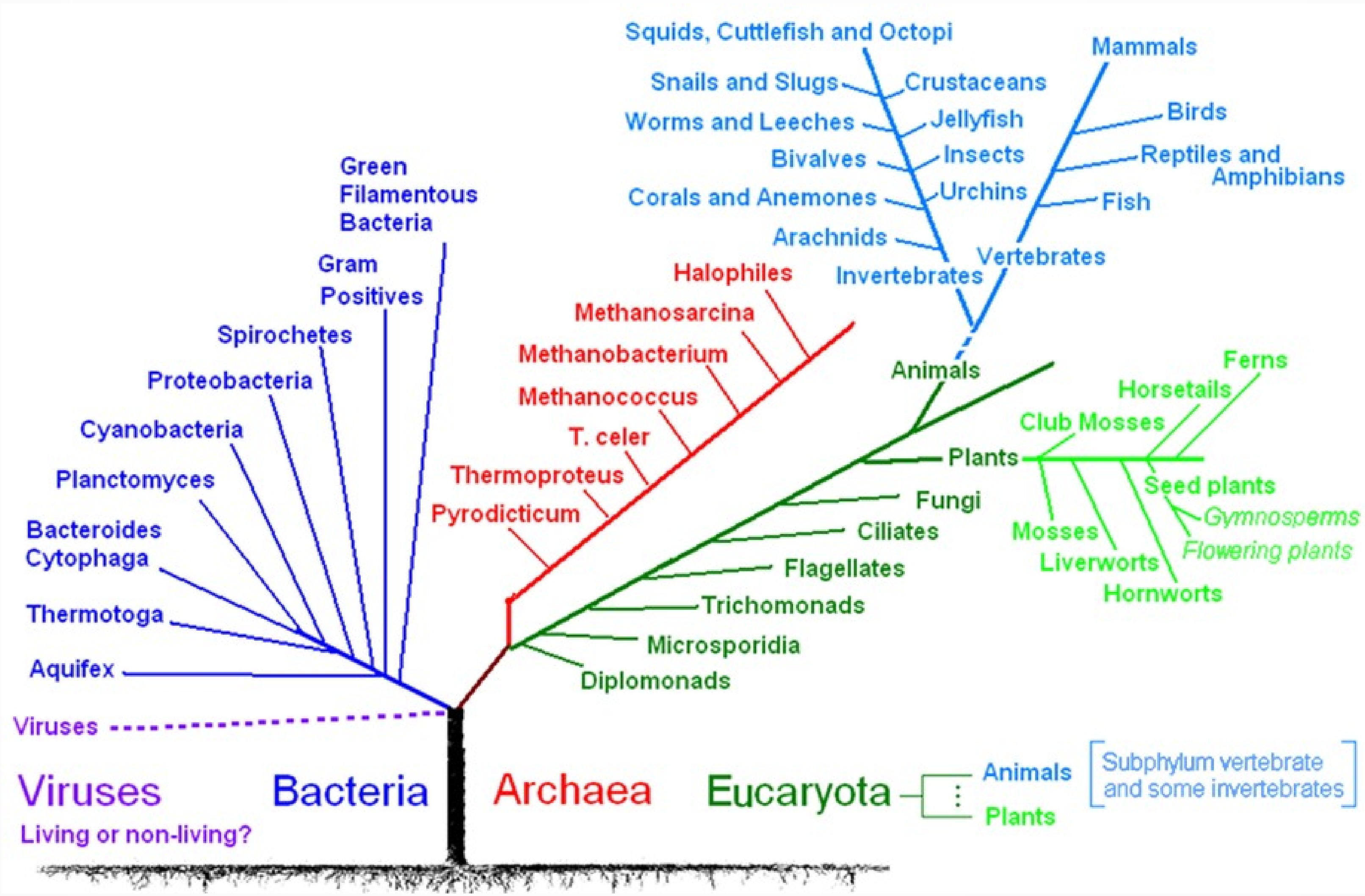
Per a què serveixen els arbres? Són a tot arreu!

- Arbres genealògics...



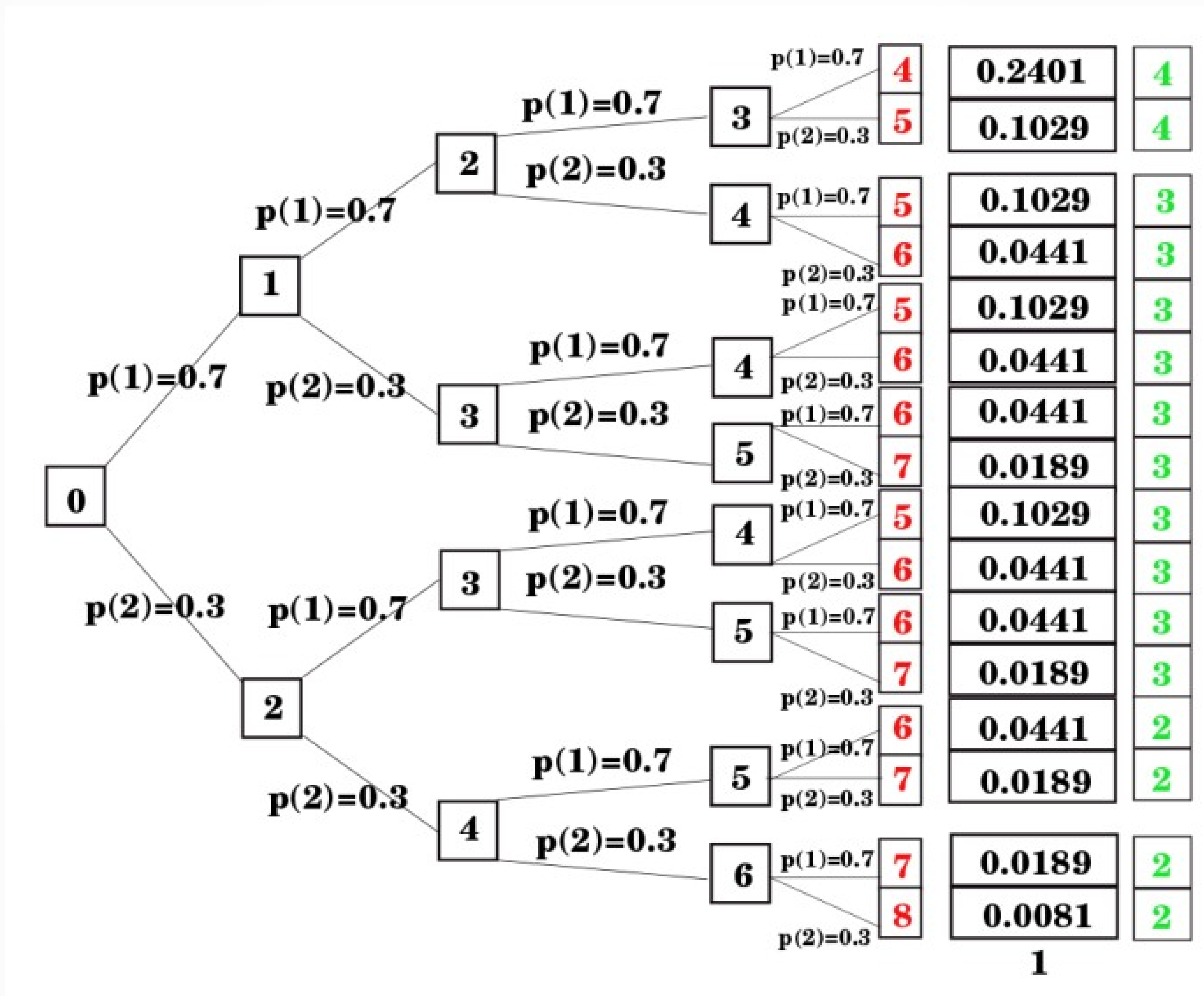
Per a què serveixen els arbres? Són a tot arreu!

- L'arbre de la vida... (`www.greennature.ca`)



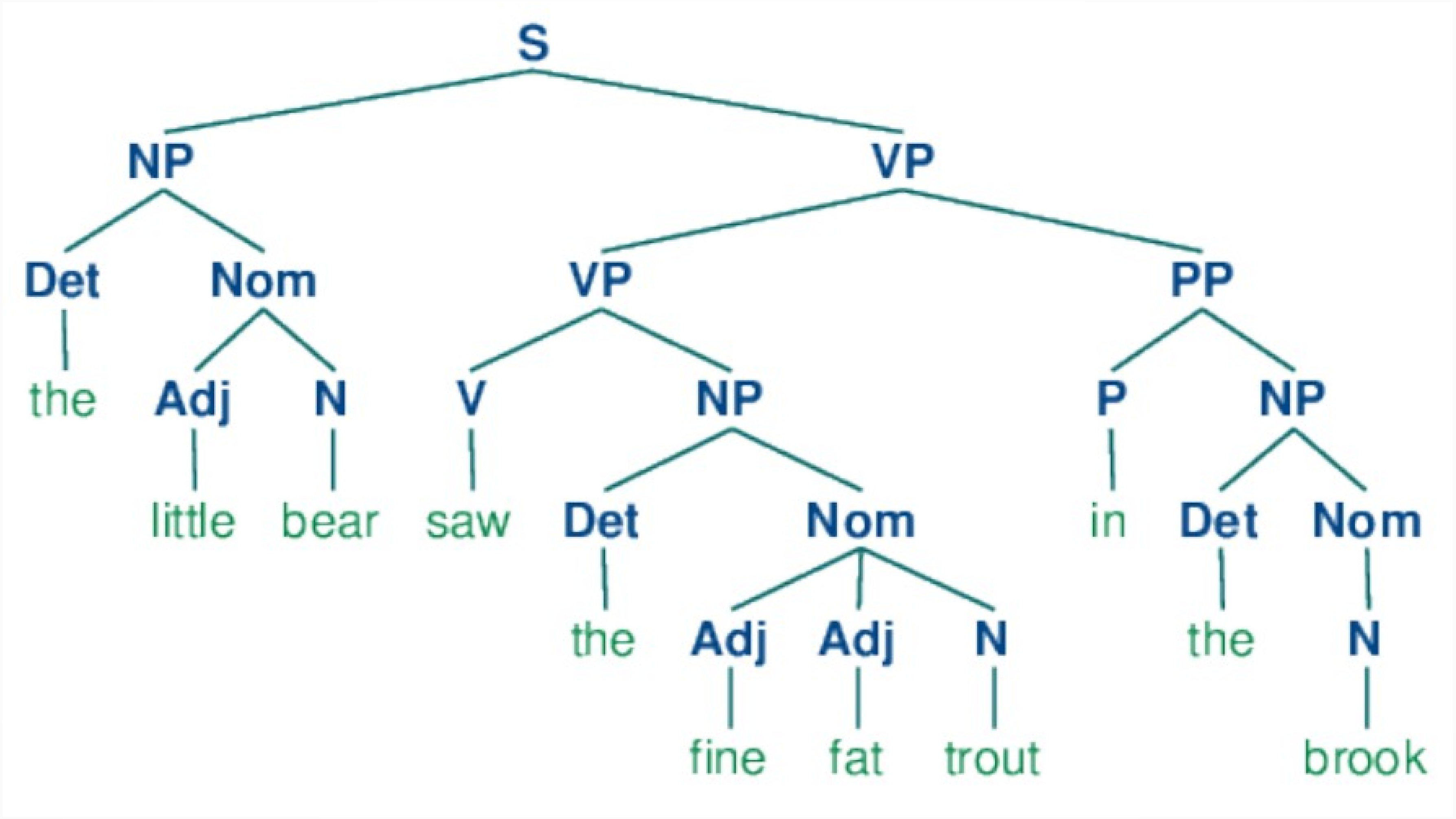
Per a què serveixen els arbres? Són a tot arreu!

- Arbres de probabilitat…



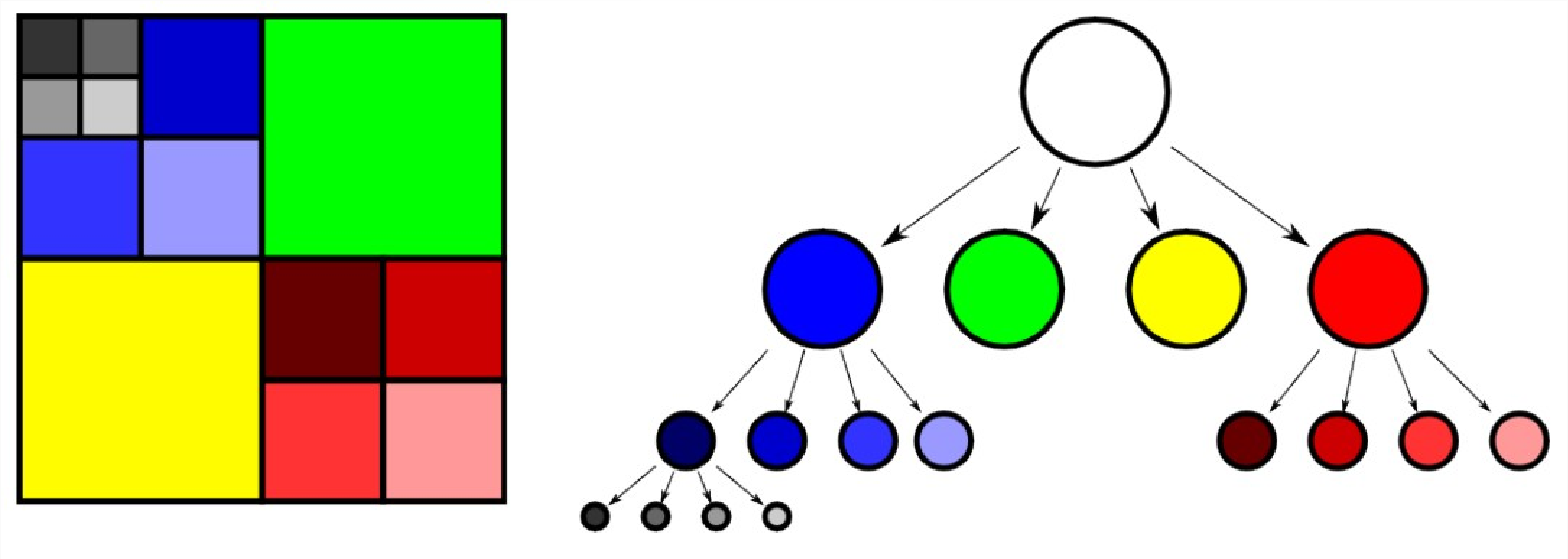
Per a què serveixen els arbres? Són a tot arreu!

- Arbres sintàctics…



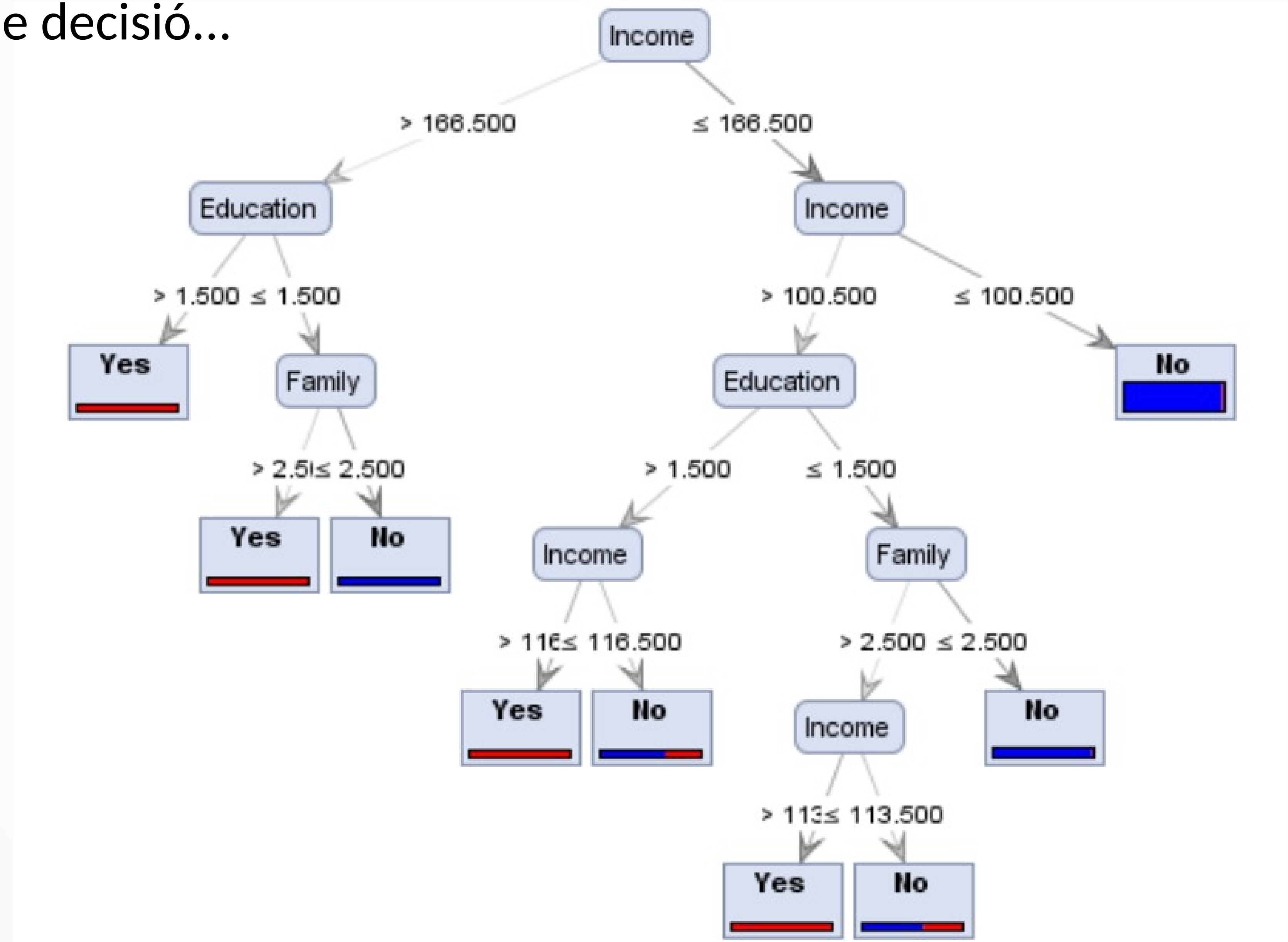
Per a què serveixen els arbres? Són a tot arreu!

- Representació d'imatges (*quadtrees*)...



Per a què serveixen els arbres? Són a tot arreu!

- Arbres de decisió…



Arbres

- Definirem una constructora, dues consultores i una funció addicional

```
def construeix_arbre(v,f):                          # Constructora
    """ Retorna un arbre amb etiqueta v a arrel i f com a
        fills de l'arrel """

def valor_arbre(a):                                 # Consultora
    """ Retorna l'etiqueta de l'arrel de l'arbre a """

def fills_arbre(a):                                 # Consultora
    """ Retorna els fills de l'arbre a """

def es_fulla(a):
    """ Retorna True si a és un node fulla """
```

- Com ho implementariem?

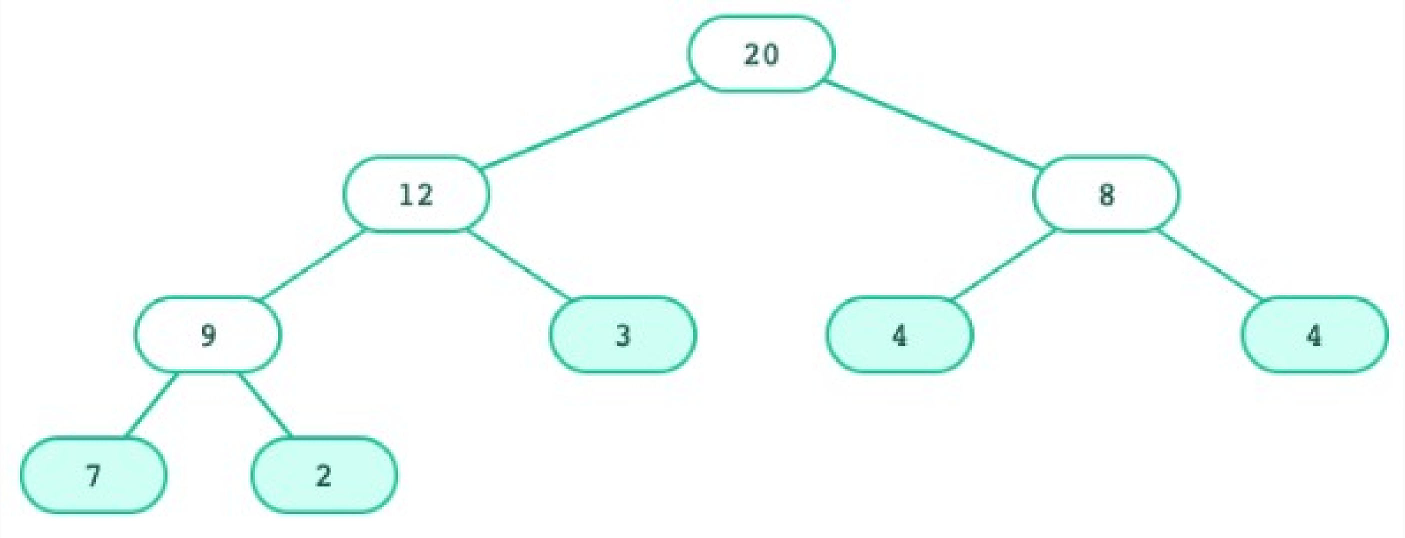
Arbres: Implementació 1

- Llista d'etiqueta + llista per a cada arbre/subarbre

```
def construeix_arbre(v,f=[]):                          # Constructora
    """ Retorna un arbre amb etiqueta v a arrel i f com a
        fills de l'arrel. f és una llista d'arbres """
    return [v] + f  # podríem posar [v] + list(f)

def valor_arbre(a):                                    # Consultora
    """ Retorna l'etiqueta de l'arrel de l'arbre a """
    return a[0]

def fills_arbre(a):                                    # Consultora
    """ Retorna els fills de l'arbre a """
    return a[1:]

def es_fulla(a):
    """ Retorna True si a és un node fulla """
    return len(fills_arbre(a)) == 0
```

Arbres: Implementació 1

- Llista d'etiqueta + llista per a cada arbre/subarbre




```
t = construeix_arbre(20, [construeix_arbre(12,
                           [construeix_arbre(9,
                              [construeix_arbre(7), construeix_arbre(2)]),
                            construeix_arbre(3)]),
                          construeix_arbre(8,
                           [construeix_arbre(4), construeix_arbre(4)])])

=> t és [20,[12,[9,[7],[2]],[3]],[8,[4],[4]]]
```

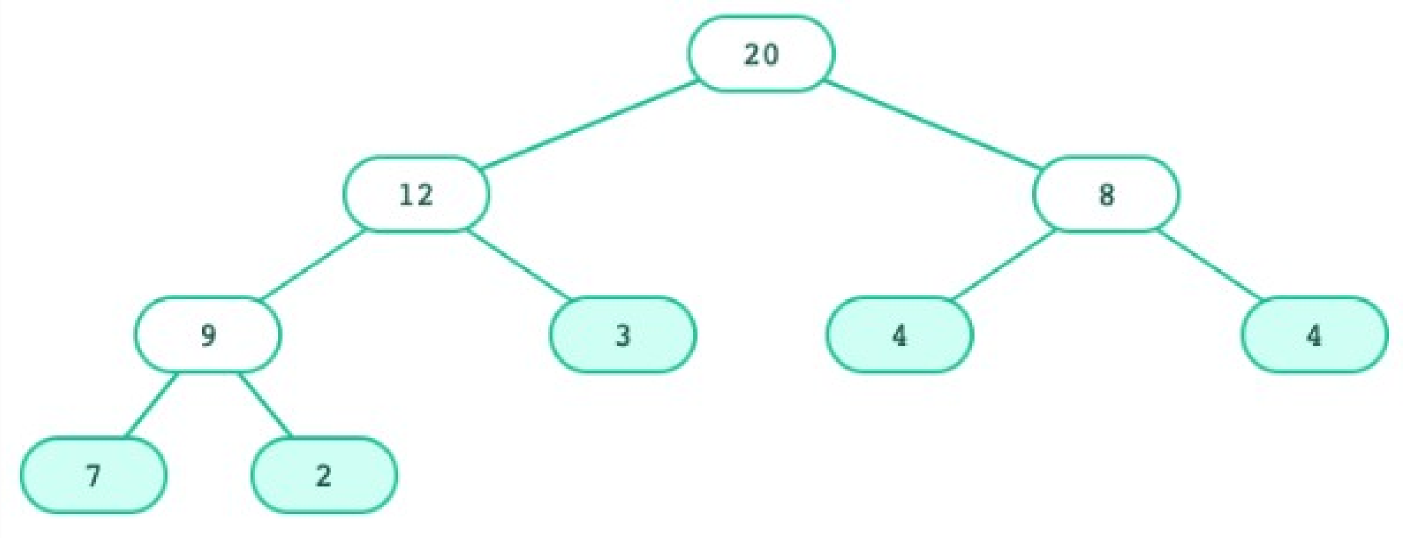
Arbres: Implementació 2

- Tupla d'(etiqueta, llista) per a cada arbre/subarbre

```
def construeix_arbre(v,f=[]):                          # Constructora
    """ Retorna un arbre amb etiqueta v a arrel i f com a
        fills de l'arrel """
    return (v,f)

def valor_arbre(a):                                    # Consultora
    """ Retorna l'etiqueta de l'arrel de l'arbre a """
    return a[0]

def fills_arbre(a):                                    # Consultora
    """ Retorna els fills de l'arbre a """
    return a[1]

def es_fulla(a):
    """ Retorna True si a és un node fulla """
    return len(fills_arbre(a)) == 0
```

Arbres: Implementació 2

- Tupla d'(etiqueta, llista) per a cada arbre/subarbre




```
t = construeix_arbre(20, [construeix_arbre(12,
                              [construeix_arbre(9,
                                 [construeix_arbre(7), construeix_arbre(2)]),
                               construeix_arbre(3)]),
                             construeix_arbre(8,
                              [construeix_arbre(4), construeix_arbre(4)])])

=> t és (20,[(12,[(9,[(7,[]),(2, [])]),(3, [])]),(8,[(4,[]),(4,[])])])
```

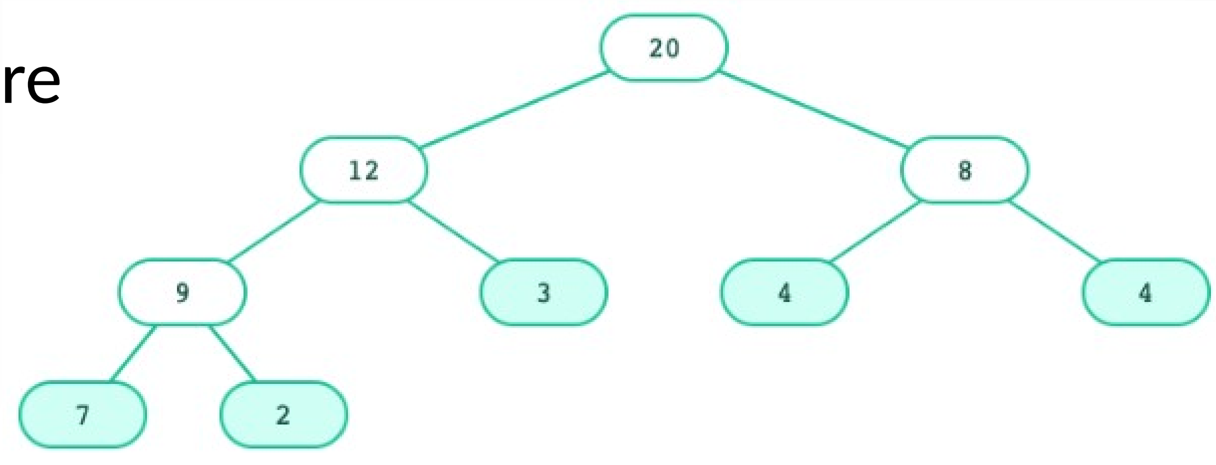
Arbres: Implementació 3

- Diccionari per a cada arbre/subarbre

```
def construeix_arbre(v,f=[]):                         # Constructora
    """ Retorna un arbre amb etiqueta v a arrel i f com a
        fills de l'arrel """
    return {"v": v, "f": f}

def valor_arbre(a):                                   # Consultora
    """ Retorna l'etiqueta de l'arrel de l'arbre a """
    return a["v"]

def fills_arbre(a):                                   # Consultora
    """ Retorna els fills de l'arbre a """
    return a["f"]

def es_fulla(a):
    """ Retorna True si a és un node fulla """
    return len(fills_arbre(a)) == 0
```

Arbres: Implementació 3

- Diccionari per a cada arbre/subarbre




```
t = construeix_arbre(20, [construeix_arbre(12,
                           [construeix_arbre(9,
                              [construeix_arbre(7), construeix_arbre(2)]),
                            construeix_arbre(3)]),
                          construeix_arbre(8,
                           [construeix_arbre(4), construeix_arbre(4)])])

=> t és {"v":20,"f":[{"v":12,"f":[{"v":9,"f":[{"v":7,"f": []},{"v":2,"f":[]}]},
                                                   {"v":3,"f":[]}]},
                     {"v":8,"f":[{"v":4,"f":[]},{"v":4,"f":[]}]}]}
```

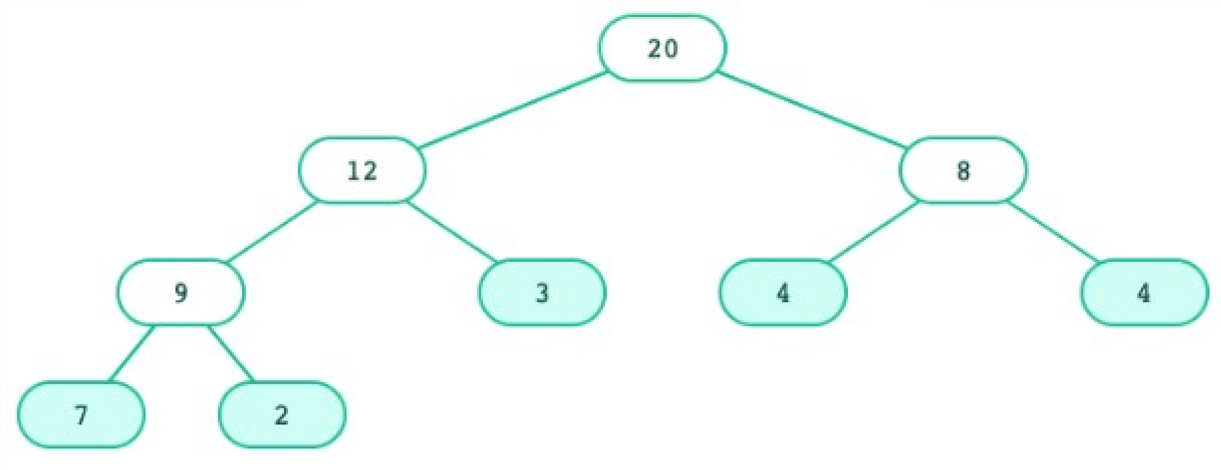
Processament d'Arbres

- Un arbre té una estructura ***recursiva***.

- Cada arbre té
  - Un valor / etiqueta
  - Zero o més fills, i cada un dels fills és un arbre

- *Estructura recursiva implica algorisme recursiu*!

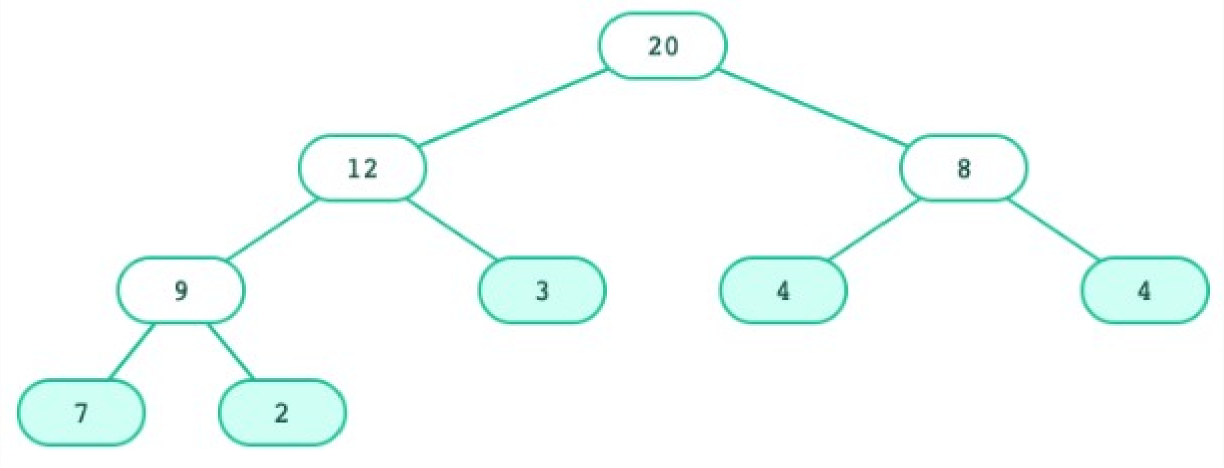
Processament d'Arbres: Comptar fulles




```
def comptar_fulles(a):
    """ Retorna el nombre de nodes fulla que té a """
    if ________:
        ________
    else:
        ________
```

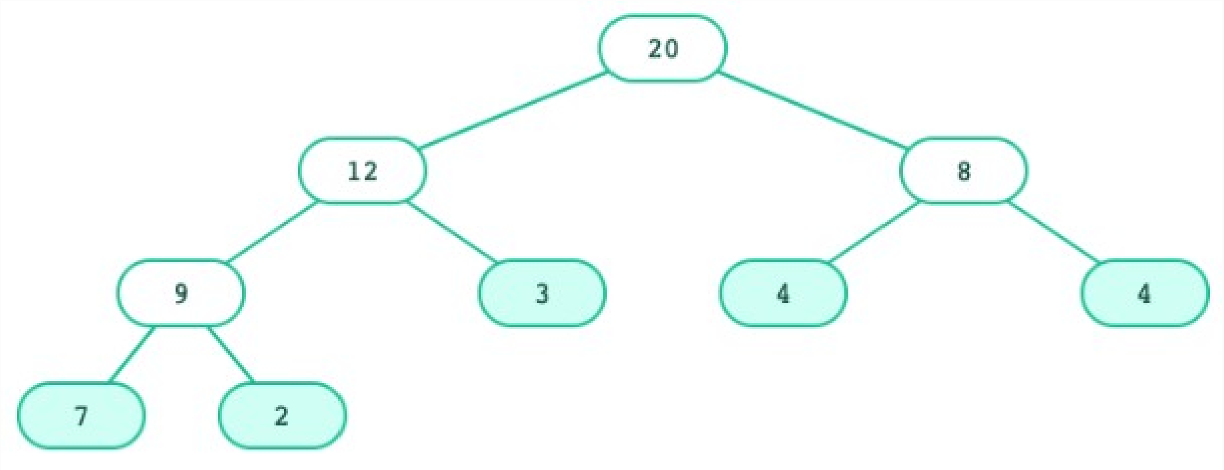
- Quin és el cas base? Quin el cas recursiu?

Processament d'Arbres: Comptar fulles




```python
def comptar_fulles(a):
    """ Retorna el nombre de nodes fulla que té a """
    if es_fulla(a):
        return 1
    else:
        fulles_fills = 0
        for f in fills_arbre(a):
            fulles_fills += comptar_fulles(f)
        return fulles_fills
```

Processament d'Arbres: Comptar fulles



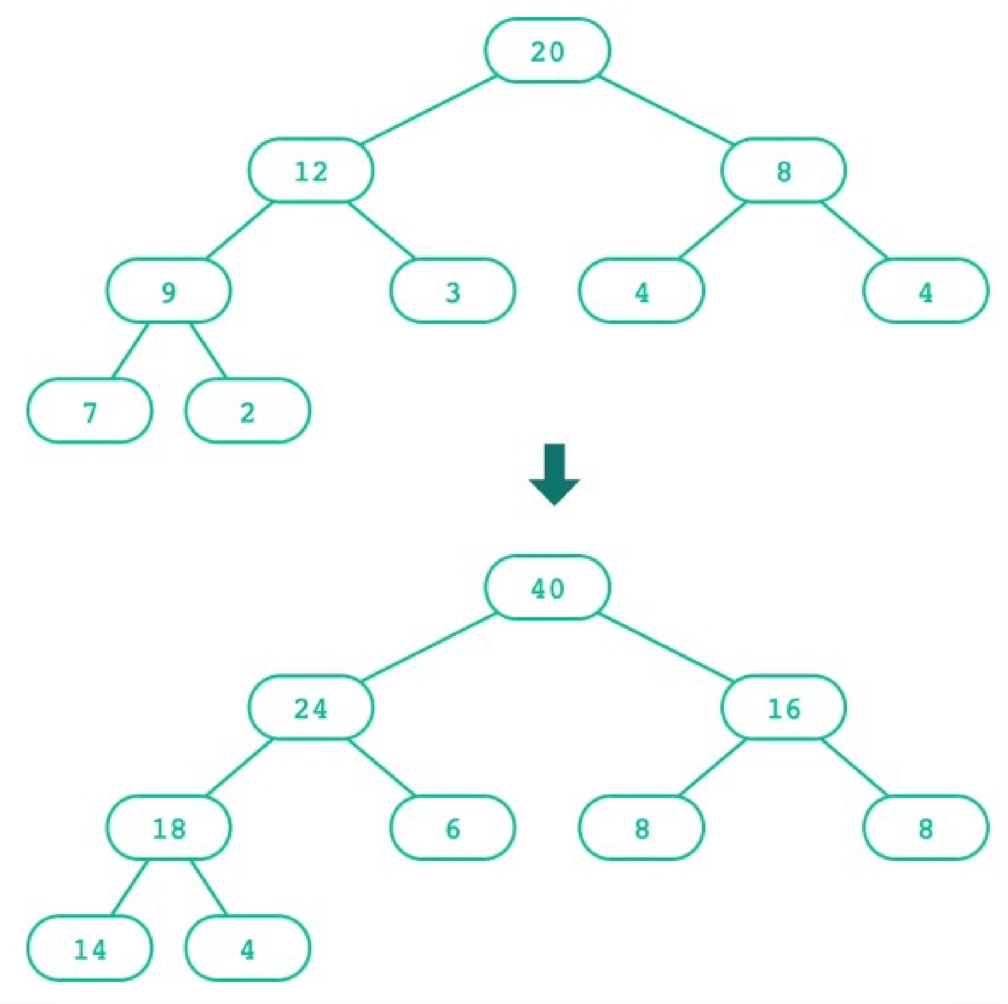

```
def comptar_fulles(a):
    """ Retorna el nombre de nodes fulla que té a """
    if es_fulla(a):
        return 1
    else:
        return sum([comptar_fulles(f) for f in fills_arbre(a)])
```

on fem servir `sum` per sumar els elements d'una llista (en realitat, un *iterable*)

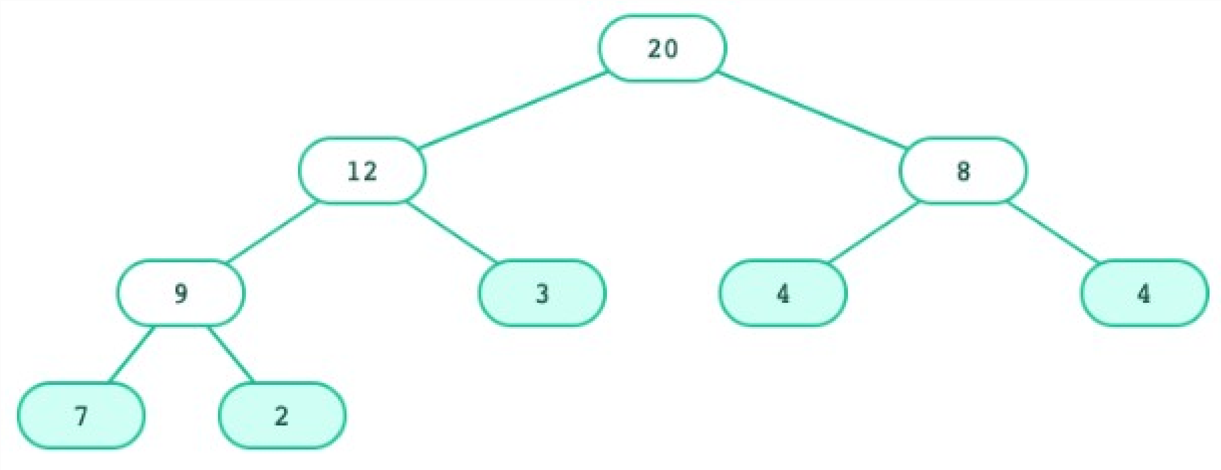
`sum([1,2,3,4]) => 10`

Creació d'arbres

- Una funció que *crea un arbre a partir d'un altre arbre* és també (sovint) ***recursiva***.



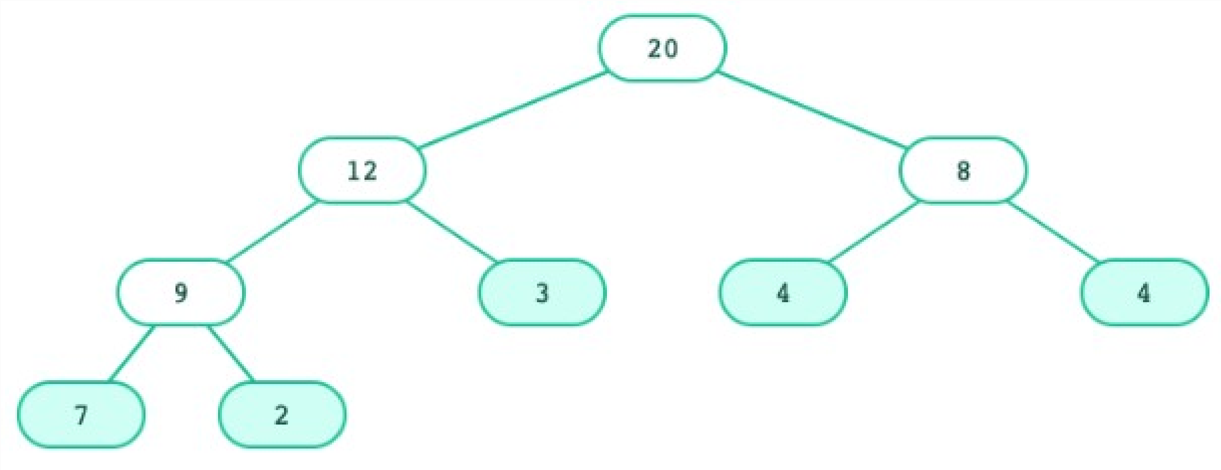
Creació d'arbres: Doblar les etiquetes




```
def doblar(a):
    """ Retorna un arbre com a, però amb les etiquetes doblades """
    if ________:
        ________
    else:
        ________
```

- Quin és el cas base? Quin el cas recursiu?

Creació d'arbres: Doblar les etiquetes




```
def doblar(a):
    """ Retorna un arbre com a, però amb les etiquetes doblades """
    if es_fulla(a):
        return construeix_arbre(2*valor_arbre(a))
    else:
        return construeix_arbre(2*valor_arbre(a),
                                       [doblar(f) for f in fills_arbre(a)])
```

Creació d'arbres: Doblar les etiquetes

- Versió llarga...

```
def doblar(a):
    """ Retorna un arbre com a, però amb les etiquetes doblades """
    if es_fulla(a):
        return construeix_arbre(2*valor_arbre(a))
    else:
        fills_doblats = []
        for f in fills_arbre(a):
            fills_doblats.append(doblar(f))
        return construeix_arbre(2*valor_arbre(a), fills_doblats)
```

Versió curta...

```
def doblar(a):
    """ Retorna un arbre com a, però amb les etiquetes doblades """
    return construeix_arbre(2*valor_arbre(a),
                            [doblar(f) for f in fills_arbre(a)])
```

Exercici: Llista de fulles

- Intenteu vosaltres:

```
def llista_de_fulles(a):
    """ Retorna una llista amb els valors a les fulles d'a

    >>> llista_de_fulles(a) # On a és l'arbre dels exemples
    [7,2,3,4,4]
    """
    if _______:
        return _______
    else:
        return _______
```

- **Pista**: *si apliqueu* `sum` *a una llista de llistes, retorna una llista amb els elements de les llistes. Cal, però, proporcionar un segon argument a* `sum`*, que és el valor inicial de* `sum`

Arbres: Capes d'Abstracció

- Així doncs, podem dividir les operacions sobre arbres d'aquesta manera:

Primitives	`1` `2` **`True`** **`False`** `( . . . )` `...[. . .]`
Representació	**`construeix_arbre`** **`valor_arbre`** **`fills_arbre`** **`es_fulla`**
Programa usuari	`comptar_fulles` `doblar`

- Les linies representen *barreres d'abstracció*.
- Cada capa només fa servir la capa per sobre

Valors Mutables: Destructiu vs. No destructiu

```
def doblar(a):
    """ Retorna un arbre com a, però amb les etiquetes doblades """
    return construeix_arbre(2*valor_arbre(a),
                            [doblar(f) for f in fills_arbre(a)])
```

- És `doblar`?
  - destructiva
  - no destructiva

Valors Mutables: Destructiu vs. No destructiu

```
def doblar(a):
    """ Retorna un arbre com a, però amb les etiquetes doblades """
    return construeix_arbre(2*valor_arbre(a),
                                      [doblar(f) for f in fills_arbre(a)])
```

- És `doblar`?
  - destructiva
  - no destructiva ←

No modifica l'arbre original, per tant la considerem no destructiva

Mutabilitat vs. Immutabilitat

- Éls arbres generats per aquesta implementació…

```
def construeix_arbre(v,f=[]):                     # Constructora
    """ Retorna un arbre amb etiqueta v a arrel i f com a
        fills de l'arrel """
    return (v,f)

def valor_arbre(a):                               # Consultora
    """ Retorna l'etiqueta de l'arrel de l'arbre a """
    return a[0]

def fills_arbre(a):                               # Consultora
    """ Retorna els fills de l'arbre a """
    return a[1]
```

  són
  - mutables?
  - immutables?

Mutabilitat vs. Immutabilitat

- Éls arbres generats per aquesta implementació…

```
def construeix_arbre(v,f=[]):                    # Constructora
    """ Retorna un arbre amb etiqueta v a arrel i f com a
        fills de l'arrel """
    return (v,f)

def valor_arbre(a):                              # Consultora
    """ Retorna l'etiqueta de l'arrel de l'arbre a """
    return a[0]

def fills_arbre(a):                              # Consultora
    """ Retorna els fills de l'arbre a """
    return a[1]
```

són

- mutables?
- **immutables**

Mentre no trenquem les barreres d'abstracció, els arbres creats amb `construeix_arbre` no es poden modificar.

Arbre mutable

- Suposem que afegim dues operacions modificadores...

```
def modifica_valor_arbre(a,v):                    # Modificadora
    """ Modifica el valor de l'arrel d'a a v """


def modifica_fills_arbre(a,f):                    # Modificadora
    """ Modifica els fills d'a a f """
```

Arbre mutable

- Suposem que afegim dues operacions modificadores...

```
def modifica_valor_arbre(a,v):                    # Modificadora
    """ Modifica el valor de l'arrel d'a a v """
    a[0] = v

def modifica_fills_arbre(a,f):                    # Modificadora
    """ Modifica els fills d'a a f """
    a[1] = f
```

- És correcte? Veiem-ho

Arbre mutable

- Suposem que afegim dues operacions modificadores...

```
def modifica_valor_arbre(a,v):                # Modificadora
    """ Modifica el valor de l'arrel d'a a v """
    a[0] = v

def modifica_fills_arbre(a,f):                # Modificadora
    """ Modifica els fills d'a a f """
    a[1] = f
```

- És correcte? Veiem-ho:

```
def construeix_arbre(v,f=[]):                # Constructora
    """ Retorna un arbre amb etiqueta v a arrel i f com a
        fills de l'arrel """
    return (v,f)
```

- La funció constructora retorna una tupla, i *les tuples són immutables*.

Arbre mutable

- Haurem de fer servir una altra implementació d'arbre:

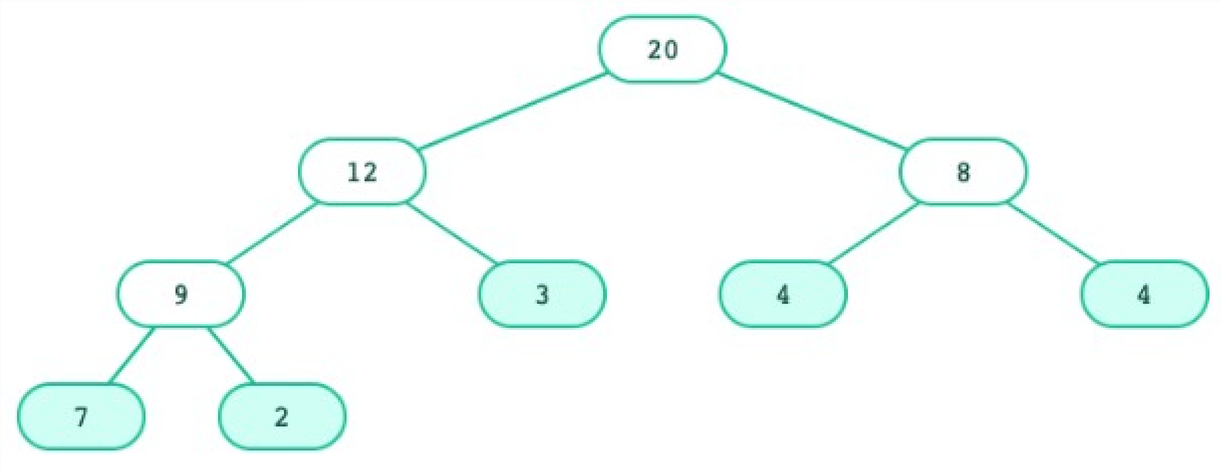
```
def construeix_arbre(v,f=[]):                    # Constructora
    """ Retorna un arbre amb etiqueta v a arrel i f com a
        fills de l'arrel. f és una llista d'arbres """
    return [v] + list(f)

def valor_arbre(a):                              # Consultora
    """ Retorna l'etiqueta de l'arrel de l'arbre a """
    return a[0]

def fills_arbre(a):                              # Consultora
    """ Retorna els fills de l'arbre a """
    return a[1:]

def es_fulla(a):
    """ Retorna True si a és un node fulla """
    return len(fills_arbre(a)) == 0

def modifica_valor_arbre(a,v):                   # Modificadora
    """ Modifica el valor de l'arrel d'a a v """
    a[0] = v

def modifica_fills_arbre(a,f):                   # Modificadora
    """ Modifica els fills d'a a f """
    a[1:] = f
```

Arbre mutable

- Haurem de fer servir una altra implementació d'arbre:

```
t = construeix_arbre(20, [construeix_arbre(12,
                           [construeix_arbre(9,
                              [construeix_arbre(7), construeix_arbre(2)]),
                            construeix_arbre(3)]),
                          construeix_arbre(8,
                           [construeix_arbre(4), construeix_arbre(4)])])

modifica_valor_arbre(t,40)

modifica_fills_arbre(t,[construeix_arbre(24)])

=> t és [40, [24]]
```

Doblar les etiquetes, destructivament



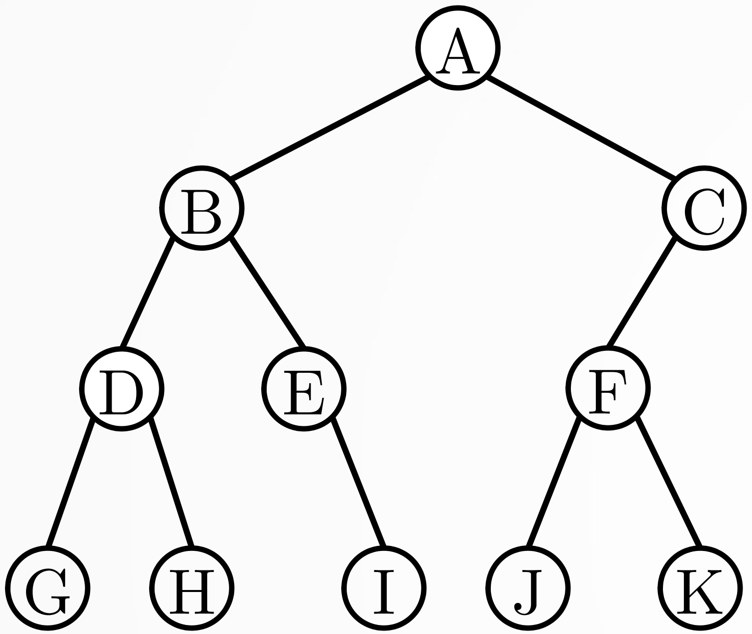

```python
def doblar(a):
    """ Dobla els valors d'a, mutant cada subarbre """
    modifica_valor_arbre(a, 2*valor_arbre(a))
    if not es_fulla(a):
        for f in fills_arbre(a):
            doblar(f)
```

Recorreguts d'arbres

- Podem fer un recorregut dels nodes d'un arbre de diferentes maneres:
  - **Pre-ordre**: Primer visitem l'arrel de l'arbre, després (*recursivament*) els fills, d'esquerra a dreta.
  - **Post-ordre**: Primer visitem (*recursivament*) els fills d'esquerra a dreta, I deixem l'arrel pel final.
  - **In-ordre** (*arbres binaris*): Primer visitem (*recursivament*) el fill esquerre, seguidament visitem l'arrel, finalment visitem (*recursivament*) el fill dret.
  - **Per nivells**: Visitem els nodes per nivells, d'esquerra a dreta.
- Veiem ara com implementar les funcions:

```
def recorregut_preordre(t, visita)
def recorregut_postordre(t, visita)
def recorregut_inordre(t, visita) # t és arbre binari
def recorregut_nivells(t, visita)
```

Recorreguts d'arbres

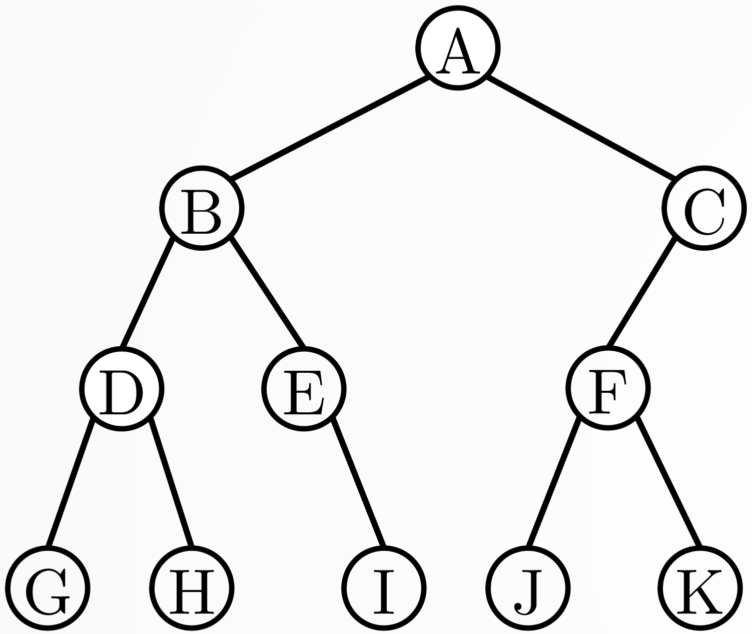


Els nodes es visiten en aquest ordre:

Pre-ordre: A B D G H E I C F J K

```
def recorregut_preordre(t, visita):
    visita(valor_arbre(t))
    if not es_fulla(t):
        for f in fills_arbre(t):
            recorregut_preordre(f,visita)
```

Recorreguts d’arbres



Els nodes es visiten en aquest ordre:

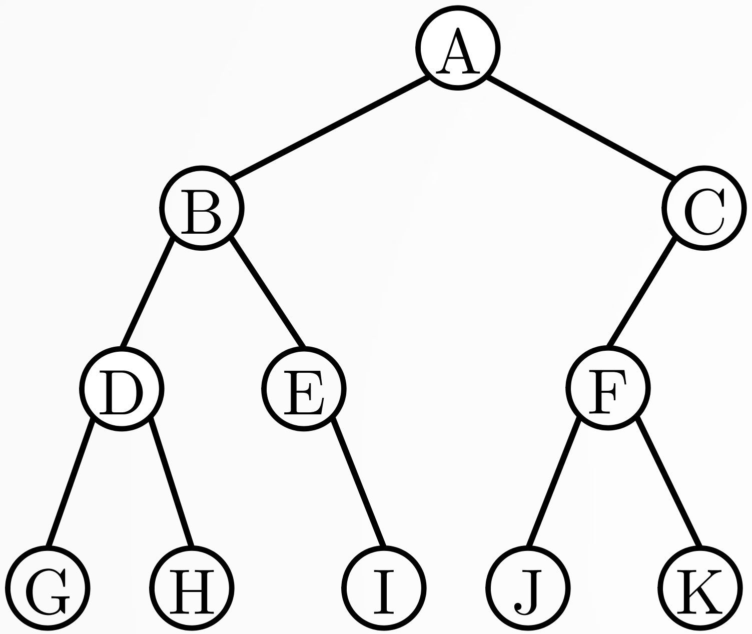
Post-ordre: G H D I E B J K F C A

```
def recorregut_postordre(t, visita):
    if not es_fulla(t):
        for f in fills_arbre(t):
            recorregut_postordre(f,visita)
    visita(valor_arbre(t))
```

Interludi: Arbre binari

```
def construeix_arbre_binari(v,fesq=None,fdre=None):         # Constructora
    """ Retorna un arbre amb etiqueta v a arrel i f com a
        fills de l'arrel """
    return {"v": v, "fesq": fesq, "fdre": fdre}

def valor_arbre_binari(a):                                  # Consultora
    """ Retorna l'etiqueta de l'arrel de l'arbre a
        L'arbre a no és None """
    return a["v"]

def fill_esq(a):                                            # Consultora
    """ Retorna el fill esquerre de l'arbre binari a """
    return a["fesq"]

def fill_dre(a):                                            # Consultora
    """ Retorna el fill dret de l'arbre binari a """
    return a["fdre"]

def es_fulla_binari(a):
    """ Retorna True si a és un node fulla """
    return a["fesq"] is None and a["fdre"] is None

def no_buit(a):
    """ Retorna True si a no és un arbre buit """
    return a is not None
```

Recorreguts d'arbres

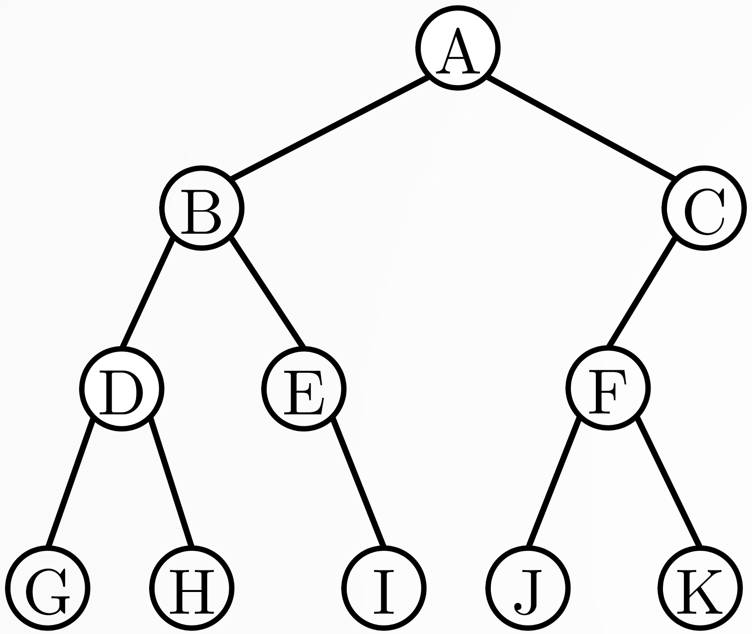


Els nodes es visiten en aquest ordre:

In-ordre: G D H B E I A J F K C

```
def recorregut_inordre(t, visita):
    # t és un arbre binari no buit
    fesq = fill_esq(t)
    fdre = fill_dre(t)
    if no_buit(fesq):
        recorregut_inordre(fesq,visita)
    visita(valor_arbre_binari(t))
    if no_buit(fdre):
        recorregut_inordre(fdre,visita)
```

Recorreguts d'arbres



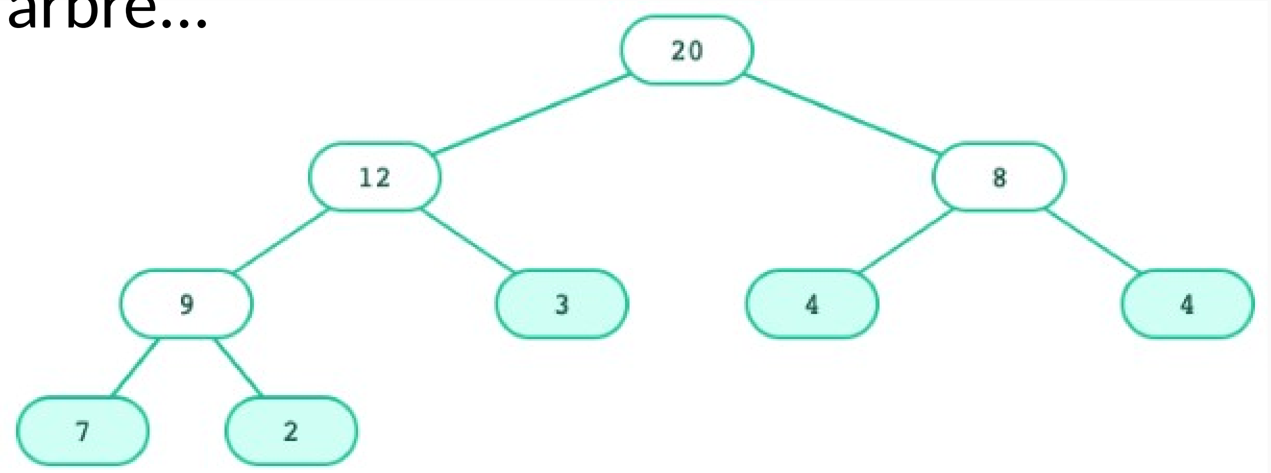
Els nodes es visiten en aquest ordre:

Per nivells: A B C D E F G H I J K

```
from collections import deque

def recorregut_nivells(t, visita):
    q = deque()
    q.append(t)
    while len(q) > 0:
        tt = q.popleft()
        visita(valor_arbre(tt))
        for f in fills_arbre(tt):
            q.append(f)
```

Generador recursiu per a arbres

- Un recorregut en pre-ordre d'un arbre...

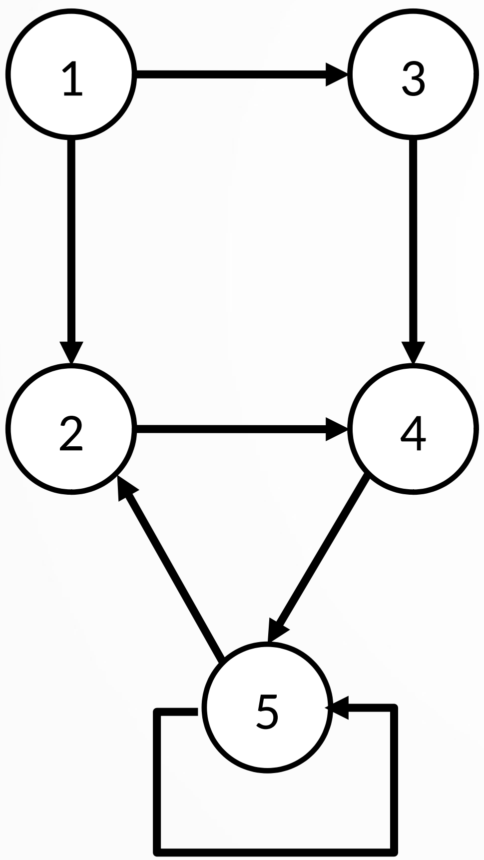



```
def preordre(a):
    yield valor_arbre(a)
    for f in fills_arbre(a):
        yield from preordre(f)

t = construeix_arbre(20, [construeix_arbre(12,
                           [construeix_arbre(9,
                              [construeix_arbre(7), construeix_arbre(2)]),
                            construeix_arbre(3)]),
                          construeix_arbre(8,
                           [construeix_arbre(4), construeix_arbre(4)])])

generador_preordre = preordre(t)
next(generador_preordre) # 20
```

Grafs

Grafs

- **ATENCIÓ!!** Repassar Fonaments Matemàtics!!
- Un ***graf*** **`(V,E)`** ve donat per un conjunt de ***vertexos*** **`V`** i un conjunt d'***arestes*** **`E`**.



$$V = \{1,2,3,4,5\}$$

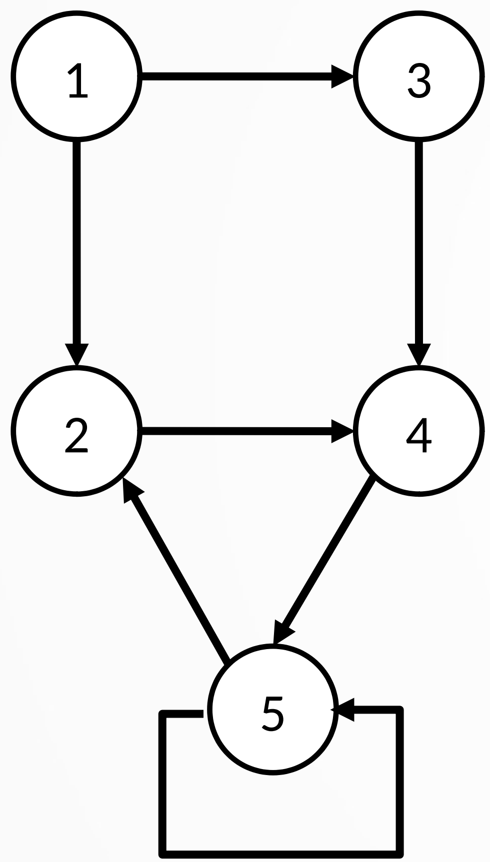
$$E = \{(1,2),(1,3),(2,4),(3,4),(4,5),(5,2),(5,5)\}$$

Els grafs poden ser ***dirigits*** o ***no dirigits***.

Les arestes E dels grafs no dirigits formen una relació simètrica.

Grafs

- Representació de grafs: ***Matriu d'Adjacència***



$$a_{i,j} = \begin{cases} 1 & \text{si hi ha una aresta } (v_i, v_j) \in E \\ 0 & \text{en altre cas} \end{cases}$$

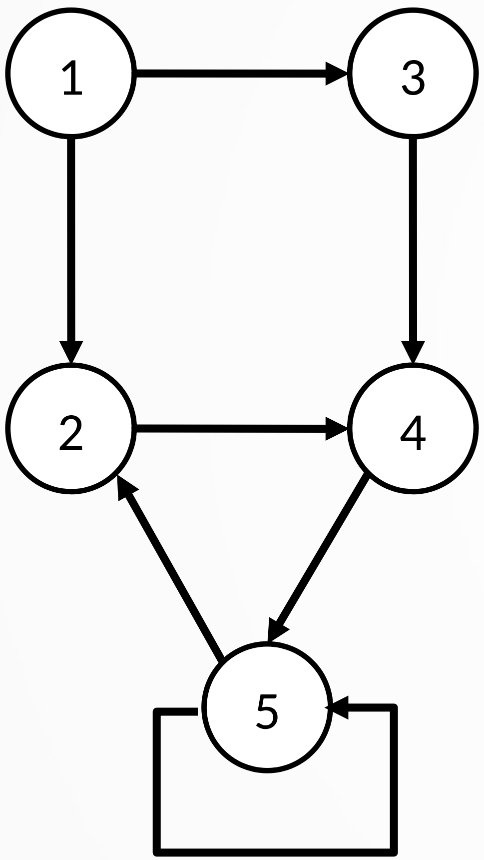
$$a = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

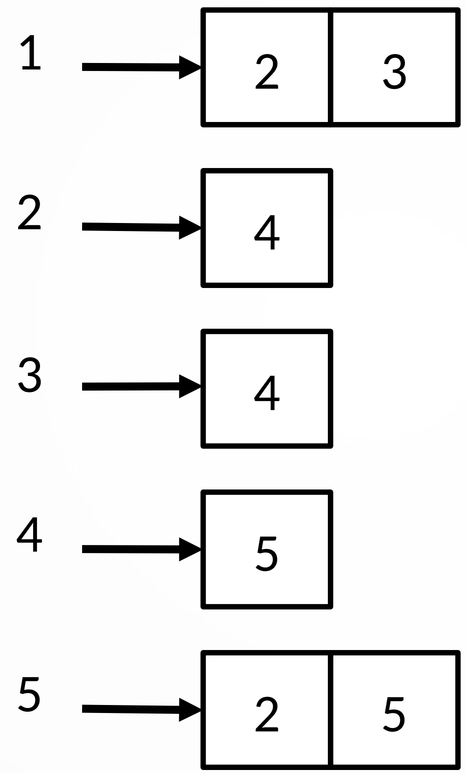
Espai: $\Theta(N^2)$

En grafs no dirigits la matriu és simètrica.

Grafs

- Representació de grafs: ***Llista d'Adjacències***



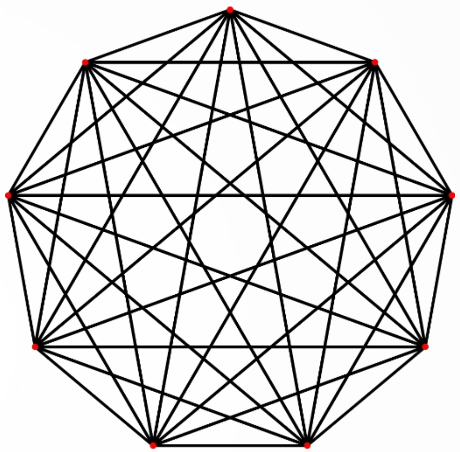


Espai: Θ (| E |)

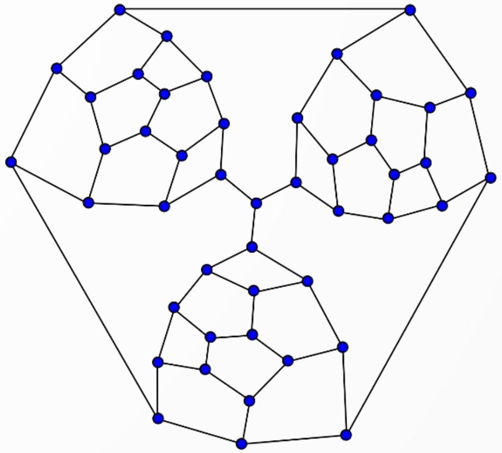
En grafs no dirigits cal afegir arestes bi-direccionals

Grafs

- Un graf amb |V| vertexos pot tenir fins a |V|(|V|-1)/2 arestes (totes les arestes possibles, si no hi ha auto-arestes).



- Direm que un graf és ***dens*** quan |E| *és proper a* $|V|^2$.



- Direm que un graf és ***espars*** (*sparse*) quan |E| *és proper a* |V|.

Matriu d'adjacència vs. Llista d'adjacència

- Espai
  - Matriu d'adjacència és $\Theta(N^2)$
  - Llistes d'adjacència és $\Theta(|E|)$

- Comprovar si existeix una aresta (v_i,v_j)
  - Matriu d'adjacència: $\Theta(1)$
  - Llistes d'adjacència: Cal recòrrer la llista adjacent a v_i.

- Quin farem servir?
  - **Grafs densos**: Matriu d'adjacència
  - **Grafs esparsos**: Llista d'adjacència

  Per a molts algorismes, iterar sobre les llistes d'adjacència no és un problema, ja que cal iterar sobre tots els veïns d'un vértex determinat. Per grafs esparsos, les llistes d'adjacència són típicament petites.

Matriu d'adjacència vs. Llista d'adjacència

n: nombre de vèrtexs / m: nombre d'arestes

operacions	matriu d'adjacència	llistes d'adjacència
espai	$\Theta(n^2)$	$\Theta(n+m)$
crear	$\Theta(n^2)$	$\Theta(n)$
afegir vèrtex	$\Theta(n)$	$\Theta(1)$
afegir aresta	$\Theta(1)$	$\mathcal{O}(n)^*$
esborrar aresta	$\Theta(1)$	$\mathcal{O}(n)$
consultar vèrtex	$\Theta(1)$	$\Theta(1)$
consultar aresta	$\Theta(1)$	$\mathcal{O}(n)$
és v aïllat?	$\Theta(n)$	$\Theta(1)$
successors*	$\Theta(n)$	$\mathcal{O}(n)$
predecessors*	$\Theta(n)$	$\mathcal{O}(m)$
adjacents+	$\Theta(n)$	$\mathcal{O}(n)$

* només en grafs dirigits

\+ només en grafs no dirigits

* g ∈ O(f) significa, a grans trets, que, *assimptòticament*, f és una *fita superior* per a g. Ho estudiarem a final de curs

Accessibilitat

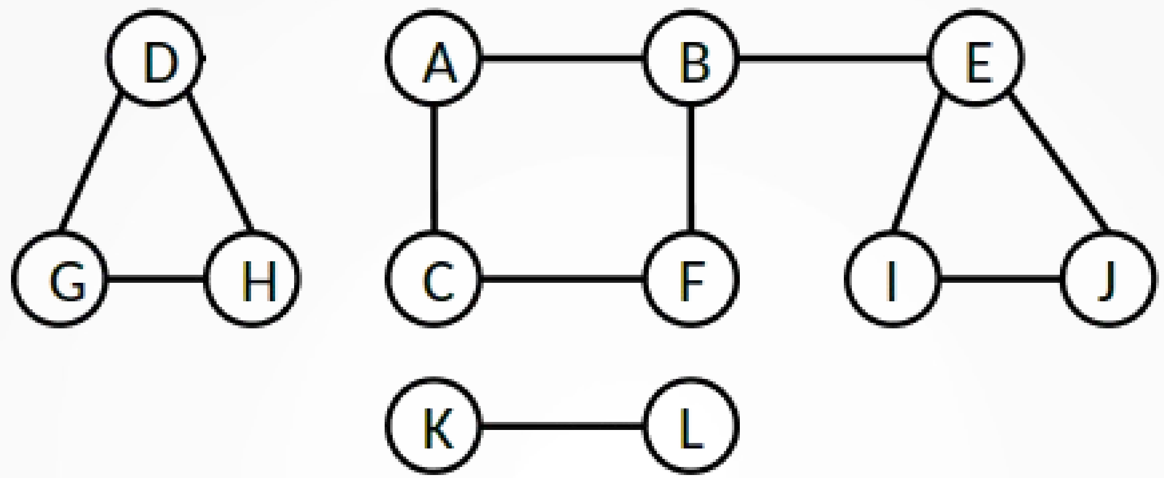
- Trobar els nodes ***accessibles*** des d'un node donat **v ∈ V***:
  - Marquem el node **v** en qüestió com a *node visitat* i hi fem alguna cosa (`accio`)
  - Seguim explorant, *recursivament*, a partir dels nodes veïns

```
def explorar(G, v, accio = (lambda x: x) ):
    # Entrada: G=(V,E) és un graf
    # Output: visitats[u] és true per a tots els
    #          nodes u accessibles des de v

    visitats[v] = True
    accio(v)
    for each aresta (v,u) ∈ E:
        if not visitats[u]:
            explorar(G, u, accio)
```

*En aquesta transparència, i altres, faré servir una mena de *pseudo-python*, per estalviar detalls irrellevants en l'explicació. Ja implementarem del tot els algorismes importants a les classes de laboratori.

Accessibilitat: Exemple

- Donat el graf G



- `explorar(G,'A')` visitats →

'A'	'B'	'C'	'D'	'E'	'F'	'G'	'H'	'I'	'J'	'K'	'L'
true	true	true	false	true	true	false	false	true	true	false	false

- `explorar(G,'D')` visitats →

'A'	'B'	'C'	'D'	'E'	'F'	'G'	'H'	'I'	'J'	'K'	'L'
false	false	false	true	false	false	true	true	false	false	false	false

Cerca en Profunditat (*Depth First Search*, DFS)

```
def DFS(G, accio):

    def DFS_vertex(G, v, ac = (lambda x: x) ): # Hem renombrat 'explorar'
        visitats[v] = True
        ac(v)
        for each aresta (v,u) ∈ E:
            if not visitats[u]:
                DFS_vertex(G, u, ac)

    visitats = {}          # visitats serà un diccionari
    for all v ∈ V:
        visitats[v] = False
    for all v ∈ V:
        if not visitats[v]:
            DFS_vertex(G,v,accio)
```

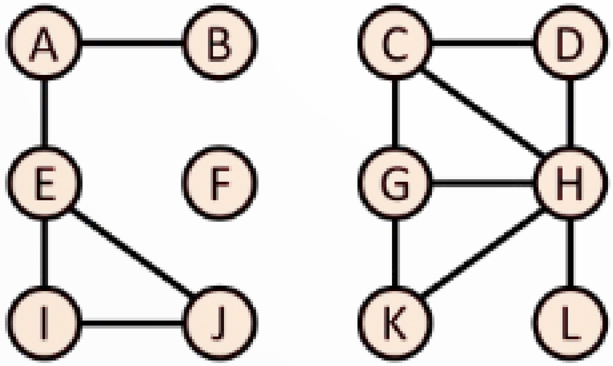
Cerca en Profunditat (*Depth First Search*, DFS)

- `DFS` accedeix a tot el graf. Fem `DFS_vertex` sobre cada un dels components connexos.

  `explorar` és en realitat un `DFS` a partir d'un vèrtex concret:

  `explorar` ≡ `DFS_vertex`.

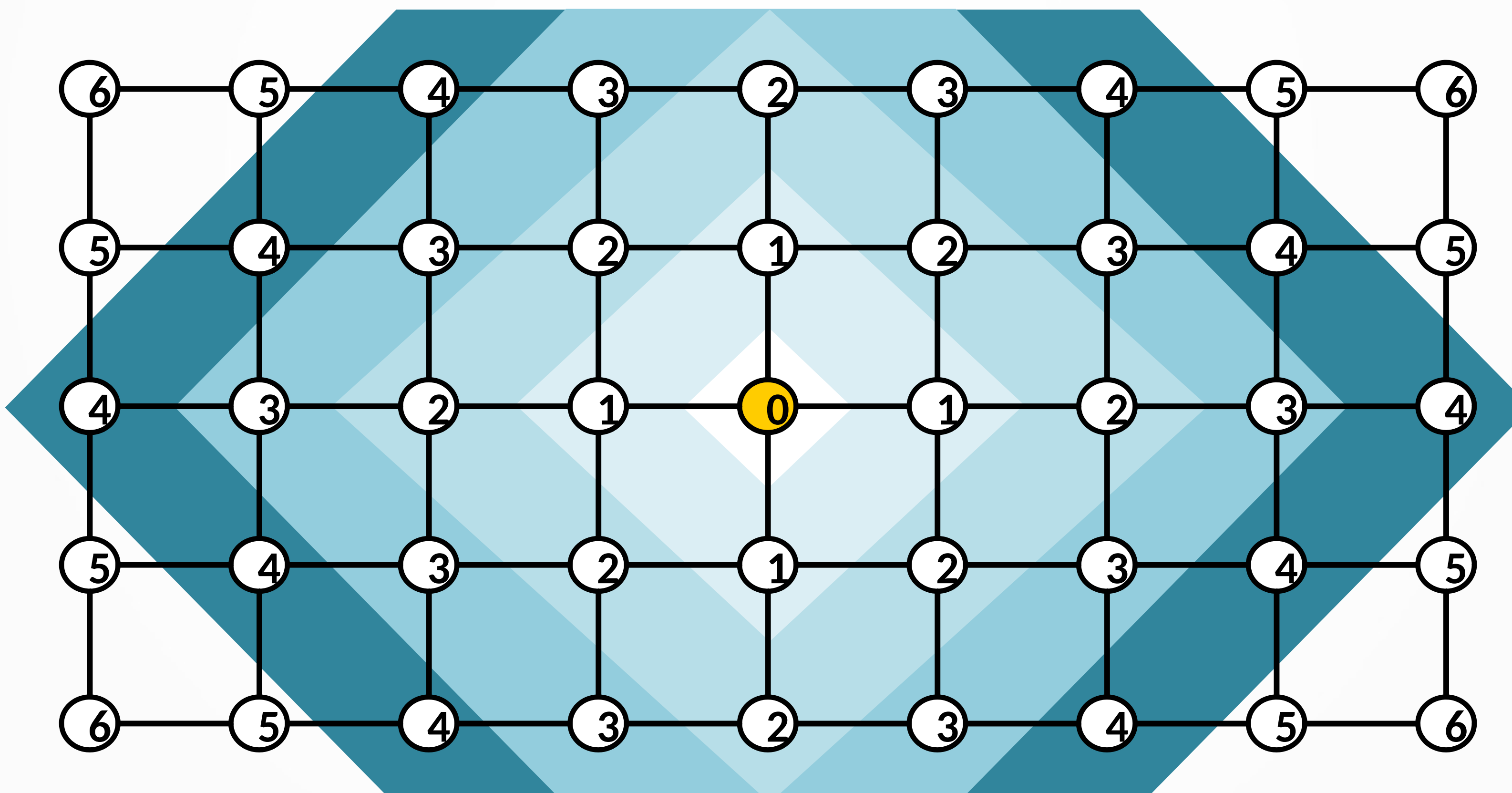
- Gràcies a l'estructura `visitats` cada vèrtex del graf es visita només **un cop**.
- El cost de `DFS` és `Θ(|V|+|E|)`, la qual cosa és difícil de millorar, ja que aquest és el cost de llegir un graf des de qualsevol entrada.
- Exemple: Si el graf és



  `DFS` farà tres crides a `DFS_vertex`, una en el vèrtex A, una altra en el vèrtex C i una altra a F (si la iteració sobre els vertexos és alfabètica)

Cerca en Amplada (*Breadth First Search*, BFS)

- La cerca (o recorregut) en amplada (BFS) avança localment des d'un vèrtex inicial s visitant els vèrtexs a distància k + 1 de s després d'haver visitat els vèrtexs a distància k de s.

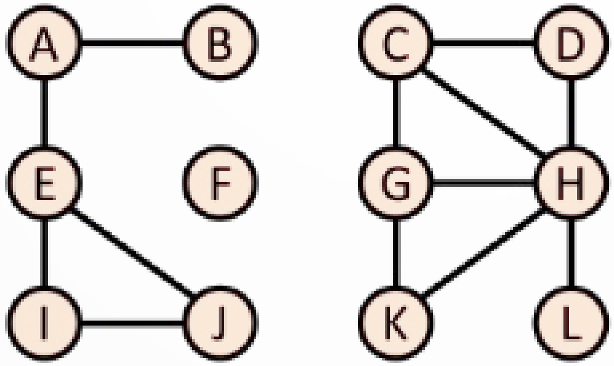


Cerca en Amplada (*Breadth First Search*, BFS)

```
def BFS_vertex(G, s, accio = (lambda x: x) ):
    # Entrada: Graf G(V, E), començant al vèrtex s.
    # Sortida: Per a cada vèrtex u, dist[u] és
    #          la distancia de s a u (nombre d'arestes).
    for all u ∈ V: dist[u] = float('inf') # podria ser un diccionari
    dist[s] = 0
    q = deque(); q.append(s) # q és la cua (només append/popleft)
    while len(q) > 0:
        u = q.popleft()
        accio(u)
        for all (u, v) ∈ E:
            if dist[v] == float('inf'):
                dist[v] = dist[u] + 1
                q.append(v)
    return dist
```

Cerca en Amplada (*Breadth First Search*, BFS)

- `BFS_vertex` accedeix només al graf accessible des de `s`.
- **Exercici**: feu un algorisme `BFS`, similar al `DFS`, per recòrrer *tot* el graf en amplada.
- Gràcies a l'estructura `dist` sabem a quantes arestes de distància estan els nodes accessibles des de `s`.
- El cost de `BFS_vertex` és $\Theta(|V|+|E|)$
- Exemple: Si el graf és

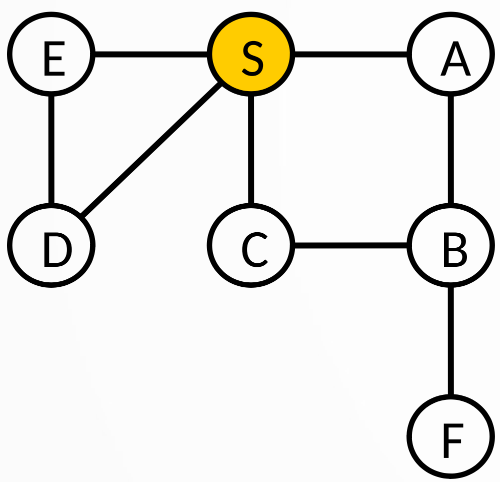


`BFS_vertex(G,'A')` retornarà

'A'	'B'	'C'	'D'	'E'	'F'	'G'	'H'	'I'	'J'	'K'	'L'
0	1	∞	∞	1	∞	∞	∞	2	2	∞	∞

Cerca en Amplada (*Breadth First Search*, BFS)

- Exemple:



	Cua q	S	A	B	C	D	E	F
	S_0	0	∞	∞	∞	∞	∞	∞
S_0	$A_1\ C_1\ D_1\ E_1$	0	1	∞	1	1	1	∞
A_1	$C_1\ D_1\ E_1\ B_2$	0	1	2	1	1	1	∞
C_1	$D_1\ E_1\ B_2$	0	1	2	1	1	1	∞
D_1	$E_1\ B_2$	0	1	2	1	1	1	∞
E_1	B_2	0	1	2	1	1	1	∞
B_2	F_3	0	1	2	1	1	1	3
F_3		0	1	2	1	1	1	3

(dist: columns S–F)

Cerca en Amplada (*Breadth First Search*, BFS)

- Per a cada `d = 0, 1, 2,...` hi ha un moment en el qual:
  - els vèrtexs a distància ≤ d de `s` tenen la distància correcta
  - els vèrtexs a distància > d de `s` tenen la distància ∞
  - la cua conté els nodes a distància d
- Els codis (*iteratius!*) de **`DFS_vertex`** i **`BFS_vertex`** són molt semblants.

  La diferència fonamental és l'ús…
  - d'una pila en **`DFS_vertex`** (**Exercici**! fer **`DFS_vertex`** iteratiu)
  - d'una cua en **`BFS_vertex`**
- Com en **`DFS_vertex`**, en **`BFS_vertex`** només s'explora el component connex del vèrtex inicial. Per explorar la resta del graf, podem recomençar la cerca des dels altres vèrtexs amb l'ajut d'un bucle (com hem fet al **`DFS`**).

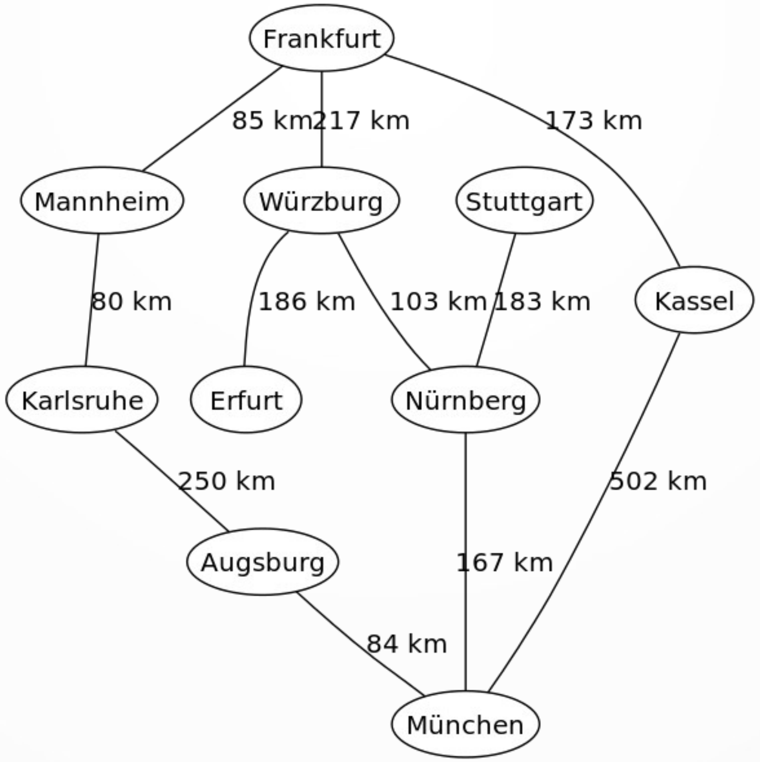
Resum: Implementació BFS & DFS

```python
def bfs(G, accio):
    # G és un graf amb el format de llegir_graf (lab)

    def bfs_vertex(G, s, ac = (lambda x: x) ):
        # Entrada: Graf G(V, E), començant al vèrtex s.
        # Sortida: Per a cada vèrtex u, dist[u] és
        #          la distancia de s a u (nombre d'arestes).
        N = len(G)
        dist = {u:float('inf') for u in range(N)}
        dist[s] = 0; visitats[s] = True
        q = deque(); q.append(s)
        while len(q) > 0:
            u = q.popleft()
            visitats[u] = True
            ac(u)
            for v in G[u]:
                if dist[v] == float('inf'):
                    dist[v] = dist[u] + 1
                    q.append(v)
        return dist

    N = len(G)   # visitats serà un diccionari
    visitats   = {u:False for u in range(N)}
    distancies = []
    for i in range(N):
        if not visitats[i]:
            d = bfs_vertex(G,i,accio)
            distancies.append(d)
    return distancies
```

```python
def dfs(G, accio):
    # G és un graf amb el format de llegir_graf (lab)

    def dfs_vertex(G, v, ac = (lambda x: x) ):
        visitats[v] = True
        accio(v)
        for u in G[v]:
            if not visitats[u]:
                dfs_vertex(G, u, ac)

    N = len(G)
    visitats = {}          # visitats serà un diccionari
    for i in range(N):
        visitats[i] = False
    for i in range(N):
        if not visitats[i]:
            dfs_vertex(G,i,accio)
```

Grafs etiquetats (*weighted graphs*)

- Un graf etiquetat `G = (V,E,w)` és un graf on `V` és el conjunt de vèrtexos, `E` és el conjunt d'arestes, i `w` és una funció que associa un nombre a cada aresta: $w : E \rightarrow \mathbb{R}$
- Típicament (però no necessàriament) aquest nombre s'interpreta com el ***cost*** de *travessar/triar* l'aresta associada:



Grafs etiquetats (*weighted graphs*)

- Una pregunta que sorgeix de manera *natural* en els grafs etiquetats és: **Quin és el camí *més curt* / *menys costós* / *òptim* (depén de com interpretem les etiquetes de les arestes) entre dos nodes donats del graf?**
- Ja vam veure que el **BFS** en un graf (*no etiquetat*) contemplava la noció de *distància entre nodes*, en nombre d'arestes. En aquest sentit, el **BFS** resol aquesta pregunta si la funció `w` és constant i positiva. En el cas general, però, no serveix.
- L'informàtic (físic teòric de formació) holandés Edsger W. Dijkstra va proposar l'any 1956 (i publicar el 1959) l'anomenat ***algorisme de Dijkstra*** per resoldre aquest problema.
- L'***algorisme de Dijkstra*** per trobar els camins mínims entre dos nodes *tampoc* resol el problema general per a qualsevol `w`.

Funciona bé només si $w(e) > 0, \forall e \in E$

Grafs etiquetats (*weighted graphs*)

- La variant de l'algorisme de Dijkstra que començarem explicant* té:
  - **Entrades**: un graf etiquetat i un *vèrtex inicial*
  - **Sortida**:
    - Un contenidor (llista o diccionari) amb les *distàncies* (suma de costos del camí des del vèrtex inicial) de cada vèrtex (accessible des del vèrtex inicial), al vèrtex inicial
    - Un altre contenidor amb el vèrtex antecessor de cada vèrtex accessible des del vèrtex inicial, en el camí que hi ha des del vèrtex inicial al vèrtex en qüestió.
- Aquesta variant de l'algorisme de Dijkstra troba els camins mínims des d'un vèrtex a tots aquells que siguin accessibles.

* Aquest algorisme que detallarem aquí no és ben bé l'algorisme de Dijkstra *original*. Podeu consultar **https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm** per veure l'algorisme original. La versió que veurem aquí és més senzilla, ja que no necessita modificar els elements del minheap.

Algorisme de Dijkstra

```
def dijkstra(G, w, s):
    # Entrada: Graf G=(V,E), vèrtex inicial s, etiquetes d'arestes w (positives)
    # Sortida: dist[u] distància de s, prev[u] predecessor en camí mínim

    for all u ∈ V:
        dist[u] = float('inf')
        prev[u] = None
    dist[s] = 0


    minheap = [(0,s)]      # La prioritat vindrà donada per la distància
    minheap = heapify(minheap)

    while not buit(minheap):
        d,u = obtenir_min(minheap)
        if d == dist[u]:
            for all (u, v) ∈ E:
                if dist[v] > dist[u] + w(u,v):
                    dist[v] = dist[u] + w(u,v)
                    prev[v] = u
                    inserir(minheap,(dist[v],v))
    return dist,prev
```

Algorisme de Dijkstra

- Fixem-nos que guardem al *heap* parelles d'elements

  (distància del vèrtex u al vèrtex inicial, u)

  Així, quan obtenim el mínim del *heap* estem obtenim la distància més petita enregistrada fins aquell moment.

```
while not buit(minheap):
        d,u = obtenir_min(minheap)
        if d == dist[u]:
            for all (u, v) ∈ E:
                if dist[v] > dist[u] + w(u,v):
                    dist[v] = dist[u] + w(u,v)
                    prev[v] = u
                    inserir(minheap,(dist[v],v))
```

- Fixem-nos també que quan trobem un vèrtex amb una distància millor que la que es tenia fins aquell moment el posem al *heap*, sense mirar si ja hi és o no. És per això que un cop obtenim una parella `(distància,vèrtex)`, mirem si guanyem alguna cosa processant-la.

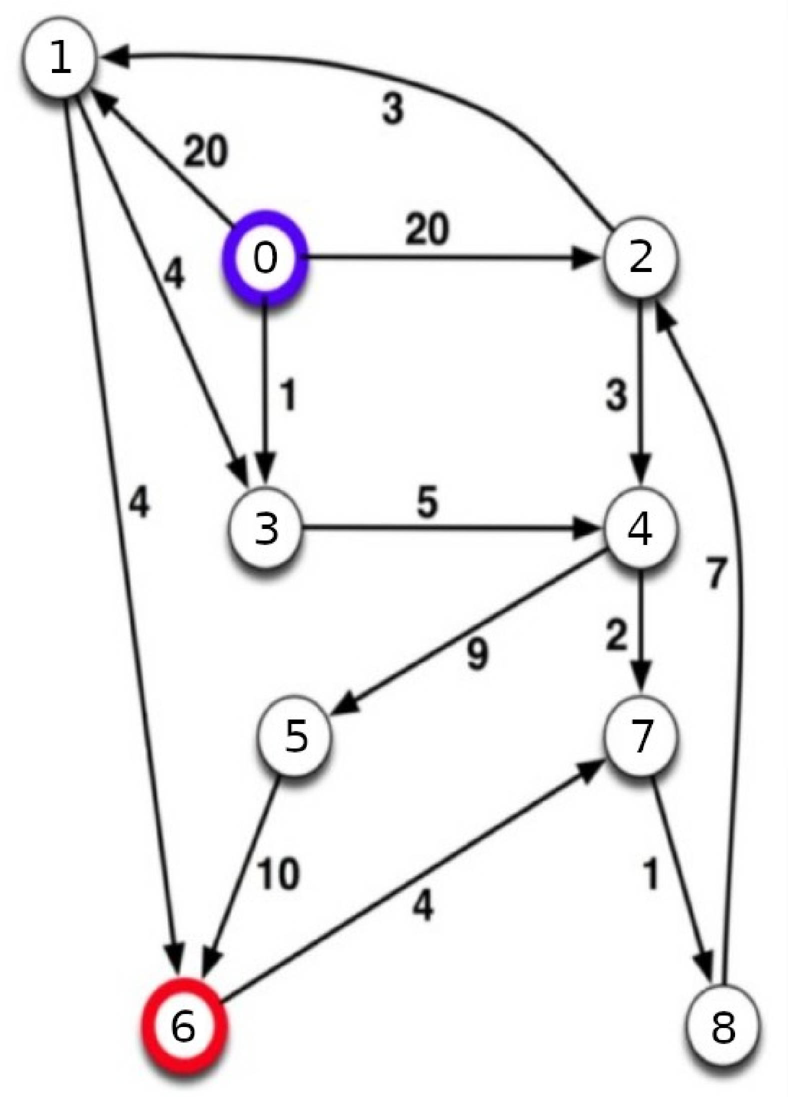
Algorisme de Dijkstra

- L'algorisme de Dijkstra és el que es coneix com a ***algorisme voraç*** (*greedy algorithm*). A grans trets, l'estratègia és triar *localment* el millor que es pot triar, amb l'esperança que així arribem a una solució *globalment* òptima.
- En el cas d'aquest algorisme aquesta tria es fa quan es decideix obtenir, per continuar el procés, el vèrtex que fins el moment té la distància més petita a `s`. Per això fem servir un *heap*. Aquesta estratègia *local* ens porta a una solució *global*, és a dir, a trobar els camins mínims des del vèrtex de partida a tots aquells que són accessibles.

```
while not buit(minheap):
       d,u = obtenir_min(minheap)
       if d == dist[u]:
           for all (u, v) ∈ E:
               if dist[v] > dist[u] + w(u,v):
                   dist[v] = dist[u] + w(u,v)
                   prev[v] = u
                   inserir(minheap,(dist[v],v))
```

Algorisme de Dijkstra

- Exemple: Trobar els camins de menys cost que parteixen del vèrtex 0



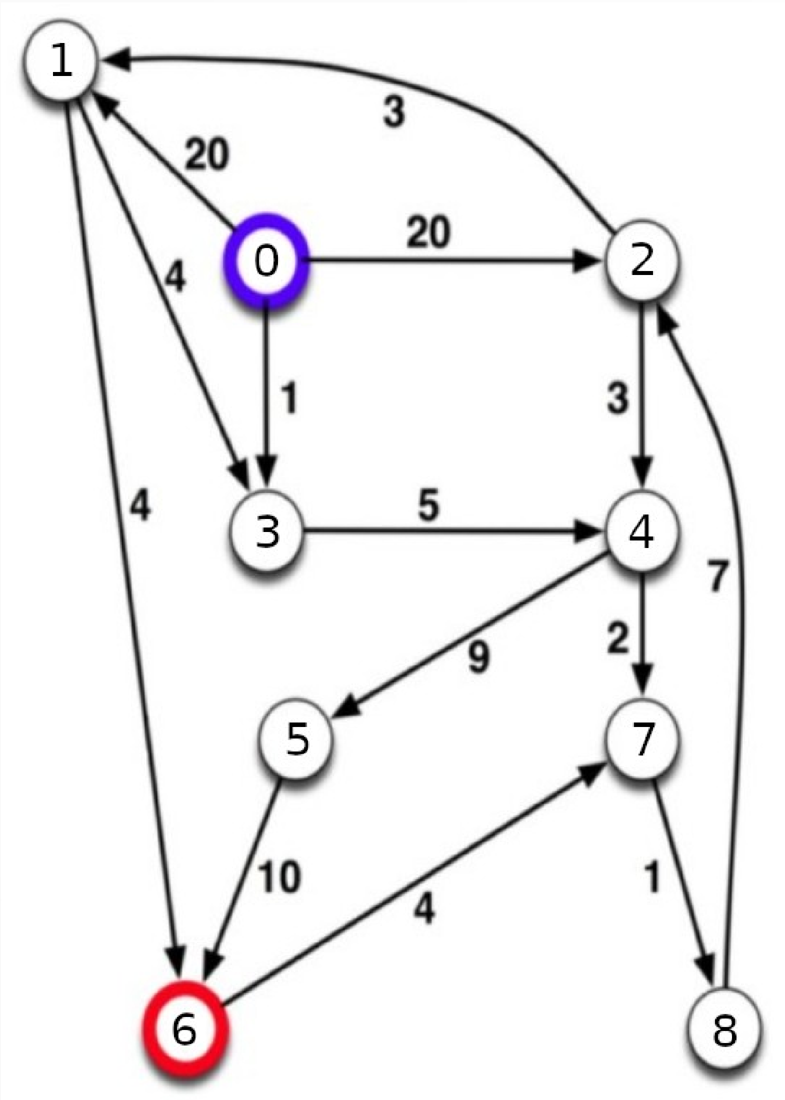
Fitxer exemple_dijkstra.inp

```
9          # nombre de vèrtexos
14         # nombre d'arestes
0 1 20     # u v w - on w és l'etiqueta de l'aresta
0 2 20
0 3 1      def llegir_graf_etiquetat():
1 3 4          n = int(f_item())*  # nombre de vertexos |V|
1 6 4          m = int(f_item())  # nombre d'arestes |E|
2 4 3          G = [[] for _ in range(n)]
2 1 3          for i in range(m):      # m parelles u,v,w:
3 4 5              u = int(f_item())  # aresta u --w--> v
4 5 9              v = int(f_item())
4 7 2              w = int(f_item())
5 6 10             G[u].append((w,v))
6 7 4          return G
7 8 1
8 2 7
```

* veure transparència següent

Algorisme de Dijkstra

- Exemple: Trobar els camins de menys cost que parteixen del vèrtex 0



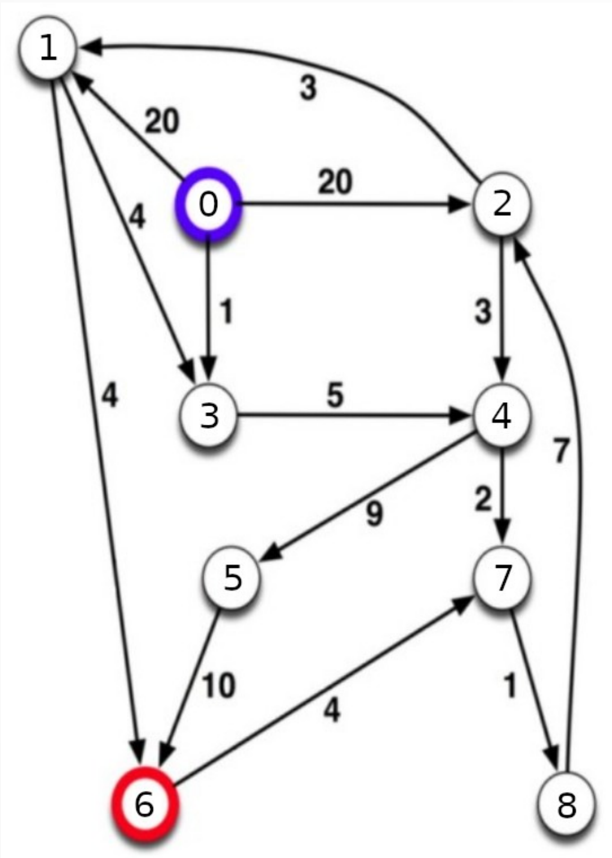
prova_dijkstra.py

```
from pytokr import make_tokr
import sys
from collections import namedtuple,deque
#-----------------------------------------------------------------
Pair = namedtuple("Pair", ["first", "second"])
#-----------------------------------------------------------------
# Implementació dels heaps que vam veure a la sessió 1 del laboratori
def surar(lst, i):...
def enfonsar(lst, i):...
def inserir(lst, x):...
def obtenir_min(lst):...
def heapify(items = []):...
#-----------------------------------------------------------------
# Lectura graf DIRIGIT i ETIQUETAT amb llistes d'adjacència, i els nodes
# són nombres enters 0...N-1 (si el graf té N nodes)
def llegir_graf_etiquetat():...    # l'hem vist a la transparència anterior
def dijkstra(G, s):...             # l'algorisme en qüestió
#-----------------------------------------------------------------
fitxer = sys.argv[1]   # Hem de passar el nom del fitxer com a argument
with open(fitxer) as f:
    _, f_item = make_tokr(f)
    G = llegir_graf_etiquetat()
d,p = dijkstra(G,0)
print(d)
print(p)
```

```
$ python3 prova_dijkstra.py exemple_dijkstra.inp
[0, 19, 16, 1, 6, 15, 23, 8, 9]
[None, 2, 8, 0, 3, 4, 1, 4, 7]
```

Algorisme de Dijkstra

- Exemple: En cada casella de la taula trobem `dist[u]/prev[u]` pel vèrtex u



0	1	2	3	4	5	6	7	8
0/None	∞/None	∞/None	∞/None	∞/None	∞/None	∞/None	∞/None	∞/None
	20/0	20/0	1/0	∞/None	∞/None	∞/None	∞/None	∞/None
	20/0	20/0		6/3	∞/None	∞/None	∞/None	∞/None
	20/0	20/0			15/4	∞/None	8/4	∞/None
	20/0	20/0			15/4	∞/None		9/7
	20/0	16/8			15/4	∞/None		
	20/0	16/8				25/5		
	19/2					25/5		
						23/1		

Camí de 0 a 6: 6 ← 1 ← 2 ← 8 ← 7 ← 4 ← 3 ← 0 amb cost 23.

Algorisme de Dijkstra

- Cost de l'algorisme de Dijkstra: L'algorisme de Dijkstra està basat en **BFS**, que té un cost $\Theta(|V|+|E|)$

  Cal comptar, a més:

  Les operacions **`obtenir_min`** i **`inserir`** tenen un cost *logarítmic* (ja ho vam veure). Ara bé, en el cas pitjor, quants cops es fa cada una?
  - **`obtenir_min`** es fa $|V|$ vegades
  - **`inserir`** es fa $|E|$ vegades

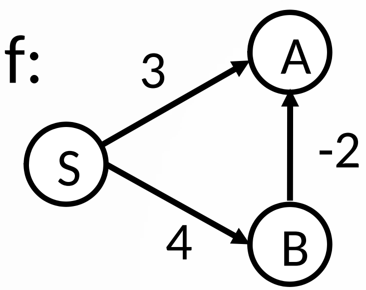
```
while not buit(minheap):
        d,u = obtenir_min(minheap)
        if d == dist[u]:
            for all (u, v) ∈ E:
                if dist[v] > dist[u] + w(u,v):
                    dist[v] = dist[u] + w(u,v)
                    prev[v] = u
                    inserir(minheap,(dist[v],v))
```

- **Cost:** $\Theta((|V|+|E|)\log|V|)$

Algorisme de Dijkstra: Limitacions

Ja hem vist que l'algorisme de Dijkstra només funciona bé si les etiquetes de les arestes són nombres positius: $w: E \to \mathbb{R}^+$

Per exemple, amb el graf:



l'algorisme de Dijkstra *original* ens diria que el millor camí de S a A té cost 3.

Fixeu-vos que si el graf té *cicles negatius* (camins que comencen i acaben al mateix lloc amb un cost total negatiu) el concepte de camí mínim *no té sentit*, ja que cada cop que passem per aquest cicle fem el cost més petit.

Però si no n'hi ha de cicles negatius, el camí més curt entre dos vèrtexos qualsevol té mida *com a molt* **`|V|-1`**

Algorisme de Dijkstra: Limitacions

També hem dit que l'algorisme de Dijkstra és voraç, ja que sempre mira d'actualitzar el vèrtex amb la distància (*cost*) més petita. És per això que necessita un *heap* (una *cua amb prioritat*, en realitat).

Si en lloc de mirar cada cop qui té la distància més petita ens limitem a actualitzar la distància **`|V|-1`** vegades (ja no és un algorisme voraç, però), podem assegurar que, *en absència de cicles negatius*, tindrem els camins mínims d'un vèrtex a qualsevol altre, fins i tot amb etiquetes negatives a les arestes. Aquest algorisme s'anomena ***algorisme de Bellman-Ford***.

L'algorisme de Bellman-Ford, però, és *menys eficient* que l'algorisme de Dijkstra, i només té sentit aplicar-lo a grafs dirigits.

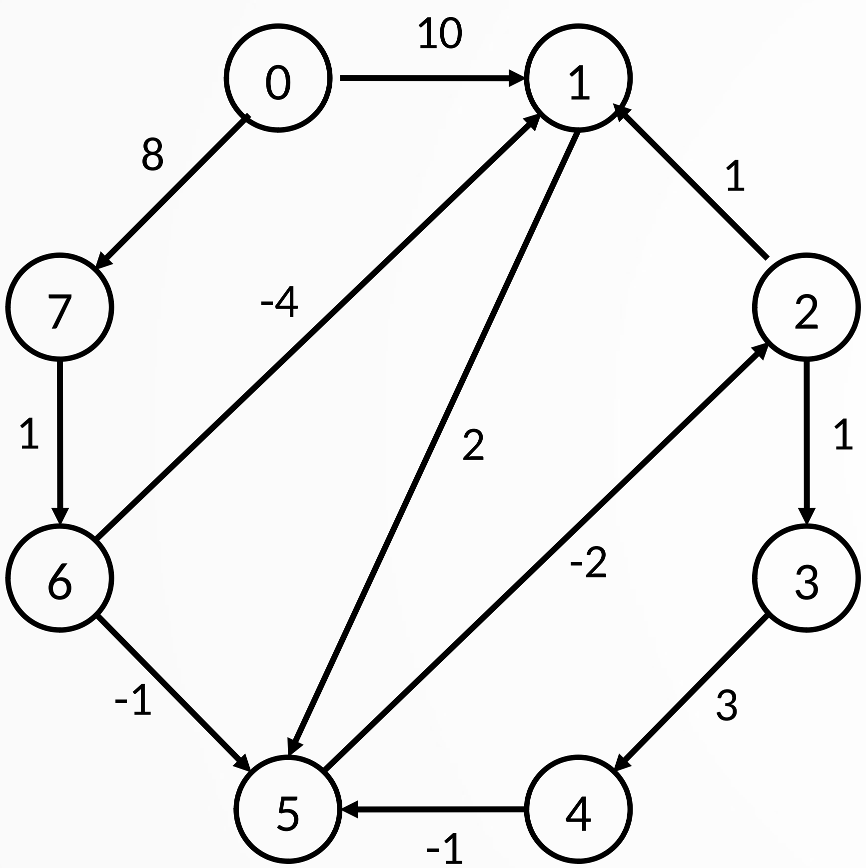
Algorisme de Bellman-Ford

```
def bellman_ford(G, w, s):
    # Entrada: Graf G=(V,E) dirigit, vèrtex inicial s, etiquetes d'arestes w
    #          No ha d'haver cicles negatius en G
    # Sortida: dist[u] distància de s, prev[u] predecessor en camí mínim

    for all u ∈ V:
        dist[u] = float('inf')
        prev[u] = None
    dist[s] = 0

    repeat |V|-1 times:
        for all (u, v) ∈ E:
            if dist[v] > dist[u] + w(u,v):
                dist[v] = dist[u] + w(u,v)
                prev[v] = u
```

Cost: $\Theta(|V| \cdot |E|)$

Algorisme de Bellman-Ford

- Exemple:



	Iteració							
Node	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0	0	0	0	0	0	0	0
1	∞	10	10	5	5	5	5	5
2	∞	∞	∞	10	6	5	5	5
3	∞	∞	∞	∞	11	7	6	6
4	∞	∞	∞	∞	∞	14	10	9
5	∞	∞	12	8	7	7	7	7
6	∞	∞	9	9	9	9	9	9
7	∞	8	8	8	8	8	8	8

Algorisme de ~~Bellman-Ford~~

Quantamagazine

Finally, a Fast Algorithm for Shortest Paths on Negative Graphs

By Ben Brubaker

January 18, 2023

Researchers can now find the shortest route through a network nearly as fast as theoretically possible, even when some steps can cancel out others.

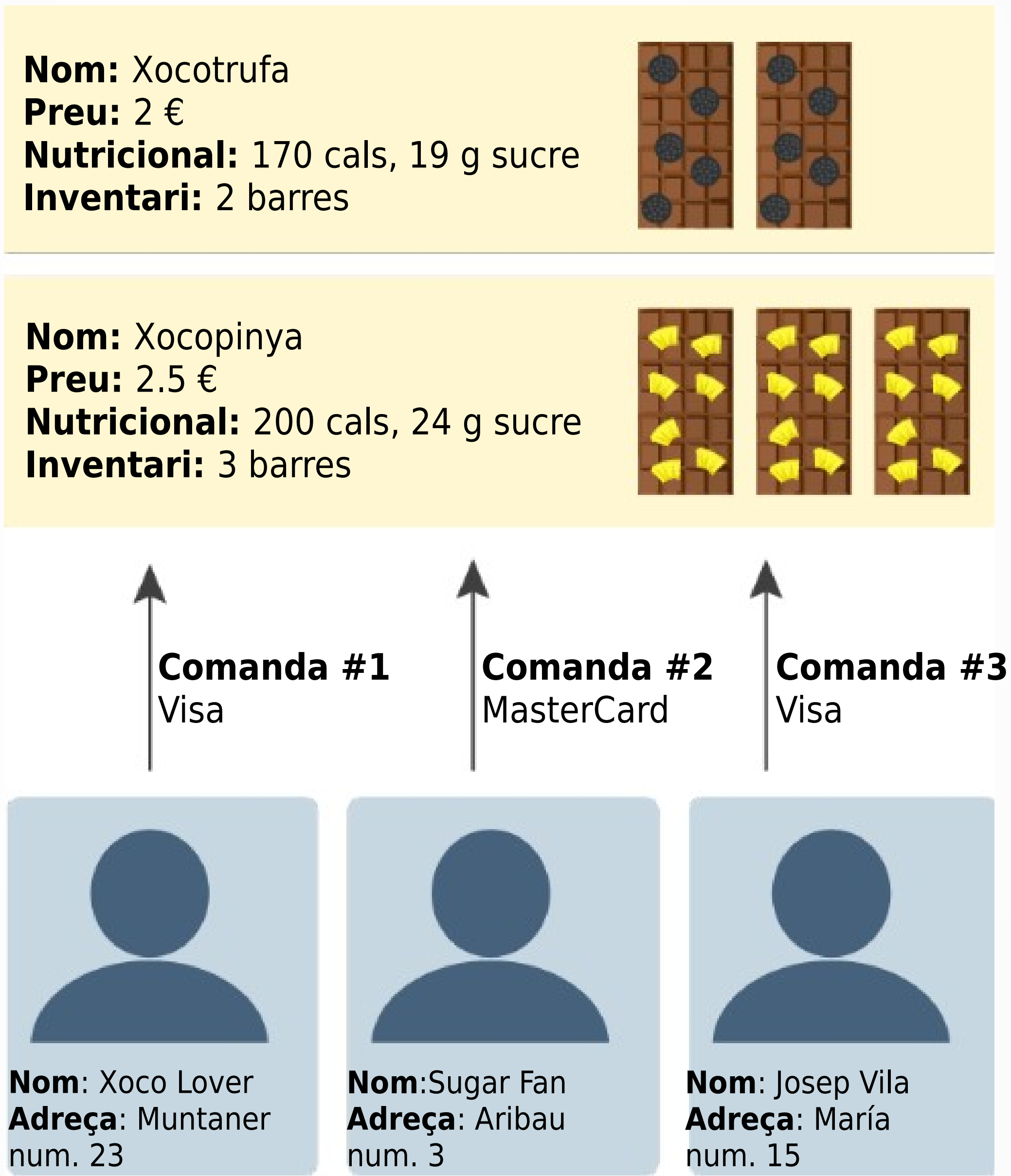
Grafs: Consideracions finals

- El món dels grafs i algorismes associats és immens. Aquí només hem vist els més bàsics i fonamentals.
- Hi ha molts problemes que es poden plantejar en termes de grafs i que tenen aplicacions en el *món real*™:
  - Trobar els arbres d'expansió mínims (MST; *minimum spanning tree*)
  - Ordenació topològica
  - Trobar els components connexos
  - Els grafs complexos (*small-world*, *scale-free*) i les seves propietats
  - Trobar camins mínims amb l'ajut d'heurístics
  - etc.
- Algorismes per resoldre alguns d'aquests problemes els veureu a **ABIA**.

Objectes (i classes)

Objectes (i classes): Motivació

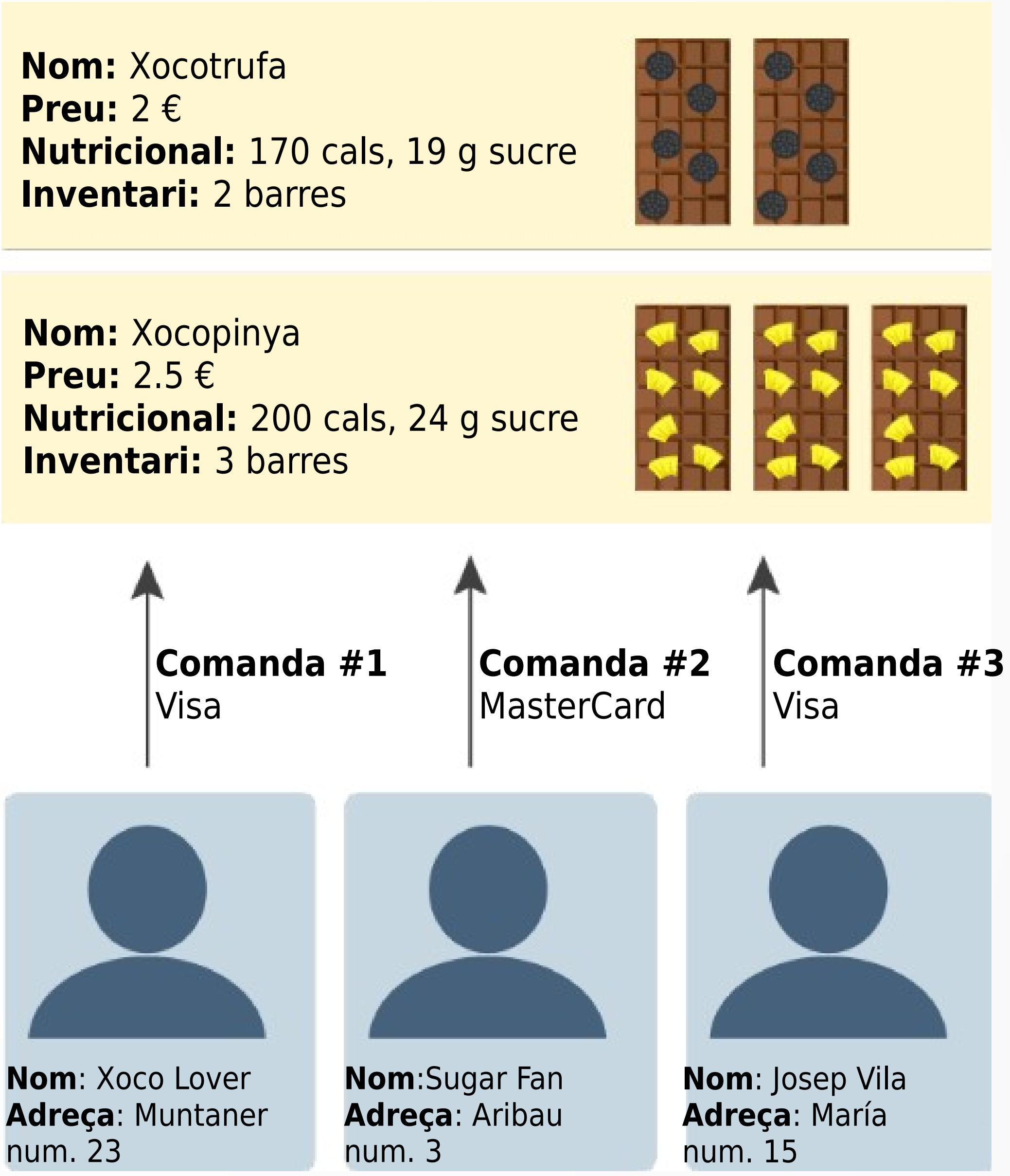
- La botiga de xocolata



Objectes (i classes): Motivació

- ***Podríem*** fer abstracció de dades fent servir només funcions...

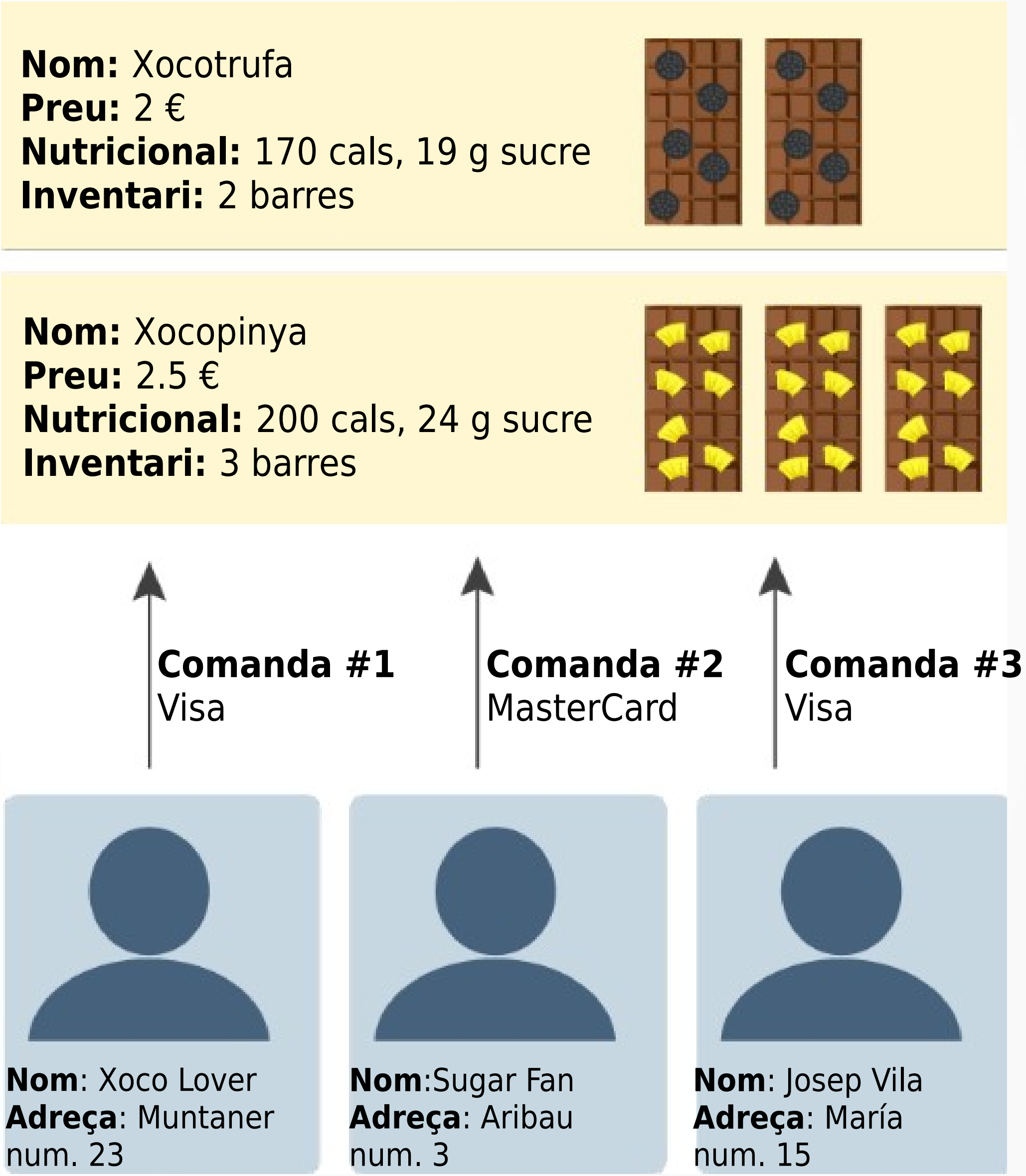
```
# Gestió de l'inventari
afegir_producte(nom, preu, nutricio)
nom(producte)
info_nutricional(producte)
incrementa_inventari(producte, quantitat)
decrementa_inventari(producte, quantitat)

# Gestió clients
nou_client(nom, adreca)
adreca_amb_format(client)

# Gestió compres
comanda(client, producte, quantitat, info)
rastreja(numero_comanda)
devolucio(numero_comanda, rao)
```



Objectes (i classes): De les funcions als objectes

- També podem organitzar aquesta abstracció de dades al voltant dels ***objectes***

```
# Gestió de l'inventari
Producte(nom, preu, nutricio)
Producte.nom()
Producte.info_nutricional()
Producte.incrementa_inventari(quantitat)
Producte.decrementa_inventari(quantitat)

# Gestió clients
Client(nom, adreca)
Client.adreca_amb_format()

# Gestió compres
Comanda(client, producte, quantitat, info)
Comanda.rastreja()
Comanda.devolucio(rao)
```



- Els objectes agrupen ***estat*** i ***comportament***

Objectes (i classes): Conceptes

- Una ***classe*** és una *plantilla* per definir nous tipus de dades.

- Un ***objecte*** és una *instància* d'una classe

- Cada objecte té atributs anomenats ***variables d'instància***, que descriuen el seu *estat*.

- Cada objecte té atributs anomenats ***mètodes***, que descriuen el seu *comportament*.

Python té una *sintaxi especial* per descriure classes i manipular objectes

Classes

- Una *classe* pot:
  - Inicialitzar els valors de les ***variables d'instància***.
  - Definir ***mètodes***, funcions associades a cada objecte ***instància*** de la ***classe***. Sovint es fan servir per canviar, o consultar, el valor de les ***variables d'instància***.

```
class Producte:

    # Inicialitzar els valors de les variables d'instància
    # Definir mètodes
```

- Anem a programar una classe!

Classes

Exemple complet:

```python
# Defineix un nou tipus de dades
class Producte:

    # Inicialitzacions
    def __init__(self, nom, preu, nutricio_info):
        self._nom = nom
        self._preu = preu
        self._nutricio_info = nutricio_info
        self._inventari = 0

    # Definir mètodes
    def incrementa_inventari(self, quantitat):
        self._inventari += quantitat

    def decrementa_inventari(self, quantitat):
        self._inventari -= quantitat

    def nom(self):
        return "Botiga de Xocolata: " + self._nom

    def informe_inventari(self):
        if self._inventari == 0:
            return "No queden barretes!"
        return f"Hi ha {self._inventari} barretes."
```

```python
# Utilització
pinya_bar = Producte("XocoPinya", 2.5, ["200 calories", "24 g sucre"])
pinya_bar.incrementa_inventari(2)
```

Classes: Anem per parts...

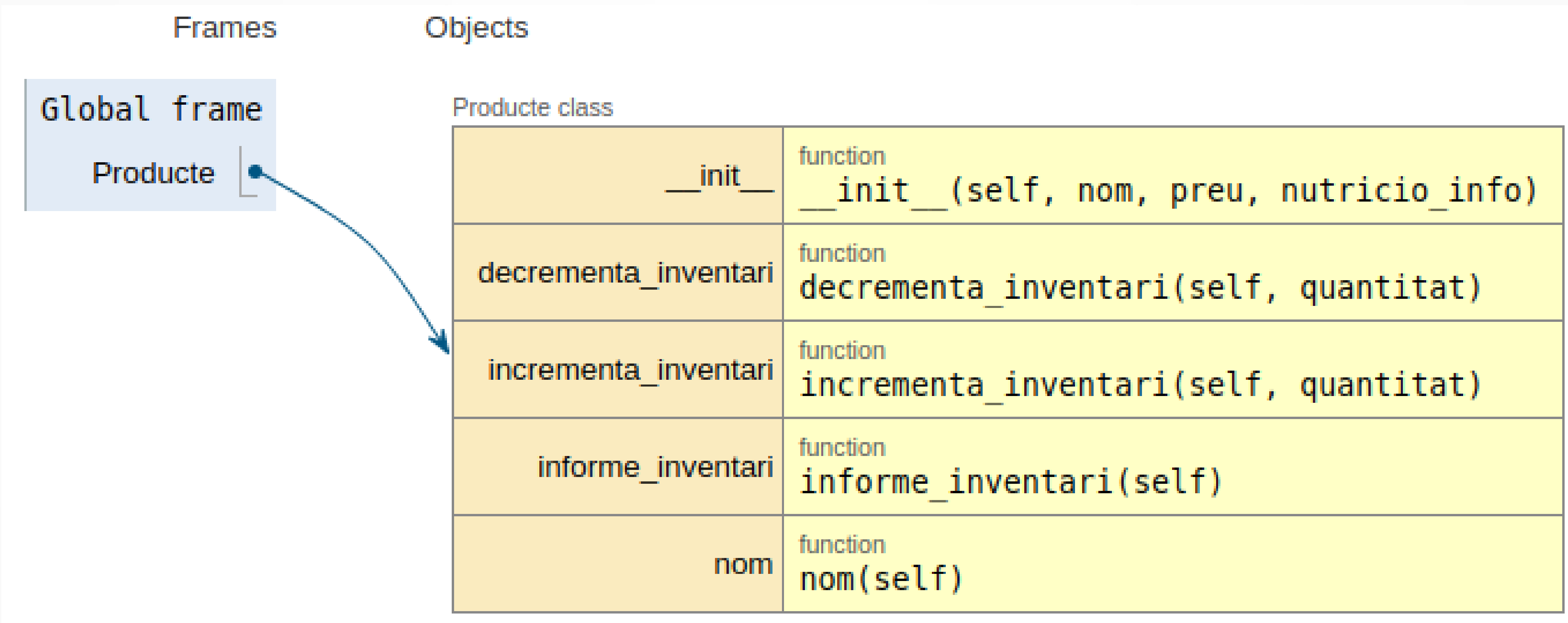
- Definició d'una classe:

```
# Defineix un nou tipus de dades
class Producte:

    # Inicialitzacions
    def __init__(self, nom, preu, nutricio_info):

    # Definir mètodes
    def incrementa_inventari(self, quantitat):
    def decrementa_inventari(self, quantitat):
    def nom(self):
    def informe_inventari(self):
```

- Una instrucció `class` defineix una nova classe i lliga la classe al nom que li hem donat dins l'entorn actual (on s'executa la definició de la classe).
- Els `def` interns creen atributs de la classe (***no** nous noms dins l'entorn*)

Classes: Anem per parts...

- Definició d'una classe:



Classes: Instanciació (creació d'objectes)

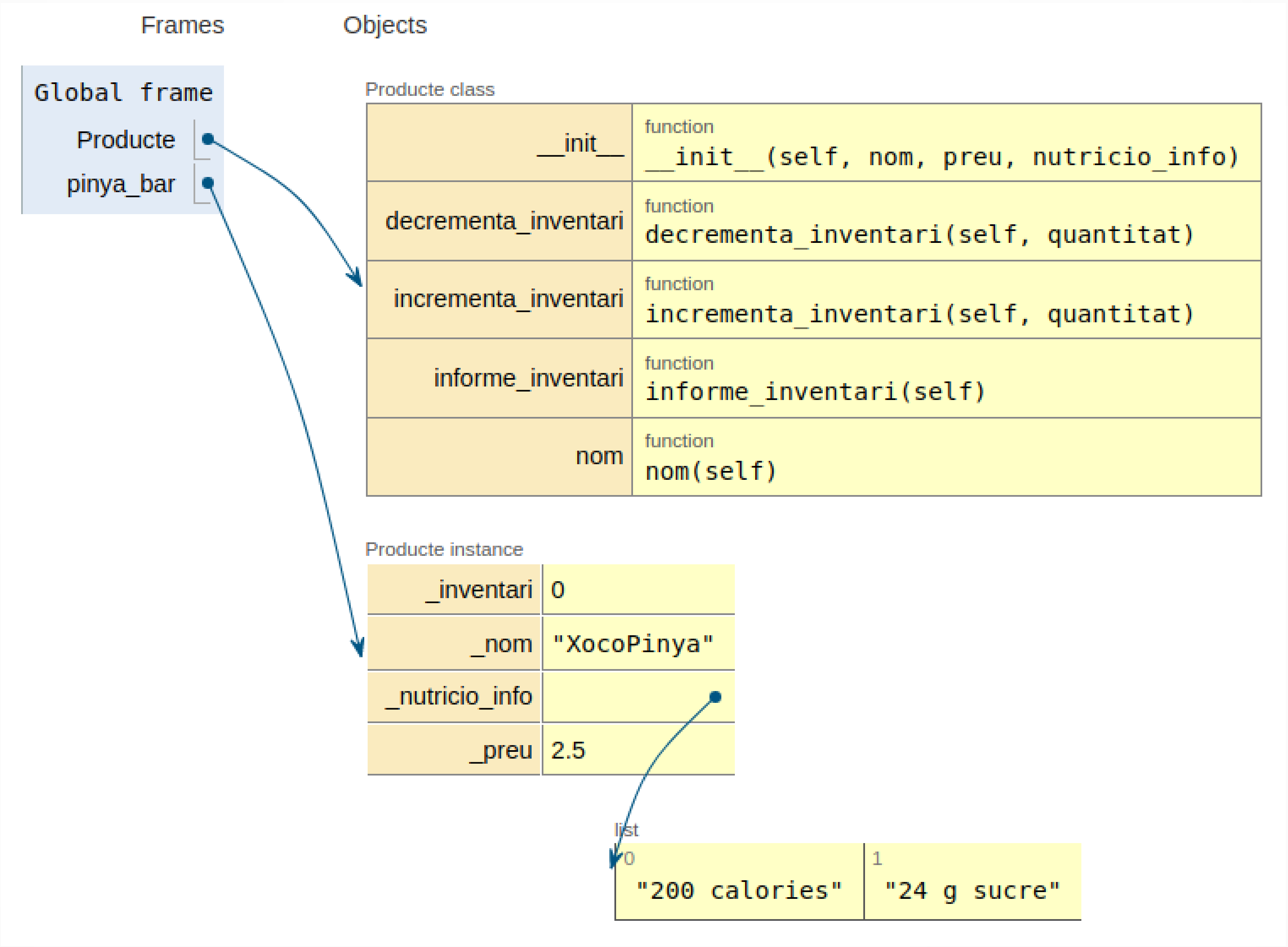
```
pinya_bar = Producte("XocoPinya", 2.5, ["200 calories", "24 g sucre"])
```

- Anomenem ***constructor*** a **`Producte(...)`**.
- Quan s'invoca el constructor:
  - Una nova ***instància*** de la classe és creada (un ***objecte*** nou)
  - El mètode **`__init__`** s'invoca *amb el nou objecte com a primer argument* (anomenat **`self`**), més els arguments proporcionats a la invocació del constructor.

```
class Producte:

    def __init__(self, nom, preu, nutricio_info):
        self._nom = nom
        self._preu = preu
        self._nutricio_info = nutricio_info
        self._inventari = 0
```

Classes: Instanciació (creació d'objectes)

```
pinya_bar = Producte("XocoPinya", 2.5, ["200 calories", "24 g sucre"])
```



Classes: La notació *del punt*

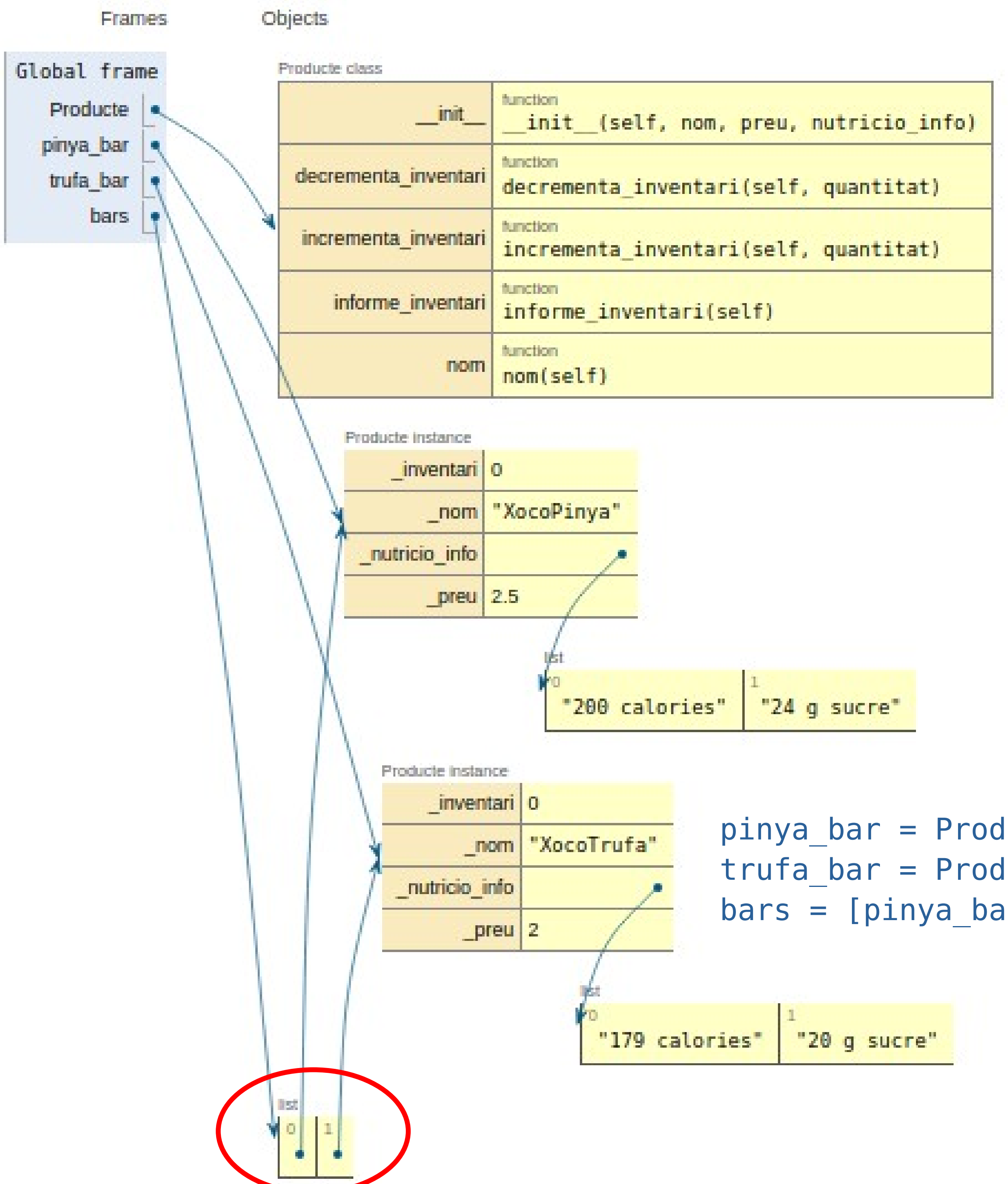
- A tots els atributs d'un objecte (això inclou mètodes i variables d'instància) es pot accedir mitjançant ***la notació del punt***.

```
pinya_bar.incrementa_inventari(2)
```

- Això s'avalua al valor de l'atribut corresponent (el mètode `incrementa_inventari`) de l'objecte vinculat al nom `pinya_bar`, i es fa el que calgui (en aquest cas s'invoca el mètode amb argument `2`),

- Al costat esquerre de la notació del punt pot haver *qualsevol* expressió que s'avalui a una referència a un objecte:

```
bars = [pinya_bar, trufa_bar]
bars[0].incrementa_inventari(2)
```




```
pinya_bar = Producte("XocoPinya", 2.5, ["200 calories", "24 g sucre"])
trufa_bar = Producte("XocoTrufa", 2, ["179 calories", "20 g sucre"])
bars = [pinya_bar, trufa_bar]
```

Classes: Variables d'Instància

- Les ***variables d'instància*** són atributs de dades que descriuen l'*estat* d'un objecte.

- En l'exemple, l'**`__init__`** inicialitza quatre variables d'instància:

```
class Producte:

    def __init__(self, nom, preu, nutricio_info):
        self._nom = nom                            # variable d'instància _nom
        self._preu = preu                          # variable d'instància _preu
        self._nutricio_info = nutricio_info  # variable d'instància _nutricio_info
        self._inventari = 0                        # variable d'instància _inventari
```

- Els mètodes poden canviar els valors de les variables d'instància, o *fins i tot crear-ne de noves!*

Classes: Invocació de mètodes

```
pinya_bar.incrementa_inventari(2)

class Producte:

    def incrementa_inventari(self, quantitat):
        self._inventari += quantitat
```

- El mètode `pinya_bar.incrementa_inventari` és un mètode ***lligat***: Una funció que té el seu primer paràmetre *pre-vinculat* (*pre-bound*) a un valor particular.
- En aquest cas, `self` està pre-vinculat a `pinya_bar`, i `quantitat` pren per valor l'argument `2`.
- És equivalent a `Producte.incrementa_inventari(pinya_bar,2)`

Classes: Variables d'Instància *Dinàmiques*

- Un objecte pot crear noves variables d'instància quan vulgui:

```
class Producte:
    # ...
    def decrementa_inventari(self, quantitat):
        if (self._inventari - quantitat) <= 0:
            self._necessita_estoc = True
        self._inventari -= quantitat
    # ...

pinya_bar = Producte("XocoPinya", 2.5, ["200 calories", "24 g sucre"])
pinya_bar.decrementa_inventari(1)
```

- Ara, `pinya_bar` té un nou valor vinculat a `_inventari` i un nou valor vinculat a una *nova* variable d'instància `_necessita_estoc` (que no era a `__init__`).

Classes: Variables de classe

- Una assignació dins d'una classe, *però no dins del cos de cap mètode*, defineix el que s'anomena ***variable de classe***.

```
class Producte:
    # ...
    _comissio_vendes = 0.07     # Variable de classe
    # ...
    def preu_total(self, quantitat):
        return (self._preu * (1 + self._comissio_vendes)) * quantitat
    # ...
pinya_bar = Producte("XocoPinya", 2.5, ["200 calories", "24 g sucre"])
trufa_bar = Producte("XocoTrufa", 2, ["179 calories", "20 g sucre"])

pinya_bar._comissio_vendes
trufa_bar._comissio_vendes
pinya_bar.preu_total(4)
trufa_bar.preu_total(2)
```

- Les variables de classe són *compartides*, en el sentit que són visibles i modificables per qualsevol instància de la classe. Són atributs de la classe, no de les instàncies.

Classes: Els atributs són públics

- Si es disposa d'una referència a un objecte, es poden consultar i modificar els seus atributs

```
pinya_bar = Producte("XocoPinya", 2.5, ["200 calories", "24 g sucre"])

pinya_bar._inventari
pinya_bar._inventari = 50000
pinya_bar._inventari = -5.00
```

- Fins i tot podem afegir noves variables d'instància a un objecte (!)

```
pinya_bar = Producte("XocoPinya", 2.5, ["200 calories", "24 g sucre"])

pinya_bar.nou_atribut = "això és força lleig"
```

Classes: Atributs "*Privats*"

- Per comunicar el nivell d'accés que es vol proporcionar als atributs, els programadors de Python fan servir la següent ***convenció***:
  - __ (doble guió baix) precedint el nom d'atributs completament privats
  - _ (un guió baix) precedint el nom d'atributs semi-privats
  - Cap guió baix davant els atributs que hom desitja públics

- Això permet a les classes amagar els detalls d'implementació i afegir comprovació d'errors addicional.

- **Atenció**! Els atributs amb nom precedit de __ (doble guió baix) ***no s'hereten*** (veure herència, més endavant), excepte si també tenen el doble guió com a sufix.

Classes: Atributs "*Privats*"

- Si definim:

```
class Producte:
    def __init__(self, nom, preu, nutricio_info):
        self._nom = nom
        self._preu = preu
        self._nutricio_info = nutricio_info
        self.__inventari = 0     # variable d'instància __inventari
    def inventari(self):
        return self.__inventari
    # ...
```

fixem-nos que...

```
>>> pinya_bar = Producte("XocoPinya", 2.5, ["200 calories", "24 g sucre"])
>>> pinya_bar.incrementa_inventari(2)
>>> pinya_bar.inventari()   # puc modificar i consultar __inventari
2
>>> pinya_bar.__inventari   # però no puc fer-ho directament
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Producte' object has no attribute '__inventari'
>>> pinya_bar._nom
'XocoPinya'
```

Qüestionari: Instàncies múltiples

- Podem tenir diverses instàncies d'una mateixa classe (suposem que tenim definides les classes **`Producte`** i **`Client`**):

```
pinya_bar = Producte("XocoPinya", 2.5, ["200 calories", "24 g sucre"])

cli1      = Client("Xoco Lover", ["Muntaner", "23"])

cli2      = Client("Sugar Fan", ["Aribau", "3"])
```

- Quantes classes intervenen en aquest codi?
- Quantes instàncies de cada una?

Qüestionari: Instàncies múltiples

- Podem tenir diverses instàncies d'una mateixa classe (suposem que tenim definides les classes **`Producte`** i **`Client`**):

```
pinya_bar = Producte("XocoPinya", 2.5, ["200 calories", "24 g sucre"])

cli1      = Client("Xoco Lover", ["Muntaner", "23"])

cli2      = Client("Sugar Fan", ["Aribau", "3"])
```

- Quantes classes intervenen en aquest codi? Dues, **`Producte`** i **`Client`**
- Quantes instàncies de cada una? 1 de **`Producte`** i 2 de **`Client`**

Qüestionari: Gestió de l'estat

- Els objectes fan servir variables d'instància per descriure el seu estat. Una ***bona pràctica*** és ***amagar la representació de l'estat i gestionar-lo totalment via crides a mètodes***.

```
>>> pinya_bar = Producte("XocoPinya", 2.5, ["200 calories", "24 g sucre"])

>>> pinya_bar.informe_inventari()
'No queden barretes!'

>>> pinya_bar.incrementa_inventari(2)
>>> pinya_bar.informe_inventari()
'Hi ha 2 barretes.'
```

- Quin és l'estat inicial?
- Què canvia l'estat?

Qüestionari: Gestió de l'estat

- Els objectes fan servir variables d'instància per descriure el seu estat. Una ***bona pràctica*** és ***amagar la representació de l'estat i gestionar-lo totalment via crides a mètodes***.

```
>>> pinya_bar = Producte("XocoPinya", 2.5, ["200 calories", "24 g sucre"])

>>> pinya_bar.informe_inventari()
'No queden barretes!'

>>> pinya_bar.incrementa_inventari(2)
>>> pinya_bar.informe_inventari()
'Hi ha 2 barretes.'
```

- Quin és l'estat inicial? 0 barretes a l'inventari
- Què canvia l'estat? L'utilització del mètode `incrementa_inventari`, que canvia la variable d'instància `_inventari`

Qüestionari: Var. d'instància vs. Var. de classe

- Suposem:

```python
class Client:
    _salutacio = "Apreciat/Apreciada"

    def __init__(self, nom, adreca):
        self._nom = nom
        self._adreca = adreca

    def salutacio(self):
        return f"{self._salutacio} {self._nom},"

    def adreca_amb_format(self):
        return "\n".join(self._adreca)

client1 = Client("Xoco Lover", ["Muntaner", "23"])
```

- Quines són les variables d'instància?
- Quines són les variables de classe?

Qüestionari: Var. d'instància vs. Var. de classe

- Suposem:

```
class Client:
    _salutacio = "Apreciat/Apreciada"

    def __init__(self, nom, adreca):
        self._nom = nom
        self._adreca = adreca

    def salutacio(self):
        return f"{self._salutacio} {self._nom},"

    def adreca_amb_format(self):
        return "\n".join(self._adreca)

client1 = Client("Xoco Lover", ["Muntaner", "23"])
```

- Quines són les variables d'instància? `_nom` i `_adreca`
- Quines són les variables de classe? `_salutacio`

Classes: Herència i Composició - Motivació

- Suposem que volem construir una aplicació: "*Conservació d'Animals*"...

Classes: Herència i Composició - Motivació

- Caldria programar les classes...

```
Panda(...)
Lleo(...)
Conill(...)
Elefant(...)
Voltor(...)
Menjar(...)
etc...
```

Classes: Herència i Composició

- Comencem pel més senzill…

```python
class Menjar:

    def __init__(self, nom, tipus, calories):
        self.nom = nom
        self.tipus = tipus
        self.calories = calories
```

i aquesta classe la fariem servir…

```python
broquil = Menjar("Broquil", "vegetal", 20)
moll_os = Menjar("Moll de l'os", "carn", 100)
```

Classes: Herència i Composició

La classe **Elefant**:

```python
class Elefant:
    nom_especie = "Elefant Sabana Africana"
    nom_cientific = "Loxodonta africana"
    requeriment_caloric = 8000

    def __init__(self, nom, edat=0):
        self.nom = nom
        self.edat = edat
        self.calories_menjades = 0
        self.felicitat = 0

    def juga(self, num_hores):
        self.felicitat += (num_hores * 4)
        print("WHEEE... HORA DE  JUGAR!")

    def menja(self, menjar):
        self.calories_menjades += menjar.calories
        print(f"nyam nyam {menjar.nom}")
        if self.calories_menjades > self.requeriment_caloric:
            self.felicitat -= 1
            print("Ugh... estic massa ple")

    def interactua_amb(self, animal2):
        self.felicitat += 1
        print(f"Yupi! què bé m'ho passo amb {animal2.nom}")
```

```python
# Utilització
el1 = Elefant("Willaby", 5)
el2 = Elefant("Wallaby", 3)
el1.juga(2)
el1.interactua_amb(el2)
```

Classes: Herència i Composició

La classe **`Conill`**:

```python
class Conill:
    nom_especie = "Conill Europeu"
    nom_cientific = "Oryctolagus cuniculus"
    requeriment_caloric = 200

    def __init__(self, nom, edat=0):
        self.nom = nom
        self.edat = edat
        self.calories_menjades = 0
        self.felicitat = 0

    def juga(self, num_hores):
        self.felicitat += (num_hores * 10)
        print("WHEEE... HORA DE  JUGAR!")

    def menja(self, menjar):
        self.calories_menjades += menjar.calories
        print(f"nyam nyam {menjar.nom}")
        if self.calories_menjades > self.requeriment_caloric:
            self.felicitat -= 1
            print("Ugh... estic massa ple")

    def interactua_amb(self, animal2):
        self.felicitat += 4
        print(f"Yupi! què bé m'ho passo amb {animal2.nom}")
```

```python
# Utilització
c1 = Conill("Mister Wabbit", 3)
c2 = Conill("Bugs Bunny", 2)
c1.menja(broquil)
c1.interactua_amb(c2)
```

Classes: Herència i Composició

- Alguna cosa en comú???

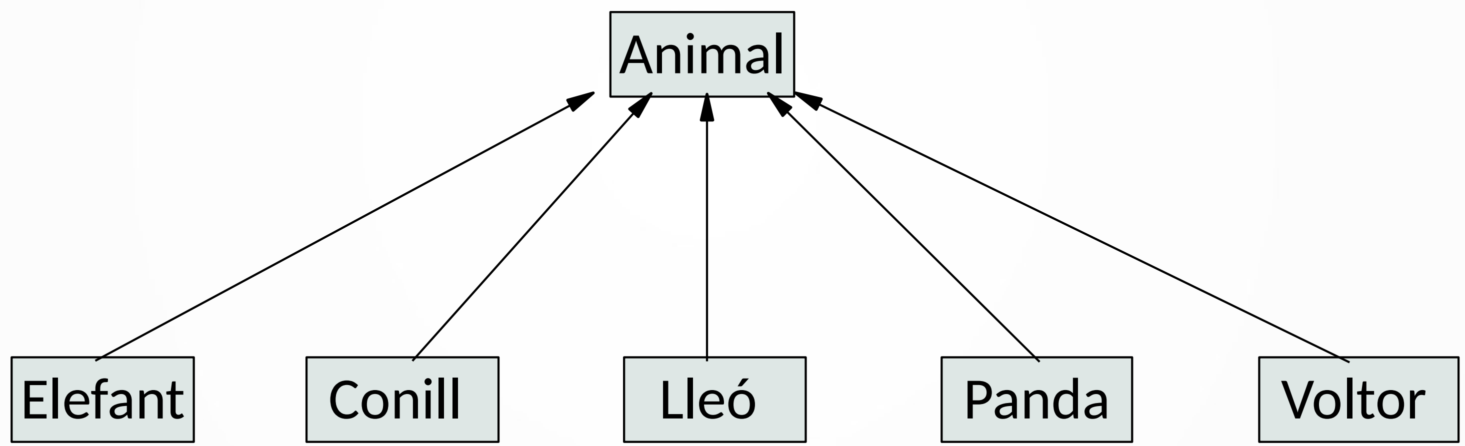
- L'**`Elefant`** i el **`Conill`** són ***animals***, de manera que tenen *atributs similars*. En lloc de ***repetir*** el codi, el que farem és ***heretar*** el codi.

```
Elefant

# Variables de classe
nom_especie
nom_cientific
requeriment_caloric


# Variables d'instància
nom
edat
calories_menjades
felicitat

# Mètodes
juga(num_hores)
menja(menjar):
interactua_amb(altre)
```

```
Conill

# Variables de classe
nom_especie
nom_cientific
requeriment_caloric


# Variables d'instància
nom
edat
calories_menjades
felicitat

# Mètodes
juga(num_hores)
menja(menjar):
interactua_amb(altre)
```

Classes: Classe base i subclasses

- Quan diverses classes comparteixen atributs similars, es pot reduir la redundància en el codi definint una ***classe base*** i ***subclasses*** que ***heretin*** d'aquesta classe base (també coneguda com a ***superclasse***).



Classes: Classe base

- La ***classe base*** conté variables d'instància (*estat*) i mètodes (*comportament*) **comuns** a les subclasses...

```python
class Animal:
    nom_especie = "Animal"
    nom_cientific = "Animalia"
    requeriment_caloric = 200
    multiplicador_joc = 2
    increment_interaccio = 1

    def __init__(self, nom, edat=0):
        self.nom = nom
        self.edat = edat
        self.calories_menjades = 0
        self.felicitat = 0

    def juga(self, num_hores):
        self.felicitat += (num_hores * self.multiplicador_joc)
        print("WHEEE... HORA DE  JUGAR!")

    def menja(self, menjar):
        self.calories_menjades += menjar.calories
        print(f"nyam nyam {menjar.nom}")
        if self.calories_menjades > self.requeriment_caloric:
            self.felicitat -= 1
            print("Ugh... estic massa ple")

    def interactua_amb(self, animal2):
        self.felicitat += self.increment_interaccio
        print(f"Yupi! què bé m'ho passo amb {animal2.nom}")
```

Classes: Subclasses

- Per definir una ***subclasse*** que ***hereti*** d'aquesta ***superclasse***. cal escriure el nom de la subclasse seguit del nom de la classe base entre parèntesi:

```python
class Panda(Animal):
```

  Aleshores les subclasses només requereixen descriure *el codi que els és propi*. Es pot redefinir qualsevol cosa d'allò heretat (*s'hereta* ***tot***!): variables de classe, definicions de mètodes, o constructor. Quan hom redefineix alguna cosa heretada s'anomena ***sobreescriptura*** (***overriding***).

- La subclasse més simple no afegeix res ni sobreescriu res:

```python
class Ameba(Animal):
    pass
```

Classes: Sobreescriure variables de classe

- Les subclasses poden sobreescriure variables de classe heretades, ***i poden afegir-ne de noves***.

```python
class Elefant(Animal):
    nom_especie = "Elefant Sabana Africana"
    nom_cientific = "Loxodonta africana"
    requeriment_caloric = 8000
    multiplicador_joc = 4
    increment_interaccio = 1
    nombre_ullals = 2


class Conill(Animal):
    nom_especie = "Conill Europeu"
    nom_cientific = "Oryctolagus cuniculus"
    requeriment_caloric = 200
    multiplicador_joc = 10
    increment_interaccio = 4
    nombre_cries = 12
```

Classes: Sobreescriure mètodes

- Si una subclasse sobreescriu un mètode, Python farà servir aquest mètode amb les instàncies de la subclasse, en lloc del mètode de la classe base.

```
class Panda(Animal):
    nom_especie = "Giant Panda"
    nom_cientific = "Ailuropoda melanoleuca"
    requeriment_caloric = 6000

    def interactua_amb(self, altre):
        print(f"Sóc un Panda, sóc solitari, ves-te'n {altre.nom}!")
```

- Com el farem servir? De cap manera especial…

```
panda1 = Panda("oset", 6)
panda2 = Panda("taqueta", 3)
panda1.interactua_amb(panda2)
```

Classes: Utilitzar mètodes heretats

- Per fer servir un mètode heretat (encara que l'estiguem sobreescrivint) cal fer-ne referència mitjançant allò retornat per la funció **super()**

```
class Lleo(Animal):
    nom_especie = "Lleó"
    nom_cientific = "Panthera Leo"
    requeriment_caloric = 3000

    def menja(self, menjar):
        if menjar.tipus == "carn":
            super().menja(menjar)
```

- Com el farem servir? De cap manera especial...

```
ossos = Menjar("Ossos", "carn")
mufasa = Lleo("Mufasa", 10)
mufasa.menja(ossos)
```

Classes: Més sobre `super()`

- L'expressió **`super().atribut`** fa referència a la definició d'**`atribut`** a la superclasse de la classe de la que **`self`** és instància. Per exemple:

```
def menja(self, menjar):
        if menjar.tipus == "carn":
            super().menja(menjar)
```

…és el mateix que:

```
def menja(self, menjar):
        if menjar.tipus == "carn":
            Animal.menja(self,menjar)
```

- És considera millor *estil* fer servir **`super()`**

Classes: Sobreescriure `__init__`

- De la mateixa manera que ja hem vist amb altres mètodes, cal cridar explícitament **`super().__init__()`** si volem fer servir la funcionalitat d'**`__init__`** de la classe base.

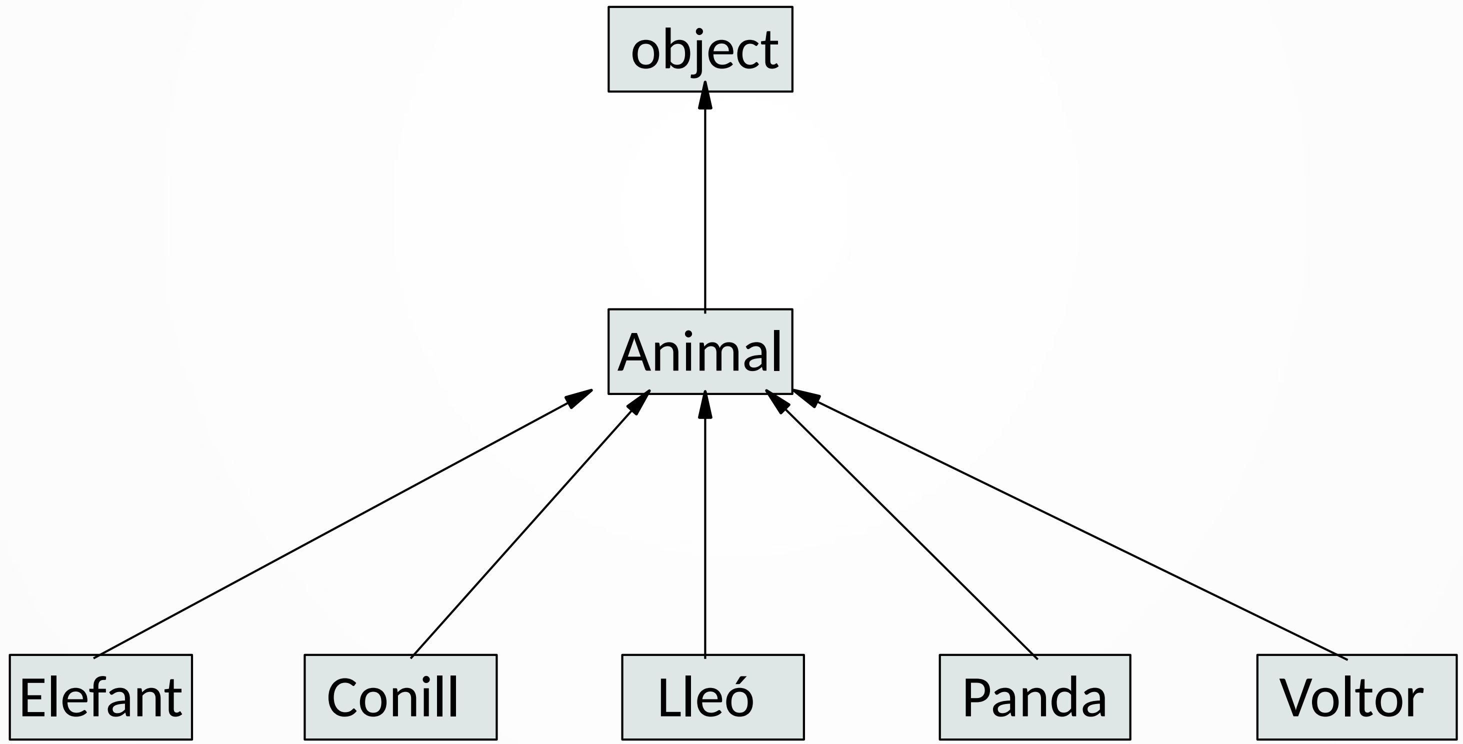
```
class Elefant(Animal):
    nom_especie = "Elefant"
    nom_cientific = "Loxodonta"
    requeriment_caloric = 8000

    def __init__(self, nom, edat=0):
        super().__init__(nom, edat)
        if edat < 1:
            self.requeriment_caloric = 1000
        elif edat < 5:
            self.requeriment_caloric = 3000
```

- Com el farem servir?

```
elly = Elefant("Ellie", 3)
elly.requeriment_caloric          # 3000
```

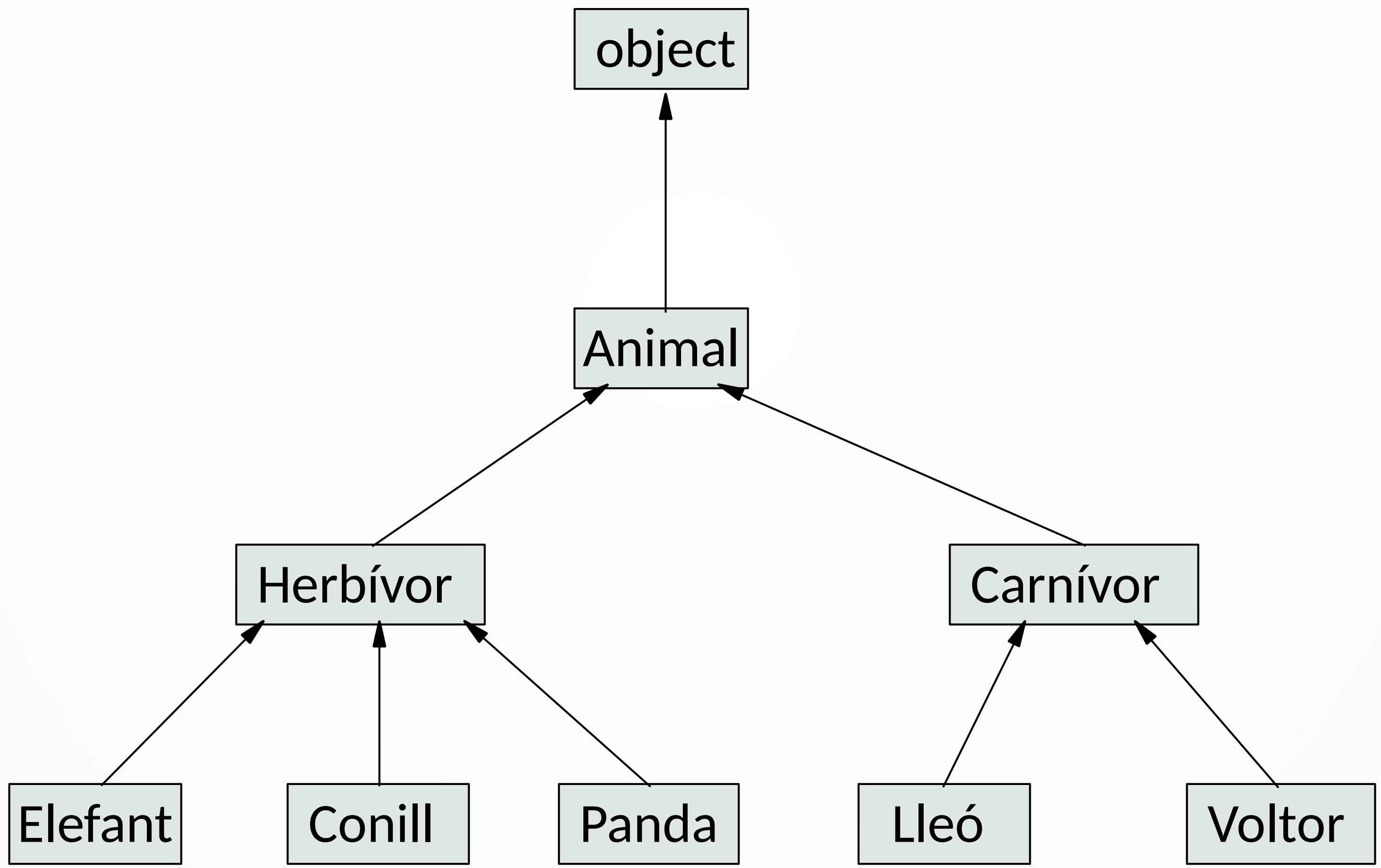
Classes: La classe `object`

- ***Totes les classes*** en Python 3 són, ***implicitament***, extensions (subclasses) de la classe **`object`**.



Classes: Afegir capes d'herència

- Les classes que defineix el programador formen jerarquies que poden tenir diversos nivells:



Classes: Afegir capes d'herència

- Primer definim les noves classes:

```python
class Herbivor(Animal):

    def menja(self, menjar):
        if menjar.tipus == "carn":
            self.felicitat -= 5
        else:
            super().menja(menjar)

class Carnivor(Animal):

    def menja(self, menjar):
        if menjar.tipus == "carn":
            super().menja(menjar)
```

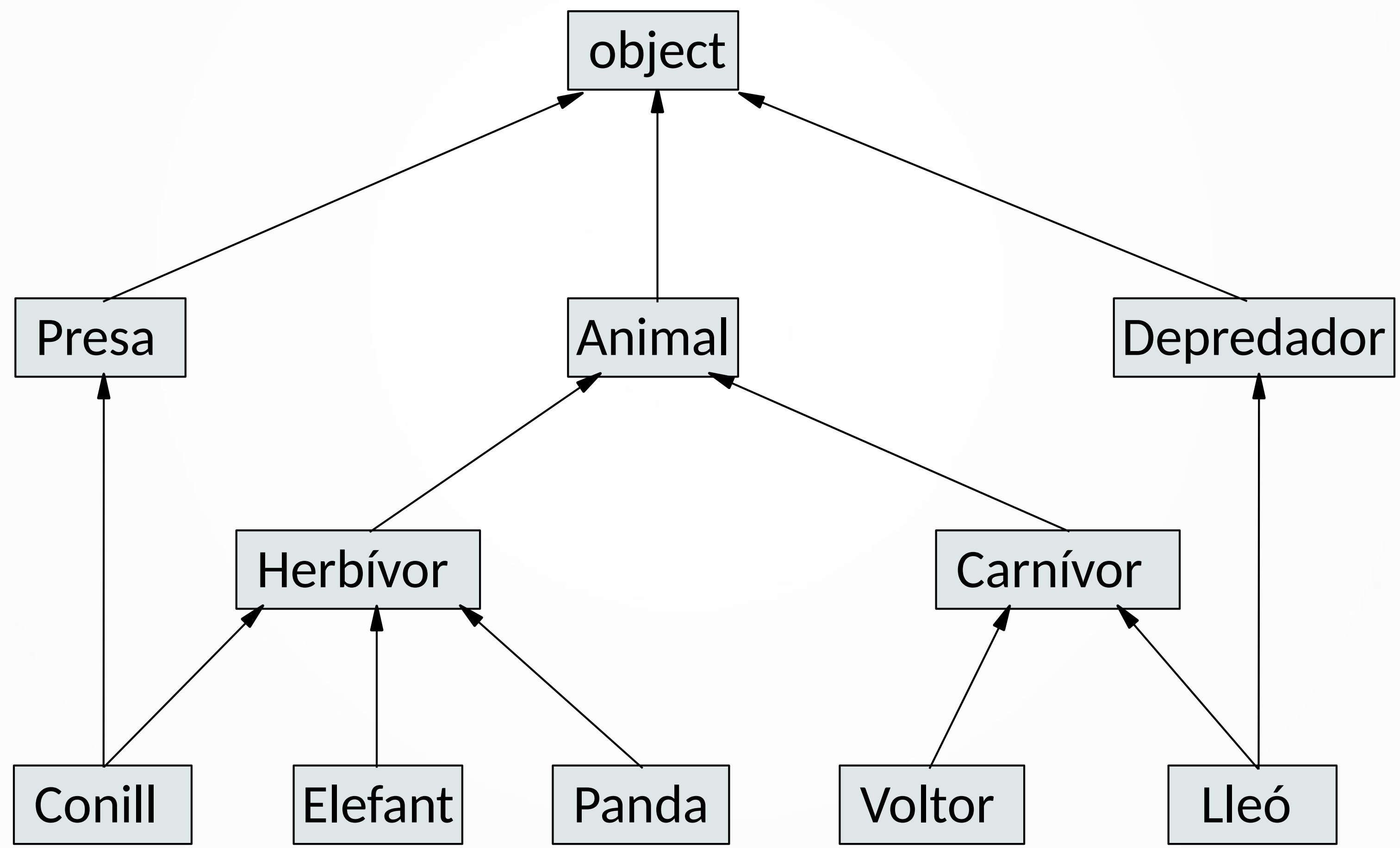
- I després canviem la classe base:

```python
class Conill(Herbivor):
class Elefant(Herbivor):
class Panda(Herbivor):
```

```python
class Lleo(Carnivor):
class Voltor(Carnivor):
```

Classes: Herència Múltiple

- A Python una classe pot heretar de ***múltiples classes base***:



Classes: Herència Múltiple

- Primer definim les noves classes base…

```
class Depredador(Animal):

    def interactua_amb(self, altre):
        if altre.tipus == "carn":
            self.menja(altre)
            print("Ho sento, sóc un depredador")
        else:
            super().interactua_amb(altre)


class Presa(Animal):
    tipus = "carn"
    calories = 200
```

Classes: Herència Múltiple

- ...aleshores l'herència s'expressa posant tots els noms entre parèntesi:

```
class Conill(Presa,Herbivor):

class Lleo(Depredador,Carnivor):
```

- Python pot trobar els atributs en qualsevol de les classes base:

```
>>> r = Conill("Bus",4)
>>> r.juga()
>>> r.tipus
>>> r.menja(Menjar("Pastanaga", "vegetal"))
>>> l = Lleo("Scar", 8)
>>> l.menja(Menjar("Zebra", "carn"))
>>> l.interactua_amb(r)
```

Classes: Herència Múltiple

- ...aleshores l'herència s'expressa posant tots els noms entre parèntesi:

```
class Conill(Presa,Herbivor):

class Lleo(Depredador,Carnivor):
```

- Python pot trobar els atributs en qualsevol de les classes base:

```
>>> r = Conill("Bus",4)                         # Animal __init__
>>> r.juga()                                    # Mètode d'Animal
>>> r.tipus                                     # Variable de classe de Presa
>>> r.menja(Menjar("Pastanaga", "vegetal"))     # Mètode d'Herbivor
>>> l = Lleo("Scar", 8)                         # Animal __init__
>>> l.menja(Menjar("Zebra", "carn"))            # Mètode de Carnivor
>>> l.interactua_amb(r)                         # Mètode de Depredador
```

Recordatori: La identitat

- **exp_0 is exp_1** s'avalua a **True** si **exp_0** i **exp_1** s'avaluen al mateix objecte

```
mufasa = Lleo("Mufasa",15)
nala = Lleo("Nala",16)

mufasa is mufasa        # True
mufasa is nala          # False
mufasa is not nala      # True
nala is not None        # True
```

Classes: Composició

- Un objecte pot contenir ***referències*** a altres objectes

- En el nostre exemple del *Conservatori d'Animals* podem trobar…
  - Un animal té una parella
  - Un animal té una mare
  - Un animal té fills
  - Un *Conservatori d'Animals* té animals

Classes: Referenciar altres instàncies

- Una variable d'instància pot fer referència a altres instàncies...

```
class Animal:

    def aparellament_amb(self, altre):
        if altre is not self and \
           altre.nom_especie == self.nom_especie:
            self.parella  = altre
            altre.parella = self
```

- Com el farem servir?

Classes: Referenciar altres instàncies

- Una variable d'instància pot fer referència a altres instàncies...

```
class Animal:

    def aparellament_amb(self, altre):
        if altre is not self and \
           altre.nom_especie == self.nom_especie:
            self.parella  = altre
            altre.parella = self
```

- Com el farem servir? De cap manera especial...

```
bugs = Conill("Bugs Bunny", 3)
lola = Conill("Lola Bunny", 2)
bugs.aparellament_amb(lola)
```

Classes: Referenciar llistes d'instàncies

- Una variable d'instància pot també fer referència a grups d'altres instàncies...

```python
class Conill(Animal):

    def reproduccio_com_conills(self):
        if self.parella is None:
            return []
        self.petits = []
        for _ in range(0, self.nombre_cries):
            self.petits.append(Conill("conillet", 0))
        return self.petits
```

- Com el farem servir?

Classes: Referenciar llistes d'instàncies

- Una variable d'instància pot també fer referència a grups d'altres instàncies...

```
class Conill(Animal):

    def reproduccio_com_conills(self):
        if self.parella is None:
            return []
        self.petits = []
        for _ in range(0, self.nombre_cries):
            self.petits.append(Conill("conillet", 0))
        return self.petits
```

- Com el farem servir? De cap manera especial...

```
bugs = Conill("Bugs Bunny", 3)
lola = Conill("Lola Bunny", 2)
bugs.aparellament_amb(lola)
fillets = lola.reproduccio_com_conills()
```

Classes: Interficie comuna

- Si totes les instàncies implementen un mètode ***amb la mateixa signatura***, (implementat *no necessàriament a la mateixa classe*) un programa pot fer servir aquest mètode en diferentes instàncies de les diferents subclasses

```
def festa(animals):
    """Suposant que animals és una llista d'Animal, fer-los interactuar
       cada un amb tots els altres exactament un cop."""
    for i in range(len(animals)):
        for j in range(i + 1, len(animals)):
            animals[i].interactua_amb(animals[j])
```

- Com el farem servir?

Classes: Interficie comuna

- Si totes les instàncies implementen un mètode ***amb la mateixa signatura***, (implementat *no necessàriament a la mateixa classe*) un programa pot fer servir aquest mètode en diferentes instàncies de les diferents subclasses

```
def festa(animals):
    """Suposant que animals és una llista d'Animal, fer-los interactuar
       cada un amb tots els altres exactament un cop."""
    for i in range(len(animals)):
        for j in range(i + 1, len(animals)):
            animals[i].interactua_amb(animals[j])
```

- Com el farem servir?

```
lola = Conill("Lola Bunny", 2)
scar = Lleo("Scar",10)
elly = Elefant("Elly",21)
pandy = Panda("Pandy",4)
festa([lola, scar, elly, pandy])
```

Classes: Herència vs. Composició

- L'**herència** és millor per representar relacions "***és-un***":
  - El conill és un tipus específic d'animal
  - Així doncs, **`Conill`** heretarà d'**`Animal`**

- La **composició** és millor per representar relacions "***té-un***"
  - Un *Conservatori d'Animals* cuida d'una col·lecció d'animals
  - Així doncs, un *Conservatori d'Animals* té una llista d'instàncies d'**`Animal`** com a variable d'instància

Classes: Exercici

```python
class Pare:
    def f(self):
        print("Pare.f")
    def g(self):
        self.f()

class Fill(Pare):
    def f(self):
        print("Fill.f")

un_fill = Fill()
un_fill.g()
```

- Què escriurà Python?

Objectes: Molts objectes!

- Quins objectes apareixen en aquest fragment de codi?

```python
class Xai:
    nom_especie = "Xai"
    nom_cientific = "Ovis aries"

    def __init__(self, nom):
        self.nom = nom

    def juga(self):
        self.felicitat = True

xai = Xai("Petit")
propietari = "María"
te_un_xai = True
vello = {"color": "blanc", "esponjos": 10}
dia = 1
```

Objectes: Molts objectes!

- Quins objectes apareixen en aquest fragment de codi?

```python
class Xai:
    nom_especie = "Xai"
    nom_cientific = "Ovis aries"

    def __init__(self, nom):
        self.nom = nom

    def juga(self):
        self.felicitat = True

xai = Xai("Petit")
propietari = "María"
te_un_xai = True
vello = {"color": "blanc", "esponjos": 10}
dia = 1
```

- `xai`, `propietari`, `te_un_xai`, `vello`, `dia`, etc. Ho podem comprovar mirant: **`objecte.__class__.__bases__`**

Objectes: Molts objectes!

- I és que **`objecte.__class__.__bases__`** retorna les *superclasses* de la classe de la que **`objecte`** és instància:

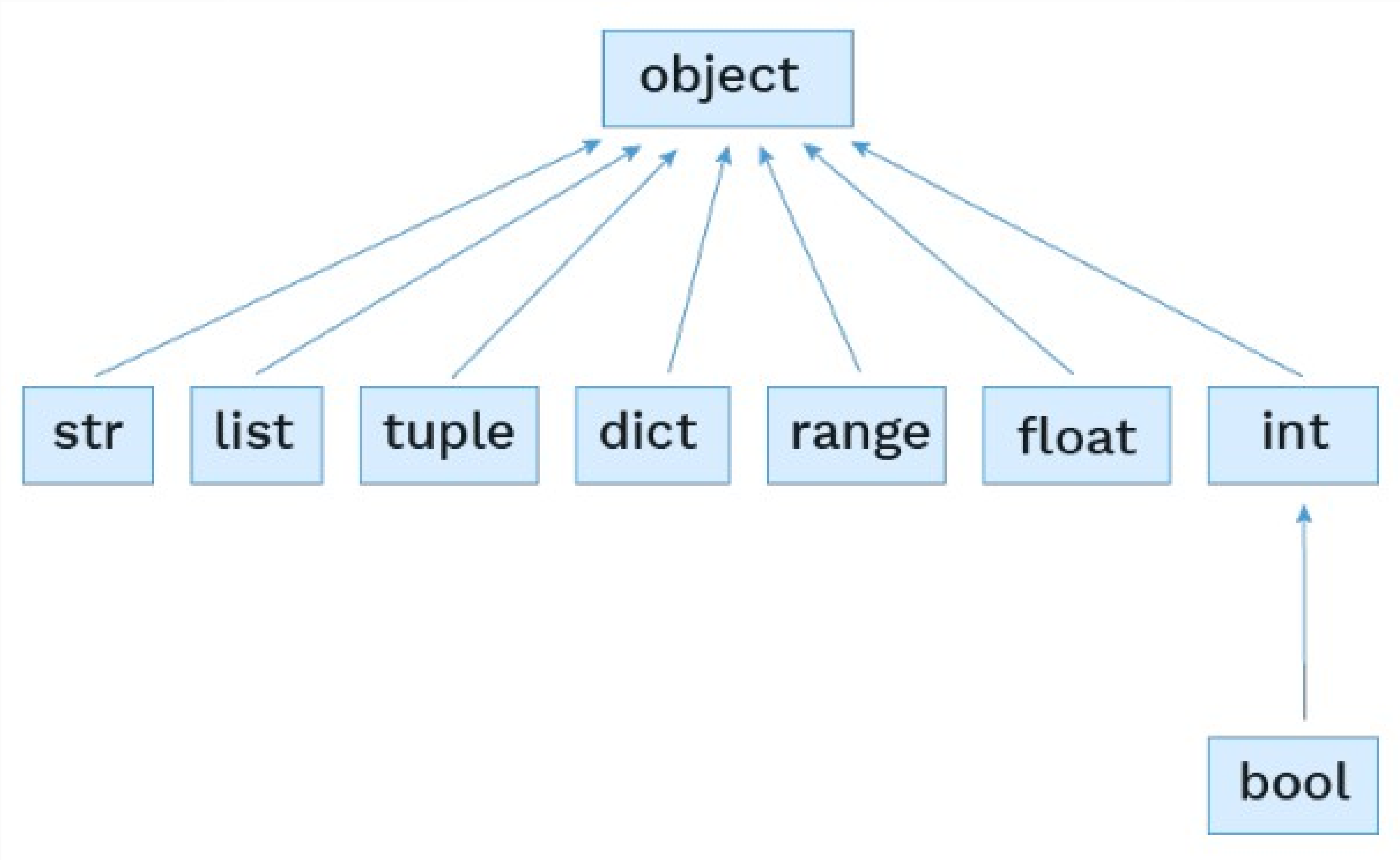
```
>>> xai.__class__.__bases__
(<class 'object'>,)
>>>
>>> propietari.__class__.__bases__
(<class 'object'>,)
>>>
>>> te_un_xai.__class__.__bases__
(<class 'int'>,)
>>>
>>> vello.__class__.__bases__
(<class 'object'>,)
>>>
>>> dia.__class__.__bases__
(<class 'object'>,)
```

Però… mireu què passa:

```
>>> xai.__class__
<class '__main__.Xai'>
>>> None.__class__
<class 'NoneType'>
>>> max.__class__
<class 'builtin_function_or_method'>
>>> max.__class__.__bases__
(<class 'object'>,)
>>> Xai.__class__
<class 'type'>
>>> Xai.__class__.__bases__
(<class 'object'>,)
```

Objectes: Tot és un objecte!

- Tots els valors que Python ens permet manipular per defecte són instàncies d'alguna classe, ***són objectes***:



Objectes: Tot és un objecte!

- Si tots els valors que Python ens permet manipular per defecte són instàncies d'alguna classe, aquesta classe hereta d'**`object`**. ***I què hereta exactament?*** Podem saber-ho amb **`dir(objecte)`**, que retorna una llista amb els atributs que té un objecte:

```
>>> dir(xai)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__',
'juga', 'nom', 'nom_cientific', 'nom_especie']
```

- Python fa servir aquests mètodes "*d'amagat*", de manera que no som conscients de la seva utilització.

Objectes: Tot és un objecte!

- Podem classificar aquests atributs heretats d'`object`:

Representació textual dels objectes:
`__repr__` , `__str__` , `__format__`

Comparacions:
`__eq__` , `__ge__` , `__gt__` , `__le__` , `__lt__` , `__ne__`

Relacionats amb classes:
`__bases__` , `__class__` , `__new__` , `__init__` , `__init_subclass__` , `__subclasshook__` , `__setattr__` , `__delattr__` , `__getattribute__`

Altres:
`__dir__` , `__hash__` , `__module__` , `__reduce__` , `__reduce_ex__`

Objectes: `__str__`

- El mètode `__str__` retorna una *string* representant textualment l'objecte de manera que sigui *llegible per humans*.

```
>>> from fractions import Fraction
>>>
>>> un_terc = 1/3
>>> una_meitat = Fraction(1,2)
>>>
>>> float.__str__(un_terc)
'0.3333333333333333'
>>> un_terc.__str__()
'0.3333333333333333'
>>>
>>> Fraction.__str__(una_meitat)
'1/2'
>>> una_meitat.__str__()
'1/2'
```

Objectes: `__str__`

- Python fa servir aquest mètode `__str__` a molts llocs: la funció `print`, el constructor `str()`, les *f-strings*, i més…

```
>>> from fractions import Fraction
>>>
>>> un_terc = 1/3
>>> una_meitat = Fraction(1,2)
>>>
>>> print(un_terc)
0.3333333333333333
>>>
>>> print(una_meitat)
1/2
>>>
>>> str(un_terc)
'0.3333333333333333'
>>>
>>> str(una_meitat)
'1/2'
>>>
>>> f"{una_meitat} > {un_terc}"
'1/2 > 0.3333333333333333'
```

Objectes: `__str__`

- Es pot sobreescriure `__str__` dins les classes fetes pel programador, per donar les instàncies d'aquesta classe una representació textual adeqüada.

```
class Xai:
    nom_especie = "Xai"
    nom_cientific = "Ovis aries"

    def __init__(self, nom):
        self.nom = nom

    def __str__(self):
        return "El " + self.nom + " fa beeee"

xai = Xai("Petit")
print(xai)                    # El Petit fa beeee
str(xai)                      # 'El Petit fa beeee'
```

Objectes: `__repr__`

- El mètode **`__repr__`** retorna una *string* que s'avaluaria a un objecte amb els mateixos valors

```
>>> from fractions import Fraction
>>>
>>> una_meitat = Fraction(1,2)
>>> una_meitat_string = Fraction.__repr__(una_meitat)
>>> una_meitat_string
'Fraction(1, 2)'
>>> una_altra_meitat = eval(Fraction.__repr__(una_meitat))  # !!!
>>> una_meitat
Fraction(1, 2)
>>> una_altra_meitat
Fraction(1, 2)
>>> una_meitat == una_altra_meitat
True
>>> una_meitat is una_altra_meitat
False
```

Objectes: `__repr__`

- Python fa servir el mètode `__repr__` a diversos llocs: Quan es crida i quan es mostra un objecte en una sessió interactiva (REPL).

```
>>> from fractions import Fraction
>>>
>>> una_meitat = Fraction(1,2)
>>> un_terc = 1/3
>>>
>>> un_terc
0.3333333333333333
>>> una_meitat
Fraction(1, 2)
>>> repr(un_terc)
'0.3333333333333333'
>>> repr(una_meitat)
'Fraction(1, 2)'
```

Objectes: `__repr__`

- Es pot sobreescriure **`__repr__`** dins les classes fetes pel programador, per donar les instàncies d'aquesta classe una representació apropiada en Python.

```
class Xai:
    nom_especie = "Xai"
    nom_cientific = "Ovis aries"

    def __init__(self, nom):
        self.nom = nom

    def __str__(self):
        return "El " + self.nom + " fa beeee"

    def __repr__(self):
        return f"Xai({repr(self.nom)})"

>>> xai = Xai("Petit")
```

Sense sobreescriure **`__repr__`**

```
>>> repr(xai)
'<__main__.Xai object at 0x7f6cf02ce3d0>'
>>> xai
<__main__.Xai object at 0x7f6cf02ce3d0>
```

Sobreescrivint **`__repr__`**

```
>>> repr(xai)
"Xai('Petit')"
>>> xai
Xai('Petit')
```

Objectes: Accés als atributs

- Usualment accedim als atributs d'un objecte (tant a l'estat com al comportament) mitjançant l'anomenada *notació del punt*:

```
class Conill:
    nom_especie = "Conill Europeu"
    nom_cientific = "Oryctolagus cuniculus"

    def __init__(self, nom):
        self.nom = nom

conillet = Conill("Bugs")

conillet.nom
conillet.nom_especie
conillet.nom_cientific
conillet.li_pengen_les_orelles  # ✗ Error!!
conillet.lliga_orelles()        # ✗ Error!!
```

Si mirem d'accedir atributs que *no existeixen*, Python genera una excepció

Objectes: Accés als atributs amb `getattr()`

- **`getattr(objecte, nom [, default])`** busca l'atribut a l'**`objecte`** pel **`nom`**. Si no està definit, retorna **`default`** (si s'ha proporcionat), o genera **`AttributeError`**.

```
>>> conillet = Conill("Bugs")

>>> getattr(conillet,"li_pengen_les_orelles")            # ✗ Error!!
>>> getattr(conillet,"li_pengen_les_orelles", False)
False
>>> getattr(conillet,"lliga_orelles")                    # ✗ Error!!
>>> getattr(conillet,"lliga_orelles", lambda: print("orelles lligades!"))
<function <lambda> at 0x7f6d1034fc10>
```

Objectes: `__getattribute__`

- Ja sigui amb la notació del punt **`objecte.nom`**, o amb **`getattr(objecte,nom)`**, Python invoca **`__getattribute__`** en l'objecte.

```
class Llum:

    def __init__(self, brillantor):
        self.brillantor = brillantor

    def __getattribute__(self, nom):
        print('__getattribute__ ', nom)
        return super().__getattribute__(nom)
```

```
>>> llum = Llum(750)
>>> llum.brillantor
__getattribute__  brillantor
750
>>> getattr(llum,"brillantor")
__getattribute__  brillantor
750
>>> Llum.__getattribute__(llum,"brillantor")
__getattribute__  brillantor
750
```

Objectes: Comprovar l'existència d'atributs

```
class Conill:
    nom_especie = "Conill Europeu"
    nom_cientific = "Oryctolagus cuniculus"
    def __init__(self, nom):
        self.nom = nom
```

Què passarà?

```
>>> conillet = Conill("Bugs")
>>>
>>> if conillet.li_pengen_les_orelles:
...     print("Sí que em pengen, sí")
... else:
...     print("No sóc orellut!")
...
```

Objectes: Comprovar l'existència d'atributs

```
class Conill:
    nom_especie = "Conill Europeu"
    nom_cientific = "Oryctolagus cuniculus"
    def __init__(self, nom):
        self.nom = nom
```

Què passarà?

```
>>> conillet = Conill("Bugs")
>>>
>>> if conillet.li_pengen_les_orelles:
...     print("Sí que em pengen, sí")
... else:
...     print("No sóc orellut!")
...
Traceback (most recent call last):                    # ✗ Error!!
  File "<stdin>", line 1, in <module>
AttributeError: 'Conill' object has no attribute 'li_pengen_les_orelles'
```

Objectes: Comprovar l'existència d'atributs

- **`hasattr(objecte,nom)`** comprova l'atribut pel **`nom`** a l'**`objecte`**, i retorna si pot trobar aquest atribut.

```
class Conill:
    nom_especie = "Conill Europeu"
    nom_cientific = "Oryctolagus cuniculus"
    def __init__(self, nom):
        self.nom = nom

conillet = Conill("Bugs")
if hasattr(conillet,"li_pengen_les_orelles"):
    print("Sí que em pengen, sí")
else:
    print("No sóc orellut!")
```

- Python implementa aquesta funció invocant **`getattr()`** i comprovant si s'ha generat una excepció, per tant aquesta funció acaba invocant **`__getattribute__`** .

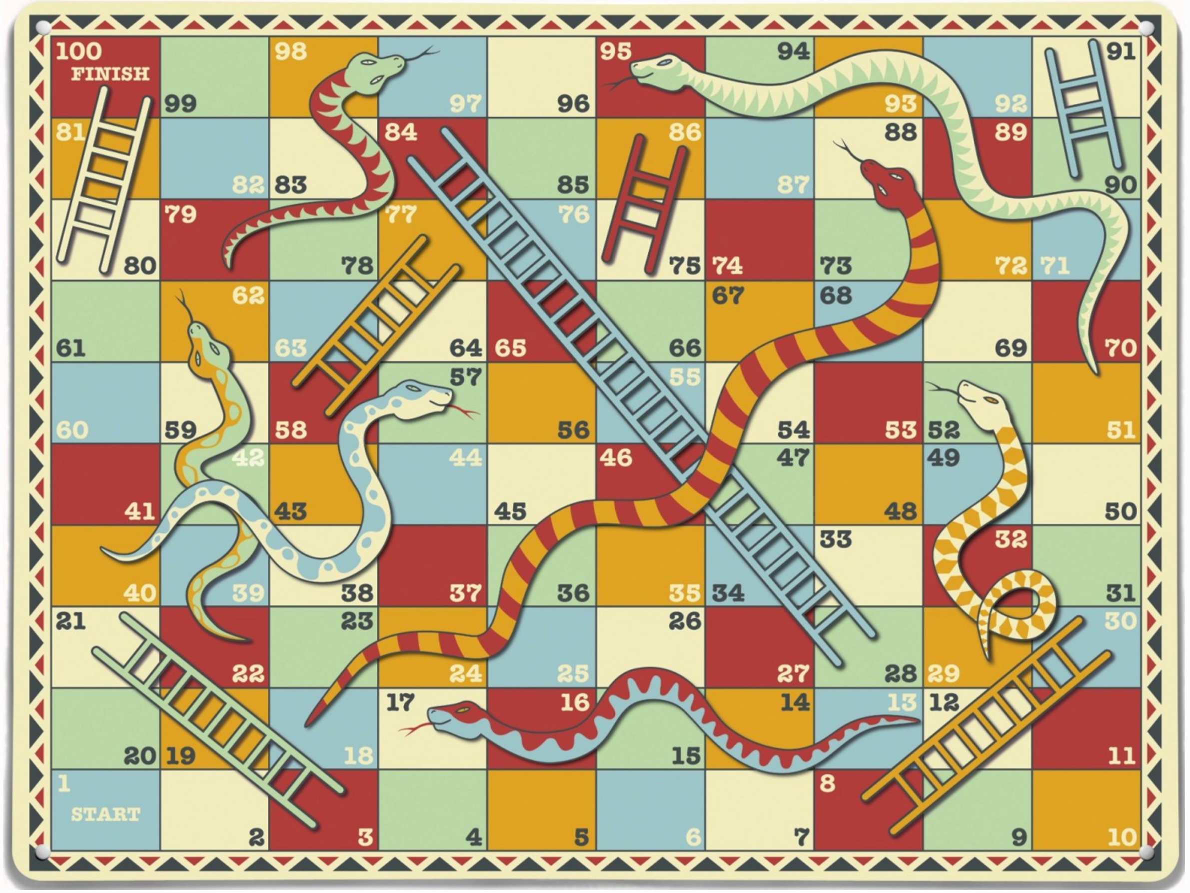
Objectes: N'hi ha molt més! Mètodes especials

- Hi ha més mètodes especials pels objectes:

Mètode	**Implementa**
`__setattr__(obj, "nom", v)`	`obj.nom = v`
`__delattr__(obj, "nom")`	`del obj.nom`
`__eq__(obj, x)`	`obj == x`
`__ne__(obj, x)`	`obj != x`
`__ge__(obj, x)`	`obj >= x`
`__gt__(obj, x)`	`obj > x`
`__le__(obj, x)`	`obj <= x`
`__lt__(obj, x)`	`obj < x`

- I això no és tot. N'hi ha molts mètodes especials més que es poden definir per personalitzar com Python opera amb els objectes.

Exemple POO: *Snakes and Ladders*



Disclaimer! Aquest vol ser un exemple de *programació* orientada a objectes, posant èmfasi en la *implementació* del *disseny* d'una possible solució. Aquest disseny serà discutit i, fins on es pugui, justificat, però vindrà essencialment donat. No podem, en aquest curs, aprofundir en qüestions de disseny de programari.

Exemple POO: *Snakes and Ladders*

Each player starts with a token on the starting square (usually the "1" grid square in the bottom left corner, ...). Players take turns rolling a single die to move their token by the number of squares indicated by the die rolled. Tokens follow a fixed route marked on the gameboard which usually follows a boustrophedon (ox-plow) track from the bottom to the top of the playing area, passing once through every square. If, on completion of a move, a player's token lands on the lower-numbered end of a "ladder", the player moves the token up to the ladder's higher-numbered square. If the player lands on the higher-numbered square of a "snake" (...), the token must be moved down to the snake's lower-numbered square.

`https://en.wikipedia.org/wiki/Snakes_and_ladders`

Exemple POO: *Snakes and Ladders*

Altres Consideracions:

- Si un jugador, en tirar el dau, *es passa* del final, torna enrera (*rebota*).
- Si un jugador, en tirar el dau, va a una casella ocupada (on hi ha un altre jugador), torna a la primera casella. Només un jugador per casella.
- Només la primera casella pot contenir més d'un jugador simultàniament.
- No es repeteix tirada encara que es tregui un 6. Cada jugador tira un cop per torn.
- Guanya el primer que arriba al final, en una tirada *exacta* (on es caigui exactament a la darrera casella).

Exemple POO: *Snakes and Ladders*

- El primer que cal fer és esbrinar quins seran els objectes que apareixen en aquest sistema, quin ha de ser el seu comportament envers els altres objectes i quina relació ha d'haver entre els diferents objectes…
- Objectes?

Exemple POO: *Snakes and Ladders*

- El primer que cal fer és esbrinar quins seran els objectes que apareixen en aquest sistema, quin ha de ser el seu comportament envers els altres objectes i quina relació ha d'haver entre els diferents objectes…
- Objectes?
  - **El joc en sí**
  - **Tauler** (*tot i que en realitat és un joc d'una dimensiò*)
  - **Caselles**
  - **Dau**
  - **Jugadors**
  - **Escales i Serps**

Exemple POO: *Snakes and Ladders*

- El primer que cal fer és esbrinar quins seran els objectes que apareixen en aquest sistema, quin ha de ser el seu comportament envers els altres objectes i quina relació ha d'haver entre els diferents objectes...
- Objectes?
  - **El joc en sí →** El joc i el tauler són en realitat la mateixa entitat
  - **Tauler →** El joc i el tauler són en realitat la mateixa entitat
  - **Caselles →** Les caselles han de considerar-se ocupades o no, com a mínim, i formaran part del Tauler
  - **Dau →** El dau és l'objecte que farem servir per saber on desplaçar un jugador
  - **Jugadors →** Els jugadors han de saber on són, i en quin joc participen
  - **Escales i Serps →** En realitat són caselles "especials"

Exemple POO: *Snakes and Ladders*

- En funció dels objectes detectats, i dels seus possibles comportaments, decidirem un conjunt inicial de classes (revisable)
- Classes?
  - **Joc ≡ Tauler ≡ SnakesAndLadders**
  - **Casella →** Casella Serp, Casella Escala… i Primera Casella
  - **Dau**
  - **Jugador**
  - **Escales i Serps →** No són més que caselles amb propietats addicionals

Exemple POO: *Snakes and Ladders*

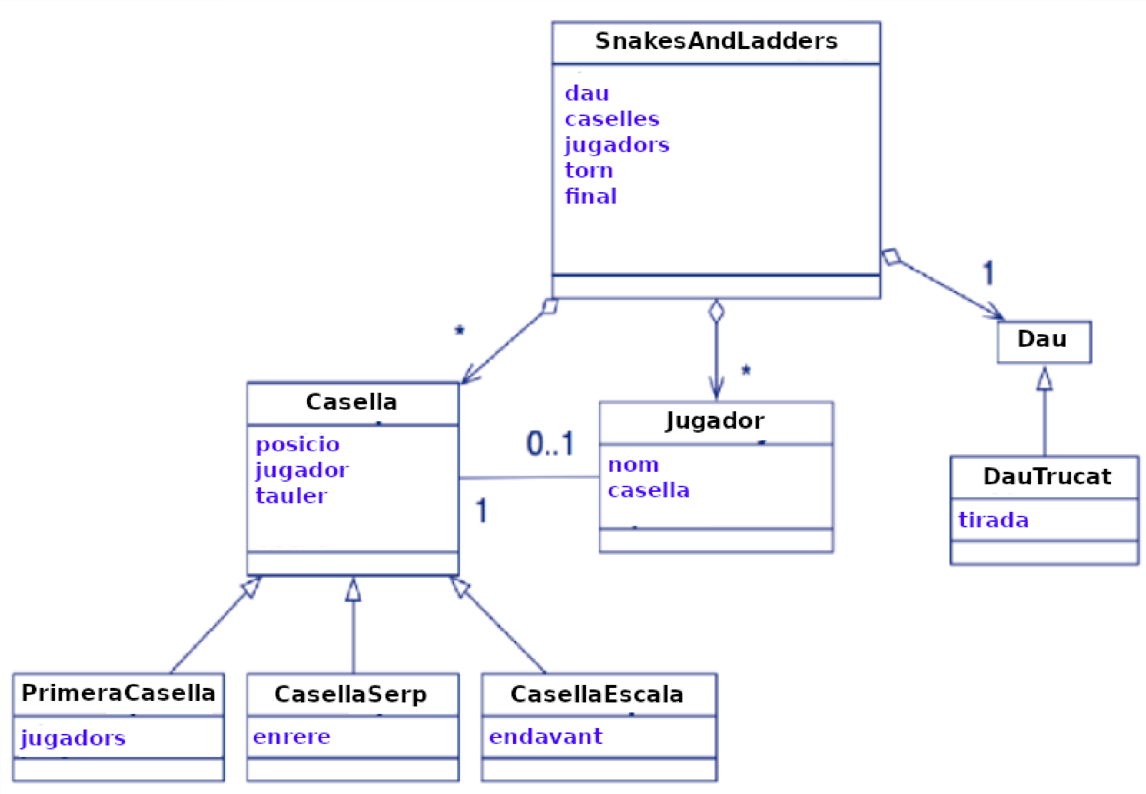
- En funció dels objectes detectats, i dels seus possibles comportaments, decidirem un conjunt inicial de classes (revisable)
- Classes?
  - **Joc ≡ Tauler ≡ SnakesAndLadders**
  - **Casella**
    - **CasellaSerp**
    - **CasellaEscala**
    - **PrimeraCasella**
  - **Dau**
    - **DauTrucat** (*anirà bé a l'hora de provar el codi*)
  - **Jugador**

Exemple POO: *Snakes and Ladders*

- En funció dels objectes detectats, i dels seus possibles comportaments, decidirem un conjunt inicial de classes (revisable)
- Relacions entre Classes?
  - El Tauler té caselles, caselles serp, caselles escala i una primera casella. També té un dau, uns jugadors i ha de saber de qui és el torn. En definitiva, si l'hem fet equivalent al joc ha de disposar de tota la informació necessària.
  - Cada casella ha de saber a quin Tauler està, quina és la seva posició dins el Tauler i pot tenir un o cap jugador (excepte la primera casella que pot tenir més d'un jugador).
  - Les caselles serp i escala han de saber quantes caselles mouen al Jugador enrere o endavant respectivament.
  - Cada Jugador té un nom i ha de saber a quina Casella està col·locat
  - Hem de ser capaços de "*programar*" el resultat del dau trucat.

Exemple POO: *Snakes and Ladders*

- Les relacions entre classes s'acostumen a representar fent servir un llenguatge formal anomenat UML (*Unified Modeling Language**). Nosaltres no hi entrarem, però aquí teniu com a exemple el ***diagrama de classes*** (*incomplet!*, falten, entre altres coses, els mètodes de les classes) del joc:



* https://en.wikipedia.org/wiki/Unified_Modeling_Language

Exemple POO: *Snakes and Ladders*

- Abans de començar a estudiar una possible implementació, *imaginem* un exemple de l'ús d'aquest programa que voldríem fer...

Programa **joc_de_prova_1.py**:

```
from snakes_and_ladders import *

def tauler_inicial():
    s = SnakesAndLadders(12)

    s.afegeix_escala(1,5)
    s.afegeix_escala(6,8)
    s.afegeix_serp(10,4)

    s.afegeix_jugador(Jugador('Jack'))
    s.afegeix_jugador(Jugador('Jill'))

    return s

joc = tauler_inicial()
dau_trucat = DauTrucat()
joc.utilitza_dau(dau_trucat)
```

Exemple POO: *Snakes and Ladders*

- A partir de la inicialització al programa **`joc_de_prova_1.py`** podem interactuar amb el joc:

```
>>> from joc_de_prova_1 import *
>>> print(joc)
[0  Jack Jill][1 ]4+>[2 ][3 ][4 ][5 ][6 ]2+>[7 ][8 ][9 ]<-6[10 ][11 ]
>>> dau_trucat.tirada(1)
>>> joc.tirada()
'Jack treu un 1 i va a la casella [5  Jack]'
>>> print(joc)
[0  Jill][1 ]4+>[2 ][3 ][4 ][5  Jack][6 ]2+>[7 ][8 ][9 ]<-6[10 ][11 ]
>>> joc.jugador_actual().nom()
'Jill'
>>> dau_trucat.tirada(5)
>>> joc.tirada()
'Jill treu un 5 i va a la casella [0  Jill]'
>>> print(joc)
[0  Jill][1 ]4+>[2 ][3 ][4 ][5  Jack][6 ]2+>[7 ][8 ][9 ]<-6[10 ][11 ]
>>> joc.jugador_actual().nom()
'Jack'
>>> joc.tirada()
'Jack treu un 5 i va a la casella [4  Jack]'
>>> print(joc)
[0  Jill][1 ]4+>[2 ][3 ][4  Jack][5 ][6 ]2+>[7 ][8 ][9 ]<-6[10 ][11 ]
>>> (etc...)
```

Exemple POO: *Snakes and Ladders*

- Aquest codi ja ens ha mostrat uns quants comportaments/mètodes que seria desitjable implementar. Comencem per la classe **SnakesAndLadders**:

```
class SnakesAndLadders:
    def __init__(self, nCaselles):
        """
        Pre: nCaselles > 0, nombre de caselles
        Cal inicialitzar les variables d'instància
        """
    def afegeix_casella(self,c,pos):
        """
        Pre: 0 <= pos < mida tauler,
             c és instància de Casella
        """
    def afegeix_escala(self,inici,fi):
        """
        Pre: inici < fi and 1 <= inici and fi < mida tauler
        """
    def afegeix_serp(self,inici,fi):
        """
        Pre: fi < inici and  1 <= fi and inici < mida tauler
        """
```

```
    def afegeix_jugador(self,jugador):
        """
        Pre: jugador és instància de Jugador
        """
    def utilitza_dau(self, dau):
        """
        Pre: dau és instància de Dau
        """
    def tirada(self):

    def jugador_actual(self):

    def __str__(self): # ja que fem print(joc)!
```

Exemple POO: *Snakes and Ladders*

- Aquest codi ja ens ha mostrat uns quants comportaments/mètodes que seria desitjable implementar. També trobem comportaments per a **`DauTrucat`** i per a **`Jugador`**.

```
class DauTrucat:
    def __init__(self):
        """
        Cal inicialitzar les variables d'instància
        """
    def tirada(self, n):
        """
        Pre: 1 ≤ n ≤ 6, n serà el resultat de la propera tirada del dau
        """

class Jugador:
    def __init__(self,nom):
        """
        Pre: nom és una string
        Cal inicialitzar les variables d'instància
        """
    def nom(self):
        """
        getter: retorna el nom del jugador
        """
```

Exemple POO: *Snakes and Ladders*

- Podem començar per la classe **Dau** i la seva subclasse **DauTrucat**, senzilles d'implementar:

```
import random

class Dau:
    def tira_dau(self):
        return random.randint(1,6)


class DauTrucat(Dau):
    def __init__(self):
        self.__tirada = 1
    def tirada(self,num):
        assert 1 <= num <= 6
        self.__tirada = num
    def tira_dau(self):
        return self.__tirada
```

- Com **Dau** no necessita estat, tampoc li cal l'**`__init__`** . El mètode important és `tira_dau`, ja que aquest serà el mètode que farem servir per obtenir la tirada d'un dau. No importa si **d** és instància de **Dau** o de **DauTrucat**, **`d.tira_dau()`** és la manera d'obtenir la tirada d'un dau.

Exemple POO: *Snakes and Ladders*

- Discutim ara una proposta de constructor per a **`SnakesAndLadders`** (compareu amb el diagrama que hem vist abans per a aquesta classe):

```
from casella import *
from jugador import *
from dau import *

class SnakesAndLadders:

    def __init__(self, nombre_de_caselles):
        self.__jugadors = []         # inicialment no hi ha jugadors
        self.__torn = 0              # és el torn del primer (nº 0) jugador
        self.__final = False         # la partida no s'ha acabat
        self.__dau = Dau()           # el Dau per defecte és no trucat
        # reservem espai per a nombre_de_caselles caselles
        self.__caselles = [None]*nombre_de_caselles
        # les inicialitzem amb instàncies de Casella, excepte la primera.
        self.afegeix_casella(PrimeraCasella(),0)
        for i in range(1,nombre_de_caselles):
            self.afegeix_casella(Casella(),i)
```

Exemple POO: *Snakes and Ladders*

- Podem fàcilment imaginar un mètode que preservi la configuració del tauler, però re-inicialitzi tota la resta, de manera que es pugui tornar a jugar:

```
# imports diversos...

class SnakesAndLadders:

    def __init__(self, nombre_de_caselles):
        # ...

    def reset(self):
        self.__dau = Dau()
        self.__torn = 0
        self.__final = False
        for j in self.__jugadors:
            j.mou_a(self.primera_casella())
```

- Fixem-nos que ara caldrà que la classe `Jugador` implementi el mètode públic `mou_a(self,casella)`, i que `SnakesAndLadders` afegeixi el mètode públic `primera_casella(self)`.

Exemple POO: *Snakes and Ladders*

- Examinem ara una possible implementació dels mètodes de **SnakesAndLadders** que afegeixen caselles o jugadors al joc. En el cas de les caselles fixem-nos en el mètode d'`afegeix_casella`, que després és re-utilitzat per `afegeix_escala` i per `afegeix_serp`. Fixem-nos també en el detall que quan s'afegeix una casella o un jugador, aquests han de ser notificats...

```
def afegeix_casella(self,casella,i):
    assert 0 <= i < len(self.__caselles)
    self.__caselles[i] = casella
    casella.posa_a_posicio(i,self)  # li diem a la casella en quina posició l'hem posat, i en quin Joc

def afegeix_escala(self,inici,fi):
    assert(inici < fi and 1 <= inici and fi < len(self.__caselles))
    self.afegeix_casella(CasellaEscala(fi-inici),inici) # CasellaEscala també és-un(a) Casella!

def afegeix_serp(self,inici,fi):
    assert(fi < inici and  1 <= fi and inici < len(self.__caselles))
    self.afegeix_casella(CasellaSerp(inici-fi),inici)   # CasellaSerp també és-un(a) Casella!

def afegeix_jugador(self,jugador):
    self.jugadors().append(jugador)
    jugador.mou_a(self.primera_casella())  # Els jugadors s'incorporen al joc a la primera casella.
    return self
```

Exemple POO: *Snakes and Ladders*

- La classe **`SnakesAndLadders`** també tindrà mètodes senzills que proporcionen informació que pot ser necessària per a altres usuaris de la classe. Com que són bàsicament trivials no farem cap comentari:

```python
def torn(self):
    return self.__torn

def jugadors(self):
    return self.__jugadors

def final(self):
    return self.__final

def utilitza_dau(self, dau):
    self.__dau = dau
```

```python
def casella(self,posicio):
    assert (0 <= posicio < len(self.__caselles))
    return self.__caselles[posicio]

def jugador_actual(self):
    return self.jugadors()[self.torn()]

def primera_casella(self):
    return self.__caselles[0]

def darrera_posicio(self):
    return len(self.__caselles) - 1
```

- No tenim *getter* per a `self.__caselles`. Això és perquè no volem fer pública aquesta informació. Tampoc hi ha *getter* pel dau, en canvi sí que tenim un *setter*.
- Pareu atenció en que, sempre que s'ha pogut, s'ha prioritzat l'ús de *getters* sobre l'accés directe a les variables d'instància. Això ho fem, insistim un cop més, per desacoblar l'accés a la informació de la implementació particular que estem fent servir.

Exemple POO: *Snakes and Ladders*

- Els mètodes de **`SnakesAndLadders`** on està el nucli del joc els podem implementar còmodament amb el que hem vist fins ara, més el comportament de **`Jugador`** i **`Casella`**.

```python
def tirada(self):
    if self.final():
        return "La partida s'ha acabat!"
    else:
        resultat = self.jugador_actual().mou_tirant(self.__dau) + self.__comprova_resultat()
        self.__actualitza_torn()
        return resultat

def partida_completa(self):
    print(self)
    while not self.final():
        print(self.tirada())   # Estem suposant que tirada retorna una string
        print(self)
    print()
```

- Fixem-nos que deleguem tota la feina de decidir la tirada i moure's dins el tauler al **`Jugador`**, tot i que aquest necessita el dau per poder fer-ho. Per això li passem el dau com a argument. La resta d'informació (continguda al tauler) el **`Jugador`** ja la té accessible.

Exemple POO: *Snakes and Ladders*

- Finalment necessitarem uns quants mètodes auxiliars que hem fet servir en els altres mètodes de la classe:

```
def __str__(self):                # Aquest mètode el fem servir quan fem print(...) del joc
    s = ''
    for c in self.__caselles:
        s = s + str(c)            # "demanem" a les caselles que retornin la seva
    return s                      # representació textual.

def __comprova_resultat(self):    # Aquest mètode el fa servir tirada()
    if self.jugador_actual().posicio() == self.darrera_posicio():
        self.__final = True
        return f"-- {str(self.jugador_actual())} ha guanyat!"
    else:
        return ''

def __actualitza_torn(self):      # Aquest mètode el fa servir tirada()
    self.__torn = (self.torn() + 1) % len(self.jugadors())
```

- Veiem que ens cal preguntar al jugador actual quina és la seva posició. Aquesta informació no té per què saber-la el joc. És pròpia del jugador.

Exemple POO: *Snakes and Ladders*

- La classe **`Jugador`** és una classe senzilla, ja que el fet de moure's pel tauler acaba delegant-lo a les caselles. L'únic estat que necessita és el nom i saber en quina casella està:

```python
class Jugador:

    def __init__(self, nom):
        self.__casella = None
        self.__nom = nom

    def deixa_casella(self):
        if self.__casella is not None:
            self.__casella.treu_jugador(self)

    def mou_a(self, casella):
        self.deixa_casella()
        self.__casella = casella.posa_jugador(self)

    def mou_tirant(self, dau):
        tirada = dau.tira_dau()
        destinacio = self.__casella.endavant(tirada)  # Atenció, aquí demanem nou comportament
        self.mou_a(destinacio)                        # sobre les caselles: mètode endavant(n)
        return f"{self.__nom} treu un {tirada} i va a la casella {str(self.__casella)}"

    def posicio(self):
        return self.__casella.posicio()

    def __str__(self):
        return self.__nom
```

Exemple POO: *Snakes and Ladders*

<u>**Incís**</u>:

- Una classe dins un programa orientat a objectes ben dissenyat (i deixeu-nos insistir en que tenim molt present que no us estem ensenyant a dissenyar; no hi cap en aquest curs) usualment està composada de <u>*molts mètodes*</u>, <u>*molt petits*</u>, ja que el comportament s'acostuma a distribuir tant com es pot, tot mantenint els mètodes al mateix nivell d'abstracció. Un exemple seria **Pharo**, una implementació del llenguatge de programació **Smalltalk** (`pharo.org`). El sistema complet està composat de 10.652 classes (en la versió 9.0), i la mida mitjana d'un mètode són 7 línies de codi!
- En el nostre exemple, fixeu-vos que el comportament associat a "*tira el dau i mou x posicions*" d'un **`Jugador`** qualsevol, el mètode `mou_tirant`, delega saber a quina casella va a parar a la casella on està posicionat (mètode `endavant` de **`Casella`**), i després es mou on la casella li "*diu*" (variable `destinacio`). Això vol dir que cal "*dir-li*" a la casella que tregui al jugador, i "*demanar-li*" a la casella destinació que posi allà al jugador. Veiem que el comportament requerit de l'objecte es desglossa en la invocació de comportaments petits, tant propis com de les caselles.

Exemple POO: *Snakes and Ladders*

- Finalment, anem a veure la implementació de la classe **`Casella`** i les seves subclasses **`PrimeraCasella`**, **`CasellaEscala`** i **`CasellaSerp`**. Com ja hem comentat anteriorment, la classe **`Casella`** mirarà de contenir *tot* el comportament *comú* a qualsevol casella:

```python
class Casella:

    def __init__(self):
        self._posicio = None
        self._jugador = None
        self._tauler  = None

    def posicio(self):
        return self._posicio

    def posa_a_posicio(self, posicio, tauler):
        self._posicio = posicio
        self._tauler = tauler

    def jugador(self):
        return self._jugador

    def tauler(self):
        return self._tauler

    def casella_seguent(self):
        return self.tauler().casella(self.posicio()+1)

    def casella_anterior(self):
        return self.tauler().casella(self.posicio()-1)

    def es_primera_casella(self):
        return self.posicio() == 0

    def es_darrera_casella(self):
        fi = self.tauler().darrera_posicio()
        return self.posicio() == fi

    def esta_ocupada(self):
        return self.jugador() is not None
```

Exemple POO: *Snakes and Ladders*

- L'estat que cada `Casella` necessita és: el tauler al que pertany, si té o no jugador (i si el té li cal saber quin jugador és aquest) i la posició que ocupa dins el tauler. Els hem declarat *semi-privats* (només un guió baix) ja que volem que siguin heretats.

- Tenim operacions a destacar:
  - Donat l'objecte instància de `Casella`, podem obtenir la casella següent i la casella anterior, però ens cal saber a quin tauler pertany la casella, ja que no podem fer una altra cosa que "*preguntar-li*" a ell. Si la posició que passem com a argument és incorrecte obtindrem un error.
  - També ens cal el tauler per a que la casella "*sàpiga*" si és o no la darrera casella, i si no ens cal per saber si la casella és la primera casella és perquè tant la classe `SnakesAndLadders` com la classe `Casella` comparteixen l'assumpció que la primera casella és la casella número 0.

Exemple POO: *Snakes and Ladders*

- Les operacions que tenen a veure amb moure el jugador es desplacen sobre el tauler fins a trobar la casella corresponent al desplaçament indicat pel dau. Fixeu-vos que la definició del mètode `destinacio` està relacionada amb aquests mètodes `endavant` i `enrere`.

```
def enrere(self,caselles):
    if caselles == 0:
        return self.destinacio()
    else:
        if self.es_primera_casella():
            return self.casella_seguent().endavant(caselles-1)  # "rebota" al principi (??)
        else:
            return self.casella_anterior().enrere(caselles-1)

def endavant(self,caselles):
    if caselles == 0:
        return self.destinacio()
    else:
        if self.es_darrera_casella():
            return self.casella_anterior().enrere(caselles-1)  # "rebota" al final!
        else:
            return self.casella_seguent().endavant(caselles-1)

def destinacio(self):                                           # cal entendre bé quin paper juga!
    return self
```

Exemple POO: *Snakes and Ladders*

- Les operacions que tenen a veure amb col·locar o treure un jugador de la casella són les que comproven que no hi hagi caselles amb més d'un jugador. Finalment, també calen un parell de funcions auxiliars.

```python
def posa_jugador(self, jugador):
    if self.esta_ocupada():
        return self.tauler().primera_casella().posa_jugador(jugador) # NO pot haver més d'un jugador
    else:                                                            # a la mateixa casella
        self._jugador = jugador
        return self

def treu_jugador(self,jugador):
    assert self.jugador() == jugador          # el jugador que treiem ha de ser el que hi ha (!)
    self._jugador = None
    return self

def _contingut(self):
    if self.esta_ocupada():
        return f" {str(self.jugador())}"
    else:
        return ''

def __str__(self):
    return f"[{self.posicio()} {self._contingut()}]"   # escriurem el nom del jugador, si n'hi ha
```

Exemple POO: *Snakes and Ladders*

- Les classes **`CasellaEscala`** i **`CasellaSerp`** són subclasses de **`Casella`**. Només cal sobreescriure `__str__` i `destinacio`. Qui s'encarrega de fer avançar/retrocedir els jugadors? Precisament els mètodes sobreescrits `destinacio`. El fet de cridar `super().__init__` dins d'`__init__` fa que les variables d'instància heretades s'inicialitzin. És per això que les variables d'instància de **`Casella`** són *semi-privades*.

```python
class CasellaEscala(Casella):

    def __init__(self,endavant=0):
        super().__init__()
        self.__endavant = endavant

    def destinacio(self):
        return self.endavant(self.__endavant)

    def __str__(self):
        s = super().__str__()
        return s + f"{self.__endavant}+>"
```

```python
class CasellaSerp(Casella):

    def __init__(self,enrere=0):
        super().__init__()
        self.__enrere = enrere

    def destinacio(self):
        return self.enrere(self.__enrere)

    def __str__(self):
        s = super().__str__()
        return f"<-{self.__enrere}" + s
```

Exemple POO: *Snakes and Ladders*

- Finalment, la classe **`PrimeraCasella`** sobreescriu més mètodes, ja que és l'única casella que pot contenir més d'un jugador. Per cada joc ha d'haver només una primera casella. Les classes que només es poden instanciar un cop s'anomenen *singletons*. Aquesta no ho és ja que no fem res per impedir que hi hagi més instàncies.
- Altre cop, cridem `super().__init__` dins d'`__init__` ja que volem inicialitzar les variables d'instància heretades.

```python
class PrimeraCasella(Casella):

    def __init__(self):
        super().__init__()
        self.__jugadors = []

    def esta_ocupada(self):
        return len(self.__jugadors) > 0

    def posa_jugador(self, jugador):
        self.__jugadors.append(jugador)
        return self

    def treu_jugador(self,jugador):
        assert jugador in self.__jugadors
        self.__jugadors.remove(jugador)

    def _contingut(self):
        s = ''
        for j in self.__jugadors:
            s = s + ' ' + str(j)
        return s
```

Per acabar: *Dynamic Dispatch*

- Segons la *wikipedia*: (...) *dynamic dispatch is the process of selecting which implementation of a polymorphic operation (method or function) to call <u>at run time</u>. It is commonly employed in, <u>and considered a prime characteristic of</u>, object-oriented programming (OOP) languages and systems.*

- En un programa orientat a objectes no podem decidir a quin mètode concret correspón una invocació d'aquell mètode fins que no s'executa el programa.

- El fet que puguem anomenar de la mateixa manera mètodes de diferents classes, o que puguem sobreescriure mètodes quan tenim herència, és un tipus de ***polimorfisme***. La seva utilitat és força més aparent en llenguatges de programació amb sistemes de tipus (no és el cas de Python).

```python
class Gat:
    def soroll(self):
        print("Miau")

class Gos:
    def soroll(self):
        print("Guau")

def parla(animal):
    # Fa un dynamic dispatch del mètode soroll
    # animal pot ser una instància de, o bé un
    # gat o bé un gos
    animal.soroll()

cat = Gat()
parla(cat)
dog = Gos()
parla(dog)
```

Així doncs, què és la POO?

- *Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data and code: data in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods).*

  A feature of objects is that an object's own procedures can access and often modify the data fields of itself (objects have a notion of this or self). In OOP, computer programs are designed by making them out of objects that interact with one another. OOP languages are diverse, but the most popular ones are class-based, meaning that objects are instances of classes, which also determine their types.

  `en.wikipedia.org/wiki/Object-oriented_programming`

- *I thought of objects being like biological cells and/or individual computers on a network, only able to communicate with messages (so messaging came at the very beginning – it took a while to see how to do messaging in a programming language efficiently enough to be useful).*

  Alan Kay, creador d'Smalltalk, `www.purl.org/stefan_ram/pub/doc_kay_oop_en`

Estructures Dinàmiques de Dades

Reduint dependències...

- Fins ara hem depès d'altres estructures de dades, ja proporcionades per Python, per poder implementar les nostres pròpies estructures de dades:
  - Pels arbres hem necessitat diccionaris o llistes
  - Pels *heaps* ens han calgut llistes
  - Pels grafs hem utilitzat llistes
- En aquests casos hem fet servir les estructures de dades de Python aprofitant una *fracció* de la seva funcionalitat potencial. En canvi l'eficiència del seu ús depèn de la implementació que n'hagi fet Python, que és una implementació genèrica que té en compte l'ús potencial d'aquestes estructures en situacions molt generals.
- Ens agradaria poder controlar les implementacions de les nostres estructures de dades, i poder ajustar l'eficiència a l'ús concret que els volem donar.

Nodes

- Farem servir objectes *senzills* per *guardar (referències a) els objectes* que vulguem emmagatzemar en les nostres estructures. Es corresponen al que hem estat anomenant informalment *nodes* o *vèrtexos* en arbres o grafs.
- L'important, però, és que aquests nodes també emmagatzemaran *enllaços* (*links*) a altres nodes. D'aquesta manera podrem construir estructures diverses amb nodes enllaçats entre sí.
- Així podrem tenir estructures *lineals* (piles, cues, llistes) i *no-lineals* (arbres i grafs), en funció de com definim els enllaços entre nodes.
- Usualment els definirem en *classes internes*, *privades*, a altres classes. Un exemple:

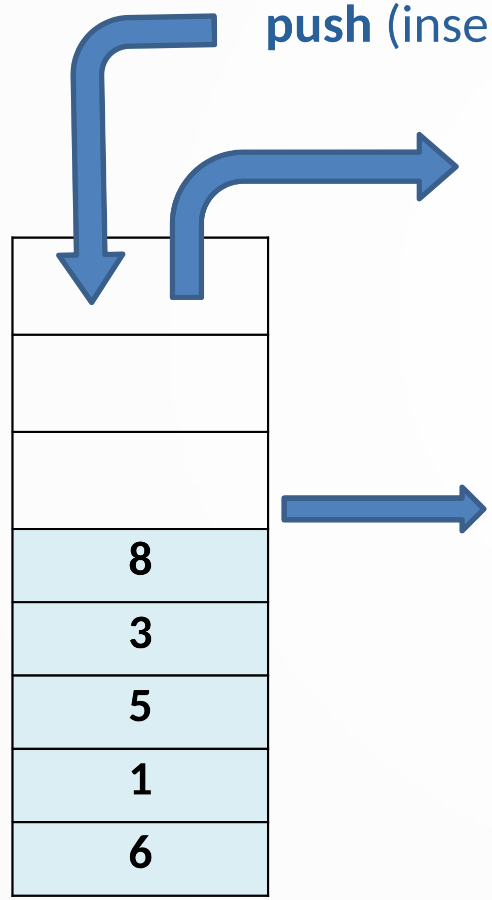
```
class _Node:
    __slots__ = '_element', '_next'        # opcional, per eficiència

    def __init__(self, element, next):     # inicialitzar els camps
        self._element = element            # ref. a l'element emmagatzemat
        self._next = next                  # referència al proper node
```

La Pila (*Stack*)

- La Pila és una estructura de dades lineal que ja hem mencionat breument relacionant-la amb l'estructura **`deque`** (*doubly ended queue*) del mòdul **`Collections`** de Python. Vam dir que fer servir *només* les operacions `pop` i `append` en un **`deque`** el feia equivalent a una Pila.
- És un contenidor on afegim i eliminem pel ***mateix extrem***.
- La Pila, com a estructura de dades, ve definida per la següent especificació, per les següents operacions:
  - constructor, que retorna una pila buida
  - `push(x)`: afegir element `x` a la pila
  - `pop()`: eliminar el darrer element afegit. Hem de suposar que la Pila no és buida
  - `top()`: consultar el darrer element afegit. Hem de suposar que la Pila no és buida
  - `buida()`: demanar si la Pila és buida
  - `mida()`: demanar la mida de la Pila

La Pila (*Stack*)

- Les Piles també es coneixen com a estructures LIFO (*Last In*, *First Out*)



push (inserir un element al cim)

pop (eliminar el darrer element inserit)

top (darrer element inserit)

buida (hi ha cap element?)

mida (nombre d'elements)

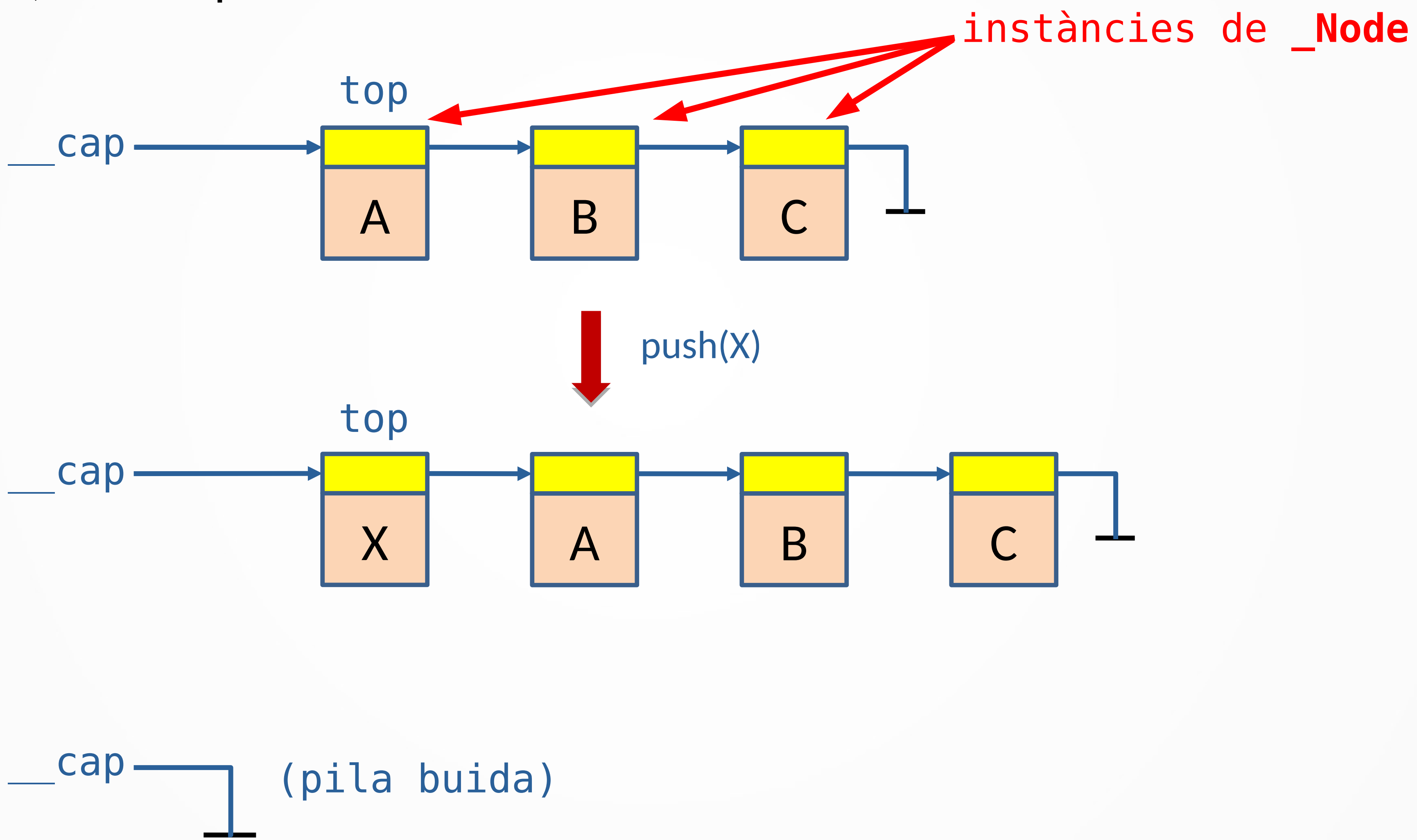
La Pila (*Stack*)

- Fent servir classes, tindríem:

```
class Pila:

    # ---------------------------------------------------
    # Classe interna per definir els elements de la pila:
    # Cada element de la pila serà una instància de _Node
    class _Node:
        __slots__ = '_element', '_next'        # opcional, per eficiència

        def __init__(self, element, next):     # inicialitzar els camps
            self._element = element            # ref. a l'element emmagatzemat
            self._next = next                  # referència al proper node
    # ---------------------------------------------------

    def __init__(self):
        self.__cap = None
        self.__mida = 0
```

La Pila (*Stack*)

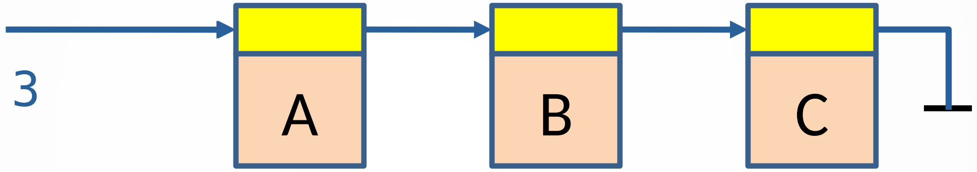
- Gràficament, volem quelcom com:



La Pila (*Stack*)

```
def push(self, e):
    self.__cap = self._Node(e, self.__cap)
    self.__mida += 1
```

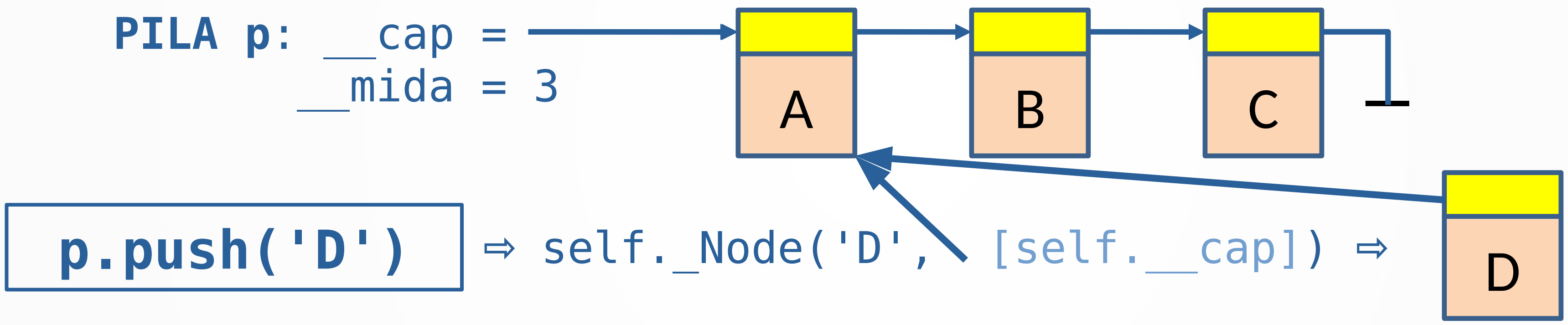
PILA p: __cap =
__mida = 3




```
p.push('D')
```

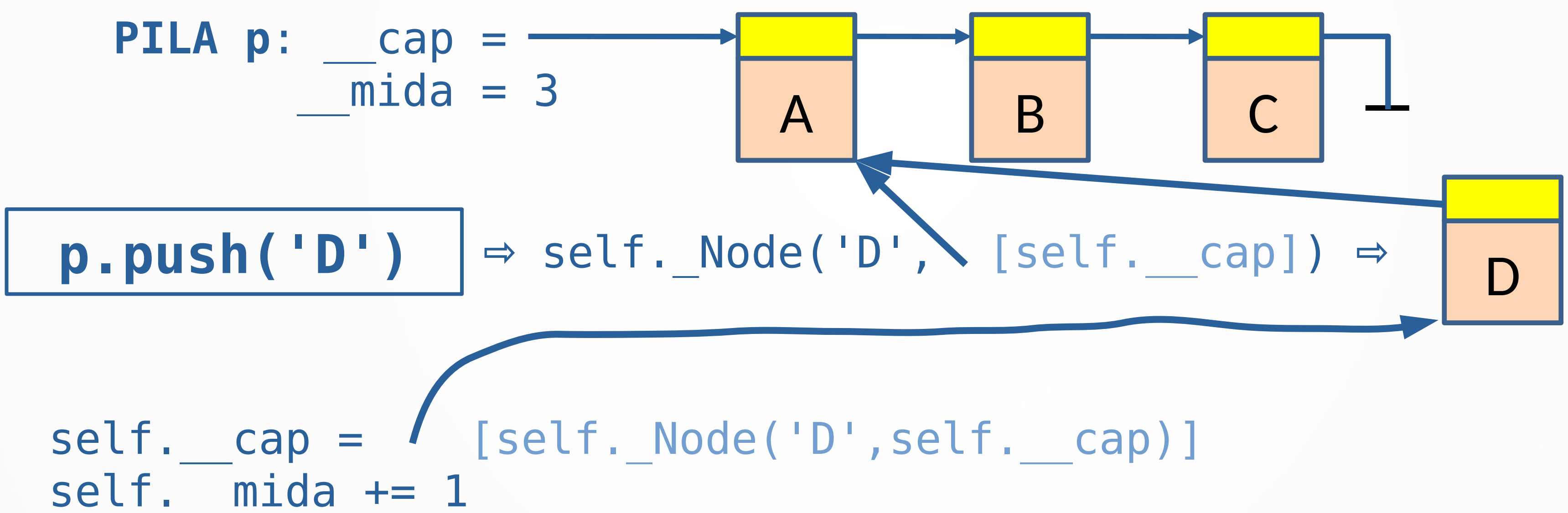
La Pila (*Stack*)

```
def push(self, e):
    self.__cap = self._Node(e, self.__cap)
    self.__mida += 1
```



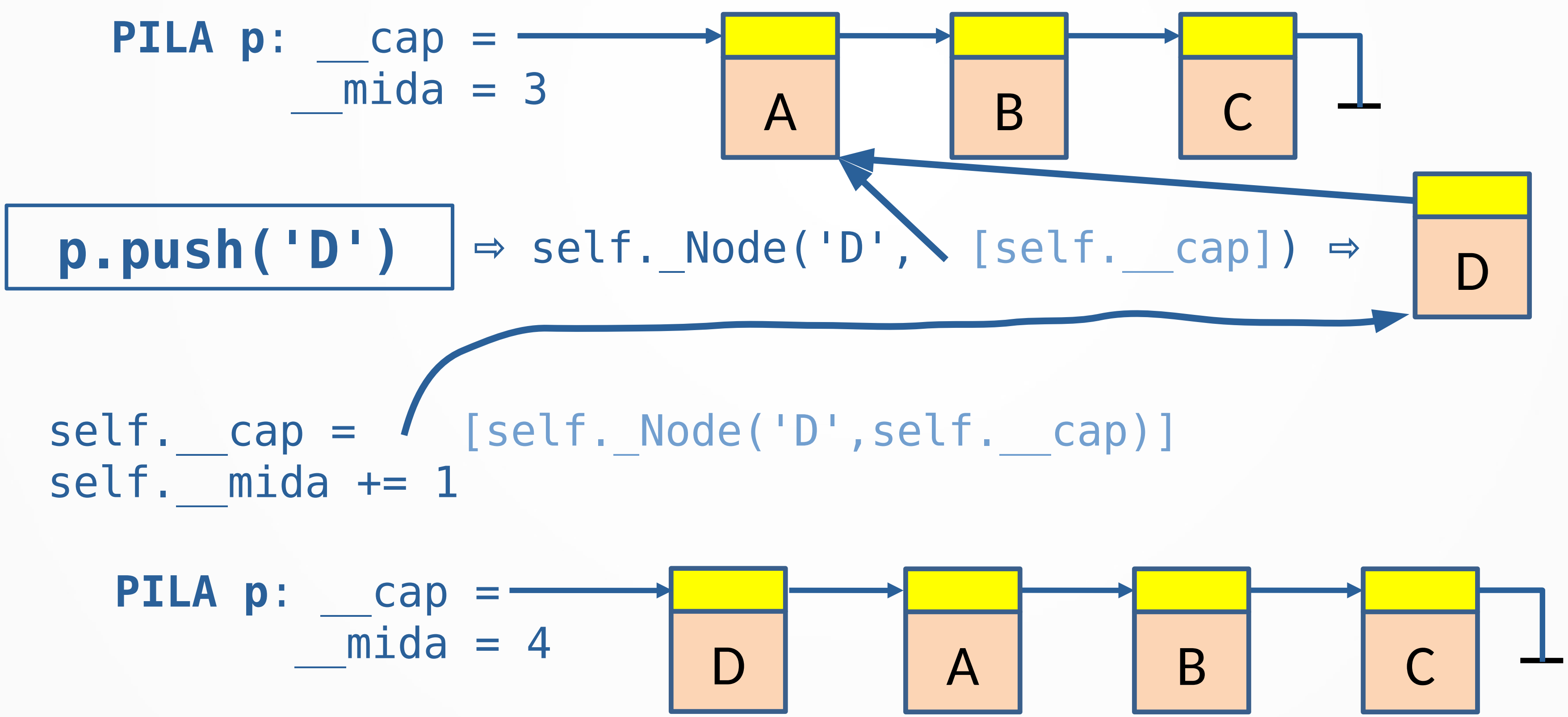
La Pila (*Stack*)

```python
def push(self, e):
    self.__cap = self._Node(e, self.__cap)
    self.__mida += 1
```



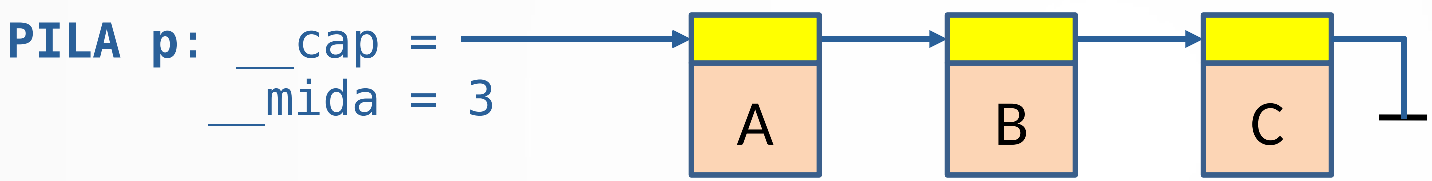
La Pila (*Stack*)

```python
def push(self, e):
    self.__cap = self._Node(e, self.__cap)
    self.__mida += 1
```



La Pila (*Stack*)

```
def pop(self):
    # Pre: La pila no és buida
    resposta = (self.__cap)._element
    self.__cap = (self.__cap)._next
    self.__mida -= 1
    return resposta
```

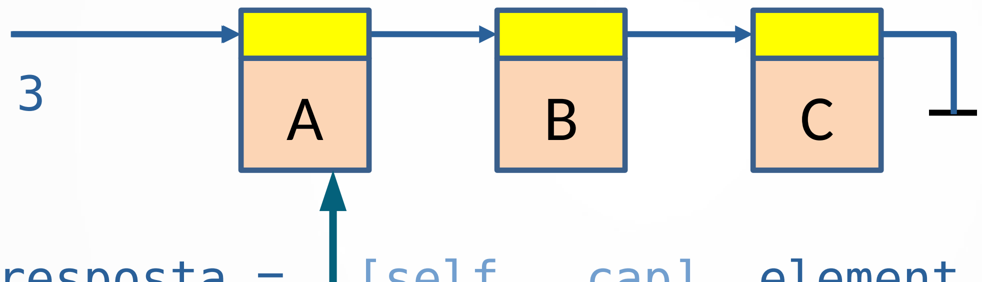



```
p.pop()
```

La Pila (*Stack*)

```
def pop(self):
    # Pre: La pila no és buida
    resposta = (self.__cap)._element
    self.__cap = (self.__cap)._next
    self.__mida -= 1
    return resposta
```

PILA p: __cap =
__mida = 3

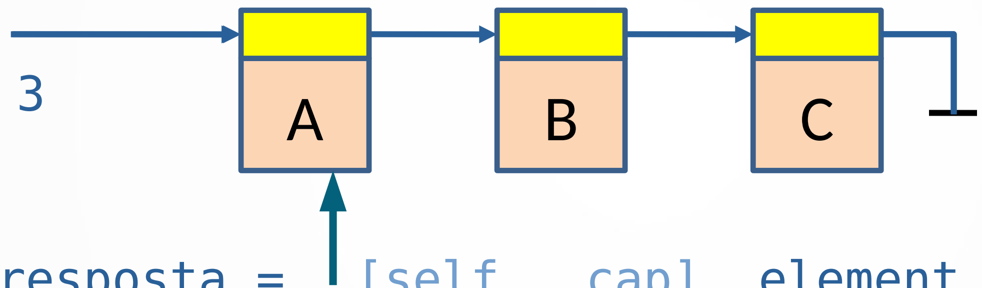


p.pop() ⇨ resposta = [self.__cap]._element ⇨ **'A'**

La Pila (*Stack*)

```
def pop(self):
    # Pre: La pila no és buida
    resposta = (self.__cap)._element
    self.__cap = (self.__cap)._next
    self.__mida -= 1
    return resposta
```

PILA p: __cap =
__mida = 3



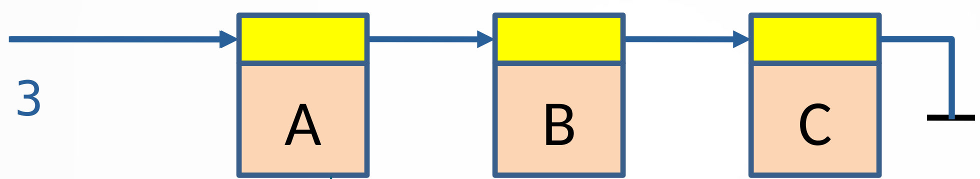
p.pop() ⇒ resposta = [self.__cap]._element ⇒ **'A'**

```
self.__cap = (self.__cap)._next
self.__mida -= 1
```

La Pila (*Stack*)

```
def pop(self):
    # Pre: La pila no és buida
    resposta = (self.__cap)._element
    self.__cap = (self.__cap)._next
    self.__mida -= 1
    return resposta
```

```
PILA p: __cap =
       __mida = 3
```

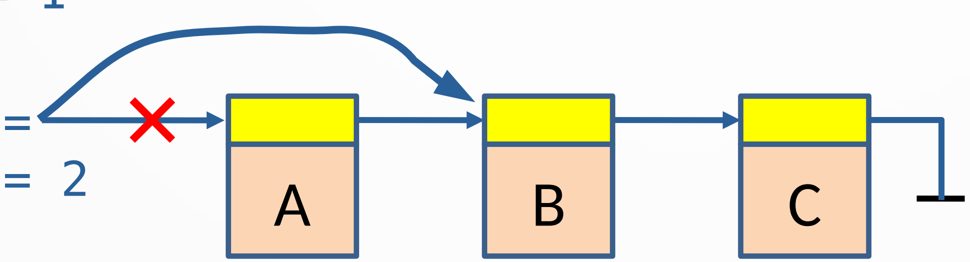



```
p.pop()
```

⇒ `resposta = [self.__cap]._element` ⇒ **'A'**

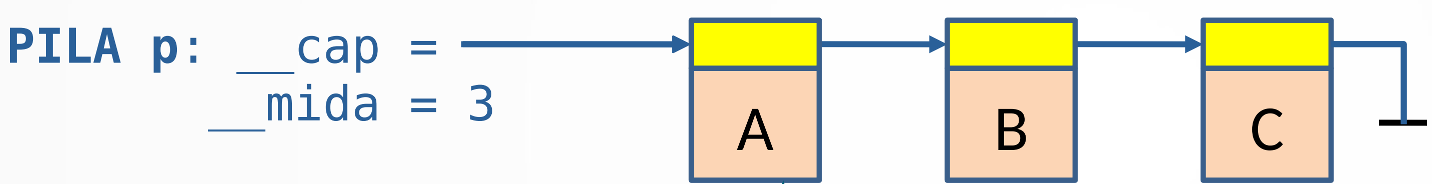
```
self.__cap = (self.__cap)._next
self.__mida -= 1
```

```
PILA p: __cap =
       __mida = 2
```



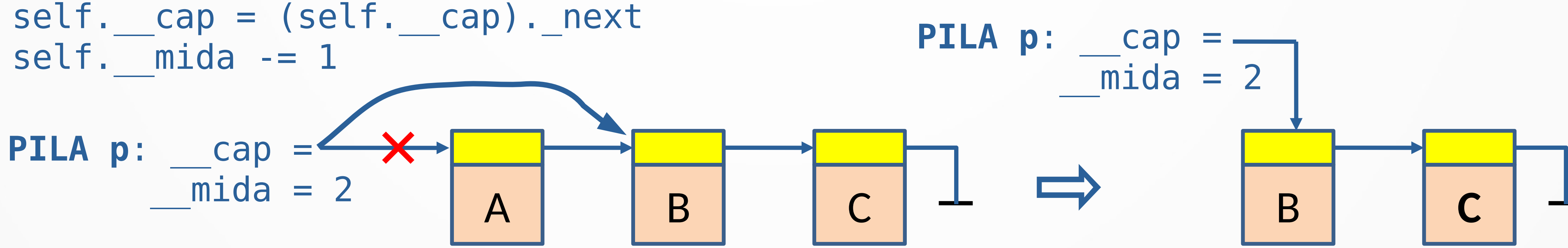
La Pila (*Stack*)

```python
def pop(self):
    # Pre: La pila no és buida
    resposta = (self.__cap)._element
    self.__cap = (self.__cap)._next
    self.__mida -= 1
    return resposta
```



p.pop() ⇒ resposta = [self.__cap]._element ⇒ **'A'**

```python
self.__cap = (self.__cap)._next
self.__mida -= 1
```



La Pila (*Stack*)

```python
class Pila:

    # ---------------------------------------------------
    # Classe interna per definir els elements de la pila:
    # Cada element de la pila serà una instància de _Node
    class _Node:
        __slots__ = '_element', '_next'

        def __init__(self, element, next):
            self._element = element
            self._next = next
    # ------------------------------------------

    def __init__(self):
        self.__cap = None
        self.__mida = 0

    def buida(self):
        return self.__mida == 0

    def mida(self):
        return self.__mida

    def push(self, e):
        self.__cap = self._Node(e, self.__cap)
        self.__mida += 1

    def top(self):
        # Pre: La pila no és buida
        return (self.__cap)._element

    def pop(self):
        # Pre: La pila no és buida
        resposta = (self.__cap)._element
        self.__cap = (self.__cap)._next
        self.__mida -= 1
        return resposta
```

Exercici: Parèntesis balancejats

- Donada una string amb caràcters '[', ']', '{', '}', '(' i ')', comprovar que els caràcters d'obrir ('[', '{' i '(') estàn ben aparellats amb els corresponents caràcters de tancar (']', '}' i ')').

  Per exemple: ({{}[()]}[{()()[]}]) és correcte, {[]{(}{)}}[(())]({}[]) no és correcte

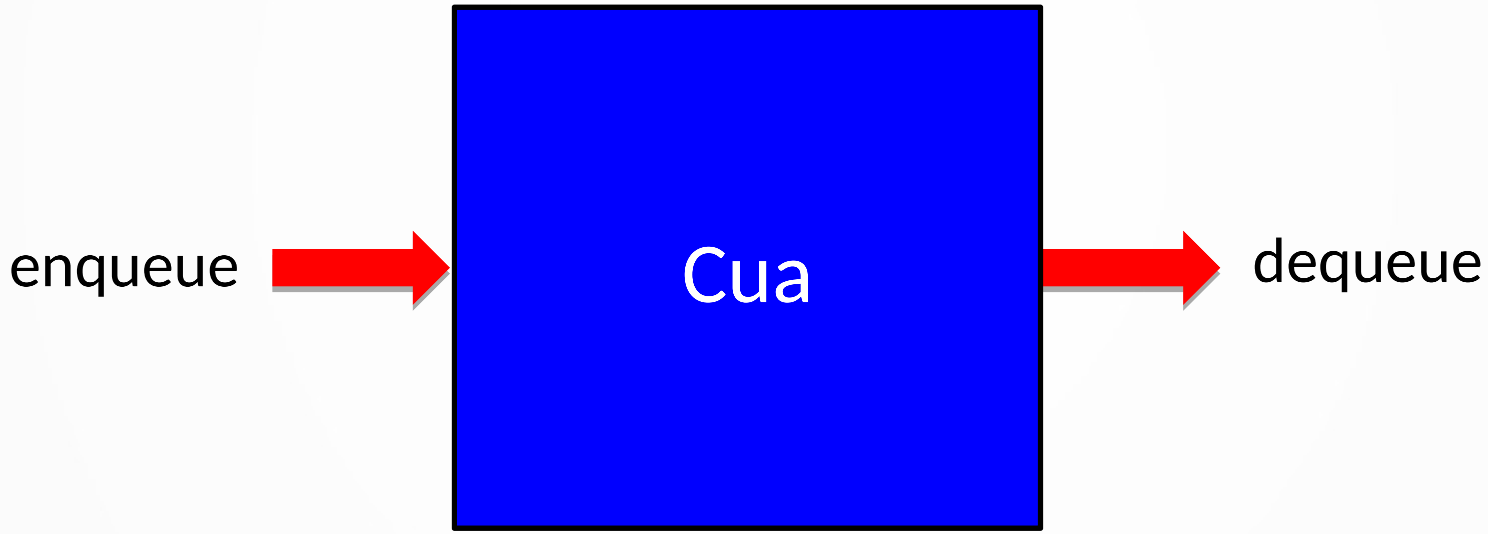
```
def correct(s):
    p = Pila()
    for c in s:
        if c in ['(', '[', '{']:
            p.push(c)
        else:
            assert c in [')', ']', '}']
            if p.buida():
                return False
            else:
                t = p.pop()
                if not ((t == '(' and c == ')') or (t == '[' and c == ']') or \
                        (t == '{' and c == '}')):
                    return False
    return p.buida()
```

La Cua (*Queue*)

- La Cua és una estructura de dades lineal que ja hem mencionat breument relacionant-la amb l'estructura **`deque`** (*doubly ended queue*) del mòdul **`Collections`** de Python. Vam dir que fer servir *només* les operacions `popleft` i `append` en un **`deque`** el feia equivalent a una Cua.
- És un contenidor on afegim elements per un extrem i els eliminem per l'altre extrem.
- La Cua, com a estructura de dades, ve definida per la següent especificació, per les següents operacions:
  - constructor, que retorna una cua buida
  - `enqueue(x)`: afegir element `x` al final de la Cua
  - `dequeue()`: eliminar el primer element. Hem de suposar que la Cua no és buida
  - `first()`: consultar el primer element. Hem de suposar que la Cua no és buida
  - `buida()`: demanar si la Cua és buida
  - `mida()`: demanar la mida de la Cua

La Cua (*Queue*)

- Les Cues també es coneixen com a estructures FIFO (*First In*, *First Out*)



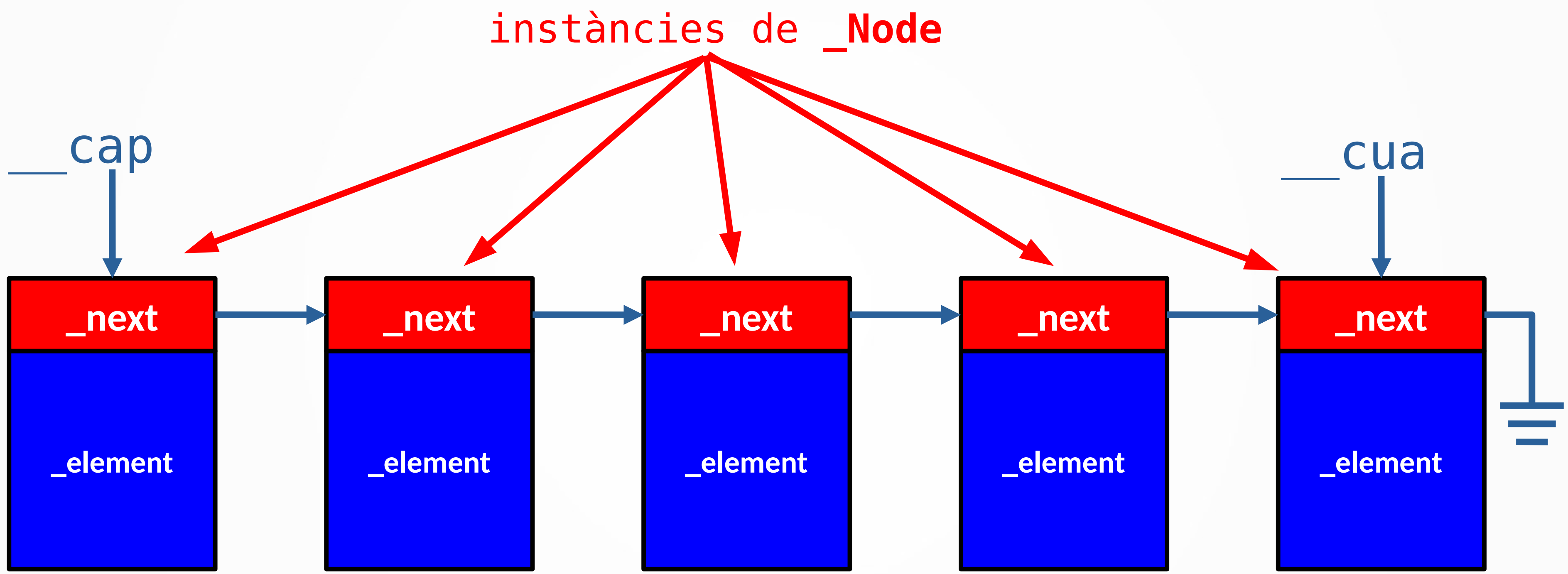
La Cua (*Queue*)

- Fent servir classes, tindríem:

```
class Cua:

    # ---------------------------------------------------
    # Classe interna per definir els elements de la cua:
    # Cada element de la cua serà una instància de _Node
    class _Node:
        __slots__ = '_element', '_next'        # opcional, per eficiència

        def __init__(self, element, next):     # inicialitzar els camps
            self._element = element            # ref. a l'element emmagatzemat
            self._next = next                  # referència al proper node
    # ---------------------------------------------------

    def __init__(self):
        self.__cap = None   # primer node de la cua
        self.__cua = None   # darrer node de la cua
        self.__mida = 0     # nombre d'elements de la cua
```

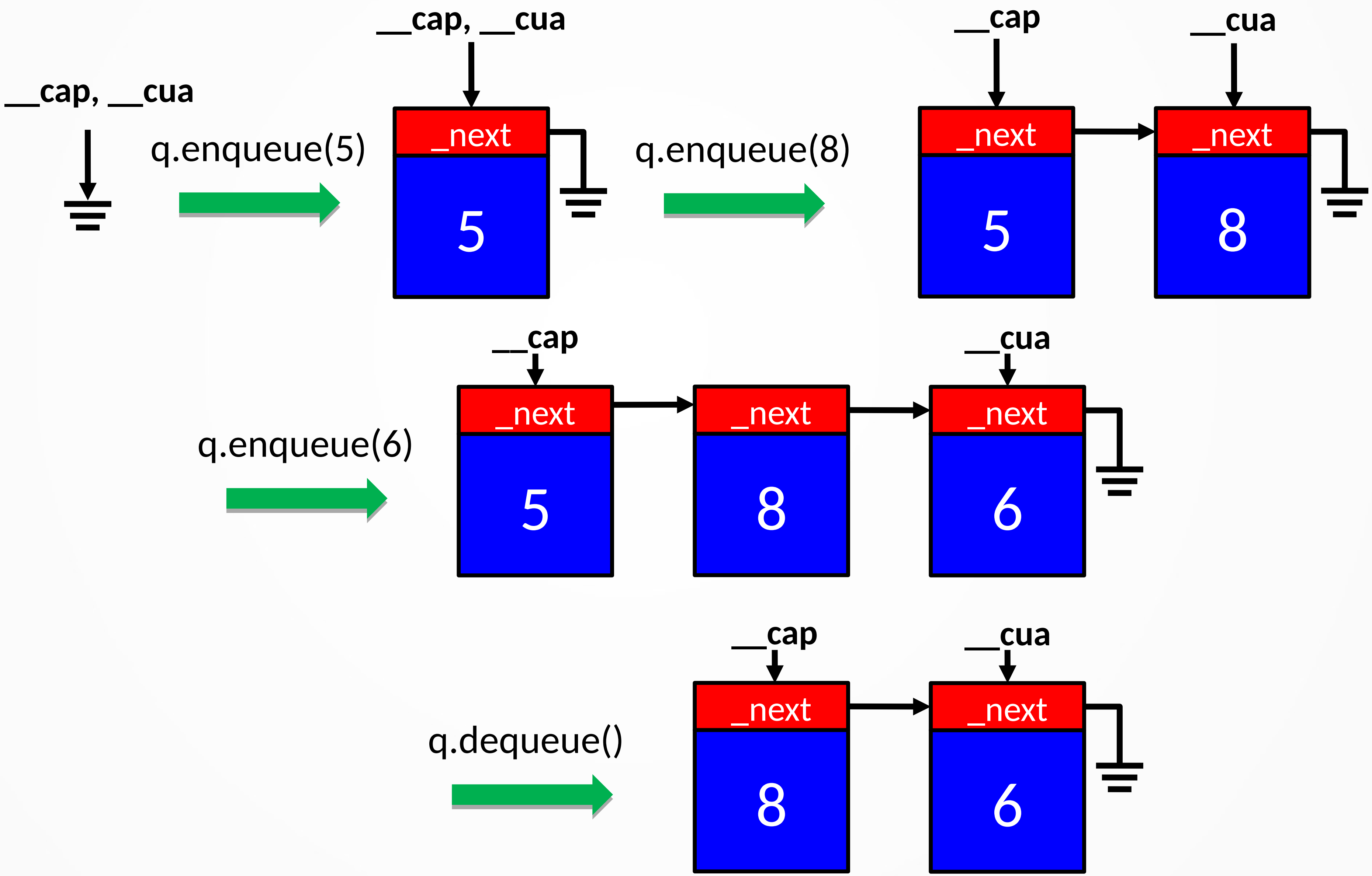
La Cua (*Queue*)

- Gràficament:



La Cua (*Queue*)

```
q = Cua()
```



La Cua (*Queue*)

```python
class Cua:
    # --------------------------------------------------------------------
    # Cada element de la cua serà una instància de _Node
    class _Node:
        __slots__ = '_element', '_next'
        def __init__(self, element, next):
            self._element = element
            self._next = next
    # ------------------------------------------------------

    def __init__(self):
        self.__cap = None
        self.__cua = None
        self.__mida = 0

    def buida(self):
        return self.__mida == 0

    def mida(self):
        return self.__mida

    def first(self):
        # Pre: La cua no és buida
        return (self.__cap)._element

    def enqueue(self, e):
        # nou node al final de la cua
        nou = self._Node(e, None)
        if self.buida():
            # cas especial, cua buida
            self.__cap = nou
        else:
            self.__cua._next = nou
        # actualitzar referència al darrer node
        self.__cua = nou
        self.__mida += 1

    def dequeue(self):
        # Pre: La cua no és buida
        resposta = self.__cap._element
        self.__cap = self.__cap._next
        self.__mida -= 1
        if self.buida():
            # cas especial, cua buida
            # el __cap eliminat també
            # era la cua
            self.__cua = None
        return resposta
```

Exercici: Revisitant el recorregut per nivells

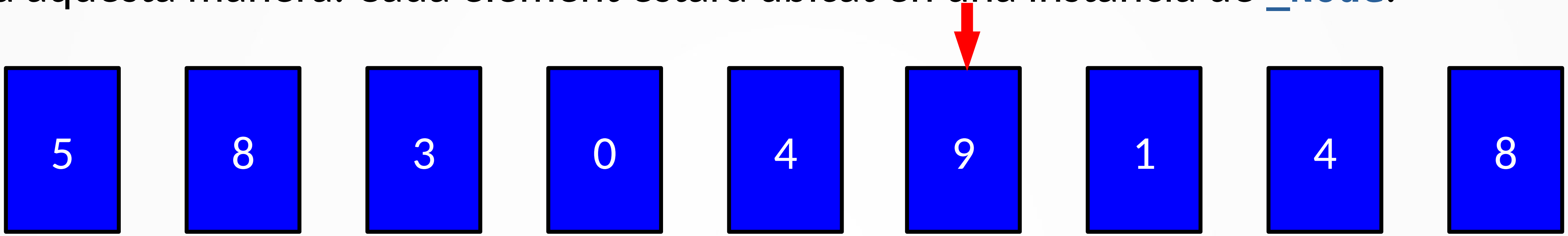
```
from cua import Cua:

class BinTree:
    # . . .
    def levelorder(self):
        """
        returns a list with the elements of the BinTree, ordered
        as is specified in the definition of the levels-order traversal.
        """
        if self.empty():
            return []
        else:
            resultat = []
            q = Cua()
            q.enqueue(self)
            while not q.buida():
                tt = q.dequeue()
                resultat.append(tt.get_root())
                if not tt.get_left().empty():
                    q.enqueue(tt.get_left())
                if not tt.get_right().empty():
                    q.enqueue(tt.get_right())
            return resultat
```

La Llista

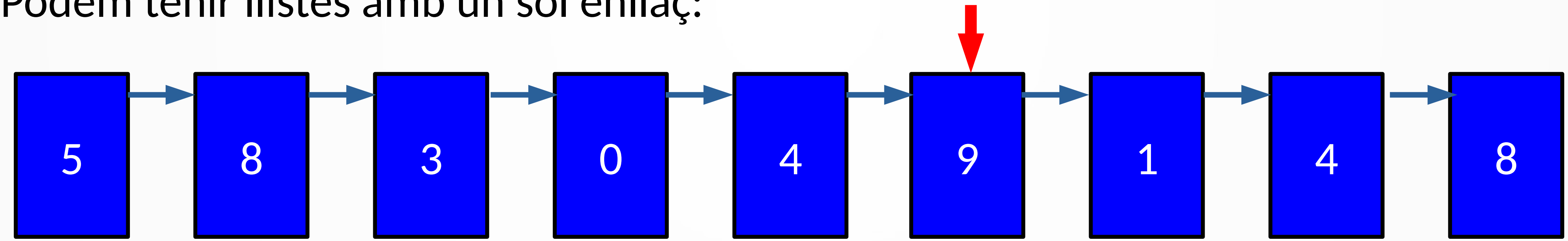
- La Llista és una estructura de dades lineal on podem inserir i esborrar a qualsevol lloc de la llista. És l'estructura lineal més flexible, tot i que ***l'accés és seqüencial***.
- Podem considerar una llista com una seqüència d'elements amb un o més <u>*cursors*</u> referenciant un o més elements de la llista. Nosaltres veurem aquí llistes amb ***un sol cursor***.
- La Llista (amb un sol cursor), com a estructura de dades, ve definida per la següent especificació, per les següents operacions:
  - constructor, que retorna una llista buida.
  - `move_forward()`: mou el cursor una posició endavant.
  - `move_backward()`: mou el cursor una posició enrere.
  - consultors relacionats amb el cursor...
  - `insert(x)`: insereix l'element x *abans* del cursor.
  - `erase()`: elimina l'element referenciat pel cursor.
  - `current()`: retorna l'element referenciat pel cursor.

La Llista

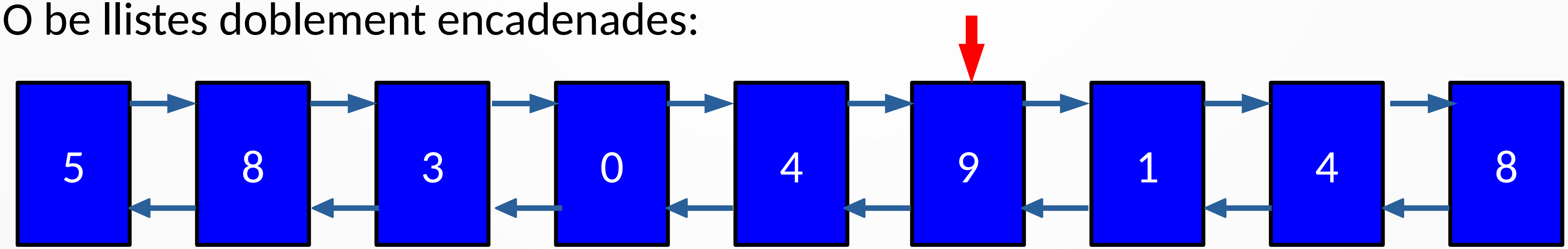
- Les Llistes les podem representar gràficament (el cursor és el dibuix vermell) d'aquesta manera. Cada element estarà ubicat en una instància de `_Node`.



- Podem tenir llistes amb un sol enllaç:

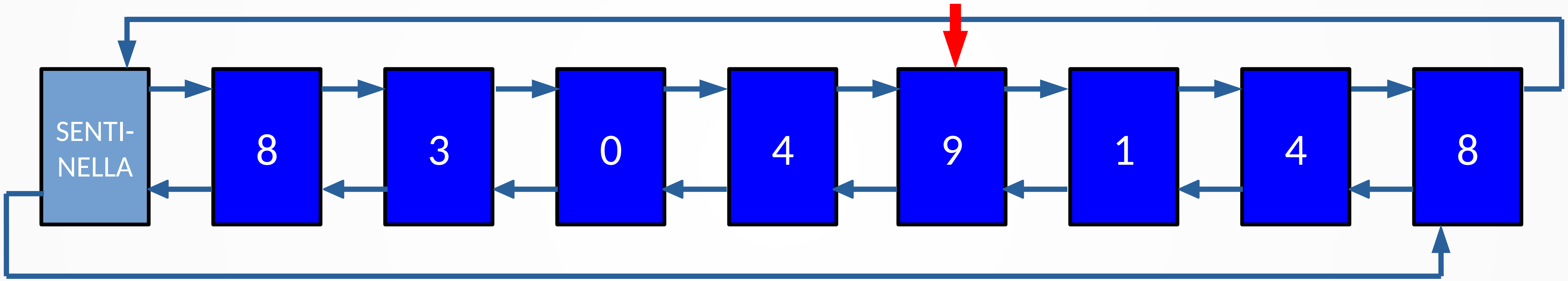


- O be llistes doblement encadenades:



La Llista

- Hi ha diverses maneres d'implementar llistes. Nosaltres presentarem una implementació basada en ***llistes doblement encadenades, amb un cursor, amb sentinella***.
- Gràficament seria quelcom com:




```python
# Cada element de la llista serà una instància de _Node
class _Node:
    __slots__ = '_element','_next','_prev' # opcional, per eficiència

    def __init__(self, prev, next, element=None):
        self._element = element            # ref. a l'element emmagatzemat
        self._next = next                  # referència al proper node
        self._prev = prev                  # referència al node anterior
```

La Llista

- Fent servir classes, tindríem:

```
class Llista:

    # ------------------------------------------------------
    # Classe interna per definir els elements de la llista:
    # Cada element de la llista serà una instància de _Node
    class _Node:
        __slots__ = '_element','_next','_prev' # opcional, per eficiència

        def __init__(self, prev, next, element=None):
            self._element = element                   # ref. a l'element emmagatzemat
            self._next = next                         # referència al proper node
            self._prev = prev                         # referència al node anterior
    # ------------------------------------------------------

    def __init__(self):
        self.__sentinella = self._Node(None,None)   # _Node sentinella
        self.__sentinella._next = self.__sentinella
        self.__sentinella._prev = self.__sentinella
        self.__cursor   = self.__sentinella
        self.__n        = 0                           # nombre d'elements (sense sentinella!)
```

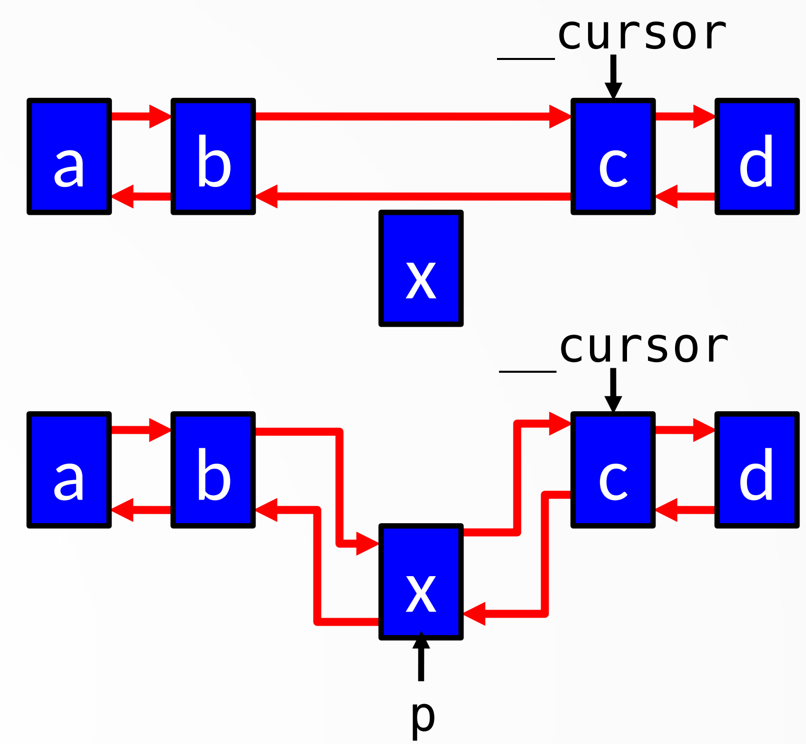
La Llista

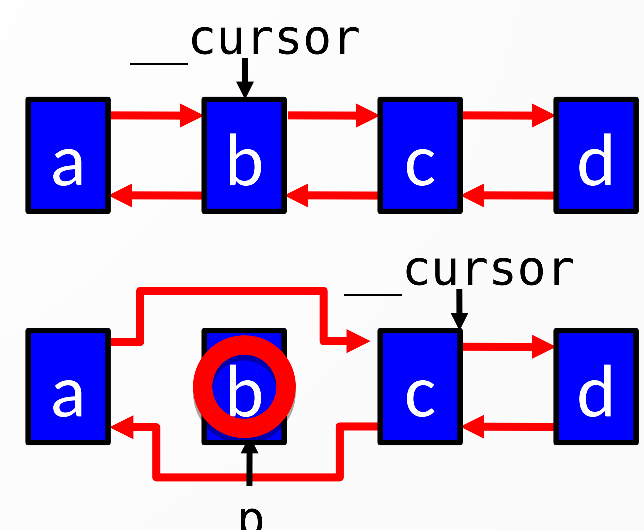
```python
class Llista:

    # . . .

    def mida(self):
        return self.__n

    def buida(self):
        return self.__n == 0

    # Comprova si el cursor és al principi
    # de la llista
    def is_at_front(self):
        return self.__cursor == self.__sentinella._next

    # Comprova si el cursor és al final
    # de la llista
    def is_at_end(self):
        return self.__cursor == self.__sentinella
```

```python
# Mou el cursor una posició enrere
def move_backward(self):
    # Pre: el cursor NO està al començament
    assert not self.is_at_front()
    self.__cursor = self.__cursor._prev

# Mou el cursor una posició endavant
def move_forward(self):
    # Pre: el cursor NO està al final
    assert not self.is_at_end()
    self.__cursor = self.__cursor._next

# Mou el cursor al principi de la llista
def move_to_front(self):
    self.__cursor = self.__sentinella._next

# Mou el cursor al final de la llista
def move_to_end(self):
    self.__cursor = self.__sentinella # !!!

# Retorna l'element referenciat pel cursor
def front(self):
    # Pre: el cursor NO està al final
    assert not self.is_at_end()
    return self.__cursor._element
```

La Llista

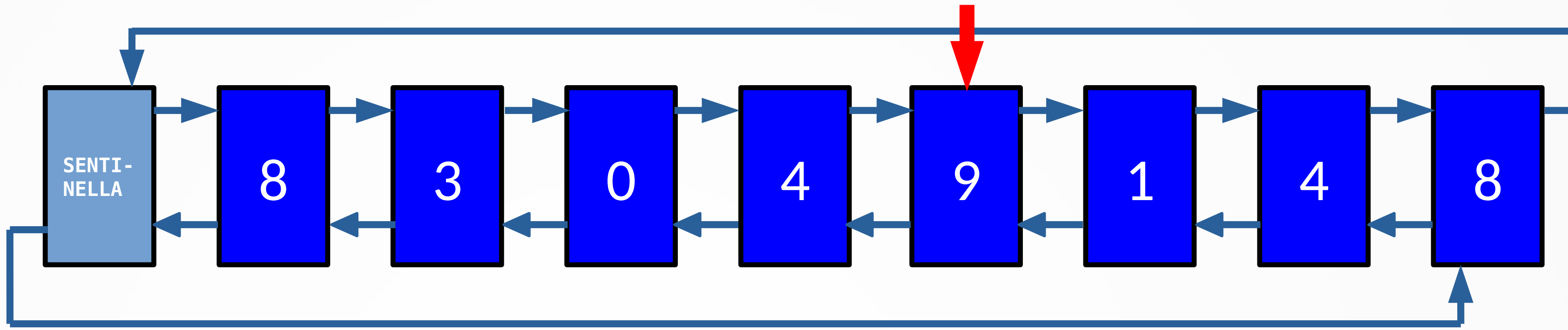
```
class Llista:
    # . . .
    # Insereix l'element x abans del cursor
    def insert(self, x):
        p = self._Node(self.__cursor._prev, self.__cursor, x)
        self.__cursor._prev._next = p
        self.__cursor._prev = p
        self.__n += 1
        return self


    # Elimina l'element ref. pel cursor
    # i avança el cursor una posició
    def erase(self):
        # Pre: el cursor NO està al final
        assert not self.is_at_end()
        p = self.__cursor
        p._next._prev = p._prev
        p._prev._next = p._next
        self.__cursor = p._next
        self.__n -= 1
```



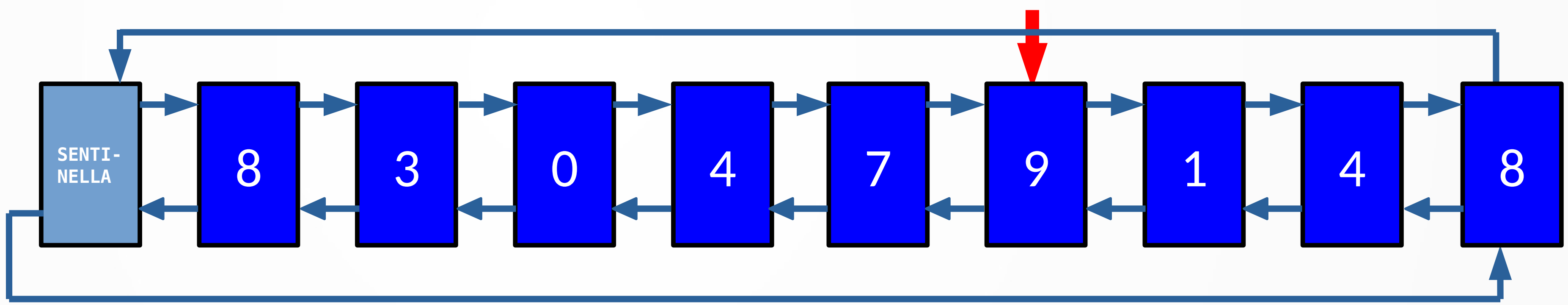


La Llista

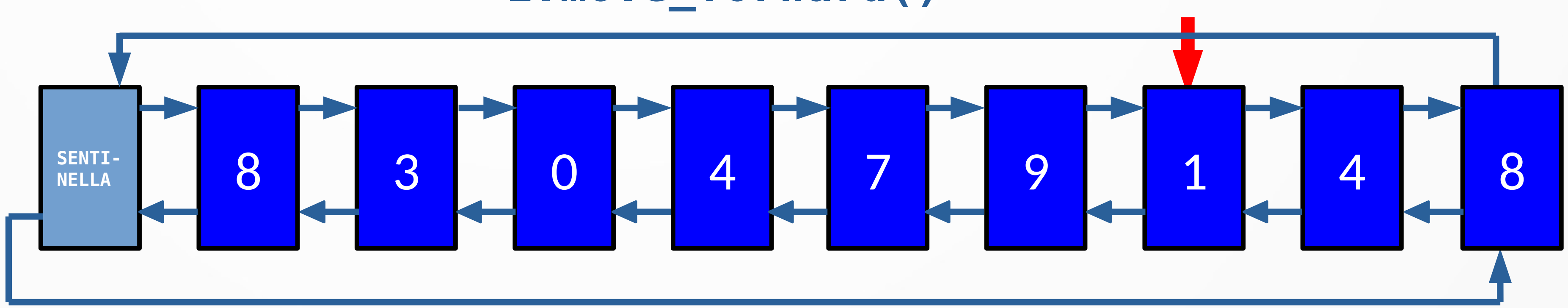
L =



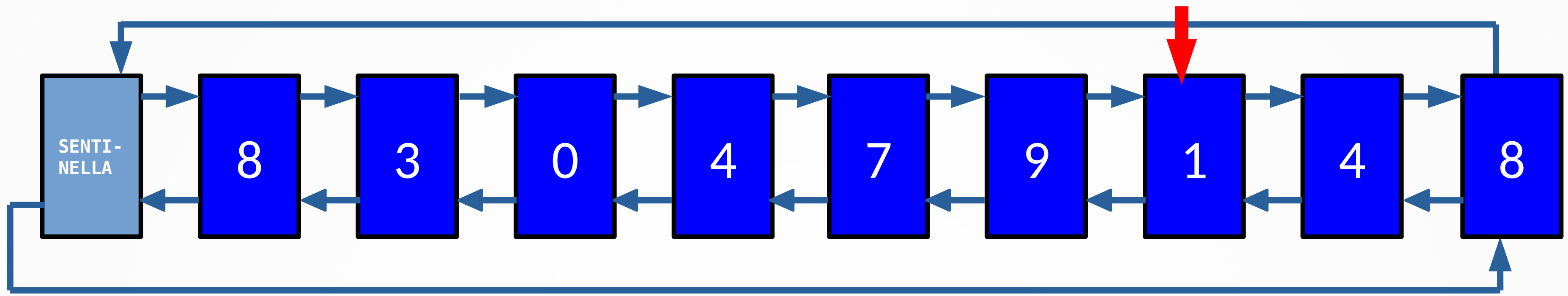
`L.insert(7)`



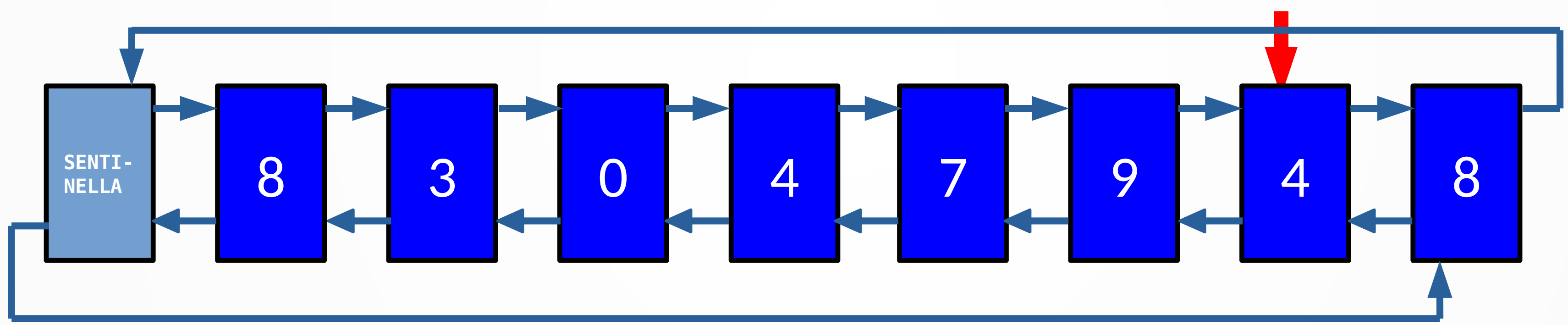
`L.move_forward()`



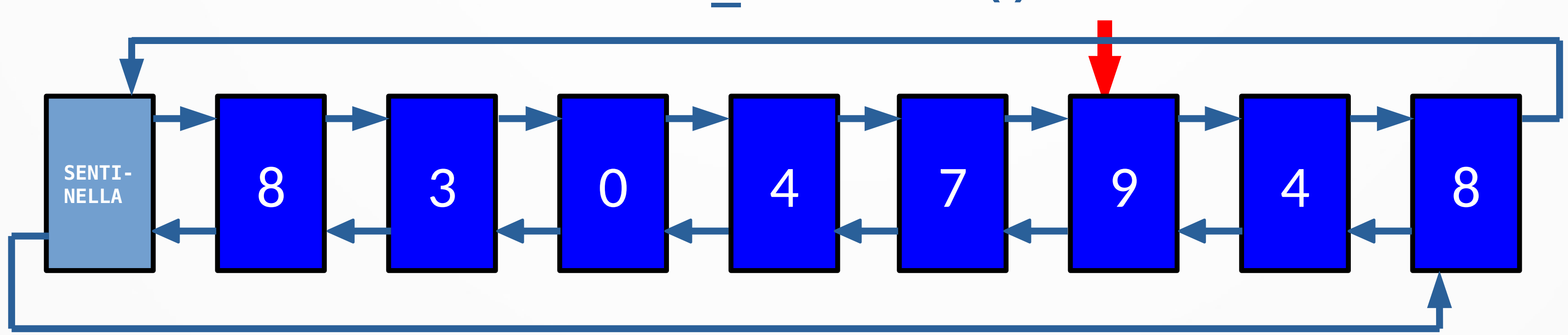
La Llista



`L.erase()`

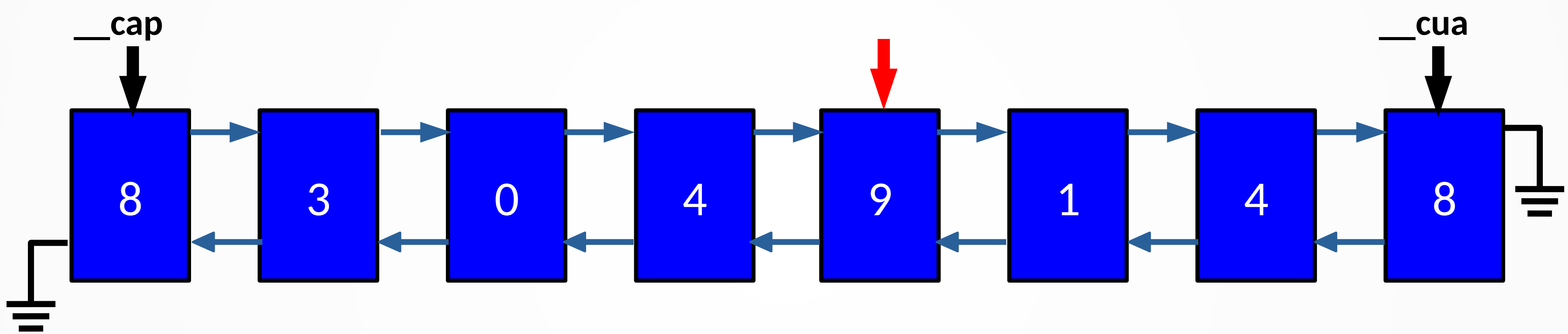


`L.move_backward()`



Exercici: Per a què serveix el sentinella?

- Imagineu-vos una llista doblement encadenada, *sense sentinella*. Si voleu, afegiu-li una referència al cap i una referència a la cua, a més del cursor:

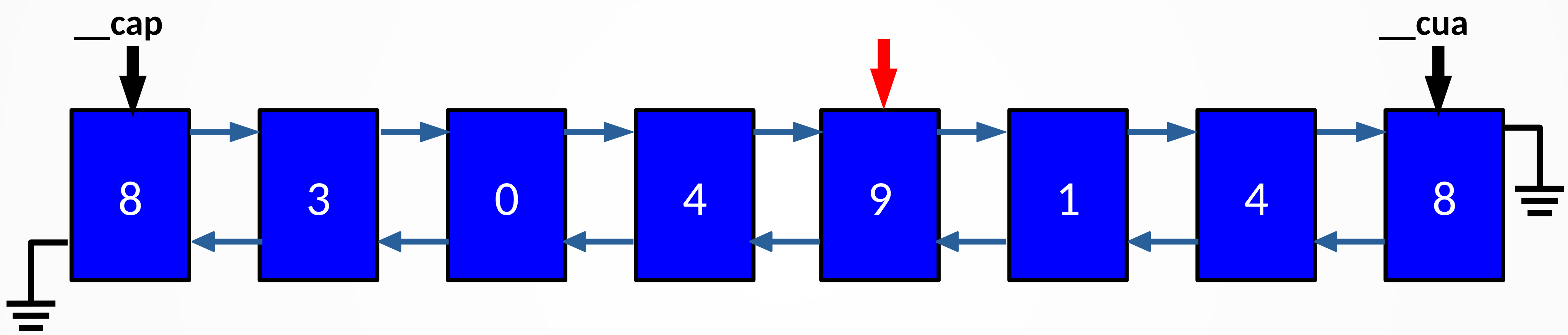


Ara, imagineu implementar els mateixos mètodes que hem vist, per a aquesta llista.

- Aquesta implementació… ***seria tan senzilla com la que hem vist?***

Exercici: Per a què serveix el sentinella?

- Imagineu-vos una llista doblement encadenada, *sense sentinella*. Si voleu, afegiu-li una referència al cap i una referència a la cua, a més del cursor:



Ara, imagineu implementar els mateixos mètodes que hem vist, per a aquesta llista.

- Aquesta implementació... ***seria tan senzilla com la que hem vist?***

NO!! El codi s'ompliria de condicionals comprovant condicions d'extrem: si estem al principi, al final, etc. El codi seria força més complicat.

Exercici: Inversió destructiva

- Afegeix un mètode `reverse` a la classe `Llista`, de manera que la llista sigui invertida *destructivament*. No es pot fer servir cap mena d'estructura de dades auxiliar, ni tan sols una altra llista, i no es poden fer còpies dels elements.

Exercici: Inversió destructiva

- Afegeix un mètode **`reverse`** a la classe **`Llista`**, de manera que la llista sigui invertida *destructivament*. No es pot fer servir cap mena d'estructura de dades auxiliar, ni tan sols una altra llista, i no es poden fer còpies dels elements.
- La solució és senzilla: *permutar el* **`_prev`** *i el* **`_next`** *de tots els elements (inclòs el sentinella)*

```
def reverse(self):
    if not self.buida():
        to_change = self.__sentinella    # comencem pel sentinella

        # simulo un do...while
        next_to_change = to_change._next  # guardo el proper a tractar i...
                                          # ...permuto les dues referències
        to_change._prev, to_change._next = to_change._next, to_change._prev
        to_change = next_to_change        # el proper a tractar és el guardat

        while not to_change == self.__sentinella:
            next_to_change = to_change._next  # guardo el proper a tractar i...
                                              # ...permuto les dues referències
            to_change._prev, to_change._next = to_change._next, to_change._prev
            to_change = next_to_change        # el proper a tractar és el guardat
```

Exercici: *Heaps*

- A l'hora d'implementar un `Heap` no ens convé fer servir una `Llista` com la que hem implementat aquí. Deixarem la implementació del `Heap` fent servir les llistes pròpies de Python.

  Per què?

Exercici: *Heaps*

- A l'hora d'implementar un `Heap` no ens convé fer servir una `Llista` com la que hem implementat aquí. Deixarem la implementació del `Heap` fent servir les llistes pròpies de Python.

  Per què?

  Perquè tal com hem dit fa unes transparències, quan introduíem les llistes, ***l'accés als elements de les llistes és seqüencial***. Així, accedir a l'i-éssim element d'una instància de `Llista` té un cost lineal. En la implementació del `Heap` ens interessa que l'accés als elements de les llistes sigui constant, que és el que ens garanteixen les llistes de Python: `wiki.python.org/moin/TimeComplexity`

Exercici: Mètodes d'ordre superior

- Afegeix els següents mètodes d'ordre superior a la classe `Llista`:

```python
# Transforma tots els elements de la llista utilitzant f,
# retorna una llista amb els elements transformats.
def transform(self, f):

# Retorna una llista amb els elements pels quals f és certa
def filter(self, f):

# S'aplica f seqüencialment a la llista i retorna un
# valor únic. Per a la llista [x1, x2, x3, ... , xn]
# retorna f(... f(f(f(x0, x1), x2), x3) ... , xn).
# Si la llista és buida, retorna x0.
def reduce(self, x0, f):
```

Exercici: Mètodes d'ordre superior

- Afegeix els següents mètodes d'ordre superior a la classe `Llista`:

```
# Transforma tots els elements de la llista utilitzant f,
# retorna una llista amb els elements transformats.
def transform(self, f):
    l = Llista()
    p = self.__sentinella._next
    while (p != self.__sentinella):
        l.insert(f(p._element))
        p = p._next
    return l
```

Exercici: Mètodes d'ordre superior

- Afegeix els següents mètodes d'ordre superior a la classe `Llista`:

```
# Retorna una llista amb els elements pels quals f és certa
def filter(self, f):
    l = Llista()
    p = self.__sentinella._next
    while (p != self.__sentinella):
       if f(p._element):
           l.insert(p._element)
       p = p._next
    return l
```

Exercici: Mètodes d'ordre superior

- Afegeix els següents mètodes d'ordre superior a la classe `Llista`:

```
# S'aplica f seqüencialment a la llista i retorna un
# valor únic. Per a la llista [x1, x2, x3, ... , xn]
# retorna f(... f(f(f(x0, x1), x2), x3) ... , xn).
# Si la llista és buida, retorna x0.
def reduce(self, x0, f):
    x = x0
    p = self.__sentinella._next
    while (p != self.__sentinella):
        x = f(x, p._element)
        p = p._next
    return x
```

La Llista com a iterable

- Ens agradaria poder tractar les instàncies de la classe `Llista`: com a ***objectes iterables***, és a dir, com a objectes als que podem associar ***un iterador***.
- Una raó podria ser utilitzar els bucles `for` sobre les llistes. Si `L` és una instància de `Llista`, voldríem poder fer bucles com: `for e in L: ...`
- Caldria implementar dins la classe `Llista` l'***iterator protocol***. Aquest consisteix en dues funcions: `__iter__` i `__next__` . La primera funció, `__iter__`, cal que inicialitzi l'objecte per poder començar la iteració. És cridada per la funció `iter`, que retorna l'objecte iterador en qüestió, sobre el que podrem invocar la funció `next`.
- L'ús d'aquestes funcions `iter` i `next` el vam veure a PA1, quan vam estudiar per primer cop els iteradors. Hauríeu de recordar que els bucles `for` no són més que sucre sintàctic sobre l'ús d'aquestes funcions:

```
for <nom> in <expressió>
    <cos>
```

≡

```
iterador = iter(<expressió>)
try:
    while True:
        <nom> = next(iterador)
        <cos>
except StopIteration:
    pass
```

La Llista com a iterable

- Aleshores, només cal afegir aquests mètodes a la classe **`Llista`**:

```python
def __iter__(self):
    self.move_to_front()
    return self
```

```python
def __next__(self):
    if self.is_at_end():
        raise StopIteration
    else:
        resultat = self.front()
        self.move_forward()
        return resultat
```

```
>>> from llista import *
>>> L = Llista().insert(3).insert(5).insert(9)
>>> for e in L:
...     print(e)
...
3
5
9
>>>
```

Arbres Binaris

- L'arbre binari és una estructura de dades ***NO-LINEAL***. La interfície pública (els mètodes públics) hauria de ser la mateixa que la que ja vam implementar a la classe **`BinTree`** (fitxer **`bintree.py`**):
  - constructor, amb les mateixes opcions que ja teníem
  - *getters*: `get_root`, `get_left`, `get_right`
  - *setters*: `set_root`, `set_left`, `set_right`
  - recorreguts: `preorder`, `postorder`, `inorder`, `levelorder`
  - funcions auxiliars: `__repr__`
- Ara, però, *canviarem la implementació* i farem servir, com hem vist amb les piles, les cues i les llistes, una classe auxiliar amb la que representar els elements de l'arbre:

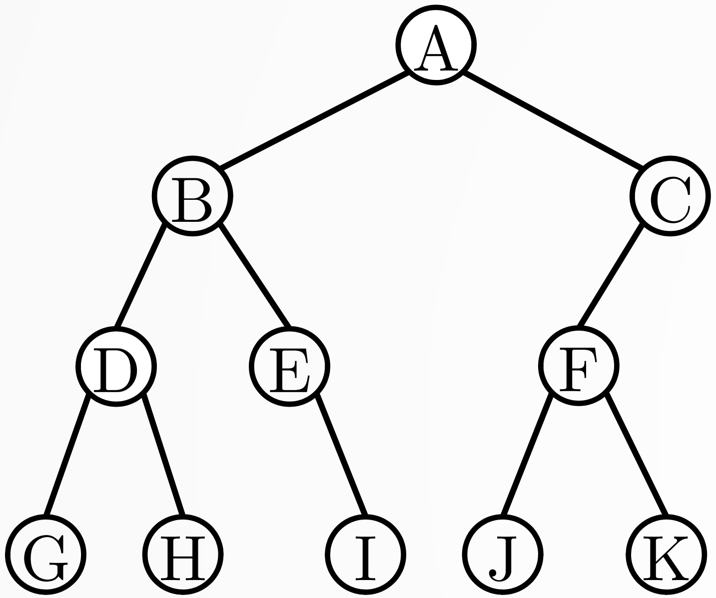
```
# Cada element de l'arbre serà una instància de _Node
class _Node:
    __slots__ = '_element','_left','_right' # opcional, per eficiència

    def __init__(self, element, left=None, right=None):
        self._element = element             # ref. a l'element emmagatzemat
        self._left  = left                  # ref. al fill esquerra (no és BinTree!)
        self._right = right                 # ref. al fill dret (no és BinTree!)
```

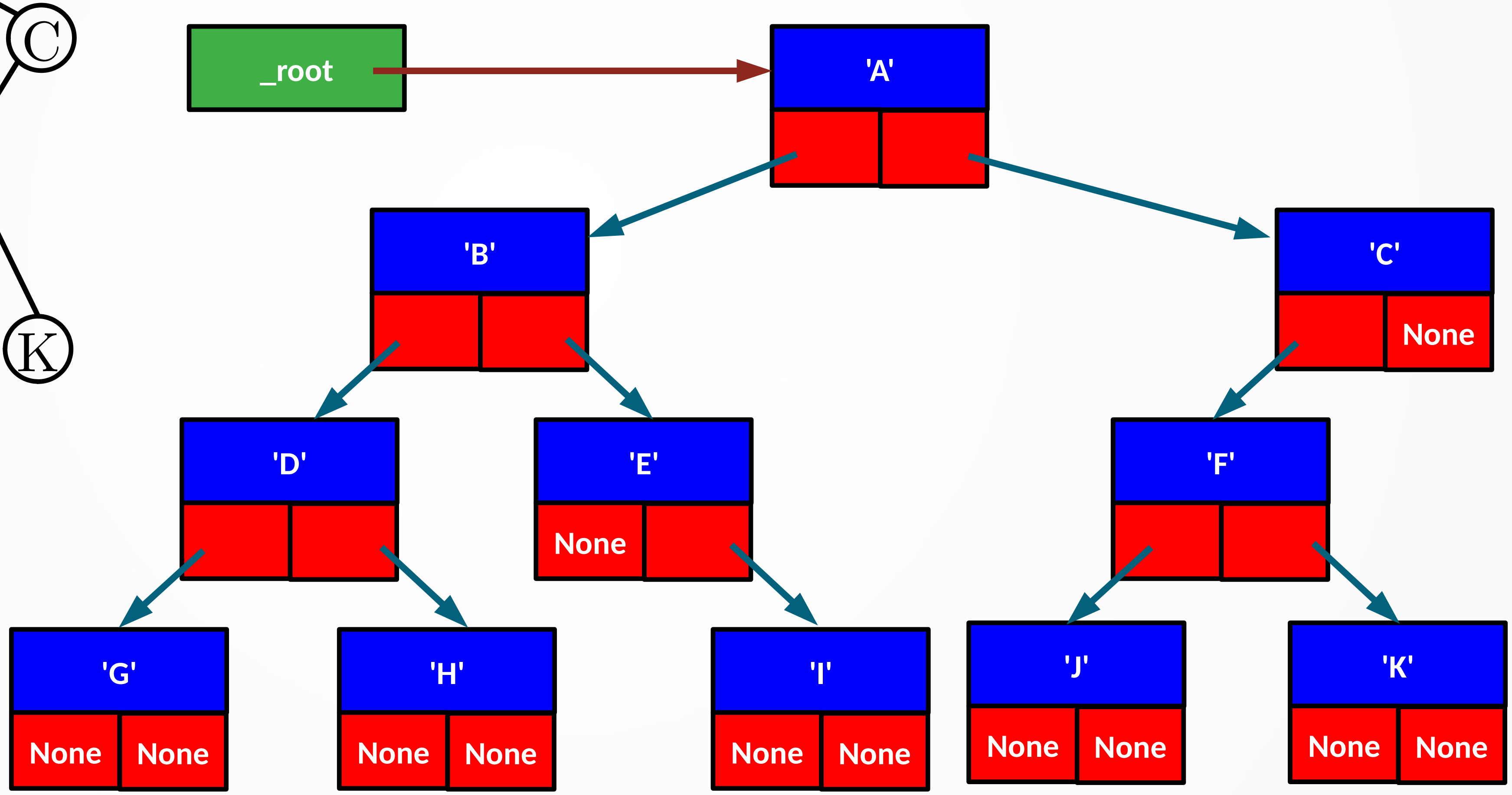
Arbres Binaris

- Així, començarem la nova implementació de la classe **BinTree**:

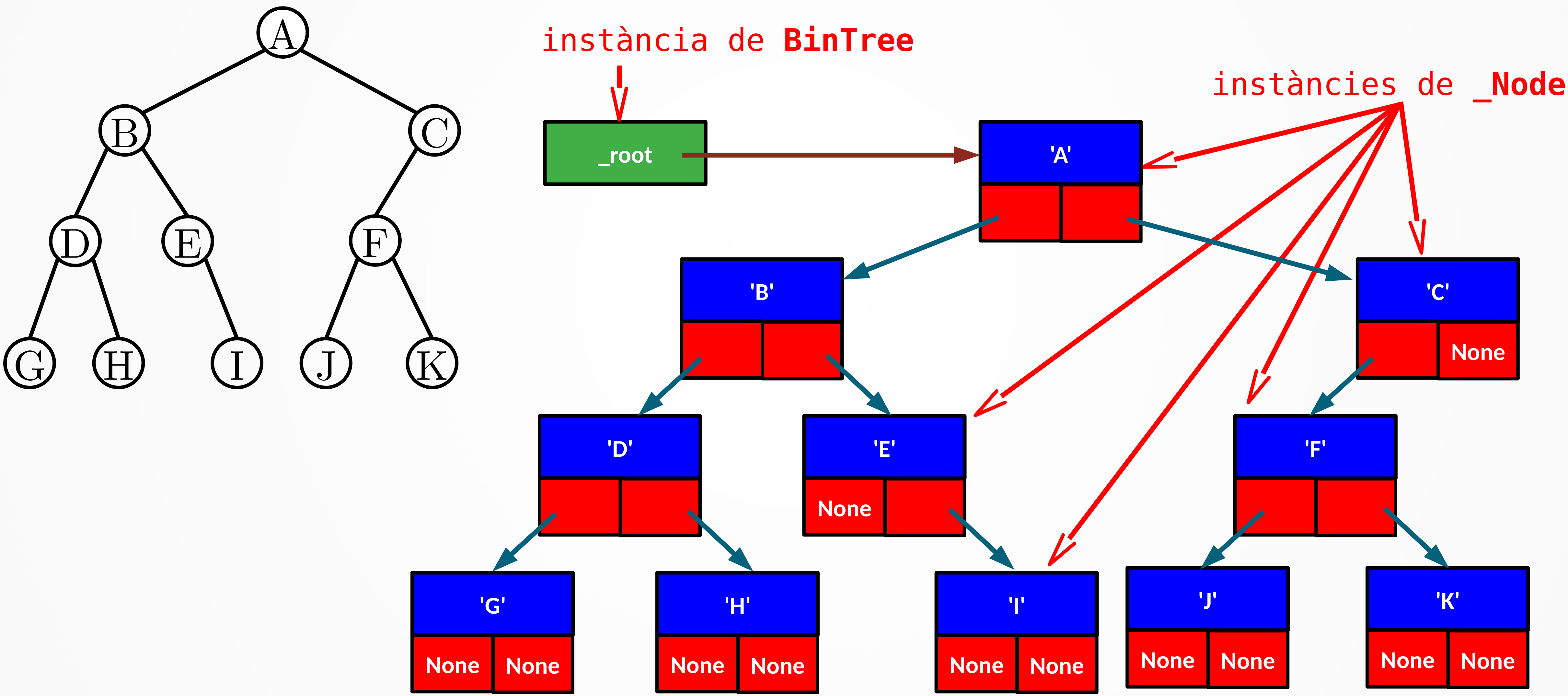
```
class BinTree:
    # --------------------------------------------------
    # Cada element de l'arbre serà una instància de _Node
     class _Node:
        __slots__ = '_element','_left','_right' # opcional, per eficiència

        def __init__(self, element, left=None, right=None):
            self._element = element             # ref. a l'element emmagatzemat
            self._left  = left                  # ref. al fill esquerra (no és BinTree!)
            self._right = right                 # ref. al fill dret (no és BinTree!)
    # --------------------------------------------------

    def __init__(self, v=None, left=None, right=None):
        assert (v is None and left is None and right is None) or v is not None
        if v is None:
            self._root = None   # Empty tree
        else:
            l = left._root  if (left  is not None) else None    # <== ATENCIÓ!!!
            r = right._root if (right is not None) else None    # <== ATENCIÓ!!!
            self._root = self._Node(v, l, r)                    # <== ATENCIÓ!!!
```

Arbres Binaris





Arbres Binaris



Arbres Binaris

- Haurem d'anar molt en compte a l'hora d'implementar els mètodes de la classe `BinTree`. Caldrà fixar-se en què cal retornar en cada moment.

- Els *getters* hauran de retornar instàncies de `BinTree`, però `self._root._left` o `self._root._right` no ho són!

- Els *setters* pels fills hauran de tenir en compte que reben com a paràmetre instàncies de `BinTree`, però `self._root._left` o `self._root._right` no prenen per valor aquestes instàncies. Cal extraure'n els arbres "*interns*" formats per instàncies de `_Node`.

- Els recorreguts poden aprofitar la implementació que ja coneixem, utilitzant els *getters*, però això introdueix una ineficiència considerable. En implementacions com la que volem fer, és millor fer un mètode públic que fa servir una funció auxiliar que treballa sobre instàncies de `_Node`.

Arbres Binaris

Els *getters* hauran de retornar instàncies de **BinTree**, però **self._root._left** o **self._root._right** no ho són!

```python
def get_root(self):
    """
    Pre: It is assumed that the BinTree is
    NOT empty
    returns the value at the root of
    the BinTree
    """
    return self._root._element
```

```python
def get_left(self):
    """
    Pre: self is NOT an empty BinTree
    returns the left child of the BinTree
    """
    lft = BinTree()
    lft._root = self._root._left
    return lft

def get_right(self):
    """
    Pre: self is NOT an empty BinTree
    returns the right child of the BinTree
    """
    rft = BinTree()
    rft._root   = self._root._right
    return rft
```

Arbres Binaris

Els *setters* pels fills hauran de tenir en compte que reben com a paràmetre instàncies de **`BinTree`**, però **`self._root._left`** o **`self._root._right`** no prenen per valor aquestes instàncies. Cal extraure'n els arbres "*interns*" formats per instàncies de **`_Node`**.

```python
def set_root(self,v):
    """
    changes the value at the root
    of the BinTree
    """
    assert v is not None
    if not self.empty():
        self._root._element = v
    else:
        self._root = self._Node(v)
```

```python
def set_left(self,left):
    """
    Pre: left and self are non-empty BinTree's
    changes the left child of the BinTree
    """
    self._root._left = left._root
```

```python
def set_right(self,right):
    """
    Pre: right and self are non-empty BinTree's
    changes the right child of the BinTree
    """
    self._root._right = right._root
```

Arbres Binaris

Els recorreguts poden aprofitar la implementació que ja coneixem, utilitzant els *getters*, però això introdueix una ineficiència considerable. En implementacions com la que volem fer, és millor fer un mètode públic que fa servir una funció auxiliar que treballa sobre instàncies de `_Node`. Per exemple:

```
def preorder(self):
    """
    returns a list with the elements of the BinTree, ordered
    as specified in the definition of the pre-order traversal.
    """
    def _preorder(t):
        if t is None:
            return []
        else:
            return [t._element] + _preorder(t._left) + _preorder(t._right)

    if self.empty():
        return []
    else:
        return _preorder(self._root)
```

Arbres Binaris

- Els recorreguts `postorder` i `inorder` queden transformats de la mateixa manera que el `preorder`

- El recorregut `levelorder` no és recursiu, per tant només cal inicialitzar la cua amb l'arrel:

```
q = Cua()
q.enqueue(self._root)
```

  i anar "*encuant*" els fills dret i esquerre (que no siguin **`None`**) de les corresponents instàncies de `_Node` que "*desencuem*" en el bucle principal.

- El mètode `__repr__` fa servir els *getters*, així que queda intacte en aquesta nova implementació.

Exercici: Poda de subarbres

- Escriure un mètode *destructiu* `def poda_subarbre(self,x)` en la classe `BinTree` que, suposant que `self` té *tots els elements diferents*, i donat un element `x`, realitzi la següent acció: Si `x` es el valor d'algun node de `self`, la funció retorna `True` i elimina de `self` el node amb valor `x` i tots els seus descendents; altrament, el resultat es `False` i `self` no varia (és a dir, es queda igual).

```
def poda_subarbre(self,x):
    """
    Pre: self té tots els elements diferents
    Si x es el valor d'algun node de self, la funció retorna True i elimina de self
    el node amb valor x i tots els seus descendents; altrament, el resultat es False
    i self no varia (és a dir, es queda igual).
    """
```

Exercici: Poda de subarbres

```
def poda_subarbre(self,x):
    """
    Pre: self té tots els elements diferents
    Si x es el valor d'algun node de self, la funció retorna True i elimina de self
    el node amb valor x i tots els seus descendents; altrament, el resultat es False
    i self no varia (és a dir, es queda igual).
    """
    def poda_auxiliar(node,x):
        # Pre: node is not None => node._element != x
        # . . .

    if self._root is None:
        return False
    else:
        if self._root._element == x:
            self._root = None
            return True
        else:
            return poda_auxiliar(self._root,x)
```

Exercici: Poda de subarbres

```
def poda_auxiliar(node,x):
    # Pre: node is not None and node._element != x
    trobat = False
    if node._left is not None:
        if node._left._element == x:              # Si x és al fill esquerre...
            trobat = True
            node._left = None                     # ...l'eliminem*
        else:
            trobat = poda_auxiliar(node._left,x)  # en altre cas continuem buscant
    if not trobat and node._right is not None:    # Si cal continuar buscant...
        if node._right._element == x:             # ...mirem si x és al fill dret...
            trobat = True
            node._right = None                    # ...i l'eliminem* si hi és
        else:
            trobat = poda_auxiliar(node._right,x) # en altre cas continuem buscant
    return trobat
```

* Del subarbre que queda "abandonat" , el que era fill amb x a l'arrel, ja se'n fa càrrec el *garbage collector*

Exercici: Poda de subarbres

```
>>> from bintree_linked import *
>>> a = BinTree(1,BinTree(2,BinTree(5),BinTree(4,BinTree(7),BinTree(6))),BinTree(3))
>>> b = BinTree(1,BinTree(2,left=BinTree(5)),BinTree(3))
>>> c = BinTree(5,BinTree(2),BinTree(7,BinTree(6),BinTree(8)))
>>>
>>> a.poda_subarbre(4)
True
>>> a            # després de podar a, es queda igual que b
BinTree(1, left=BinTree(2, left=BinTree(5)), right=BinTree(3))
>>> b
BinTree(1, left=BinTree(2, left=BinTree(5)), right=BinTree(3))
>>>
>>> c.poda_subarbre(3)
False
>>> c            # després de podar c, es queda igual, ja que no hi ha 3
BinTree(5, left=BinTree(2), right=BinTree(7, left=BinTree(6), right=BinTree(8)))
>>>
```

Exercici: Poda de subarbres

- Si haguéssim implementat la classe **`BinTree`** amb una classe **`_Node`** que tingués una referència **`_parent`** al node pare (valdria **`None`** a l'arrel, és clar), la funció **`poda_auxiliar`** seria ***molt*** més senzilla d'implementar:

```
def poda_auxiliar(node,x):
    # Pre: node == self._root => node._element != x
    if node is None:
        return False
    if node._element == x:
        if node._parent._left == node:
            node._parent._left  = None
        else:
            node._parent._right = None
        return True
    else:
        trobat = poda_auxiliar(node._left,x)
        return trobat or poda_auxiliar(node._right,x)
```

- **Exercici**: Implementar la classe **`BinTree`** amb una classe **`_Node`** que tingui una referència **`_parent`** al node pare. Cal anar amb molt de compte amb el que retornen els *getters*!

Diccionaris i la seva Implementació
(taules de dispersió i BSTs)

Diccionaris

- Ja hem vist i utilitzat el tipus de dades **`Dictionary`** de Python. S'inicialitzava amb **`{}`** o amb **`dict()`**. Ara hi tornem, però perquè volem implementar els diccionaris nosaltres mateixos.
- Un diccionari (altrament conegut com a *array associatiu*) és un contenidor de parelles **`(clau,valor)`** amb les següents operacions:
  - constructor, per crear un diccionari buit
  - `assignar` / `inserir` / `afegir`: afegir un element **`(clau,valor)`** al diccionari. Si ja existia un element amb la mateixa **`clau`**, es sobreescriu el **`valor`**.
  - `esborrar` / `eliminar`: donada una **`clau`**, s'esborra l'element que té aquella clau. Si no hi ha cap element amb la **`clau`**, no es fa res.
  - `present`: donada una **`clau`**, es retorna un booleà que indica si el diccionari conté un element amb aquella **`clau`**.
  - `cerca`: donada una **`clau`**, retorna una *referència* a la parella **`(clau,valor)`** corresponent.
  - `consulta`: donada una **`clau`**, retorna una *referència* al **`valor`** associat a aquella **`clau`**.
  - `mida`: retorna el nombre de parelles **`(clau,valor)`** del diccionari.

Diccionaris

- Un diccionari es pot implementar de moltes maneres. A la *wikipedia* podem trobar una taula comparativa:

Underlying data structure	Lookup or Removal		Insertion		Ordered
	average	worst case	average	worst case	
Hash table	O(1) *	O(*n*)	O(1)	O(*n*)	No
Self-balancing binary search tree	O(log *n*)	O(log *n*)	O(log *n*)	O(log *n*)	Yes
unbalanced binary search tree	O(log *n*)	O(*n*)	O(log *n*)	O(*n*)	Yes
Sequential container of key–value pairs (e.g. association list)	O(*n*)	O(*n*)	O(1)	O(1)	No

- Nosaltres estudiarem les ***hash tables*** i els (***unbalanced***) ***binary search trees***
- Podem voler fer d'un diccionari un objecte *iterable*, però la conveniència d'això dependrà de la seva implementació. Eventualment, ja sabem com fer-ho (mètodes `__iter__` i `__next__`).

* g ∈ O(f) significa, a grans trets, que, *assimptòticament*, f és una *fita superior* per a g. Ho estudiarem a final de curs

Diccionaris: Taules de Dispersió (*hash tables*)

- Les ***taules de dispersió*** (*hash tables*) són estructures de dades que permeten una implementació eficient dels diccionaris. Python les fa servir per implementar els seus diccionaris (tot i que d'una manera més sofisticada que el que nosaltres veurem aquí).
- La primera proposta és de 1957 (!):

  W. W. Peterson. *Addressing for random access storage*. IBM Journal of Research and Development, 1(2), Abril 1957.

W. W. Peterson

Addressing for Random-Access Storage*

IBM JOURNAL • APRIL 1957

Abstract: Estimates are made of the amount of searching required for the exact location of a record in several types of storage systems, including the index-table method of addressing and the sorted-file method. Detailed data and formulas for access time are given for an "open" system which offers high flexibility and speed of access. Experimental results are given for actual record files.

Diccionaris: Taules de Dispersió (*hash tables*)

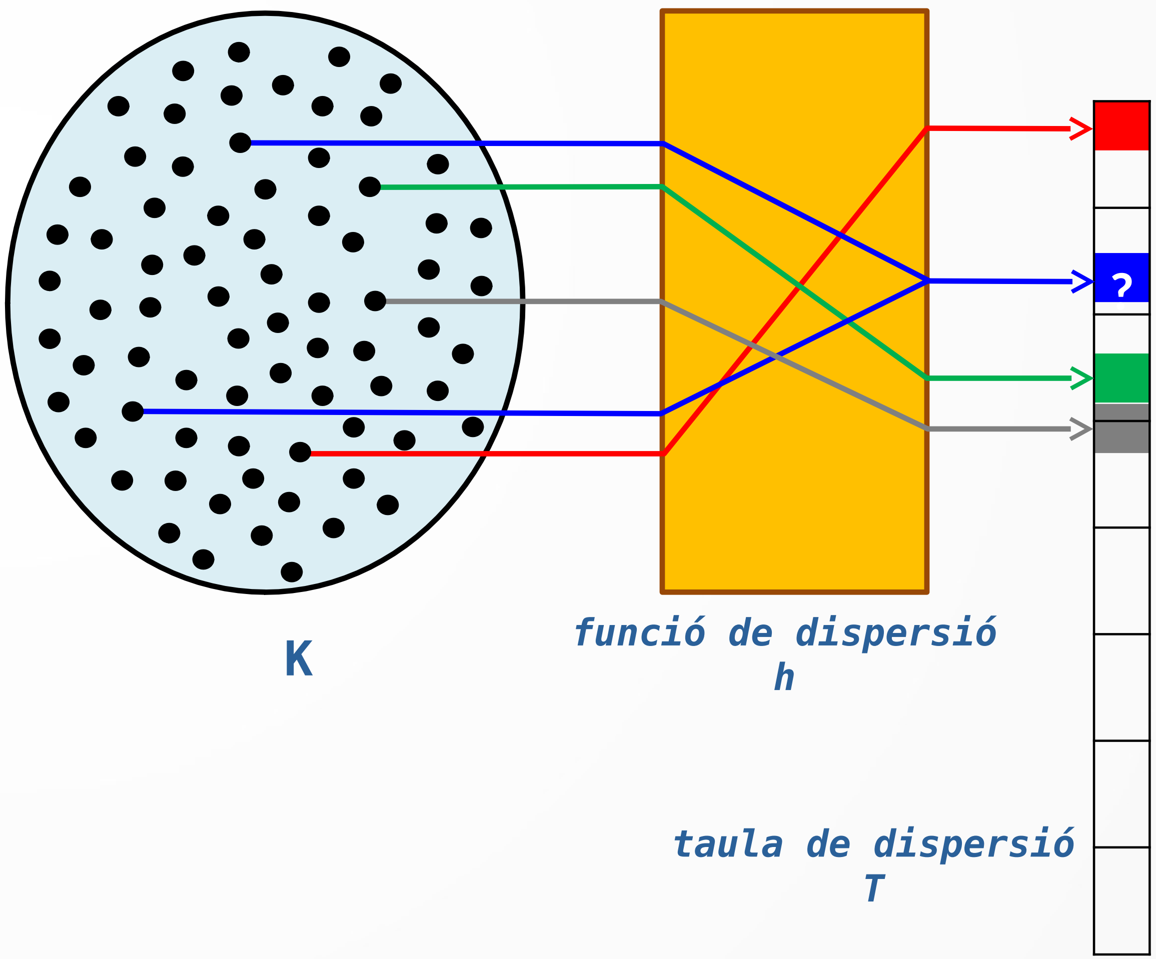
- Sigui `K` el conjunt de claus possibles, `|K| >> n`, on `n` seria el màxim nombre de parelles `(clau,valor)` que tenim *intenció* d'emmagatzemar. Reservem una `taula T` (en Python farem servir una llista convencional) de mida `m ≥ n`.

Aleshores, un element `(clau,valor)` s'emmagatzemarà a la posició `h(clau)` de la ***taula de dispersió*** `T`, on

`h: K → {0,1,...,m-1}`.

La funció `h` és el que coneixem com a ***funció de dispersió*** (*hash function*) .

El problema és que com `|K| >> m`, és segur que hi haurà ***col·lisions***, és a dir, claus *diferents* `k1` i `k2` tal que `h(k1)==h(k2)`



- Així doncs, ***com resolem les col·lisions?***

Diccionaris: Taules de Dispersió (*hash tables*)

- Les col·lisions són quelcom *a evitar*, quelcom que hauria de passar amb poca freqüència. Són, com hem vist, inevitables per una qüestió de cardinalitat.
- Que les col·lisions passin rares vegades depén essencialment de la funció de dispersió:
  - Ha de ser una funció que distribueixi les claus *aparentment* **a l'atzar** i de manera **uniforme**.
  - La funció, però, ha de ser **consistent**, és a dir, entrades iguals han de generar la mateixa sortida: $k_1 == k_2 => h(k_1) == h(k_2)$.
- Un exemple de funció de dispersió per a *strings*, sobre una taula de mida `m`, seria:

$$h(x) = \left(\sum_{i=0}^{n-1} x_i \cdot p^i\right) mod\, m$$

on `p` és un nombre primer, `n` és la mida de l'*string* `x` i `x`$_i$ és (el codi numèric de) l'i-éssim caràcter de l'*string*.

Diccionaris: Taules de Dispersió (*hash tables*)

- Veiem un altre exemple de funció de dispersió, el *mètode de la multiplicació*:
  - Sigui una clau *numèrica* `k` (si les claus no són números, cal transformar-les)
  - Multiplicar la clau per una constant `A`, on `0 < A < 1` i ens quedem amb la part fraccional: $k \cdot A - \lfloor k \cdot A \rfloor$
  - Multipliquem per `m` (mida de la taula), i ens quedem la part entera, per sota:

$$h(k) = \lfloor m \cdot (k \cdot A - \lfloor k \cdot A \rfloor) \rfloor$$

- Una bona tria per a `m` seria una potencia de 2. Una bona tria per a `A` (segons Knuth) és: $A \approx (\sqrt{5}-1)/2 = 0.6180339887...$
- Aquest mètode té l'avantatge que el valor de `m` no és crític (en altres mètodes sí que ho és). Té el desavantatge que és més lent de calcular que d'altres mètodes.

- Nosaltres sempre podem fer servir la funció `hash` que Python ens proporciona! Si la clau és un objecte *hashable* `ob`, i la taula té mida `m`, tenim `h(ob) = hash(ob) % m`

Diccionaris: Taules de Dispersió (*hash tables*)

- Hi ha diverses maneres de gestionar les col·lisions:
  - ***Encadenament separat*** (*separate chaining*): Els elements que tenen claus amb el mateix valor de la funció de dispersió s'emmagatzemen en forma de *llista*, a la posició determinada per la funció de dispersió.

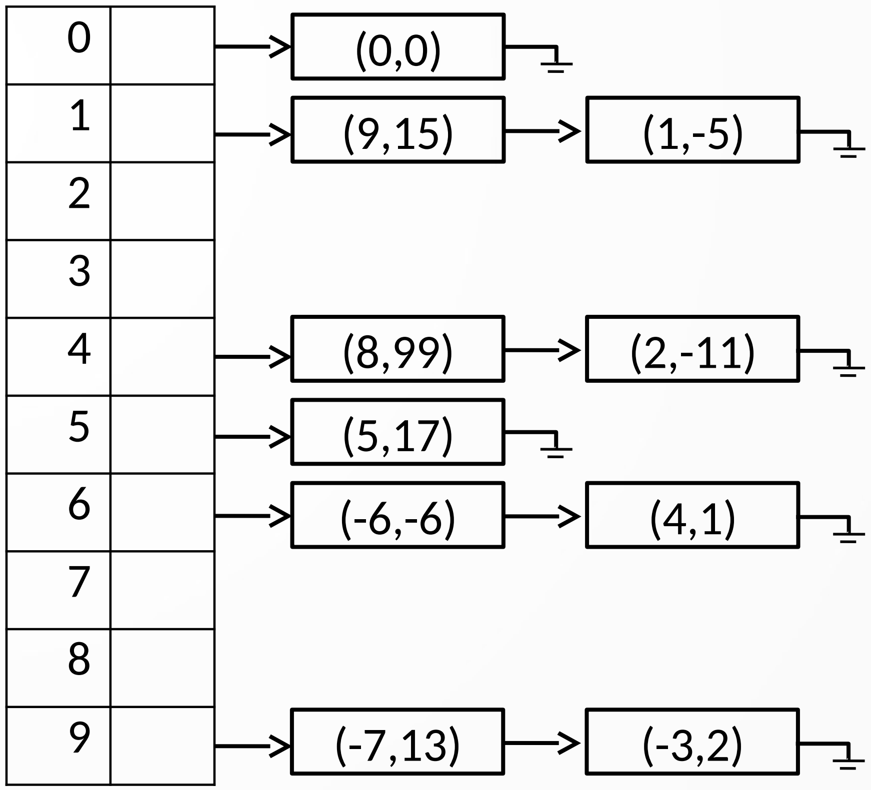
    Per exemple:

    Suposem que volem emmagatzemar les parelles `(clau,valor)` següents:

    `(0,0)`,`(9,15)`,`(1,-5)`,`(8,99)`,`(2,-11)`,
    `(5,17)`,`(-6,-6)`,`(4,1)`,`(-7,13)`,`(-3,2)`

    amb funció de dispersió:
    $\texttt{h(clau) = clau}^2\ \texttt{mod 10}$

    en una taula de mida 10



Diccionaris: Taules de Dispersió (*hash tables*)

- Hi ha diverses maneres de gestionar les col·lisions:
  - ***Encadenament separat*** (*separate chaining*): Els elements que tenen claus amb el mateix valor de la funció de dispersió s'emmagatzemen en forma de *llista*, a la posició determinada per la funció de dispersió.

- Quin és el cas pitjor en les operacions de cerca o d'esborrat? El cas pitjor seria tenir mala sort i que totes les **n** claus dels elements inserits tinguin el mateix valor en la funció de dispersió, és a dir, *que tot fossin col·lisions*. En aquest cas, la cerca o l'esborrat (que requereix una cerca prèvia) tenen una complexitat $\Theta(n)$ en cas pitjor, és a dir, lineal en el nombre d'elements emmagatzemats a la taula.

- I així, per quina raó pensem que les taules de dispersió amb encadenament separat són *eficients*?!?!

Diccionaris: Taules de Dispersió (*hash tables*)

- Hi ha diverses maneres de gestionar les col·lisions:
  - ***Encadenament separat*** (*separate chaining*): Els elements que tenen claus amb el mateix valor de la funció de dispersió s'emmagatzemen en forma de *llista*, a la posició determinada per la funció de dispersió.

- Sigui **`α = n/m`** (**`n`** = nombre d'elements, **`m`** = mida de la taula) el ***factor de càrrega*** (*load factor*). És una mesura de la mida mitjana de les llistes associades a cada posició de la taula.
- Aleshores, es pot demostrar que el cost *en cas mitjà* de fer una cerca o un esborrat és **`Θ(1+α)`**, és a dir, *quasi*-constant. Tractaríem, doncs, de mantenir **`α`** *petita*.
- Si mai ens trobem amb **`α > 1`** cal fer *rehashing*: Augmentar la mida de la taula, i tornar a situar els elements. Si triem la mida de la taula amb prou habilitat (respecte al nombre d'elements que hi volem emmagatzemar), aquesta operació serà poc freqüent. Té un cost **`Θ(n)`**.

Diccionaris: Taules de Dispersió (*hash tables*)

- Hi ha diverses maneres de gestionar les col·lisions:
  - ***Adreçament obert*** (*open addressing*): Els elements que tenen claus amb el mateix valor de la funció de dispersió s'emmagatzemen *a la mateixa taula*.
- Tenim diverses estratègies dins l'adreçament obert:
  - *Linear probing*: Si per a una clau `k` la posició `h(k)` està ocupada, mirem `h(k)+1`, `h(k)+2`, etc. Situem la parella `(clau,valor)` a la primera posició buida que trobem.

    Assignar i buscar elements donada una clau no presenta problema. Eliminar una parella `(clau,valor)` sí que és problemàtic, ja que les caselles buides també són marcadors de "*fi de cerca*". Usualment cal buscar més endavant una altra parella `(clau,valor)` (amb la mateixa `h(clau)`) per situar al lloc de la parella que volem eliminar. I així successivament fins no trobar-ne més.

    Aquí també té sentit definir un factor de càrrega `α = n/m`. Típicament es dobla la mida de la taula (*rehashing* ⇨ taula mida `2m`) quan `α > 1/2`.

Diccionaris: Taules de Dispersió (*hash tables*)

- Hi ha diverses maneres de gestionar les col·lisions:
  - ***Adreçament obert*** (*open addressing*): Els elements que tenen claus amb el mateix valor de la funció de dispersió s'emmagatzemen *a la mateixa taula*.
- Tenim diverses estratègies dins l'adreçament obert:
  - *Double Hashing*: En aquest cas calen *dues* funcions de dispersió, h_1 i h_2. Definim la funció $h(i,k) = (h_1(k) + i \cdot h_2(k)) \bmod m$. Si per a una clau `k` la posició `h(0,k)` està ocupada, mirem `h(1,k)`, `h(2,k)`, etc.

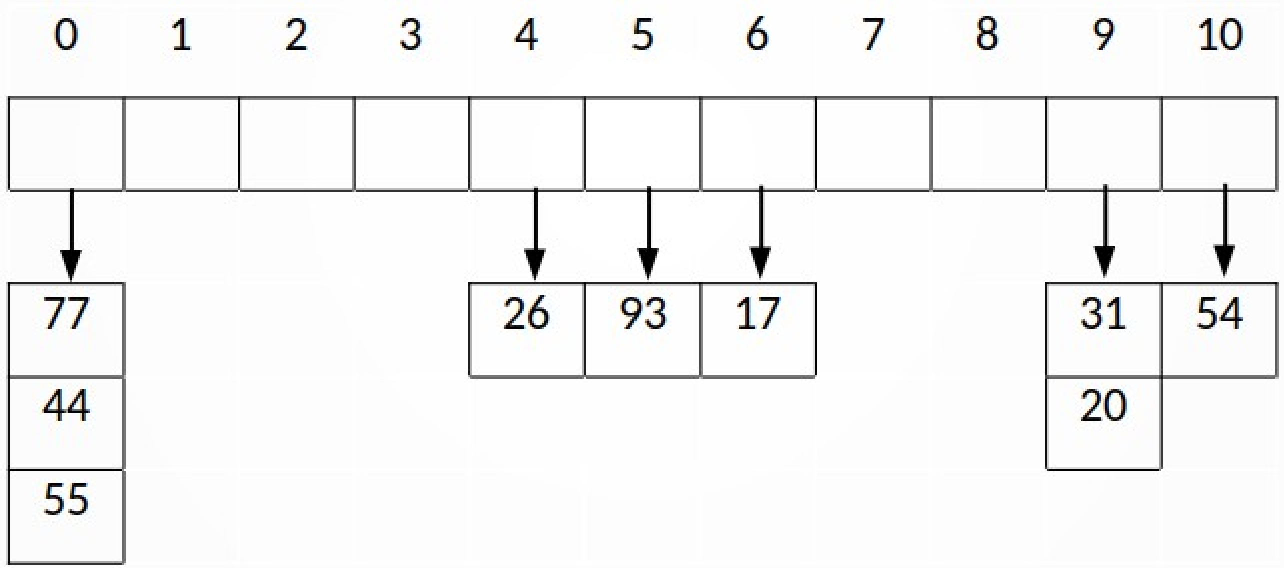
    El rang d'h_2 ha de ser `{1,2,...,m-1}`.

Exercici: Taules de Dispersió (*hash tables*)

- Inserir els elements amb claus 54, 26, 93, 17, 77, 31, 44, 55, 20 (només ens fixem en les claus, ignorem els valors associats en aquest exercici) en una taula de dispersió de mida `m = 11` i amb funció de dispersió: $h(x) = x \bmod 11$
- Feu servir una taula de dispersió amb *encadenament separat* per resoldre col·lisions:

Exercici: Taules de Dispersió (*hash tables*)

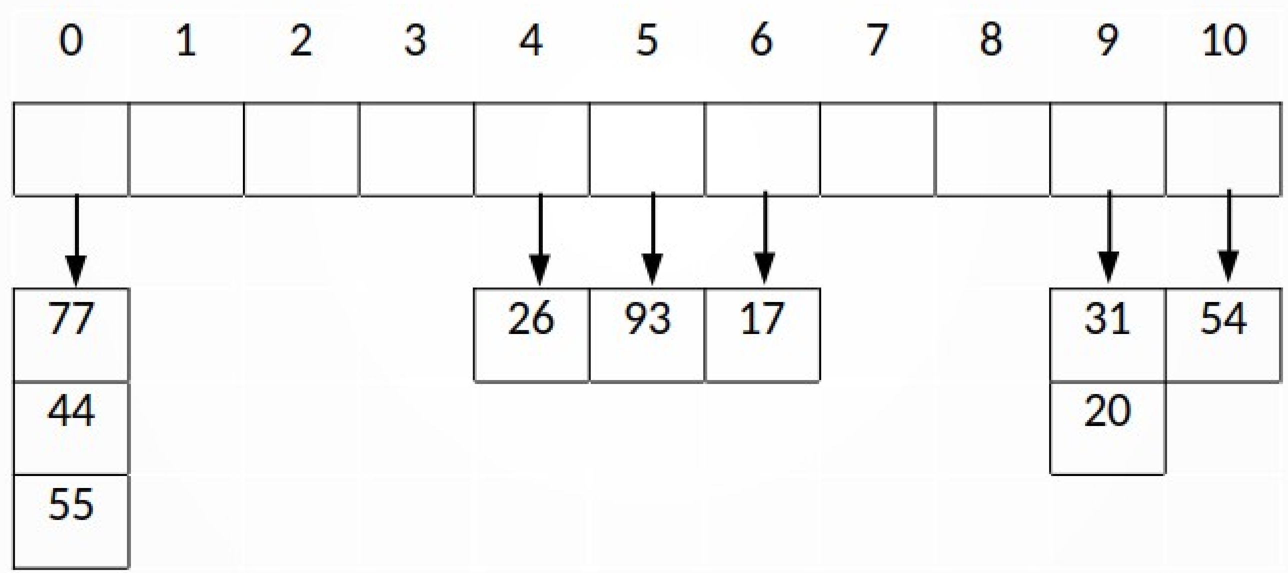
- Inserir els elements amb claus 54, 26, 93, 17, 77, 31, 44, 55, 20 (només ens fixem en les claus, ignorem els valors associats en aquest exercici) en una taula de dispersió de mida `m = 11` i amb funció de dispersió: $h(x) = x \bmod 11$
- Feu servir una taula de dispersió amb *encadenament separat* per resoldre col·lisions:



- Feu servir una taula de dispersió amb adreçament obert (*linear probing*):

Exercici: Taules de Dispersió (*hash tables*)

- Inserir els elements amb claus 54, 26, 93, 17, 77, 31, 44, 55, 20 (només ens fixem en les claus, ignorem els valors associats en aquest exercici) en una taula de dispersió de mida `m = 11` i amb funció de dispersió: h(x) = x mod 11
- Feu servir una taula de dispersió amb *encadenament separat* per resoldre col·lisions:



- Feu servir una taula de dispersió amb adreçament obert (*linear probing*):

0	1	2	3	4	5	6	7	8	9	10
77	44	55	20	26	93	17			31	54

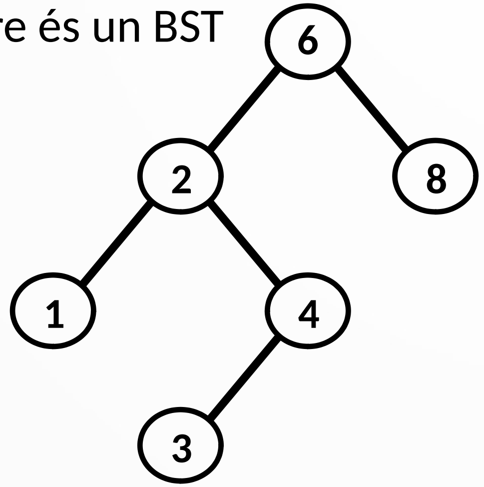
Diccionaris: Arbres Binaris de Cerca (*Binary Search Trees*)

- Els arbres binaris de cerca (*Binary Search Trees*, BST) són un tipus d'arbre binari que podem utilitzar per implementar diccionaris i conjunts (*sets*).
- Un **BST** és un **arbre binari** que verifica la...

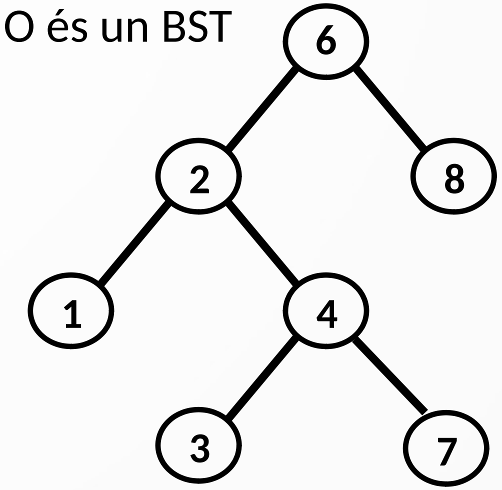
  Propietat BST: Una arbre binari A té la propietat BST si, *per a tot node de l'arbre A* (anomenem V al corresponent element del node), tots els nodes del fill esquerre són més petits que V, i tots els nodes del fill dret són més grans que V.

  ⇨ per tant, els elements dels nodes d'un BST han de ser *comparables*.

Aquest arbre és un BST

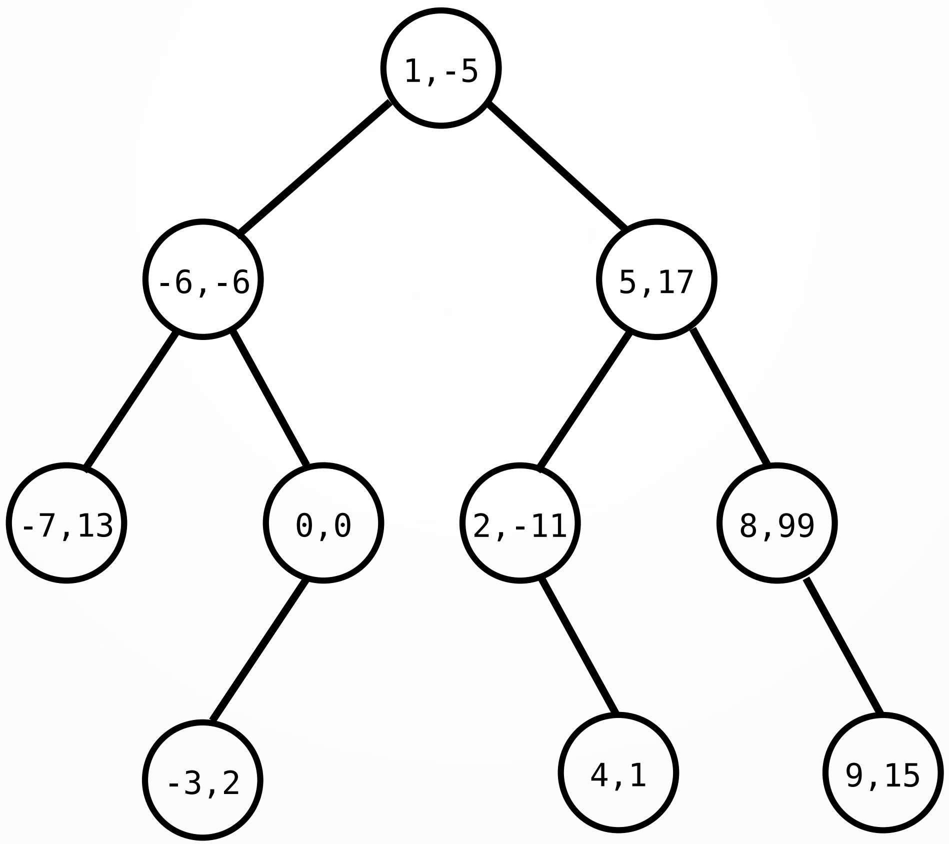


Aquest arbre NO és un BST



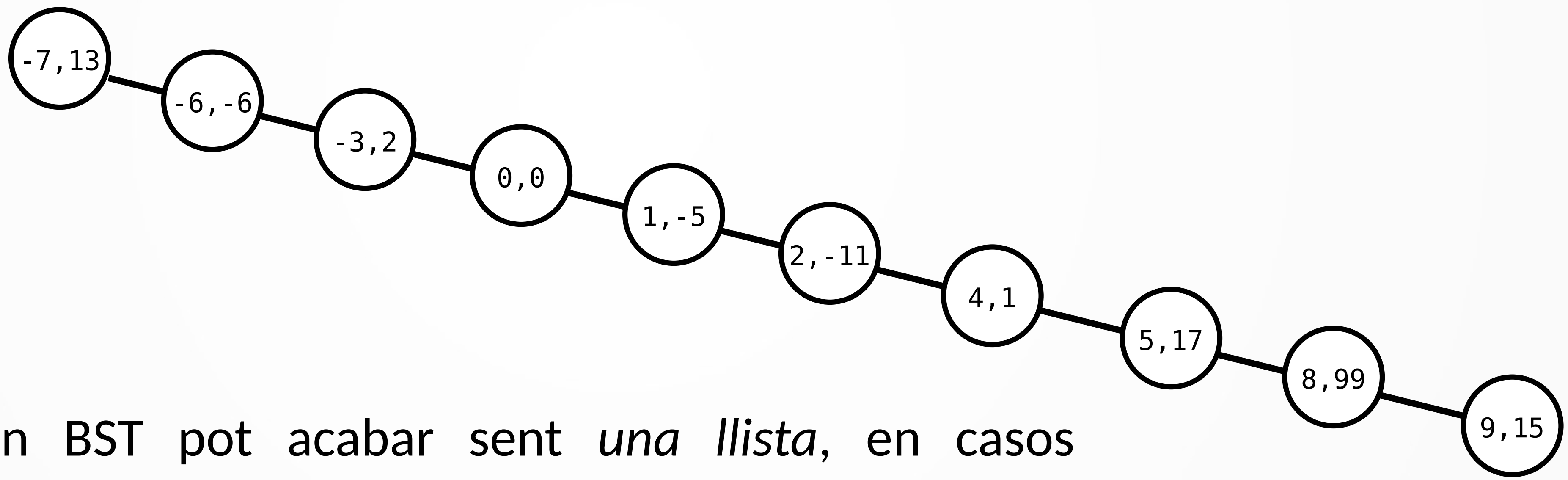
Diccionaris: Arbres Binaris de Cerca (*Binary Search Trees*)

- Veiem un exemple: Suposem que volem emmagatzemar en un BST les parelles `(clau,valor)` següents. Comparem les *claus* per construir l'arbre: `(0,0)`,`(9,15)`,`(1,-5)`,`(8,99)`,`(2,-11)`,`(5,17)`,`(-6,-6)`,`(4,1)`,`(-7,13)`,`(-3,2)`
- Un possible BST seria:
  (*no és l'únic*!)



Diccionaris: Arbres Binaris de Cerca (*Binary Search Trees*)

- Veiem un exemple: Suposem que volem emmagatzemar en un BST les parelles `(clau,valor)` següents. Comparem les *claus* per construir l'arbre: `(0,0)`,`(9,15)`,`(1,-5)`,`(8,99)`,`(2,-11)`,`(5,17)`,`(-6,-6)`,`(4,1)`,`(-7,13)`,`(-3,2)`
- Això també és un BST!!:



- Així doncs, un BST pot acabar sent *una llista*, en casos especials. Això té conseqüències de cara a la complexitat *en cas pitjor* de les operacions essencials del diccionari. Posposarem la discussió fins haver vist una implementació d'aquestes operacions.

Diccionaris: Arbres Binaris de Cerca (*Binary Search Trees*)

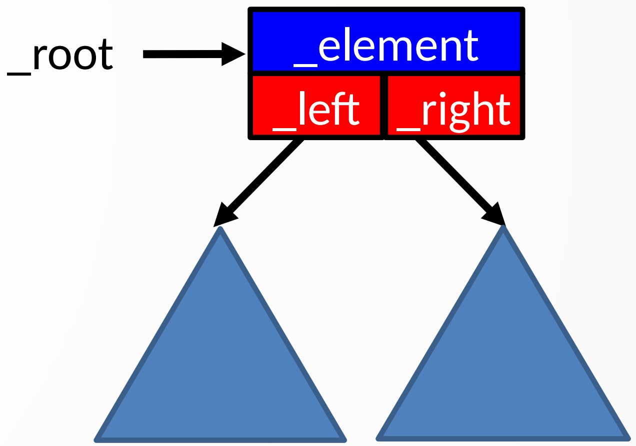
- La implementació d'un diccionari implementat amb un BST és molt similar a la implementació que ja vam veure dels arbres binaris. No en fem subclasse ja que no volem pràcticament cap de les operacions que heretaríem de l'arbre binari.

```
class Diccionari:
    # ----------------------------------------------------
    # Cada element de l'arbre serà una instància de _Node
     class _Node:
        __slots__ = '_element','_left','_right' # opcional, per eficiència

        def __init__(self, element, left=None, right=None):
            self._element = element             # ref. a l'element emmagatzemat
            self._left  = left                  # ref. al fill esquerra
            self._right = right                 # ref. al fill dret
    # ----------------------------------------------------

    def __init__(self):
        self._root = None
        self._n    = 0
```

Diccionaris: Arbres Binaris de Cerca (*Binary Search Trees*)

- La implementació d'un diccionari implementat amb un BST és molt similar a la implementació que ja vam veure dels arbres binaris. No en fem subclasse ja que no volem pràcticament cap de les operacions que heretaríem de l'arbre binari.

```python
class Diccionari:
    # ---------------------------------------------------
    # Cada element de l'arbre serà una instància de _Node
     class _Node:
        __slots__ = '_element','_left','_right'

        def __init__(self, element, left=None, right=None):
            self._element = element
            self._left  = left
            self._right = right
    # ---------------------------------------------------

    def __init__(self):
        self._root = None
        self._n    = 0
```



Les operacions sobre un BST han de ***<u>preservar</u>*** la propietat BST!

Diccionaris: Arbres Binaris de Cerca (*Binary Search Trees*)

- Tal com vam fer amb les taules de dispersió, les parelles **`(clau,valor)`** seran **`namedtuples`**: `Element = namedtuple("Element", ["clau", "valor"])`
- Aleshores, l'operació d'`assignar` / `inserir` / `afegir` una parella **`(clau,valor)`**:

```
def assigna(self, clau, valor):
    """
    Assigna informació a una clau. Si la clau ja hi és dins
    el diccionari, la informació és modificada.
    cas pitjor: Theta(n)
    """
    self._root = self._assigna(Element(clau,valor), self._root)

def _assigna(self, x, t):
    # Pre: x és un Element
    if t is None:
        self._n += 1
        return self._Node(x)
    elif x.clau < t._element.clau:
        t._left = self._assigna(x, t._left)
    elif x.clau > t._element.clau:
        t._right = self._assigna(x, t._right)
    else:  # x.clau == t._element.clau
        t._element = t._element._replace(valor=x.valor)
    return t
```

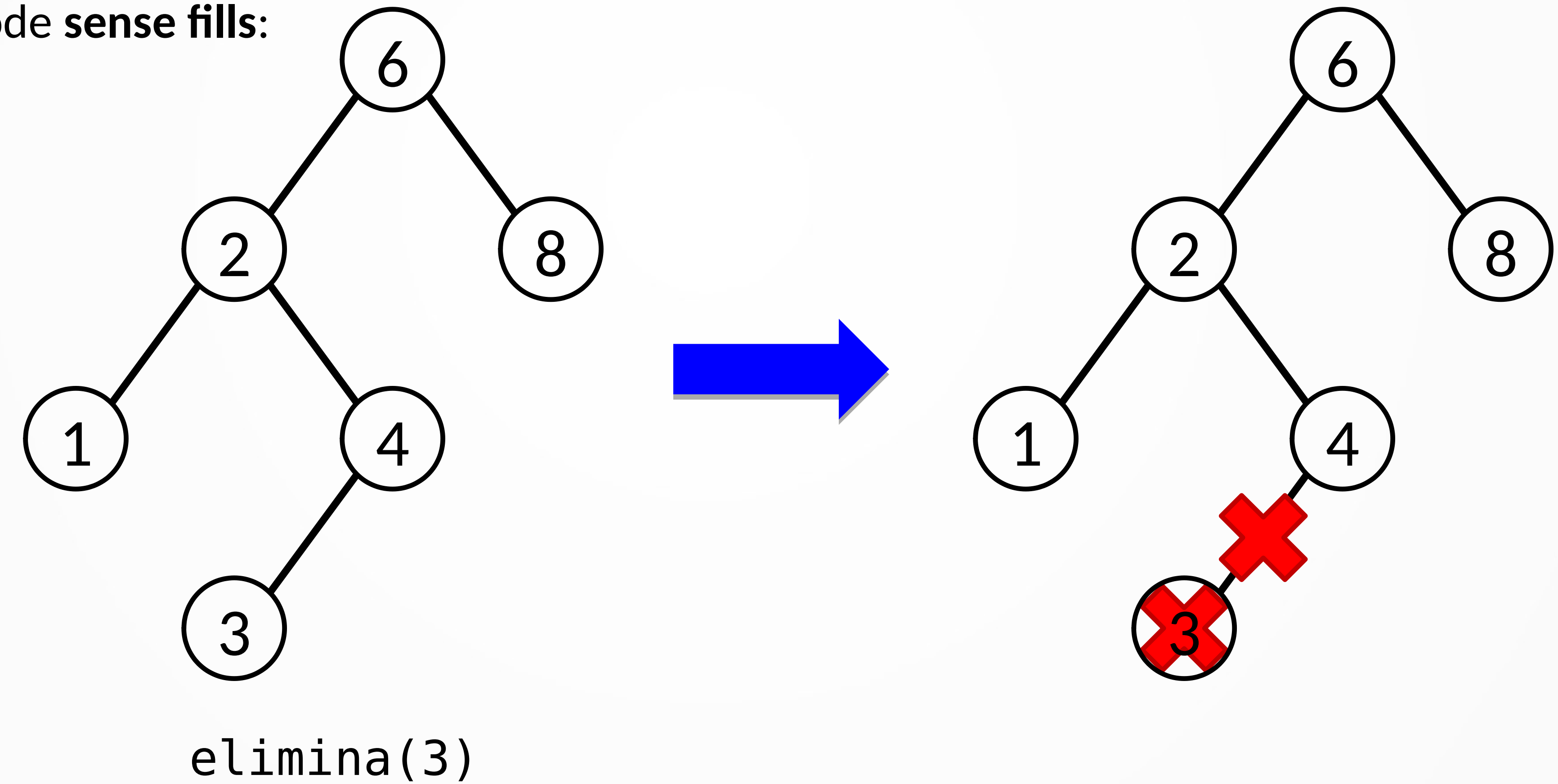
$\left.\right\}$ públic

$\left.\right\}$ privat

Diccionaris: Arbres Binaris de Cerca (*Binary Search Trees*)

- És obvi que, per construcció, l'operació `assigna` preserva la propietat BST. En el cas de l'eliminació d'una parella **`(clau,valor)`** la qüestió és una mica més complicada. Farem una anàlisi per casos (ho il·lustrarem només amb les claus, ignorem els valors):

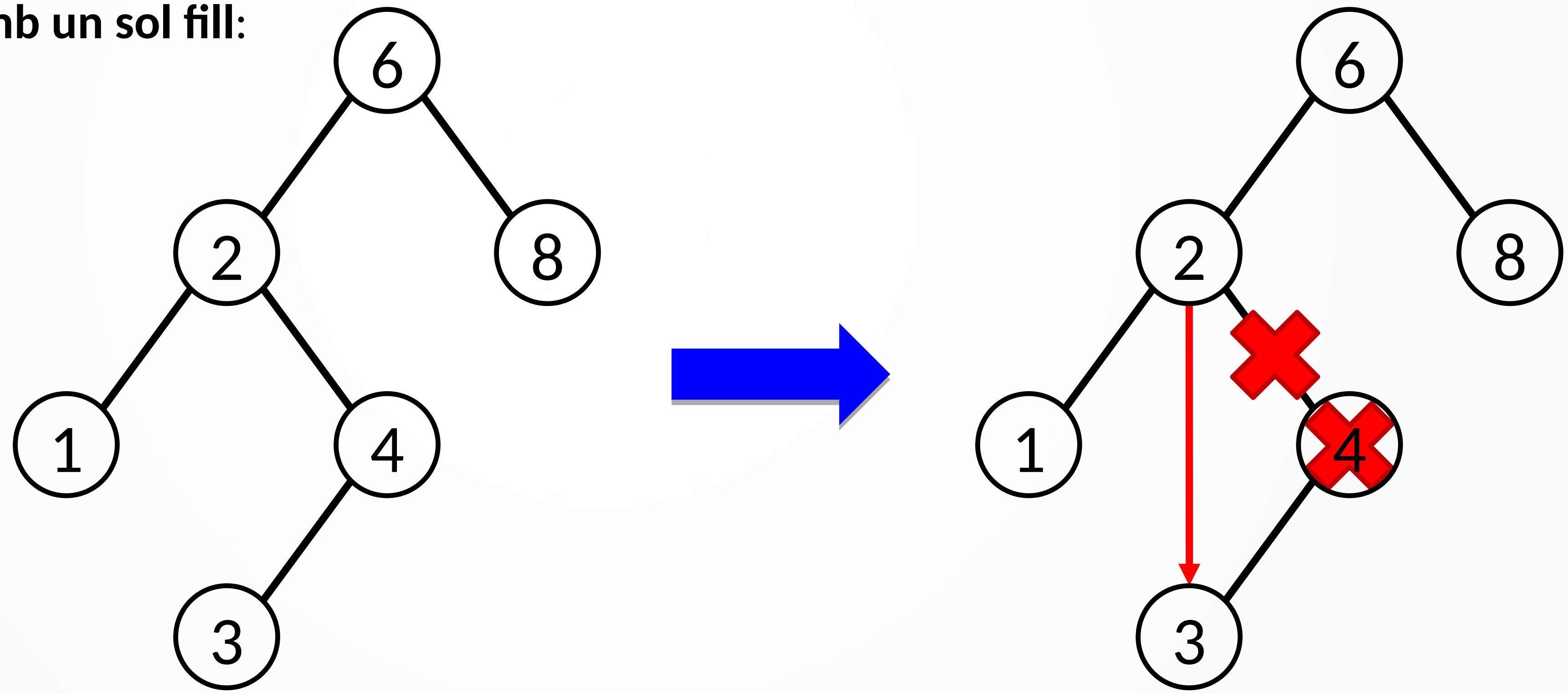
Eliminar un node **sense fills**:



Diccionaris: Arbres Binaris de Cerca (*Binary Search Trees*)

- És obvi que, per construcció, l'operació `assigna` preserva la propietat BST. En el cas de l'eliminació d'una parella **`(clau,valor)`** la qüestió és una mica més complicada. Farem una anàlisi per casos (ho il·lustrarem només amb les claus, ignorem els valors):

Eliminar un node **amb un sol fill**:



`elimina(4)`

Diccionaris: Arbres Binaris de Cerca (*Binary Search Trees*)

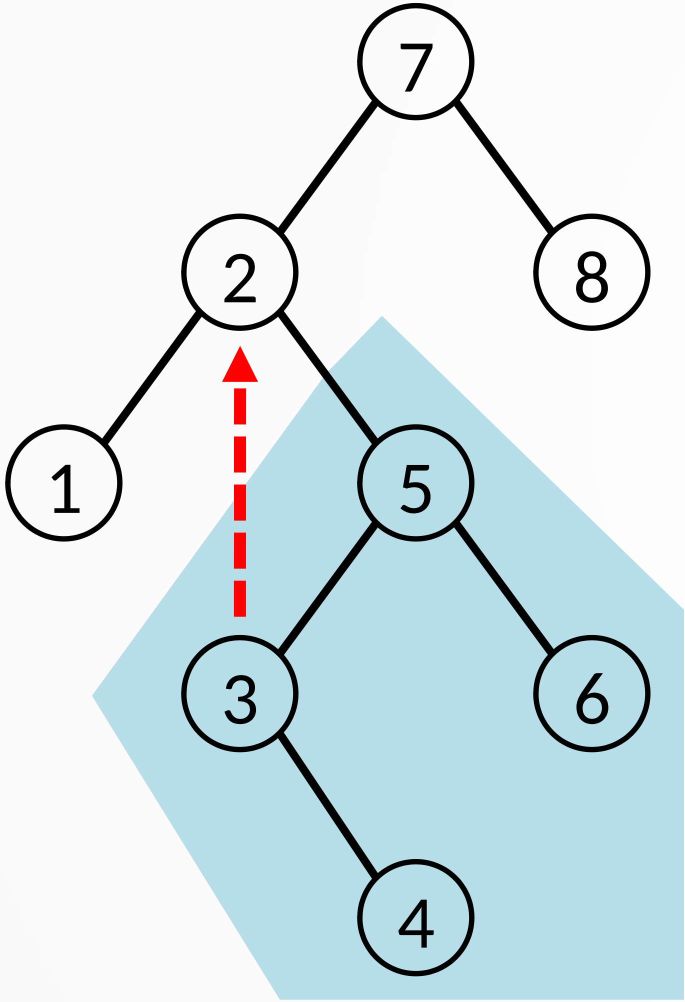
- Aquests dos casos que hem vist són els casos fàcils. El codi corresponent seria:

```
def elimina(self, clau):
    """
    Elimina la parella (clau, valor) del diccionari.
    Si la clau no pertany al diccionari, res canvia.
    cas pitjor: Theta(n).
    """
    self._root = self._elimina(clau, self._root)
```

```
def _elimina(self, clau, t):
    if t != None:
        if clau < t._element.clau:
            t._left = self._elimina(clau, t._left)
        elif clau > t._element.clau:
            t._right = self._elimina(clau, t._right)
        else:  # clau == t._element.clau
            # t té 0 o 1 fill
            if t._left == None:
                self._n -= 1
                return t._right
            elif t._right == None:
                self._n -= 1
                return t._left
            else: # t té dos fills
                # ...
        return t
```

Diccionaris: Arbres Binaris de Cerca (*Binary Search Trees*)

- Quan el node a eliminar té dos fills:



`elimina(2)`

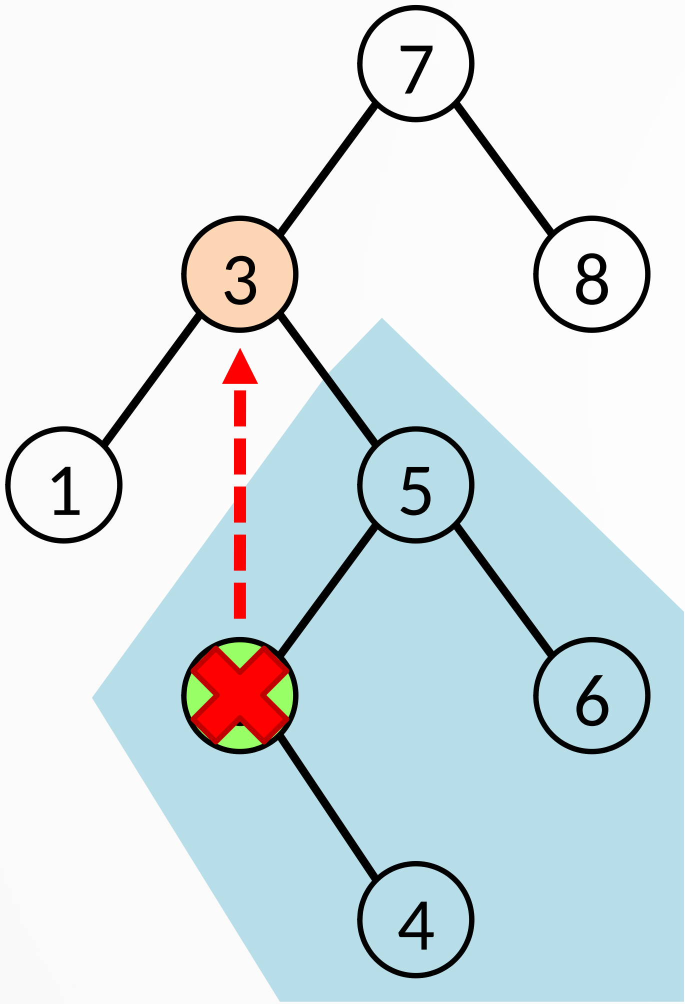
1. Trobar l'element `(clau,valor)` a partir de la clau

2. Trobar l'element mínim del fill dret

3. Copiar aquest element mínim al node del que volem eliminar l'element

Diccionaris: Arbres Binaris de Cerca (*Binary Search Trees*)

- Quan el node a eliminar té dos fills:

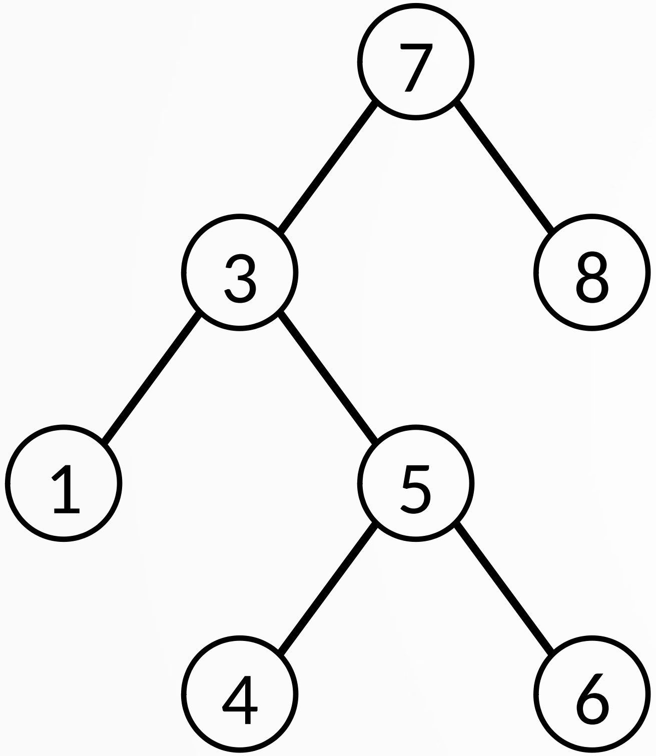


`elimina(2)`

1. Trobar l'element `(clau,valor)` a partir de la clau
2. Trobar l'element mínim del fill dret
3. Copiar aquest element mínim al node del que volem eliminar l'element
4. Elimina el valor mínim del fill dret

Diccionaris: Arbres Binaris de Cerca (*Binary Search Trees*)

- Quan el node a eliminar té dos fills:



1. Trobar l'element `(clau,valor)` a partir de la clau
2. Trobar l'element mínim del fill dret
3. Copiar aquest element mínim al node del que volem eliminar l'element
4. Elimina el valor mínim del fill dret (en l'exemple és un cas simple)

`elimina(2)`

Diccionaris: Arbres Binaris de Cerca (*Binary Search Trees*)

- La versió completa seria:

```python
def _elimina(self, clau, t):
    if t != None:
        if clau < t._element.clau:
            t._left = self._elimina(clau, t._left)
        elif clau > t._element.clau:
            t._right = self._elimina(clau, t._right)
        else:  # clau == t._element.clau
            # t té 0 o 1 fill
            if t._left == None:
                self._n -= 1
                return t._right
            elif t._right == None:
                self._n -= 1
                return t._left
            else: # t té dos fills
                 m = self._troba_minim(t._right)._element
                 t._element = m
                 t._right = self._elimina(m.clau, t._right)
        return t
```

```python
def _troba_minim(self, t):
    # Pre: t no és None
    if t._left == None:
        return t
    return self._troba_minim(t._left)
```

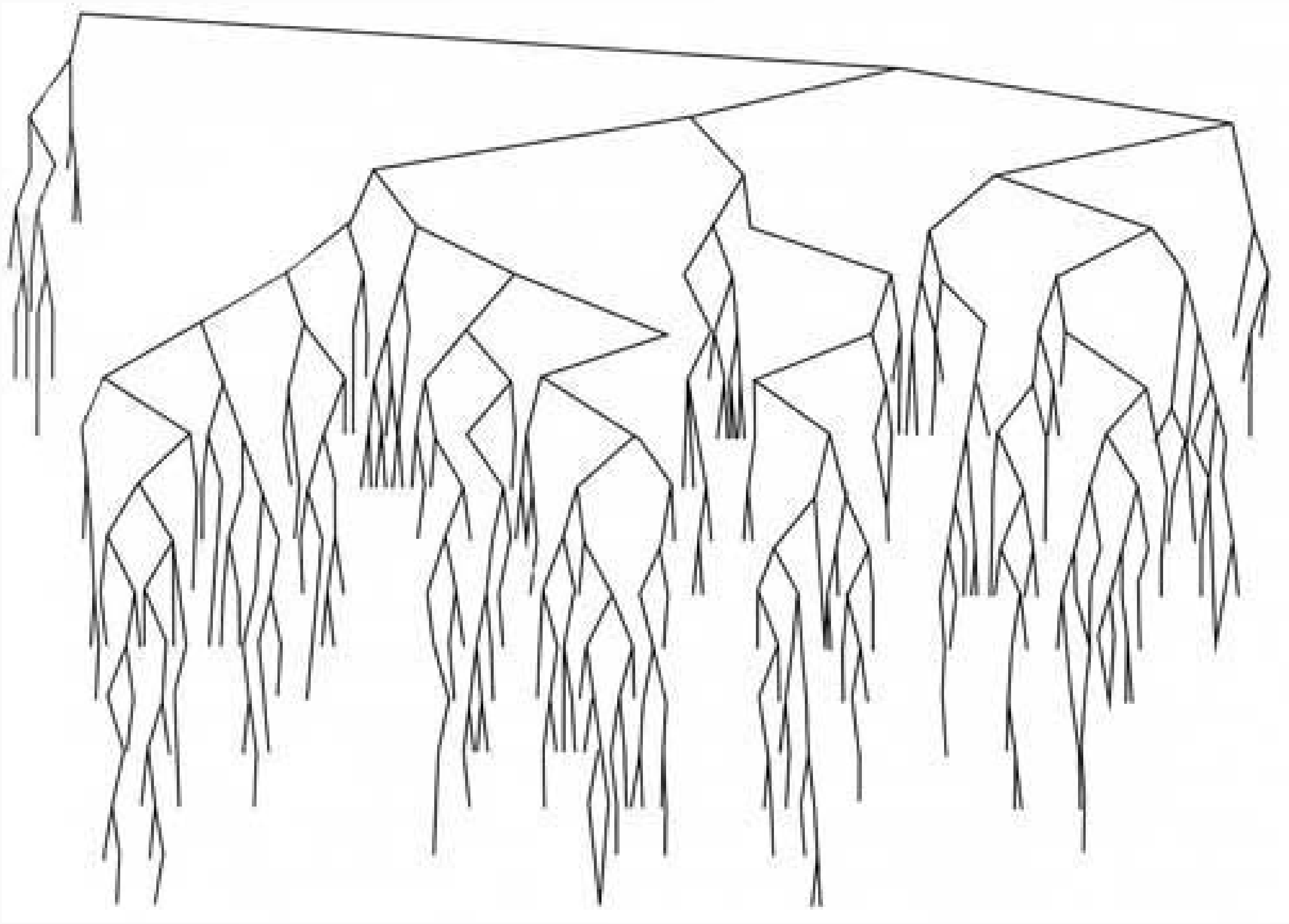
Diccionaris: Arbres Binaris de Cerca (*Binary Search Trees*)

- Finalment, per acabar la implementació:

```python
def mida(self):
    return self._n

def valor(self, clau):
    """
    retorna el valor associat a una clau, None si la clau no hi és
    cas pitjor: Theta(n)
    """
    return self._valor(clau, self._root)

def _valor(self, clau, t):
    if t is None:
        return None
    elif clau < t._element.clau:
        return self._valor(clau, t._left)
    elif clau > t._element.clau:
        return self._valor(clau, t._right)
    else:  # clau == t._element.clau
        return t._element.valor
```

Diccionaris: Arbres Binaris de Cerca (*Binary Search Trees*)

- Examinant amb atenció les operacions és fàcil adonar-se que el cost d'aquestes operacions, en la nostra implementació, és `Θ(d)`, on `d` és la profunditat del node sobre el que fem l'operació. La pregunta important, naturalment, és, ***com de gran és*** `d`?



BST amb 500 nodes generat a l'atzar (claus aleatòries). Aquest arbre és el resultat de realitzar l'operació `assigna` 500 vegades.

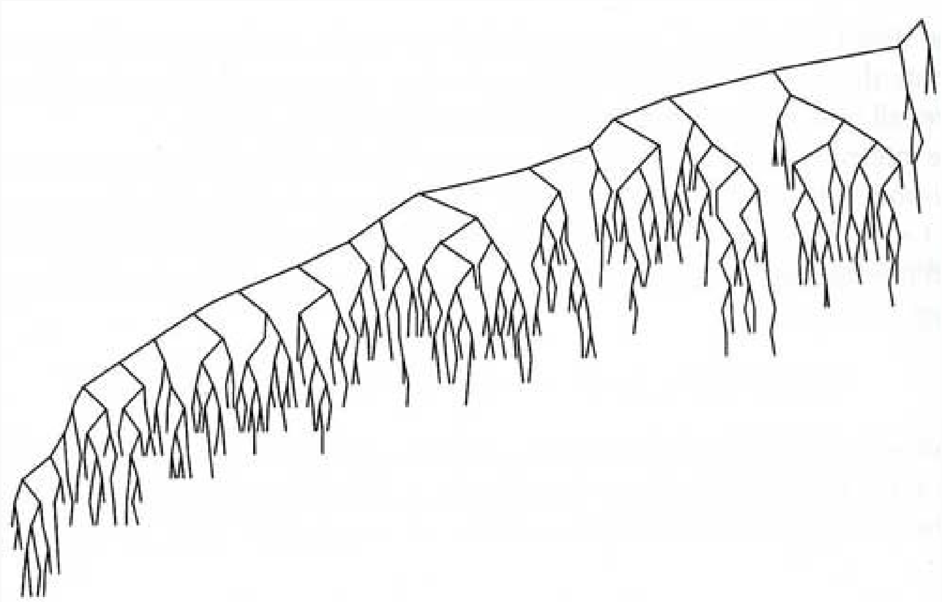
plana 143 de Weiss, M.A., *Data Structures and Algorithm Analysis in C++* (4th ed.), Addison-Wesley 2014

Diccionaris: Arbres Binaris de Cerca (*Binary Search Trees*)

- Es pot demostrar que, després de `n` insercions (sense eliminar mai cap node), la profunditat *mitjana* d'un node és `Θ(log n)`. Per tant la complexitat mitjana de les operacions del diccionari per a un BST amb `n` elements seria `Θ(log n)`... correcte?

  ⇨ Doncs **NO**.

- El problema és que l'eliminació d'elements està esbiaixada, ja que sempre eliminem elements del fill dret del node a eliminar. Això provoca que l'arbre vagi "*degenerant*" i no pugui romandre "*equilibrat*" (en un sentit intuïtiu, no hem definit "*arbre equilibrat*").



El BST de la transparència anterior, després de fer 250.000 operacions `assigna`/`elimina` (sempre amb claus aleatòries). S'observa que, en un BST amb `n` elements i després de `Θ(n²)` operacions `assigna`/`elimina`, la profunditat d'un node és `Θ(√n)`

plana 143 de Weiss, M.A., *Data Structures and Algorithm Analysis in C++* (4th ed.), Addison-Wesley 2014

Diccionaris: Arbres Binaris de Cerca (*Binary Search Trees*)

- En definitiva, una col·lecció d'`n` elements (comparables) pot acabar en un BST de profunditat `n`, essent per tant, a efectes pràctics, una llista. Així doncs, la <u>*complexitat en cas pitjor*</u> de les operacions principals d'un diccionari (implementat amb BST): $\Theta(n)$, és a dir, lineal en el nombre d'elements del BST.

  La <u>*complexitat mitjana*</u> és difícil de precisar (podem fer variants de l'eliminació d'elements per mirar de suprimir el biaix), però **no** està garantit el tan desitjat cost logarítmic.

- S'han proposat diverses variants dels BST per aconseguir el que s'anomena ***BST equilibrats*** (*self-balanced BSTs*). Tant l'assignació com l'eliminació d'elements estan definits de manera que procuren, no només preservar la propietat BST, sinò a més preservar l'equilibri del BST.

  Alguns exemples són els *AVL trees* (Adelson-Velsky and Landis, 1962), els *Red-Black trees* (Bayer, 1972), els *2-3 trees* (Hopcroft, 1970) o els *B-trees* (Bayer and McCreight, 1970).

  ⇨ En BST *equilibrats* la <u>*complexitat en cas pitjor*</u> de les operacions d'un diccionari és $\Theta(\log n)$.

Exercici: La bossa de les paraules (*jutge P84415*)

- Tot i això, si ens demanen el valor de la clau més gran o la clau més petita, fins i tot un BST pot ser més eficient que una taula de dipersió (si l'arbre no *degenera* massa). Veiem-ne un exemple:
- Teniu una bossa inicialment buida, on hi podeu guardar paraules, i també esborrar-ne. Les paraules poden estar repetides. Esborrar una paraula vol dir esborrar una de les seves aparicions. Esborrar una paraula que no hi és no té cap efecte. En qualsevol moment us poden preguntar quina és la paraula (*lexicogràficament*) més gran de la bossa, i quantes vegades apareix. També us poden preguntar el mateix respecte de la paraula més petita.

  Feu un programa que simuli aquest procés, i que respongui totes les preguntes sobre màxims i mínims que es facin en qualsevol moment.

  Entrada

  L'entrada consisteix en diverses línies. Cada línia conté "*guarda p*", on p és una paraula, o bé "*elimina p*", on p és una paraula, o bé "*maxim?*", o bé "*minim?*". Les paraules tenen exclusivament una o més lletres minúscules.

  Sortida

  Per a cada pregunta, escriviu quina és la paraula més gran (o més petita) continguda a la bossa en aquell moment. Si, en el moment de respondre alguna pregunta, la bossa estigués buida, cal indicar-ho.

Exercici: La bossa de les paraules (*jutge P84415*)

- Per exemple:

```
$ more prova.inp
minim?
guarda hi
minim?
elimina bye
guarda hi
maxim?
minim?
guarda bye
minim?
elimina bye
elimina hi
elimina hi
maxim?
elimina hi
maxim?
$ python3 bossa_dict.py < prova.inp
mínim indefinit
mínim: hi, 1 vegades
màxim: hi, 2 vegades
mínim: hi, 2 vegades
mínim: bye, 1 vegades
màxim indefinit
màxim indefinit
```

Exercici: La bossa de les paraules (*jutge P84415*)

- Aquest problema és un problema típic de diccionaris. El qüid de la qüestió està en *les peticions del mínim o el màxim*. Aquestes operacions són *cares* (lineal en el nombre d'elements) si el diccionari està implementat amb taules de dispersió. Amb BSTs equilibrats són operacions logarítmiques, i amb BSTs, si tenim sort (depén de com construïm l'arbre i per tant de l'entrada) i l'arbre no degenera massa, també.

  La solució en Python (utilitzant els diccionaris de Python) ⇨

Fitxer **bossa_dict.py** (calen els imports)

```
M = dict()
s = read(str)
while s is not None:
    if s == "guarda":
        t = read(str)
        if t not in M:
            M.update({t : 1})
        else:
            M[t] += 1
    elif s == "elimina":
        t = read(str)
        if t in M:
            M[t] -= 1
            if M[t] <= 0:
                M.pop(t)
    elif s == "maxim?":
        if not bool(M):
            print("màxim indefinit")
        else:
            l = max(M)
            print("màxim: ", l, ", ", M[l], " vegades", sep='')
    else:
        if not bool(M):
            print("mínim indefinit")
        else:
            l = min(M)
            print("mínim: ", l, ", ", M[l], " vegades", sep='')
    s = read(str)
```

Exercici: La bossa de les paraules (*jutge P84415*)

- La solució en Python, utilitzant els diccionaris implementats amb BSTs que tot just hem estudiat.

  Cal fer **from diccionari_bst import *** ⇨

Fitxer **bossa_diccionari.py** (calen els imports)

```
M = Diccionari()
s = read(str)
while s is not None:
    if s == "guarda":
        t = read(str)
        v = M.valor(t)
        M.assigna(t,(v+1) if v is not None else 1)
    elif s == "elimina":
        t = read(str)
        v = M.valor(t)
        v = v if v is not None else 0
        if v <= 1:
            M.elimina(t)
        else:
            M.assigna(t,v-1)
    elif s == "maxim?":
        if M.buit():
            print("màxim indefinit")
        else:
            l = M.troba_maxim()
            print("màxim: ", l.clau, ", ", l.valor, " vegades", sep='')
    else:
        if M.buit():
            print("mínim indefinit")
        else:
            l = M.troba_minim()
            print("mínim: ", l.clau, ", ", l.valor, " vegades", sep='')
    s = read(str)
```

Exercici: La bossa de les paraules (*jutge P84415*)

- Si fem servir un joc de proves prou gran podem veure l'eficiència guanyada amb la solució amb BSTs. Farem servir un fitxer amb 120.000 ordres anomenat `prova.inp`: (tenim la solució al fitxer `prova.cor`)

```
$ time python3 bossa_dict.py < prova.inp > prova_dict.jor

real 0m29,586s
user 0m28,468s
sys  0m0,848s

$ diff prova.cor prova_dict.jor    <== Ens assegurem que la solució és bona
$


$ time python3 bossa_diccionari.py < prova.inp > prova_diccionari.jor

real 0m4,583s
user 0m3,974s
sys  0m0,609s

$ diff prova.cor prova_diccionari.jor    <== Ens assegurem que la solució és bona
$
```

Complexitat (II)

Dèiem a PA1...

- Suposem que **f(n)** i **g(n)** són funcions que retornen nombres reals.
- Per a nosaltres, **f(n)** serà generalment alguna mena de ***funció de cost***: El temps d'execució per a un problema de mida **n**.

- Utilitzarem la notació Θ**(g(n))** (*theta de* **g(n)**) per indicar "*el conjunt de totes les funcions els valors absoluts de les quals són, eventualment, proporcionals a* **g(n)**"
- Escriurem **f(n) ∈** Θ**(g(n))** per indicar el següent:
  "*per a* **n** *prou gran*, $K_1|g(n)| \leq |f(n)| \leq K_2|g(n)|$, *on* $0 < K_1 < K_2$ *són constants*"
- En altres paraules, "**|f(n)|** *és aproximadament proporcional a* **|g(n)|**".

- Aquesta notació es pot utilitzar per expressar la taxa de creixement de qualsevol funció, tot i que nosaltres estem interessats en funcions de cost

La Notació Asimptòtica, continuació...

- En realitat $\Theta(f(n))$ s'acostuma a definir en funció d'altres conjunts de funcions, anomenats l'"*O gran*" i l'"*Omega gran*":

  Definirem $O(f(n))$ i $\Omega(f(n))$ de la manera següent:

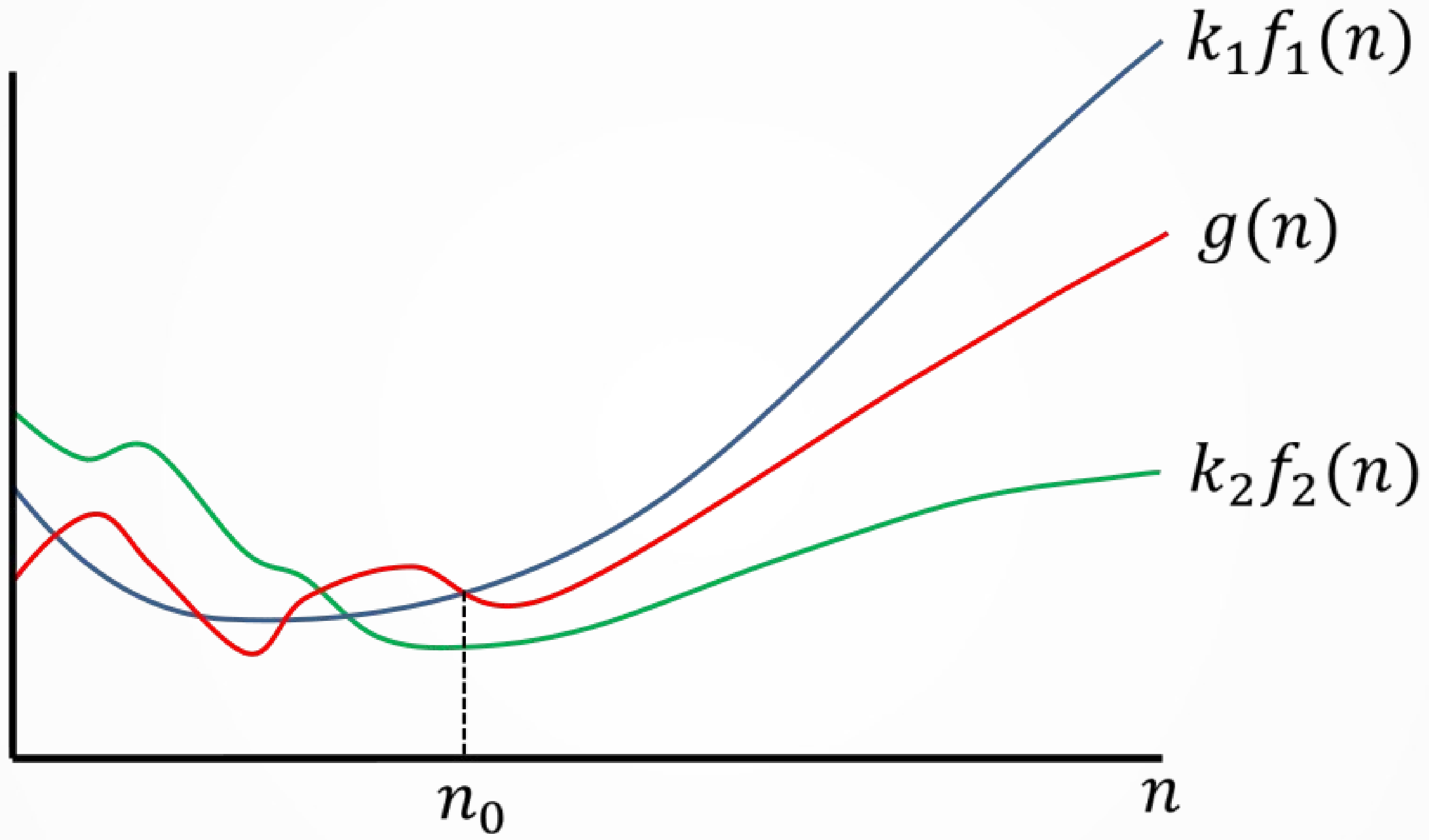
  $O(f(n)) = \{\ g(n) : \exists\ k > 0,\ \exists\ n_0,\ \forall\ n \geq n_0\ \ g(n) \leq k \cdot f(n)\ \}$

  $\Omega(f(n)) = \{\ g(n) : \exists\ k > 0,\ \exists\ n_0,\ \forall\ n \geq n_0\ \ g(n) \geq k \cdot f(n)\ \}$

  Aleshores $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$

- Recordem que, per a nosaltres, $f(n)$ serà generalment alguna mena de ***funció de cost***: El temps d'execució per a un problema de mida n.
- Escriurem $g(n) \in O(f(n))$ per indicar que "$f(n)$ *és, asimptòticament, una fita superior de* $g(n)$".
- Escriurem $g(n) \in \Omega(f(n))$ per indicar que "$f(n)$ *és, asimptòticament, una fita inferior de* $g(n)$".

La Notació Asimptòtica, continuació...



$$g(n) \in \mathrm{O}\big(f_1(n)\big)$$

$$g(n) \in \Omega(f_2(n))$$

La Notació Asimptòtica, exemples

$$13n^3 - 4n + 8 \in \mathrm{O}\left(n^3\right)$$
$$2n - 5 \in \mathrm{O}\left(n\right)$$
$$n^2 \notin \mathrm{O}\left(n\right)$$
$$2^n \in \mathrm{O}\left(n!\right)$$
$$3^n \notin \mathrm{O}\left(2^n\right)$$
$$3\log_2 n \in \mathrm{O}\left(\log n\right)$$
$$3n\log_2 n \in \mathrm{O}\left(n^2\right)$$
$$\mathrm{O}\left(n^2\right) \subseteq \mathrm{O}\left(n^3\right)$$

$$13n^3 - 4n + 8 \in \Omega(n^3)$$
$$n^2 \in \Omega(n)$$
$$n^2 \notin \Omega(n^3)$$
$$n! \in \Omega(2^n)$$
$$3^n \in \Omega(2^n)$$
$$3\log_2 n \in \Omega(\log n)$$
$$n\log_2 n \in \Omega(n)$$
$$\mathrm{O}\left(n^3\right) \subseteq \Omega(n^2)$$

La Notació Asimptòtica, propietats

- Totes aquestes propietats (excepte l'última) també es poden aplicar a **Ω** i a Θ.

- $f \in O(f)$
- $\forall c > 0,\ O(f) = O(c \cdot f)$
- $f \in O(g) \wedge g \in O(h) \Rightarrow f \in O(h)$
- $f1 \in O(g1) \wedge f2 \in O(g2) \Rightarrow f1+f2 \in O(g1+g2) = O(\max\{g1,g2\})$
- $f \in O(g) \Rightarrow f+g \in O(g)$
- $f1 \in O(g1) \wedge f2 \in O(g2) \Rightarrow f1 \cdot f2 \in O(g1 \cdot g2)$
- $f \in O(g) \Leftrightarrow g \in \Omega(f)$

La Notació Asimptòtica, la regla del límit

- Suposem que L existeix (pot ser ∞) on:

$$L = \lim_{n \to \infty} \frac{f(n)}{g(n)}$$

Aleshores,

$$
\begin{array}{lcl}
\mathrm{L} = 0 & \Rightarrow & \mathrm{f} \in \mathrm{O}(\mathrm{g}) \\
0 < \mathrm{L} < \infty & \Rightarrow & \mathrm{f} \in \Theta(\mathrm{g}) \\
\mathrm{L} = \infty & \Rightarrow & \mathrm{f} \in \Omega(\mathrm{g})
\end{array}
$$

- Si els dos límits són 0 o ∞, podem fer servir la regla de L'Hôpital:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{f'(n)}{g'(n)}$$

Exemple: Ordenació per Selecció

Segur que recordeu l'***ordenació per selecció*** (PA1: problema 53, sessió 10 de laboratori)

```
def ordenacio_per_seleccio(lst):
    n = len(lst)
    for i in range(n-1):
        # Inv: lst[0:i] esta ordenat i els elements de lst[i:n]
        # son tots mes grans que els de lst[0:i]
        minim = i
        for j in range(i+1, n):
            if lst[j] < lst[minim]:
                minim = j
        lst[i], lst[minim] = lst[minim], lst[i]
```

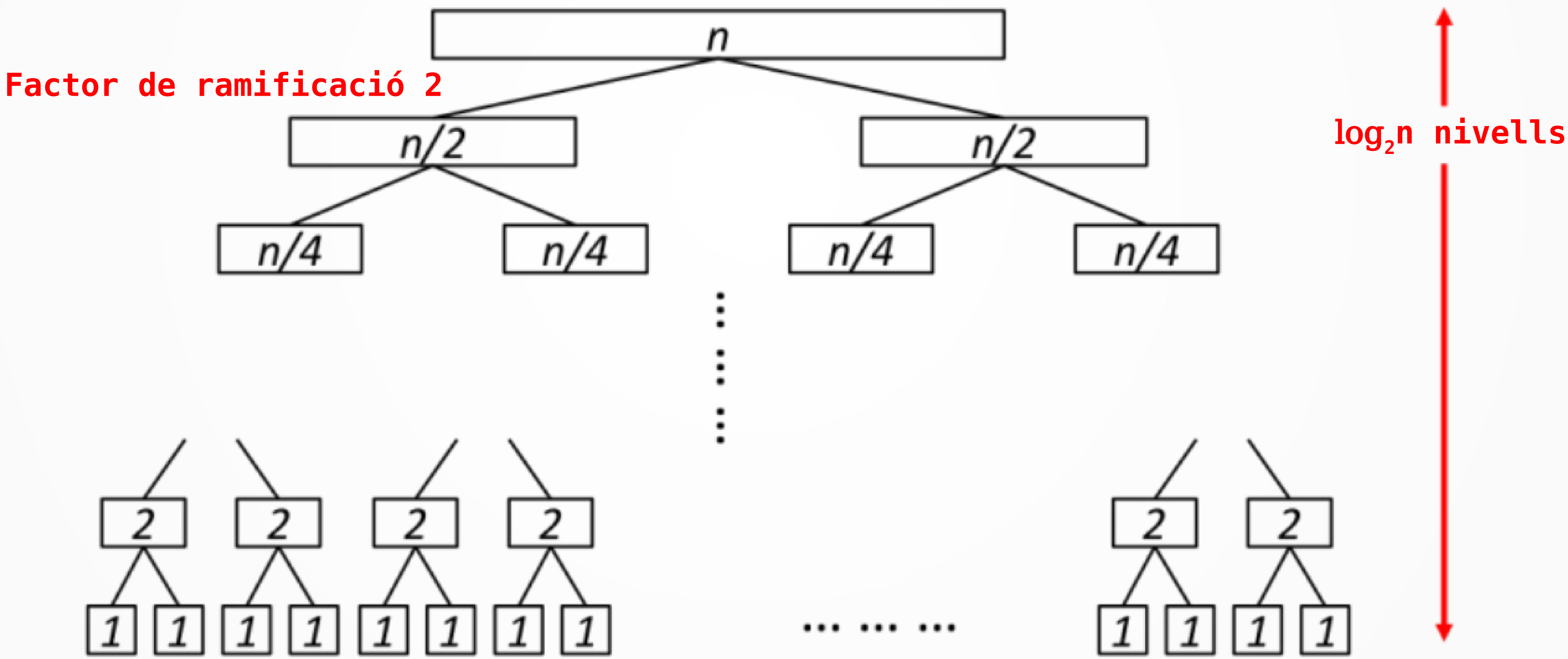
Volem calcular el temps que triga, en cas pitjor, aquesta ordenació:

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} \Theta(1) = \Theta(1) \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \Theta(1) \sum_{i=0}^{n-2} (n-i-1)$$

$$= \Theta(1) \left(\frac{n}{2}(n-1) \right) = \Theta(1) \cdot \Theta(n^2) = \Theta(n^2)$$

Algorismes recursius

Imaginem una funció *recursiva* que té com a paràmetre un objecte de mida n (p.ex. una llista amb n elements) que realitza dues crides recursives sobre objectes de mida n/2. Podem *visualitzar* el procés amb un arbre com el següent...



Algorismes recursius

El temps $T(n)$ que triga la funció serà $2 \cdot T(n/2)$, més la feina que calgui fer addicional, que pot ser $\Theta(1)$, $\Theta(n)$, $\Theta(n^2)$, etc. En general podem suposar que aquesta feina addicional a fer en cada nivell és $\Theta(n^c)$. Això ens permet plantejar una *eqüació recurrent*, o *recurrència*, pel cost $T(n)$.

A PA1 vam veure alguns problemes amb solucions recursives a les que podríem aplicar aquest raonament, l'estructura de les quals s'ajusta a aquest arbre. En aquests casos podrem raonar sobre la seva complexitat i trobar una aproximació asimptòtica del seu cost. Un parell d'exemples d'algorismes importants que vam veure:

Cerca eficient (*binary search*): Fèiem *una* crida recursiva amb la meitat de la llista, o del fragment de llista, amb una feina addicional constant (no depenia de la mida de la llista, o del fragment de llista, que teníem com a paràmetre):

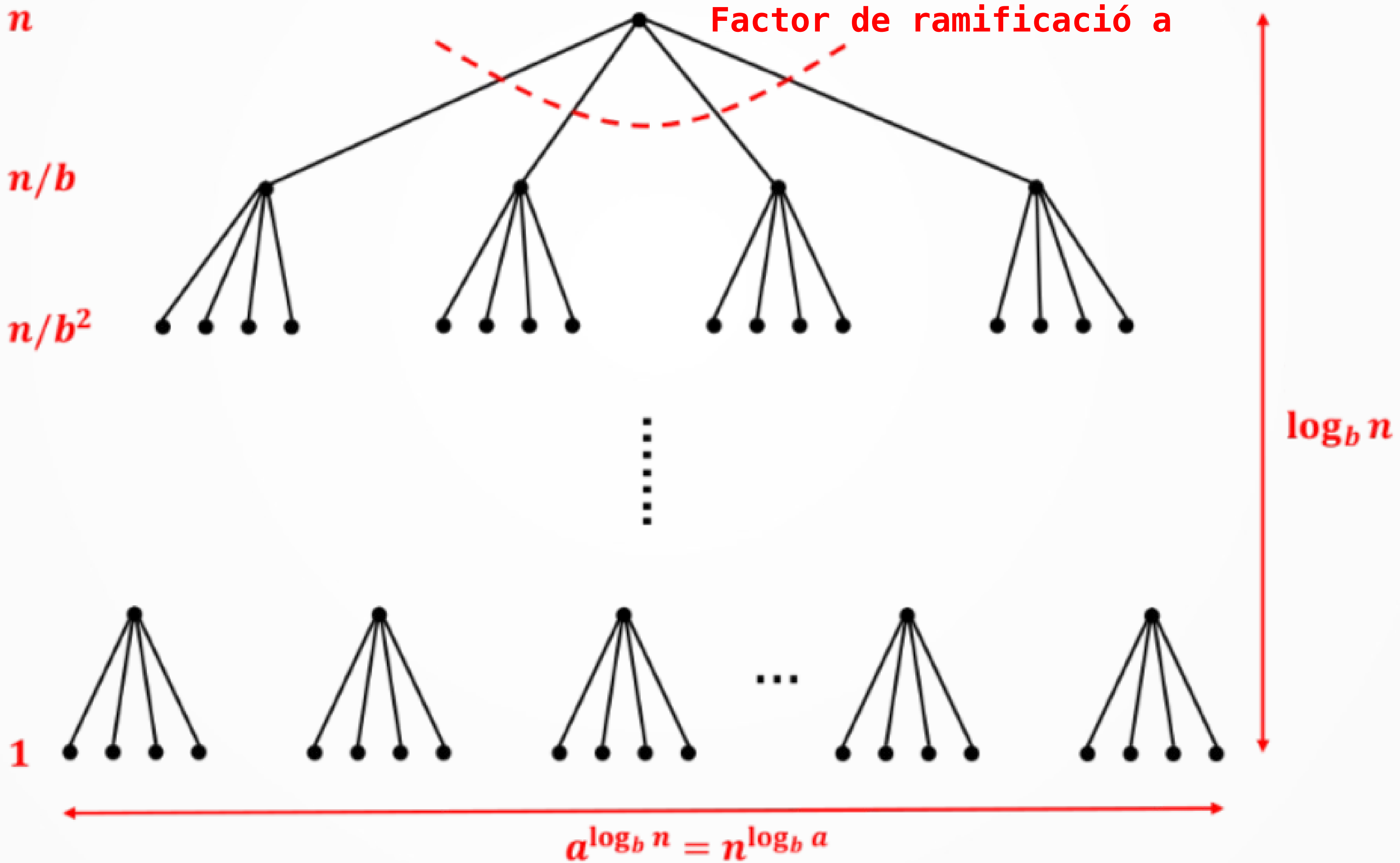
⇨ $T(n) = T(n/2) + \Theta(1)$

Ordenació per fusió (*merge sort*): Fèiem *dues* crides recursives amb les diferents meitats de la llista, o del fragment de llista, amb una feina addicional (fusionar) que era lineal en el nombre d'elements:

⇨ $T(n) = 2 \cdot T(n/2) + \Theta(n)$

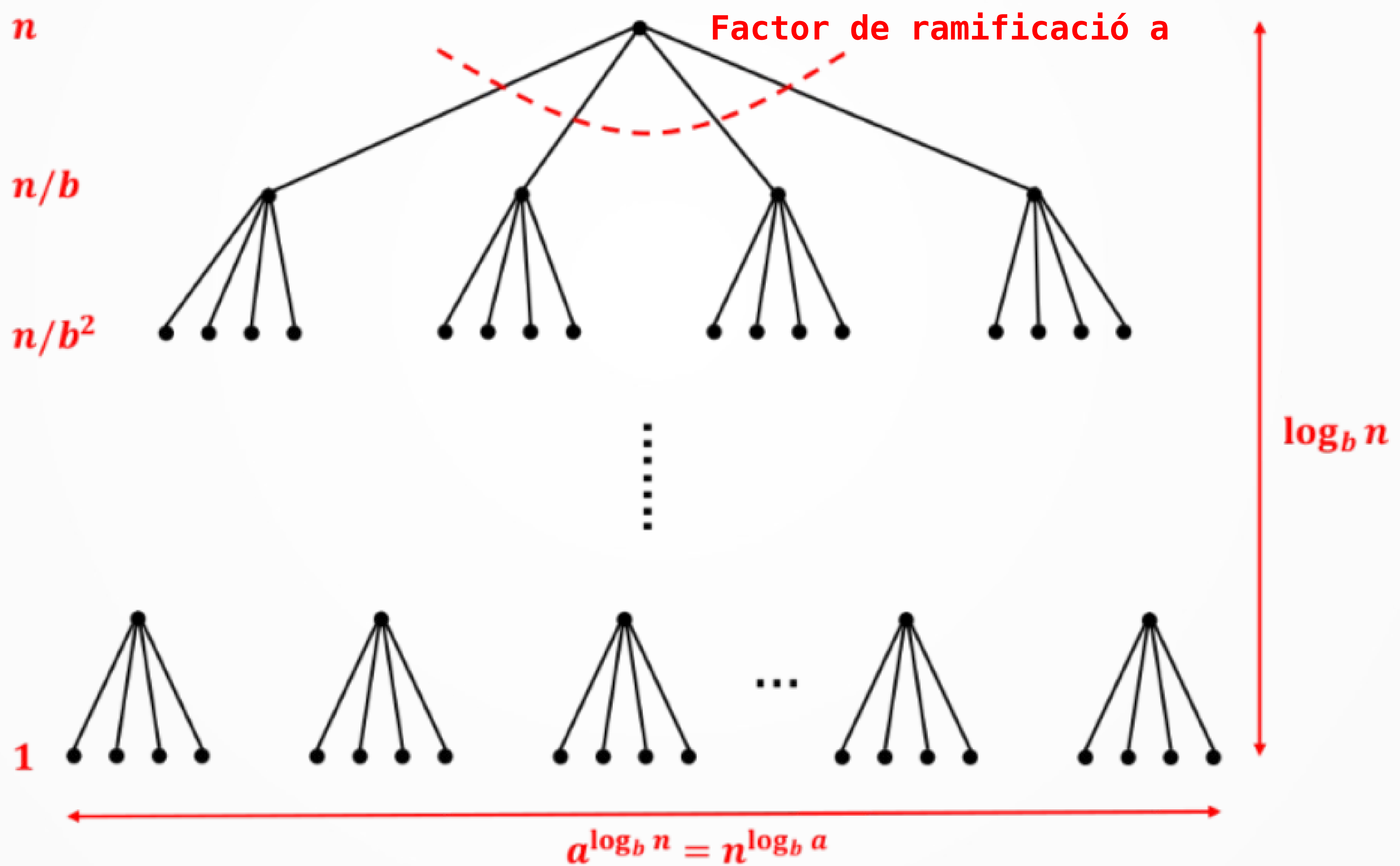
Algorismes recursius

Aquest esquema el podem generalitzar:



Algorismes recursius

En aquest cas podríem plantejar la recurrència general: **T(n) = a·T(n/b) + Θ(n^c)**



El *Master Theorem* per recurrències divisores

El ***Master Theorem*** ens diu que, donada la recurrència general: $T(n) = a \cdot T(n/b) + \Theta(n^c)$ (on a, b i c són *constants*, és a dir, no depenen d'n) podem afirmar:

Si $a < b^c$ aleshores $T(n) = \Theta(n^c)$

Si $a = b^c$ aleshores $T(n) = \Theta(n^c \cdot \log(n))$

Si $a > b^c$ aleshores $T(n) = \Theta(n^{\log_b(a)})$

Cerca eficient (*binary search*): Fèiem *una* crida recursiva amb la meitat de la llista, o del fragment de llista, amb una feina addicional constant (no depenia de la mida de la llista, o del fragment de llista, que teníem com a paràmetre):

⇨ $T(n) = T(n/2) + \Theta(1)$, així, aplicant el *Master Theorem*, $T(n) = \Theta(\log(n))$

Ordenació per fusió (*merge sort*): Fèiem *dues* crides recursives amb les diferents meitats de la llista, o del fragment de llista, amb una feina addicional (fusionar) que era lineal en el nombre d'elements:

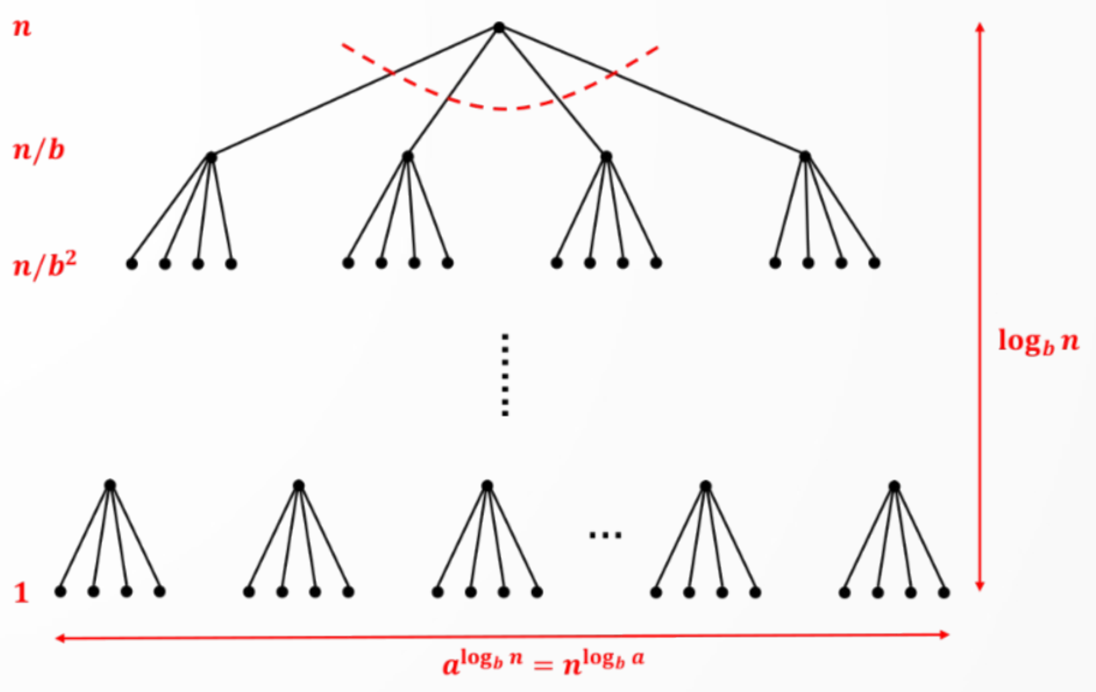
⇨ $T(n) = 2 \cdot T(n/2) + \Theta(n)$, així, aplicant el *Master Theorem*, $T(n) = \Theta(n \cdot \log(n))$

Estratègia Dividir i Vèncer (*Divide and Conquer*)

Aquesta estructura es troba en algorismes resultants d'aplicar una coneguda estratègia de disseny algorísmic anomenada ***dividir i vèncer***.

A grans trets, l'estratègia consisteix en, donat un cert problema amb una entrada de mida n:

1.- *Dividir el problema en subproblemes més petits del mateix tipus de problema*
2.- *Resoldre els subproblemes de manera recursiva*
3.- *Combinar les respostes per resoldre el problema original*



La feina es fa:

- En dividir el problema en subproblemes
- En resoldre els casos base de la recursivitat
- En fusionar les respostes dels subproblemes per obtenir la solució del problema original

Altres algorismes recursius

Hi ha algorismes recursius que NO s'ajusten a aquesta estructura. Un exemple en seria trobar (*recursivament*) el màxim d'una llista d'elements comparables, que ens permet trobar la recurrència pel cost $T(n)$: $T(n) = T(n-1) + \Theta(1)$

Hi ha casos *senzills* on es pot resoldre la recurrència *manualment*, desplegant-la, p.ex:

$$T(n) = T(n-1)+\Theta(1) = T(n-2)+2\cdot\Theta(1) =...= T(0)+n\cdot\Theta(1) \Rightarrow T(n) = \Theta(n)$$

És fàcil veure que:

$$T(n) = T(n-1) + \Theta(n^c) \Rightarrow T(n) = \Theta(n^{c+1})$$

El *Master Theorem* per recurrències subtractives

El ***Master Theorem*** ens diu que, donada la recurrència general: $T(n) = a \cdot T(n-c) + \Theta(n^k)$ (on a, c i $k \geq 0$ són *constants*, és a dir, no depenen d'n) podem afirmar:

Si $a < 1$ aleshores $T(n) = \Theta(n^k)$

Si $a = 1$ aleshores $T(n) = \Theta(n^{k+1})$

Si $a > 1$ aleshores $T(n) = \Theta(a^{n/c})$

Factorial: Fèiem *una* crida recursiva l'argument menys 1, amb una feina addicional constant:
⇨ $T(n) = T(n-1) + \Theta(1)$, així, aplicant el *Master Theorem*, $T(n) = \Theta(n)$

Complexitat d'algorismes vs. complexitat de problemes

Quan calculem la complexitat *en cas pitjor* d'un *programa* en Python per resoldre un *problema*, aquest *programa* és una implementació concreta (en Python) d'un *algorisme abstracte* per resoldre l'esmentat problema.

De vegades som capaços de determinar clarament quin és el cas pitjor per a aquell algorisme, i per tant podem calcular el cost en cas pitjor de l'algorisme, en funció de la mida **n** de l'entrada, en termes de $\Theta(f(n))$.

Ara bé, què ens diu això respecte de la complexitat del *problema*?

Exemple: L'*ordenació per selecció* i l'*ordenació per fusió* són algorismes *diferents* per resoldre el *mateix problema*: L'*ordenació d'una llista de mida* **n**:

Cost en cas pitjor de l'ordenació per selecció: $\Theta(n^2)$

Cost en cas pitjor de l'ordenació per fusió: $\Theta(n \cdot \log(n))$

Complexitat d'algorismes vs. complexitat de problemes

Què podem dir sobre el cost en cas pitjor de l'ordenació d'una llista de mida **n**?

Sabem que es pot resoldre amb un cost $\Theta(n \cdot \log(n))$. És a dir, aquest cost és la <u>*fita superior*</u> del cost real que comporta resoldre el problema. Potser podem resoldre el problema millor, però sabem que, com a mínim, ho podem fer així. Per tant, tot el que podem afirmar en aquest moment és que el problema de l'ordenació d'una llista de mida **n** té un cost en cas pitjor $O(n \cdot \log(n))$
Ho podem fer millor? Amb la informació proporcionada fins ara no ho sabem.

Afortunadament s'ha pogut demostrar que el cost en cas pitjor de resoldre aquest problema també pertany a $\Omega(n \cdot \log(n))$. És a dir, aquesta funció és també una fita *inferior* pel cost en cas pitjor del problema de l'ordenació de llistes de mida **n***.

Finalment, com que (el cost en cas pitjor de resoldre) aquest problema pertany a $O(n \cdot \log(n))$ i a $\Omega(n \cdot \log(n))$, podem dir que pertany a $\Theta(n \cdot \log(n))$.

* *si l'ordenació està basada en comparacions*

Complexitat d'algorismes vs. complexitat de problemes

En general, podem dir que el fet de tenir un algorisme de cost, en cas pitjor, Θ`(f(n))` per resoldre un determinat problema ens permet afirmar que el cost en cas pitjor de resoldre el <u>*problema*</u> està en O`(f(n))`.

Per poder anar més enllà, i afirmar que el cost en cas pitjor de resoldre el problema és exactament Θ`(f(n))`, cal poder demostrar *abans* que el cost en cas pitjor de resoldre l'esmentat problema també està en Ω`(f(n))`.

Això vol dir que cal demostrar que una funció del conjunt Ω`(f(n))` és una <u>*fita inferior*</u> pel cost de resoldre, en cas pitjor, aquest problema. I això, en general, és **difícil**.

Exercici: Llegiu el còmic `https://xkcd.com/342/` i expliqueu per què la darrera vinyeta no fa gràcia, és confusa: