FIB

PROGRAMACIÓ i ALGORÍSMIA - 2

Guies de Laboratori

Jordi Delgado i José Luis Balcázar
(Dept. CS, UPC)

Grau d'IA, 2022-23
FIB (UPC)

Sessió 1: *Heaps* i *Heapsort*

Suposem el següent problema:

Donada una llista d'elements comparables (és a dir, que donats dos elements qualsevol té sentit la pregunta *quin és més petit*, o dit d'una altra manera, aquests elements poden ser comparats amb l'operador `<`), volem ser capaços d'obtenir, i eliminar de la llista, l'element més petit repetides vegades, i també volem poder inserir-hi nous elements. Per fixar idees, els elements comparables seran nombres. La solució del problema la podem expressar fent dues funcions: `obtenir_min(l)`, i `inserir(l,x)`. Totes dues operacions són destructives.

Així, si `l = [6,2,9,7,3,0,-1,8,3]`, amb 9 elements:

```
>>> obtenir_min(l)
-1
>>> len(l)          # el -1 ja no hi és a l
8
>>> obtenir_min(l)
0
>>> obtenir_min(l)
2
>>> inserir(l,-3)
>>> len(l)          # hem tret el 0 i el 2, però hem afegit el -3
7
>>> obtenir_min(l)
-3
>>> len(l)
6
```

Segur que molts de vosaltres heu pensat que la solució és ordenar la llista. Així, cada cop que inserim un element mirem de fer-ho de manera que la llista continuï ordenada i obtenir l'element més petit és senzillament fer un `pop(0)` (o qualsevol de les altres possibles maneres d'eliminar el primer element d'una llista). Bé, aquesta és una solució, però és la *millor* solució? Comencem mirant quin és el cost d'aquesta solució:

- Ordenar la llista (d'`N` elements): $\Theta(N \cdot \log N)$
- Inserir un element, mantenint la llista ordenada: $\Theta(N)$ (*encara que fem una cerca binària per saber on cal inserir, inserir on toca és lineal ja que s'han de desplaçar els elements que calgui*)
- Obtenir l'element mínim: $\Theta(1)$ (*si no entrem massa en detalls, ja que cal eliminar l'element de la llista i això podria costar fins a* $\Theta(N)$)

En aquesta sessió de laboratori us presentarem una nova estructura de dades (o, millor dit, una nova forma d'*estructurar* les dades) de manera que aquestes operacions siguin *molt* més eficients. Farem els exemples fent servir nombres, però ja sabeu que ho podem generalitzar sense problema.

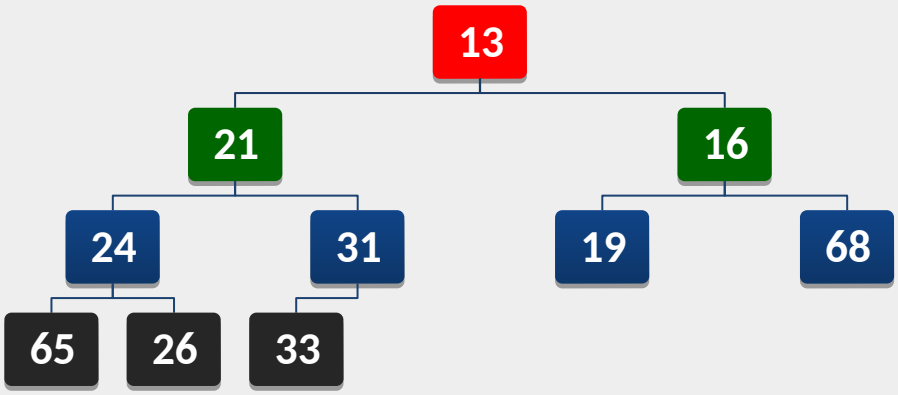
Farem servir un ***heap*** (més concretament, un ***binary heap***). Els elements de la llista continuen estant en una llista, però aquesta llista està organitzada d'una determinada manera:

> Una llista L és un (*binary*) ***heap*** si L és tal que, per a tot índex `1 ≤ k < len(L)`, tenim `L[k] ≤ L[2k]` (si `2k < len(L)`) i `L[k] ≤ L[2k+1]` (si `2k+1 < len(L)`).

Aquesta definició, tot i ser clara des d'un punt de vista matemàtic, no aporta gaire intuició sobre què és i com funciona un *heap*.

Per explicar els *heaps* va molt bé representar els elements en forma d'arbre binari, tot i que l'estructura de dades que farem servir en els programes és una llista. El recurs *gràfic* de la representació en arbre és útil només de cara a conceptualitzar el problema.

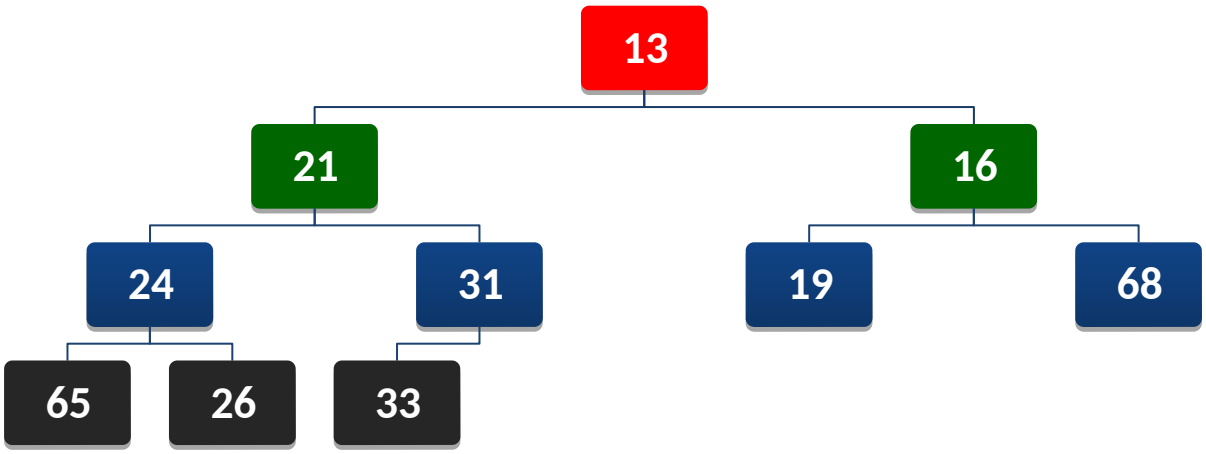
Primer, una mica de *terminologia* respecte als arbres. Per exemple, donat l'arbre binari:



direm que el node vermell, amb el nombre 13, és l'*arrel*. Direm també que els nodes tenen *fills* (*esquerre* i *dret*) i *pares*: El 21 té com a fill esquerre el 24 i com a fill dret el 31 (i aquests dos nodes tenen com a pare el 21), el 24 té dos fills, el 65 (esquerre) i el 26 (dret), i aquests dos nodes tenen com a pare el 24, el 31 té només un fill (esquerre), el 33. El 16 té al 19 i al 68 com a fills, i aquests tenen com a pare el 16. I els nodes 65, 26, 33, 19 i 68 no tenen cap fill. Als nodes sense fills se'ls anomena *fulles*. Direm que aquest arbre, com a cada node hi ha com a molt dos fills, que poden ser esquerre o dret, és un arbre *binari*. Cada node té una determinada *profunditat*: El 13 té profunditat 0, el 21 i el 16 tenen profunditat 1, etc. Cada node té com a profunditat, la profunditat del seu pare + 1. Els nodes amb la mateixa profunditat formen els *nivells*: 13 està a nivell 0, 21 i 16 estan a nivell 1, a nivell 2 trobem el 24, el 31, el 19 i el 68, etc. Un arbre binari amb tots els nivells *plens* és un arbre *complet*. Si tots els nivells estan plens excepte el darrer nivell, i aquest no té "*forats*", direm que l'arbre és *semi-complet*. L'arbre de l'exemple és semi-complet.

Veiem, doncs, la relació entre els *heaps* i els arbres binaris amb un exemple:

Suposem la llista de 10 elements `l = [16,21,13,65,19,31,24,26,33,68]`, i representem-la d'aquesta manera:



Aquest arbre té dues propietats:

- *Heap-order property*: Qualsevol node (excepte les fulles) és més petit que els seus fills.
- *Completesa*: L'arbre és *semi-complet* o *complet*.

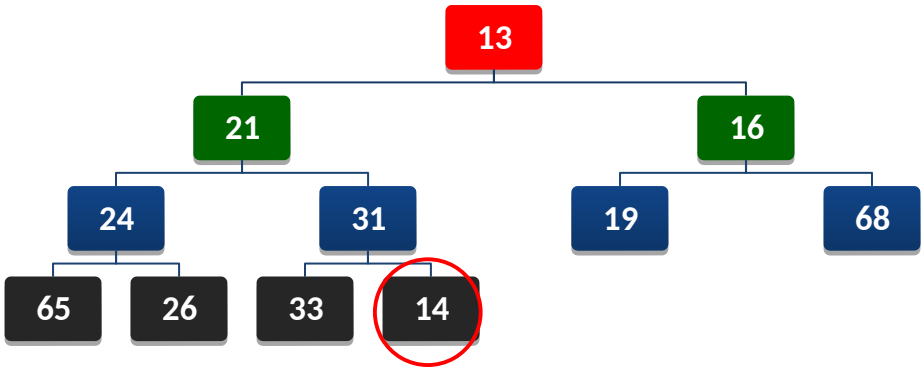
Fixeu-vos que l'element més petit està dalt de tot, a l'arrel. També, fixeu-vos que la representació de la llista `l` com a arbre que verifica aquestes propietats no és única.

I què fem amb aquest arbre?

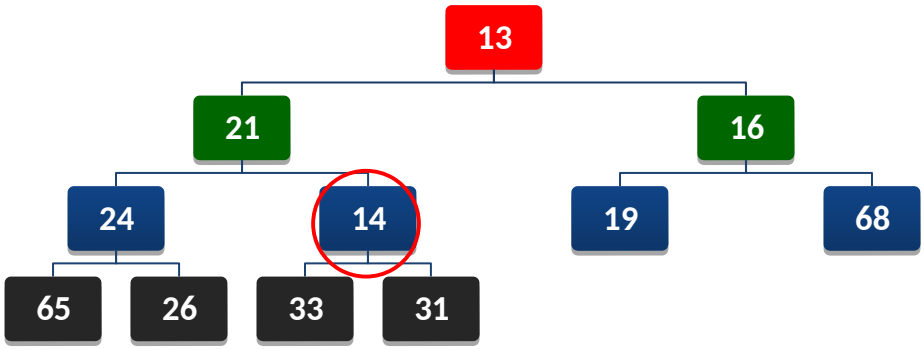
Veiem si podem implementar les operacions desitjades *si tinguéssim arbres en lloc de llistes*. Pensem com haurien de funcionar `obtenir_min(arbre)` i `inserir(arbre,x)`. La idea és que després de fer cada una de les operacions `obtenir_min(arbre)`, i `inserir(arbre,x)` l'arbre continuï satisfent les propietats de completesa i *heap-order*.

Suposem que `arbre` és el de la figura més amunt i volem fer `inserir(arbre,14)`:

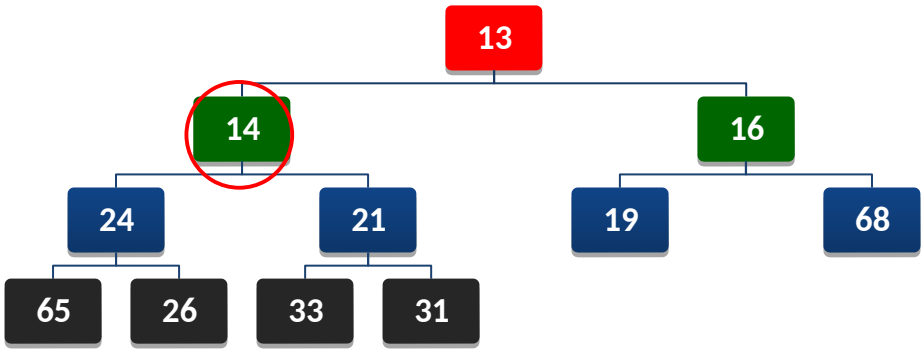
El primer pas és inserir l'element al primer lloc lliure en el darrer nivell (el lloc *lliure* de més a l'esquerra, o un nou node en el següent nivell en cas que el darrer nivell estigui ple)



Ara, aquest arbre no satisfà la propietat *heap-order*: El pare del 14 (el 31) és més gran que el fill (el 14). Com ho arreglem? Permutem el pare i el fill (d'això en direm *surar* un cop):



Hem aconseguit recuperar la propietat *heap-order*? No encara. El pare (21) és encara més gran que el fill (14). Tornem a *surar* un cop (recordem, permutar de lloc el pare i el fill):



I ara sí, tenim completesa (òbviament) i la propietat *heap-order* es satisfà.

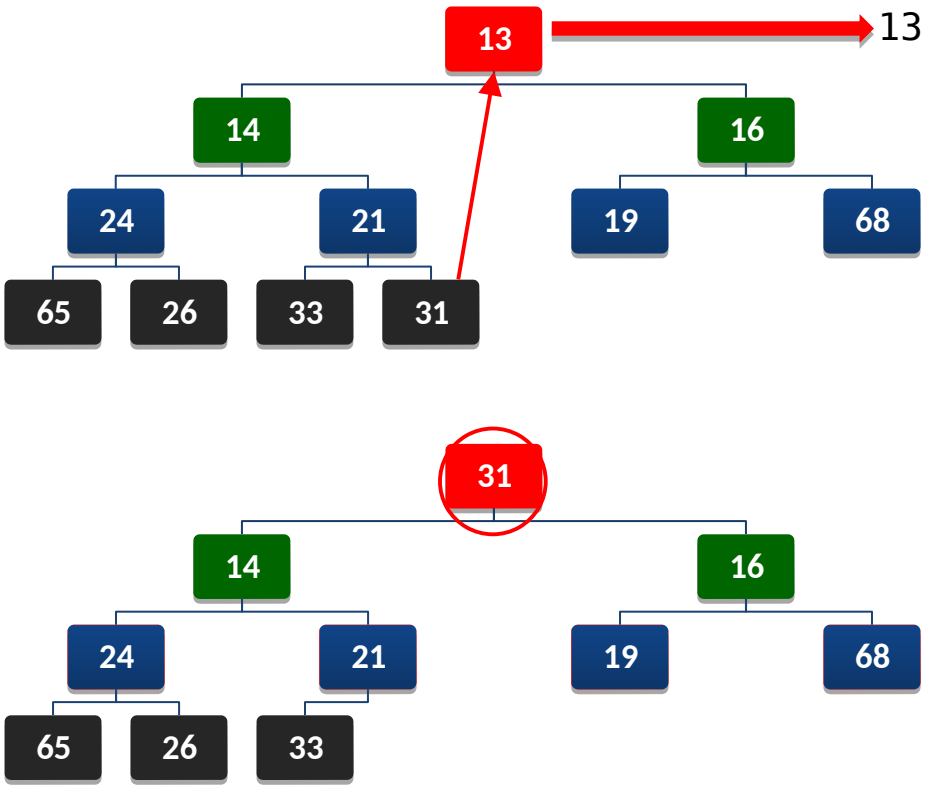
Així doncs:

`inserir(arbre,x)` → Inserir `x` al final de l'`arbre`, i *surar* tants cops com calgui fins que el resultat satisfaci les propietats de *heap*.

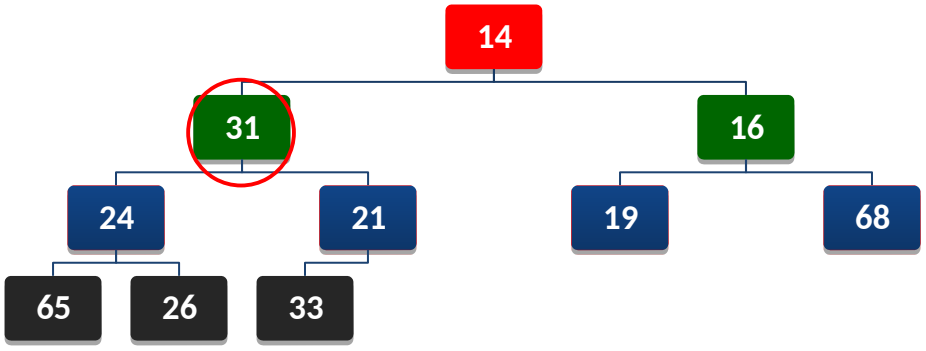
Partim ara de l'`arbre` resultant de fer `inserir(arbre,14)`. Volem fer `obtenir_min(arbre)`.

Consultem l'arbre i obtenim el mínim a l'arrel (el 13). Això és senzill. Ara cal eliminar el 13 de l'arbre i que aquest continuï satisfent les propietats de *heap-order* i completesa.

Mourem el darrer element (darrer nivell, més a la dreta) a l'arrel de l'arbre:

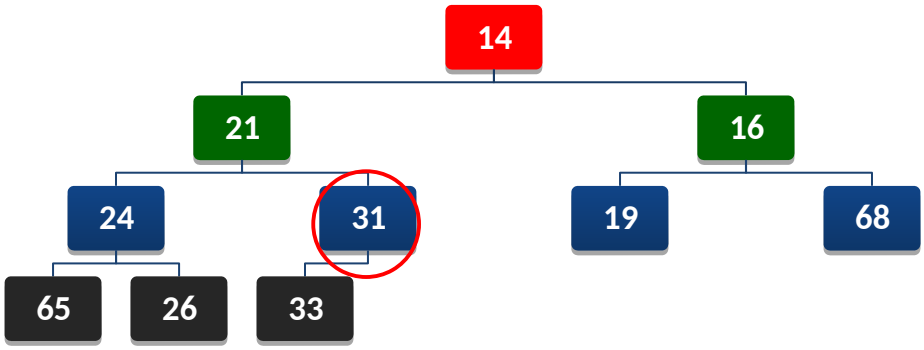


Ara mirem si es satisfà la propietat *heap-order* i veiem que no. Fem el mateix que abans (permutar pare i fill), però des del punt de vista del pare. D'això en direm *enfonsar* un cop. Amb quin fill permutem? Si volem preservar la propietat *heap-order* caldrà fer-ho amb el més petit dels dos:



Continuem enfonsant fins que l'arbre recuperi la propietat *heap-order*. Fixem-nos que la completesa no perilla en cap moment, l'arbre es manté (semi-)complet durant tot el procés.

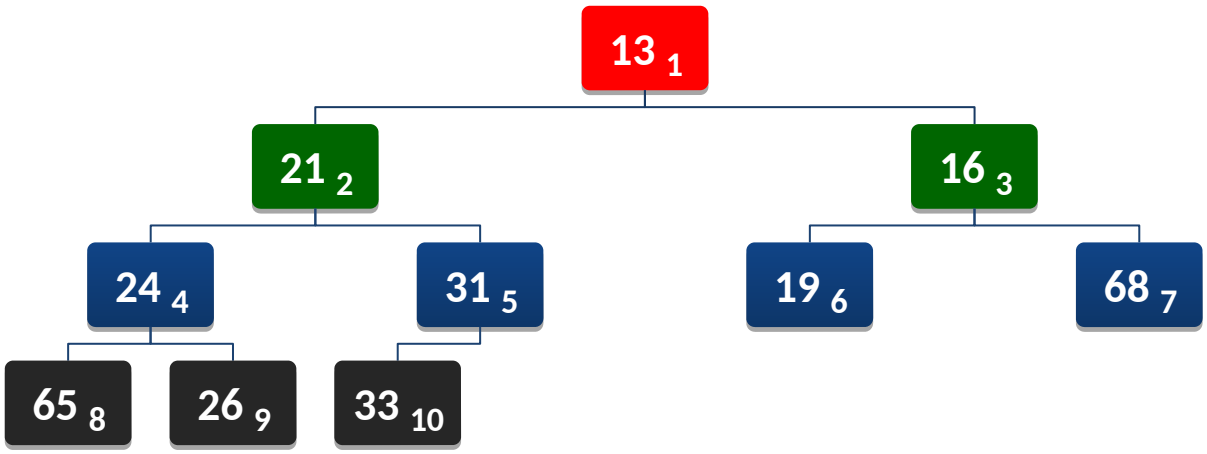
Així doncs, en aquest exemple només ens queda enfonsar un cop més per arribar a satisfer les propietats requerides:



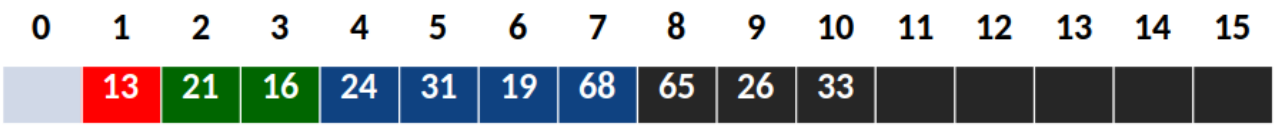
`obtenir_min(arbre)` → Guardar l'arrel de l'`arbre` (el mínim) per retornar-lo després, posar el darrer element a l'arrel i *enfonsar* fins que el resultat satisfaci la propietat *heap-order*.

Però... de moment nosaltres no sabem res d'arbres! Ja hem dit al començament que, tot i que veurem els arbres ben aviat, ara mateix només ens fan falta com a eina conceptual. Per què? Perquè podem entendre tot el que hem explicat fins ara utilitzant els arbres en termes de llistes. Veiem-ho.

Tornem a l'arbre original de l'exemple, i enumerem els nodes per nivells (primer el del primer nivell, després els del segon nivell, etc.):



Ara podem construir una llista de manera que aquesta enumeració es correspongui a les posicions de cada element:



és a dir, `l = [None,13,21,16,24,31,19,68,65,26,33]`. Fixem-nos que la posició `0` no la farem servir, per això hi posem `None`.

La clau de la correspondència *llista* ↔ *arbre* està en adonar-se de la relació entre la posició dels nodes, la posició dels seus fills i la posició del seu pare.

Si un node està en la posició `i` (`> 0`) de la llista `l`, és a dir, si un node és a `l[i]`:

- Els seus fills són a `l[2*i]` i a `l[2*i+1]`. Fixeu-vos que pot passar que no tingui fills (si `2*i` és més gran que la mida de la llista) o que només en tingui un (en aquest cas existeix `l[2*i]`, però no `l[2*i+1]`. És *impossible*, però, que existeixi `l[2*i+1]`, però no `l[2*i]`, per què?).

- El seu pare és a `l[i//2]` (si `i//2 > 0`, és a dir, si el node no és el mínim)[1].

La llista `l`, obtinguda a partir de l'arbre que hem fet servir per pensar les operacions que volem fer, és un ***heap***, d'acord a la definició que hem donat al començament, ja que l'arbre satisfà la propietat de *heap-order*. Aquesta no és més que la definició de *heap*, i la completesa implica, senzillament, que la llista no té forats.

Així, ja tenim la solució! Reformulem el que hem vist amb arbres, però ara fent servir *heaps* (llistes):

```
def inserir(lst,x):
    # Pre: lst és un heap
    lst.append(x)
    surar(lst,len(lst)-1)
```

```
def obtenir_min(lst):
    # Pre: lst és un heap no buit
    primer = lst[1]
    darrer = lst.pop()
    if len(lst) > 1:
        lst[1] = darrer
        enfonsar(lst,1)
    return primer
```

1.- *És clar que ara cal implementar* `surar(lst,index)` *i* `enfonsar(lst,index)` *(on suposem que* `1 ≤ index < len(lst)`*) Aquest serà el primer exercici d'aquesta sessió. Pista: Són funcions recursives.*

Hi ha un detall que hem deixat pel final: Donada una llista qualsevol... *com convertim aquesta llista en un heap*?

Hi ha una manera immediata: Partim d'un *heap* buit i fem `N` (el nombre d'elements de la llista) insercions. Però això té un cost $\Theta(N \cdot \log N)$.

2.- *Per què té aquest cost? Ens falta saber el cost d'*`inserir` *i* `obtenir_min`*, i per tant estem obligats a calcular el cost de* `surar` *i* `enfonsar`*. Aquest és el segon exercici d'aquesta sessió. Calculeu els costos d'aquestes funcions que tot just heu implementat.*

---

1 Fixem-nos que si haguéssim fet servir una llista des de la posició 0 (en lloc de posar-hi `None`), com habitualment fem en Python, els fills del node k estarien a 2k+1 i 2k+2, i el pare del node k dependria de si aquest node és un fill esquerre (en aquest cas el pare estaria a k//2) o un fill dret (en aquest cas el pare estaria a k//2-1). Aquesta complicació és la que ens estalviem posant un element *dummy* a la posició 0 i fent servir la llista des de la posició 1.

Ho podem fer molt millor, però. Suposem que la llista té `N` elements. Aquesta llista, amb `None` a la posició `0`, és tal que el darrer índex és precisament `N`. Aleshores, el pare del darrer element (posició `N`) estaria situat a la posició `N//2`, i podem inferir que cap element des de la posició `N//2` fins a la `N` té fills, és a dir, la meitat dels elements són fulles.

Així, donada una llista qualsevol, podem construir una llista amb `None` al davant, i després tractar aquesta llista com un *heap per construir*. Així, anirem enfonsant tots els nodes que trobem que tinguin fills:

```
def heapify(items = []):    # operació NO destructiva
    lst = [None]            # element a ignorar en la posició 0
    for e in items:
        lst.append(e)
    darrer_index = len(lst)-1
    for i in range(darrer_index//2, 0, -1):
        enfonsar(lst,i)
    return lst
```

Es pot demostrar que el cost de `heapify` sobre llistes amb N elements és **Θ(N)** (nosaltres no ho demostrarem).

Tota aquesta explicació s'ha fet pensant que l'element que ens interessa obtenir eficientment de la llista és el més petit, el mínim. Per a això, el que nosaltres hem anomenat *heap* es diu en realitat ***binary min-heap***.

3.- *Mireu d'implementar un **binary max-heap**, on el que ens interessa és obtenir eficientment el màxim. Les operacions a implementar són:* `obtenir_max(l)`*, i* `inserir(l,x)` *(amb els corresponents* `surar` *i* `enfonsar`*). Si teniu ben implementat el min-heap, el max-heap és molt fàcil de pensar i de fer.*

4.- *Per acabar, aprofitant això que hem vist, com faríeu un algorisme d'ordenació? L'algorisme d'ordenació a partir de heaps s'anomena **heapsort**. Hi ha una versió fàcil, que fa servir una llista auxiliar, i una altra (que ens recorda a l'ordenació per selecció que vam veure a PA1) que ordena a la mateixa llista (in-place, es diu en anglès). Quin cost té el heapsort (les dues versions tenen el mateix cost en termes de Θ)?*

Aquesta estructura de dades és extremadament útil, i per això Python té un mòdul per treballar amb *heaps*: **heapq** (`https://docs.python.org/3/library/heapq.html`). Els *heaps* s'utilitzen per implementar el concepte de ***cua amb prioritat*** (*priority queue*).

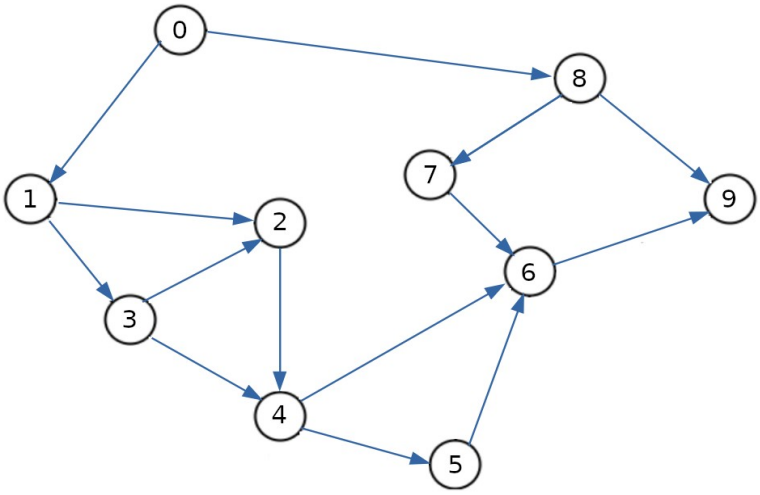
Sessió 2: Grafs

ATENCIÓ!
Primer de tot, cal que vingueu a la sessió de laboratori amb el que vau veure de Grafs a *Fonaments Matemàtics* (quadrimestre passat) repassat. Si voleu, a les transparències de teoria (p. 59-74) teniu un resum del que us caldria saber per fer aquesta sessió de laboratori. Aquestes transparències les veurem ràpidament en aquesta sessió de laboratori. Hi són per ajudar-vos a revisar el que us fa falta per a la sessió.

Com que volem fer programes que manipulin grafs, les explicacions teòriques i el codi en *pseudo-python* de les transparències s'han de concretar d'alguna manera. Mirarem de fer precisament això en aquesta sessió de laboratori.

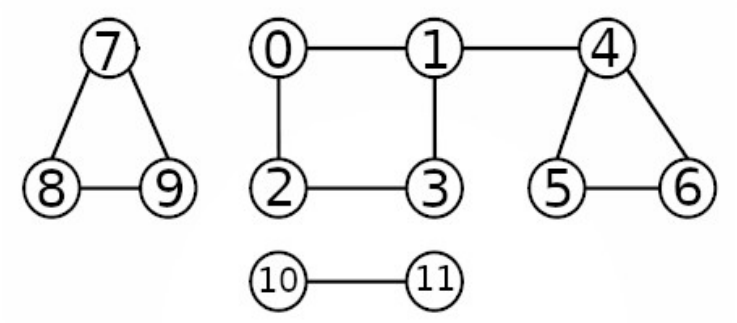
Tenim essencialment dues maneres d'implementar grafs: matriu d'adjacències i llista d'adjacències. Una implementació immediata en Python és la llista de llistes. En el primer cas cada llista té la mateixa mida, ja que una manera d'implementar una matriu `N×N` és amb una llista d'`N` llistes (una per cada vèrtex), cada una amb `N` elements (un per cada vèrtex). En aquesta sessió de laboratori, però, farem servir la implementació amb **llistes d'adjacència**. En aquest cas tindrem una llista amb `N` llistes (una per cada vèrtex), i cada llista tindrà com a elements els vèrtexos adjacents al vèrtex corresponent.

Els nostres programes necessitaran que els grafs, sobre els que els programes treballaran, es descriguin en alguna notació. De moment, farem servir fitxers de text, on els vèrtexos d'un graf amb `N` vèrtexos els etiquetarem amb els nombres `0,1,...,N-1`. Veiem uns exemples: El graf dirigit:



el representarem en un fitxer de text que indicarà el nombre de vèrtexos (10), el nombre d'arestes (14) i les arestes amb parells de nombres. El fitxer `exemple1.inp`, per tant, conté els nombres: 10 14 0 1 0 8 1 3 1 2 2 4 3 2 3 4 4 5 4 6 5 6 6 9 7 6 8 7 8 9 per descriure el graf de la figura.

El graf no dirigit



el representarem en un fitxer de text que indicarà el nombre de vèrtexos (12), el nombre d'arestes (12×2=24) i les arestes amb parells de nombres, considerant que *cal duplicar-les, ja que el graf és no dirigit* (l'aresta entre **i** i **j** es representa amb **i j**, però també amb **j i**). El fitxer `exemple2.inp`, per tant, contindria els nombres: 12 24 0 1 1 0 0 2 2 0 1 3 3 1 2 3 3 2 1 4 4 1 4 5 5 4 4 6 6 4 5 6 6 5 7 8 8 7 7 9 9 7 8 9 9 8 10 11 11 10. Fixem-nos, doncs, que estem representant un graf no dirigit com un graf dirigit on cada aresta connecta els dos vèrtexos en ambdues direccions.

Com llegirem aquests grafs? Farem servir la següent funció, que llegeix un graf en aquest format d'un fitxer **f**:

```
import sys
from pytokr import make_tokr

def llegir_graf():
    n = int(f_item())      # nombre de vertexos |V|
    m = int(f_item())      # nombre d'arestes |E|
    G = [[] for _ in range(n)]
    for i in range(m): # m parelles u,v: aresta u -> v
        u = int(f_item())
        v = int(f_item())
        G[u].append(v)
    return G
```

on, en el programa principal, caldrà fer:

```
fitxer = sys.argv[1]   # Hem de passar el nom del fitxer com a argument
                       # de la crida
with open(fitxer) as f:
    _, f_item = make_tokr(f)
    G = llegir_graf()
```

El programa que faci servir aquesta funció, diguem `prova_grafs.py`, s'haurà d'executar donant-li el nom del fitxer on està descrit el graf. Per exemple:

```
$ python3 prova_grafs.py exemple1.inp
```

Ara ja tenim un graf G, implementat amb llistes d'adjacència, amb el que podem treballar.

1.- *Mireu d'implementar el programa* `DFS(G,f)` *(amb la funció recursiva* `DFS_vertex`*). Partiu del codi en pseudo-python que teniu a les transparències, i proveu-lo amb els exemples que acabem de veure.*

2.- *Un cop hagueu fet l'exercici anterior, mireu d'implementar* `BFS(G,f)`*, que visiti en amplada tot el graf utilitzant el programa* `BFS_vertex(G,v,f)`*. Per a cada component connex podem guardar les distàncies (*`dist` *al codi en pseudo-python de la transparència; pot ser una llista o un diccionari) en una llista. Veure l'exemple més avall. Per implementar* `BFS(G,f)` *(amb la funció iterativa* `BFS_vertex(G,v,f)`*) podeu partir del codi en pseudo-python que teniu a les transparències. Proveu-lo amb els exemples que acabem de veure.*

FIxeu-vos que el codi de les transparències afegeix una funció com a paràmetre, per executar cada cop que visitem un vèrtex. És interessant veure en quin ordre es visiten els nodes, així que podem fer servir la funció:

```
def crea_accio():
    ordre = 0
    def ff(x):
        nonlocal ordre
        ordre += 1
        print(ordre,'=>',x)
    return ff
```

i fer programes senzills, per exemple, si anomenem aquest codi `prova_dfs.py`:

```
import sys
from pytokr import make_tokr

def llegir_graf():    # aquí vindria la funció que hem vist més amunt...

def dfs(G, accio):    # aquí vindria la funció que heu de fer vosaltres

def crea_accio():     # aquí vindria la funció que hem vist més amunt...

fitxer = sys.argv[1]  # Hem de passar el nom del fitxer com a argument
                      # de la crida, p.ex:
                      # python3 prova_dfs.py exemple1.inp
with open(fitxer) as f:
    _, f_item = make_tokr(f)
    G = llegir_graf()

dfs(G,crea_accio())
```

i el resultat d'executar-lo amb els exemples seria:

```
$ python3 prova_dfs.py exemple1.inp
1 => 0
2 => 1
3 => 3
4 => 2
5 => 4
6 => 5
7 => 6
8 => 9
9 => 8
10 => 7
```

```
$ python3 prova_dfs.py exemple2.inp
1 => 0
2 => 1
3 => 3
4 => 2
5 => 4
6 => 5
7 => 6
8 => 7
9 => 8
10 => 9
11 => 10
12 => 11
```

Si heu resolt el segon problema podem fer el programa `prova_bfs.py`:

```
import sys
from pytokr import make_tokr
from collections import deque

def llegir_graf():   # aquí vindria la funció que hem vist més amunt...

def bfs(G, accio):   # aquí vindria la funció que heu de fer vosaltres

def crea_accio():    # aquí vindria la funció que hem vist més amunt...

fitxer = sys.argv[1] # Hem de passar el nom del fitxer com a argument
                     # de la crida, p.ex:
                     # python3 prova_bfs.py exemple1.inp
with open(fitxer) as f:
    _, f_item = make_tokr(f)
    G = llegir_graf()

bfs(G,crea_accio())
```

i provar la cerca en amplada amb els exemples:

```
$ python3 prova_bfs.py exemple1.inp
1 => 0
2 => 1
3 => 8
4 => 3
5 => 2
6 => 7
7 => 9
8 => 4
9 => 6
10 => 5
```

```
$ python3 prova_bfs.py exemple2.inp
1 => 0
2 => 1
3 => 2
4 => 3
5 => 4
6 => 5
7 => 6
8 => 7
9 => 8
10 => 9
11 => 10
12 => 11
```

Si afegim el càlcul de les distàncies per a cada component connex del graf (on `dist` en el codi *pseudo-python* de la transparència seria un diccionari), tindríem:

```
distancies = bfs(G,crea_accio())
print(distancies)
```

i amb els exemples que hem vist el resultat seria:

`exemple1` (un sol component connex, per tant un sol diccionari de distàncies)

```
[{0: 0, 1: 1, 2: 2, 3: 2, 4: 3, 5: 4, 6: 3, 7: 2, 8: 1, 9: 2}]
```

`exemple2` (tres components connexos, per tant tres diccionaris de distàncies)

```
[{0: 0, 1: 1, 2: 1, 3: 2, 4: 2, 5: 3, 6: 3, 7: inf, 8: inf, 9: inf, 10: inf,
11: inf},
{0: inf, 1: inf, 2: inf, 3: inf, 4: inf, 5: inf, 6: inf, 7: 0, 8: 1, 9: 1,
10: inf, 11: inf},
{0: inf, 1: inf, 2: inf, 3: inf, 4: inf, 5: inf, 6: inf, 7: inf, 8: inf, 9: inf,
10: 0, 11: 1}]
```

a més d'escriure l'ordre en que visitem els vertexos, com hem vist més amunt.

Finalment, cal considerar que els *arbres* són en realitat grafs dirigits, per tant podríem preguntar-nos si l'estructura particular dels arbres és susceptible de simplificar els recorreguts en amplada i en profunditat que acabem d'estudiar. No cal pensar-hi més, perquè ja ho sabem. El *recorregut en preordre* d'un arbre és en realitat un DFS, i el *recorregut per nivells* d'un arbre és en realitat un BFS. Tots dos algorismes ja els vam veure a classe de teoria.

Sessió 3: Grafs: El mòdul **NetworkX** (networkx.org). Introducció a les Classes i Objectes

Les implementacions que hem vist a classe de grafs i algorismes sobre grafs són, diguem-ne, merament il·lustratives des d'un punt de vista pedagògic. Si es vol treballar *seriosament* amb grafs (i estudiants d'un grau d'IA haurien de voler) des del llenguatge de programació Python cal considerar el mòdul **NetworkX**[2], amb implementacions eficients de grafs i algorismes relacionats.

Aquest mòdul és molt senzill de fer servir. Podem recuperar els exemples que ja coneixem de la sessió anterior de laboratori i de la classe de teoria sobre l'algorisme de Dijkstra. Partim dels mateixos fitxers que ja coneixem (`exemple1.inp`, `exemple2.inp`, `exemple_dijkstra.inp`), codificant els grafs de la manera que ja vam descriure a classe.

Les funcions per llegir aquests grafs serien (cal fer `import networkx as nx`):

```python
def llegir_graf():
    # Suposo que tot són grafs dirigits
    G = nx.DiGraph()

    # nombre de vertexos |V|
    n = int(f_item())

    # Afegeixo nodes a partir d'un iterable
    G.add_nodes_from(range(n))

    # nombre d'arestes |E|
    m = int(f_item())

    # m parelles u,v: aresta u -> v
    for _ in range(m):
        u = int(f_item())
        v = int(f_item())
        G.add_edge(u,v)

    return G
```

```python
def llegir_graf_etiquetat():
    # Suposo que tot són grafs dirigits
    G = nx.DiGraph()

    # nombre de vertexos |V|
    n = int(f_item())

    # Afegeixo nodes a partir d'un iterable
    G.add_nodes_from(range(n))

    # nombre d'arestes |E|
    m = int(f_item())

    # m ternes u,v,w: aresta u --w--> v
    for _ in range(m):
        u = int(f_item())
        v = int(f_item())
        w = int(f_item())
        G.add_edge(u,v,weight=w)

    return G
```

Un cop fem `G = llegir_graf()` / `G = llegir_graf_etiquetat()`, ja podem fer servir l'***objecte*** `G`, que representa el graf llegit. Aquest objecte direm que és una ***instància*** de la ***classe*** `DiGraph` (que vol dir *Directed Graph*, o graf dirigit). És a dir, la classe ens ha servit de *plantilla* per crear l'objecte d'acord al que descriu la classe. Hem creat aquest objecte invocant la funció `nx.DiGraph()` (que a l'hora és el nom de la classe).

Aquest objecte té un ***estat***: La representació interna (implementació) del graf, que no sabem, *ni ens importa*, com és, i un ***comportament***, és a dir, una sèrie de ***mètodes*** (funcions) associats a l'objecte que podem fer servir per consultar informació, i/o

2 Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart, "*Exploring network structure, dynamics, and function using NetworkX*", in Proceedings of the 7th Python in Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008

modificar l'objecte (ja n'hem vist abans: **append** o **popleft** són mètodes de **deque**).

Podem demanar a l'objecte que ens proporcioni part de la informació que guarda en una forma que ens resulti còmode de tractar, per exemple **G.nodes** o **G.edges**. També podem utilitzar el comportament associat (els mètodes) a l'objecte per alterar aquesta representació interna (que, insistim, desconeixem, i així ha de ser), per exemple **G.add_nodes_from(...)** o **G.add_edge(...)**.

El mòdul també proporciona funcions amb les que processar aquest objecte, però que no formen part del comportament de l'objecte, tal com està descrit a la classe de la que és instància. Aquestes funcions ens permeten calcular informació interessant. Si ens centrem en els principals algorismes que hem estat veient, podem demanar que calculi un BFS a partir d'un cert vèrtex **v** (la nostra funció **BFS_vertex**): **nx.bfs_edges(G,source=v)** o un DFS a partir d'un cert vèrtex **v** (la nostra funció **DFS_vertex**): **nx.dfs_edges(G,source=v)** o l'algorisme de Dijkstra sobre un graf etiquetat a partir del vèrtex **v**: **nx.single_source_dijkstra(G,v)**.

És important fer notar la diferència entre *funció* i *mètode*. Diem que **add_nodes_from(...)** és un mètode perquè la invocació està associada a l'objecte: **G**.add_nodes_from(...), i diem que **nx.bfs_edges(...)** és una funció perquè l'objecte forma part dels seus arguments: nx.bfs_edges(**G**,source=0). Els mètodes els defineix la classe, i les funcions es defineixen de manera independent.

El mòdul **NetworkX** té definida una gran varietat de classes i funcions per treballar amb grafs. Podeu consultar la informació important sobre aquest mòdul a:

- https://networkx.org/documentation/stable/tutorial.html
- https://networkx.org/documentation/stable/reference/index.html

i els conceptes que hem introduït aquí (classe, objecte, etc.) els estudiarem a teoria.

Anem a veure alguns exemples. Donades les funcions de lectura dels grafs que hem vist més amunt, els programes que podem fer per reproduir els resultats de la sessió anterior són ara força senzills. Per exemple, per veure el recorregut BFS i DFS a partir del vèrtex 0, farem servir aquest programa, que anomenem **dfs_bfs_nx.py**:

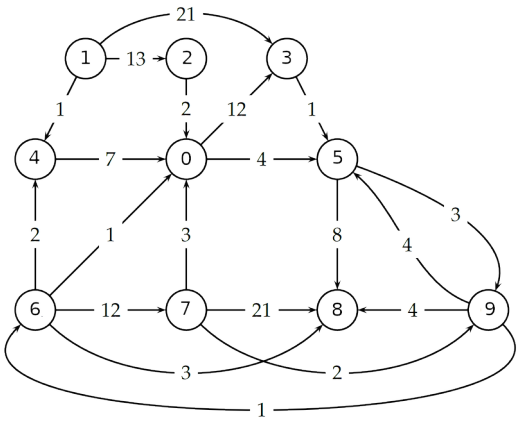
```
import sys
from pytokr import make_tokr
import networkx as nx

def llegir_graf():... # La funció que acabem de veure

fitxer = sys.argv[1]   # Hem de passar el nom del fitxer com a argument
                       # de la crida, p.ex: python3 dfs_bfs_nx.py exemple1.inp
with open(fitxer) as f:
    _, f_item = make_tokr(f)
    G = llegir_graf()
print(list(nx.dfs_edges(G,source=0)))  # Mostra les arestes visitades, i l'ordre
print(list(nx.bfs_edges(G,source=0)))  # en què es visiten, en tots dos casos
```

O bé, per aplicar l'algorisme de Dijkstra i conèixer els camins de cost mínim des del vèrtex inicial (el vèrtex 0) fins a qualsevol altre vèrtex podem fer el programa següent, que anomenarem **dijkstra_nx.py**:

```
import sys
from pytokr import make_tokr
import networkx as nx
def llegir_graf_etiquetat():... # La funció que acabem de veure

fitxer = sys.argv[1]  # Hem de passar el nom del fitxer com a argument
                      # de la crida, per exemple:
                      # python3 dijkstra_nx.py exemple_dijkstra.inp
with open(fitxer) as f:
    _, f_item = make_tokr(f)
    G = llegir_graf_etiquetat()
d,p = nx.single_source_dijkstra(G,source=0)  # d i p són diccionaris

print('--------------------')
for v in G.nodes:
    if v in d:
        print('cost',v,'=',d[v])
        print('camí',v,'=',p[v])
    else:
        print('cost',v,'= inf')       # per si v és inaccessible des de 0
        print('camí',v,'= None')
    print('--------------------')
```

Us passem pel Racó els programes propis i els que fan servir **NetworkX**, més els fitxers d'exemples (fitxer **Sessio3_Lab_PA2.zip**). Hem modificat lleugerament les funcions `BFS_vertex`, `DFS_vertex` i `dijkstra` que ja vam veure a classe, per poder obtenir les mateixes sortides que les funcions `dfs_edges`, `bfs_edges` i `single_source_dijkstra` del mòdul **NetworkX**, respectivament. Estudieu els programes **dfs_bfs.py** i **dijkstra.py** *detalladament*. Si heu entés el que s'ha explicat del tema de Grafs a classe no us hauria de costar gaire d'entendre.

1.- *Utilitzeu els programes que acabem de veure,* **dfs_bfs_nx.py** *i* **dijkstra_nx.py**, *i executeu-los sobre els exemples que ja coneixeu. Veieu quin és el resultat i compareu-lo amb el resultat que obtenim amb els algorismes que hem vist a classe (programes* **dfs_bfs.py** *i* **dijkstra.py**). *Seria un bon exercici que féssiu el mateix amb grafs inventats per vosaltres; per exemple, codifiqueu aquest graf etiquetat i trobeu els camins mínims des del vèrtex 0:*



2.- *Dins el fitxer* zip *mencionat més amunt teniu el programa* **`bellman_ford.py`**, *que podeu fer servir sobre l'exemple que vam veure a classe (transparència 88 de teoria), codificat en el fitxer* **`exemple_bellman_ford.inp`**. *Executeu el programa amb el fitxer d'exemple i compareu el resultat amb la taula de la transparència.*

3.- *Escriviu el programa* **`bellman_ford_nx.py`**, *seguint els exemples vistos en aquesta sessió, per calcular el mateix que el programa* **`bellman_ford.py`**, *però utilitzant el mòdul* **`NetworkX`**.

Sessió 4: Funcions vs. Objectes (i una mica de GUI). Classes a tot arreu!

La sessió d'avui pretén que compareu la versatilitat dels objectes respecte a fer la mateixa feina amb funcions. Veureu que el fet que les funcions puguin *capturar* variables (recordeu els entorns?) permet emular, *fins a cert punt*, el que podem fer amb objectes, però que la manera *natural*, si més no en Python, de treballar és utilitzar objectes (instàncies de classes). A més, com a *bonus*, treballarem una miqueta amb entorns gràfics, en concret amb el **`tkinter`**, que ja ve per defecte amb Python i no cal instal·lar-lo.

Igual que vam fer a la sessió 3, adjuntem via Racó un fitxer amb el codi que haureu de fer servir a aquesta sessió: **`Sessio4_Lab_PA2.zip`**.

Comencem amb un programa d'exemple molt senzill per convertir iardes a metres i metres a iardes, que fa servir **`tkinter`**. El programa es diu **`convert.py`**:

```
import tkinter

gui = tkinter.Tk()

label_meters = tkinter.Label(gui, text="Enter meters")
label_meters.grid(row=0, column=0)
e_meters = tkinter.Entry(gui)
e_meters.grid(row=0, column=1, columnspan=2)

label_or = tkinter.Label(gui, text="or")
label_or.grid(row=1, column=0)

label_yards = tkinter.Label(gui, text="Enter yards")
label_yards.grid(row=2, column=0)
e_yards = tkinter.Entry(gui)
e_yards.grid(row=2, column=1, columnspan=2)

def y2m():
    "convert yards to meters"
    e_meters.delete(0, tkinter.END)
    e_meters.insert(0, float(e_yards.get()) * 0.9144)

def m2y():
    "convert meters to yards"
    e_yards.delete(0, tkinter.END)
    e_yards.insert(0, float(e_meters.get()) / 0.9144)

b_up = tkinter.Button(gui, text="yards to meters   ^", command=y2m)
b_up.grid(row=1,column=1)
b_down = tkinter.Button(gui, text="v   meters to yards", command=m2y)
b_down.grid(row=1,column=2)

gui.mainloop()
```

Si executeu aquest programa veureu una finestra amb text, un parell de llocs per escriure i dos botons. Feu **`python3 convert.py`** i mireu què surt...

Primer definim l'aplicació gràfica amb **`tkinter.Tk()`**, i l'anomenem **`gui`**. Ara ja hauríem d'entendre aquesta expressió: Del mòdul **`tkinter`** fem servir la classe **`Tk`**, cridem al constructor sense paràmetres **`Tk()`** creant un objecte, instància de **`Tk`**, que lliguem a la variable **`gui`**. Ara associem a l'aplicació diversos objectes: etiquetes, o textos que apareixen dins la finestra, amb `tkinter.`**`Label(gui,...)`** , espais per introduir text amb `tkinter.`**`Entry(gui)`**, i botons, fent servir `tkinter.`**`Button(gui,...)`**. Un cop més **`Label`**, **`Entry`** i **`Button`** són classes (del mòdul `tkinter`) de les que cridem el constructor per instanciar-les, és a dir, crear objectes que s'han de comportar d'acord al que s'ha especificat a cada classe.

Fixeu-vos que quan cridem els constructors de les classes **`Label`**, **`Entry`** i **`Button`** cal dir a quina interficie gràfica volem que pertanyin aquests objectes. Per això els passem com a primer argument **`gui`**. També cal dir-los on volem que apareguin *dins* la finestra. Podem pensar la finestra com una taula amb un cert nombre de files i columnes. Així, després de instanciar les classes corresponents, invoquem en els objectes el mètode **`grid`**, amb una fila (`row`) i una columna (`column`). Eventualment també podem dir quin espai han d'ocupar amb `columnspan`. Compareu el codi amb el que veieu.

Cada classe definirà un estat i un comportament que depén de què estem representant amb la classe. Els textos o els espais per entrar text no requereixen més informació que la que ja veieu en la crida als corresponents constructors. En canvi, els botons necessiten saber què han de fer quan fem *click* a sobre. Això els ho indiquem amb el paràmetre `command`, amb el que diem al botó quina funció cal invocar quan el botó es prem. En aquest cas tenim dues funcions que manipulen les variables que referencien els objectes que representen els espais on entrar text, `e_meters` i `e_yards`, anomenades **`y2m`** i **`m2y`**. Aquestes funcions invoquen sobre les corresponents instàncies de **`Entry`** els mètodes **`delete`** (que esborra el contingut de l'espai per entrar text) i **`insert`** (escriu en el corresponent espai allò que se li passa com a paràmetre; en el nostre cas són nombres reals). Fixeu-vos que **`y2m`** i **`m2y`** no retornen res, ja que manipulen els objectes directament, modificant el seu estat via els mètodes esmentats. El **`0`** que passem als mètodes d'**`Entry`** es refereix a la posició del text dins el rectangle que tenim disponible (a l'inici), i `tkinter.`**`END`** significa "*fins que s'acabi el text*". La conversió de nombres a *strings* és automàtica.

Finalment, tot plegat es posa en marxa invocant el mètode **`mainloop`** (definit a la classe **`Tk`**) sobre l'objecte **`gui`**.

1.- *Invertiu una miqueta de temps analitzant el codi mentre feu servir l'aplicació, mirant d'entendre tot allò que us hem explicat.*

2.- *Modifiqueu el programa* **`convert.py`** *per convertir altres unitats de mesura: graus celsius i graus Farenheit, litres i centímetres cúbics, etc. Allò que vulgueu.*

Ara que us hem explicat una mica com hem d'entendre l'ús del mòdul **`tkinter`**, centrem-nos en els fitxers **`query_area_c.py`** i **`query_area_o.py`**. La part que pertoca la interfície gràfica d'usuari d'aquests programes és idèntica en tots dos programes, i a partir de l'exemple anterior l'hauríeu de poder entendre sense més problemes.

Centrem-nos doncs en la principal diferència entre tots dos programes: funcions vs. objectes. El programa **`query_area_c.py`** fa servir una funció amb diverses variables (`a_entry`, `n_entry` i `areas`) *capturades*, és a dir, que pertanyen a un entorn *pare*, i el programa **`query_area_o.py`** fa servir una classe de la que instancia un sol objecte, `q`.

Comencem amb **`query_area_c.py`**. Fixeu-vos en allò (`command`) que vinculem al botó `b`. En el cas del programa **`query_area_c.py`** és el resultat d'invocar la funció `make_query(n_entry, a_entry, "areas.json")`, que anomenem **`query`**. Aleshores fem:

```
query = make_query(n_entry, a_entry, "areas.json")
b = tkinter.Button(gui, text="Query area of country", command=query)
```

La funció **`query`** és el que retorna la invocació a `make_query(...)`. Si parem atenció al codi de `make_query` podem veure quelcom de bastant familiar, ja que es crea una funció local **`query`** que manipula variables que venen de l'entorn on s'ha creat aquesta funció. Veiem-ho amb una mica més de detall (i així repassem conceptes de PA1): Quan s'executa `make_query` es crea un entorn nou, amb les variables `a_entry`, `n_entry` i `areas` (també `datafile` i `df`, encara que no són rellevants a la discussió), i la funció **`query`**. Aquest entorn creat per l'execució de `make_query` hauria de desaparèixer quan s'acaba l'execució de `make_query`, però *no ho fa*, ja que precisament aquest entorn serà l'entorn *pare* de l'entorn creat quan executem **`query`**. La funció **`query`** fa servir les variables `a_entry`, `n_entry` i `areas`, que trobarà en l'entorn pare quan les necessiti. Com que no hi *assignem* res, no cal fer explícita la pertinença d'aquestes variables a un entorn pare amb **`nonlocal`**.

Això que tot just acabem d'explicar és una manifestació particular d'una característica de Python: ***Les funcions capturen el seu context lèxic*** (aquesta és la manera de dir això que us acabem d'explicar i que ja vam veure a PA1). Aquesta característica la tenen molts llenguatges de programació, p.ex. Javascript, Lisp, Scheme, Smalltalk, Ruby, etc. És per això que diem que les funcions a Python són ***closures*** (clausures o tancaments, si traduïm literalment). Les *closures* són ***funcions + context lèxic*** (capturat).

Tornant al nostre cas, l'entorn pare dels entorns d'execució de **`query`** no ha desaparegut (ja que ha estat *capturat*), per tant cada cop que executem **`query`** es crea un entorn nou d'execució, l'entorn pare del qual és l'entorn capturat, i trobem les mateixes variables `a_entry`, `n_entry` i `areas` amb els mateixos objectes referenciats. El lligam de les variables amb els objectes que referencien no canvia amb cada execució diferent de **`query`** (no hi ha assignacions a aquestes variables dins del codi de **`query`**). I quan

s'executa **query**? Cada cop que pitgem el botó `Query area of country`!!.

El fet que pugueu treballar amb funcions que capturen l'entorn on han estat creades té una conseqüència: Aquestes variables poden guardar el valor que tenen entre execucions de la funció, i per tant poden servir per emmagatzemar informació. Ho vau fer servir també a la pràctica de PA1!

Això que acabem de descriure... *no us evoca alguna cosa*? *No us recorda res*?

Recordeu, si us plau, el que vam explicar a classe de teoria sobre què és, essencialment, un objecte: *estat* + *comportament*. Així doncs, una funció, o millor dit, un *closure*, és *COM UN* objecte, on l'*estat* és l'entorn capturat en el moment de crear la funció i el *comportament* és la mateixa funció. A més, recordeu la pràctica de PA1, on vau fer servir que el mateix entorn pot ser capturat per *més d'una* funció! És a dir, aquest estat pot tenir més d'un comportament. Resumint:

	Objectes	***Closures***
Estat	`Variables d'Instància`	`Entorn capturat en crear la funció`
Comportament	`Mètodes`	`Funcions compartint aquest entorn`

En aquesta sessió tenim també el programa **query_area_o.py**, on fem exactament el mateix que amb el programa **query_area_c.py**, però amb classes i objectes, en lloc de fer servir només funcions/*closures*.

Comencem l'anàlisi al mateix lloc que hem començat abans, amb el paràmetre `command` del botó `b`. Ara tenim:

```
q = Query(n_entry, a_entry, "areas.json")
b = tkinter.Button(gui, text="Query area of country", command=q.query)
```

Creem un objecte **q**, *instància* de la classe **Query**, i l'acció que farà el botó `b` cada cop que el pitgem serà el mètode **q.query**. El mètode **q.query** és essencialment igual que la funció **query** de **query_area_c.py**. Ara, on és aquell *estat* que manipulava **query** gràcies a haver capturat l'entorn d'execució de **make_query**? És a dir, on són les variables `a_entry`, `n_entry` i `areas`?

Aquestes variables són ara *les variables d'instància de la classe* **Query**, de manera que definiran l'estat de cada instància d'aquesta classe. Fixeu-vos, però, que estan jugant el mateix paper que jugaven abans. L'objecte **q** del programa **query_area_o.py** és equivalent a, fa el mateix paper que, la funció **query** del programa **query_area_c.py**. En el cas de l'objecte **q** el comportament (l'únic mètode que té disponible) queda dins de l'objecte, **q.query**, mentre que la funció **query** el fa explícit.

Uns quants detalls de **query_area_o.py**: Les variables d'instància que s'han declarat a

la classe `Query` són públiques (podríem haver-les anomenat `__a_entry`, `__n_entry` i `__areas` si les volguéssim fer "*privades*"). Fixeu-vos també en l'equivalència entre crear el closure `query` (amb `make_query`) i instanciar la classe `Query`, estem passant els mateixos arguments, per tant fan essencialment la mateixa feina.

3.- *En el fitxer* **`Sessio4_Lab_PA2.zip`** *trobareu el programa* **`browse_areas_c.py`**. *Aquest és un programa una mica més complicat que els programes que hem estat comentant en el text, però no massa més. Aquest programa està fet amb* ***closures***. *Executeu-lo mentre analitzeu el codi, per entendre bé què fa.*

4.- *Feu un programa* **`browse_areas_o.py`** *que faci el mateix que* **`browse_areas_c.py`**, *però amb classes i objectes. Imiteu el que heu vist comparant els programes* **`query_area_c.py`** *i* **`query_area_o.py`**.

Alguns comentaris sobre el fitxer **`area.json`** que fem servir en aquesta sessió. Aquest fitxer està en un format de text (no binari) i en una notació estàndard per guardar i recuperar objectes (*Javascript Object Notation*, `en.wikipedia.org/wiki/JSON`, `www.json.org/json-es.html`). Si obriu el fitxer veureu que el format és idèntic al d'un diccionari Python. En els programes d'aquesta sessió en tenim prou, per recuperar el contingut del fitxer, amb fer servir la funció **`load`** del mòdul **`json`** (d'aquí el `from json import load`). Així doncs, la variable **`area`** és un diccionari amb *strings* com a claus (noms de països) i nombres reals com a valors (la superficie en km quadrats del país corresponent).

Sessió 5: Classes i Objectes (I)

Ara que ja hem vist els conceptes de classe, instància, mètode, variable d'instància, etc, podem començar a re-visitar les estructures de dades noves que hem vist (*heaps*, arbres, arbres binaris, grafs) per implementar-les *com cal*, és a dir, amb classes.

Un exemple seria el tipus **`Racional`** (transparències teoria 6-15). Podríem fer la classe **`Racional`** amb el codi que vam veure a classe de teoria:

```
class Racional:

    def __init__(self,n,d):
        """ Retorna el nombre racional n/d, suposant n i d són enters i d != 0 """
        assert(d != 0)
        g = gcd(n, d)
        self.__numerador   = n // g
        self.__denominador = d // g

    # Mètodes consultors / 'getters'
    def numerador(self):
        """ Retorna el numerador del nombre racional r amb valor absolut més petit """
        return self.__numerador

    def denominador(self):
        """ Retorna el denominador del nombre racional r amb valor > 0 més petit """
        return self.__denominador

    # Operacions amb racionals
    def suma_racional(self,q):
        n1 = self.numerador() * q.denominador()
        n2 = q.numerador() * self.denominador()
        d  = self.denominador() * q.denominador()
        return Racional(n1+n2,d)

    def producte_racional(self,q):
        n = self.numerador() * q.numerador()
        d = self.denominador() * q.denominador()
        return Racional(n,d)

    def str_racional(self):
        n = self.numerador()
        d = self.denominador()
        return str(n) if d == 1 else f"{n}/{d}"

    def igual(self,q):
        n1 = self.numerador() * q.denominador()
        n2 = q.numerador() * self.denominador()
        return (n1 == n2)
```

Ja podem fer :

```
def nombre_harmonic_exacte(n):
    s = Racional(0,1)
    for k in range(1,n+1):
        s = s.suma_racional(Racional(1,k))
    return s.str_racional()
```

Veiem ara l'estructura de dades que vam estudiar a la primera sessió: El tipus de dades **MinHeap**. Us presentem la part que implementa les operacions públiques:

```
class MinHeap:

    def __init__(self, items=[]):
        # __heapify crea i inicialitza la variable d'instància __lst
        self.__heapify(items)

    # Operacions públiques del min-heap

    def inserir(self, x):
        (self.__lst).append(x)
        self.__surar(len(self.__lst)-1)

    def obtenir_min(self):
        # Pre: lst no pot ser buida
        assert not self.buit()
        primer = self.__lst[1]
        darrer = (self.__lst).pop()
        if len(self.__lst) > 1:
            self.__lst[1] = darrer
            self.__enfonsar(1)
        return primer

    def buit(self):
        return self.__lst == [None]

    # Operacions privades del min-heap

    def __heapify(self,ll=[]):
        # ...

    def __surar(self,i):
        # ...

    def __enfonsar(self,i):
        # ...
```

1.- *Completeu la classe* **MinHeap** *amb les funcions privades que falten. Us heu de fixar en el codi que ja teniu per a aquestes operacions, i considerar els canvis necessaris (p.ex,* `__surar` *i* `__enfonsar` *no necessiten cap paràmetre llista) a partir de la comparació que heu de fer de les operacions públiques amb les funcions que ja coneixeu.*

Ara seria oportú fer una classe **MaxHeap**, però triarem una opció més abstracta. Farem una classe **Heap** més general que el que hem vist, ja que *la mateixa classe* ens podrà proporcionar objectes que seran *min-heaps*, i objectes que seran *max-heaps*. La idea és que podrem crear *heaps* de la següent manera:

```
>>> from operator import lt, gt  # operadors de comparació < i > com a funcions
>>> from heap import *           # tindrem la classe Heap en el fitxer heap.py
>>> h1 = Heap()                       # h1 és min-heap buit
>>> h2 = Heap(items=[12,45,3,67])     # h2 és min-heap amb els elements d'items
>>> h3 = Heap(cmp=gt)                 # h3 és max-heap buit
>>> h4 = Heap(items=[12,45,3],cmp=gt) # h4 és max-heap amb els elements d'items
```

La classe la podem escriure:

```
from operator import lt, gt

class Heap:

    def __init__(self, items=[], cmp=lt): # per defecte tenim un min-heap
        assert cmp == lt or cmp == gt     # només acceptem aquestes funcions
        self.__cmp = cmp
        # __heapify crea i inicialitza la variable d'instància __lst
        self.__heapify(items)

    # Operacions públiques del heap

    def inserir(self, x):
        (self.__lst).append(x)
        self.__surar(len(self.__lst)-1)

    def obtenir(self):
        # Pre: lst no pot ser buida
        assert not self.buit()
        primer = self.__lst[1]
        darrer = (self.__lst).pop()
        if len(self.__lst) > 1:
            self.__lst[1] = darrer
            self.__enfonsar(1)
        return primer

    def buit(self):
        return self.__lst == [None]

    # Operacions privades del heap

    def __heapify(self,ll=[]):
        # ...

    def __surar(self,i):
        # ...

    def __enfonsar(self,i):
        # ...
```

2.- *Completeu ara la classe* **Heap**, *tenint en compte que la funció de comparació dels elements del* **Heap** *és a* `self.__cmp`.

Sessió 6: Classes i Objectes (II)

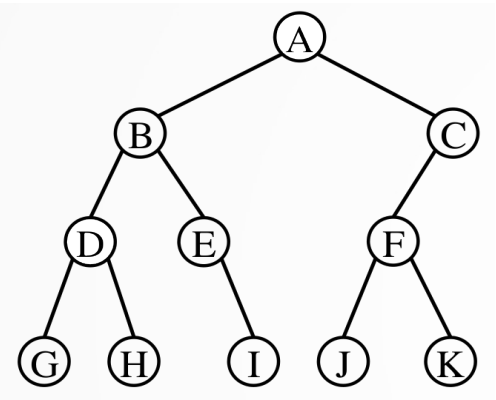
La sessió anterior vam convertir la implementació dels *heaps*, que vam veure a la primera sessió, en classes. En aquesta sessió la idea és acabar la feina i convertir la implementació que vam estudiar dels arbres, els arbres binaris i els grafs en classes que puguem instanciar i fer servir com a objectes Python. De moment, us demanem que implementeu l'arbre binari:

1.- *Convertiu en classe la implementació d'arbre binari que vam estudiar, considerant que la implementació d'algunes operacions no tenen per què ser implementades igual. La classe ha de tenir les operacions públiques (ho farem en anglés):*

```
class BinTree:

    # Constructor
    def __init__(self,v=None,left=None,right=None):
        """
        Pre: left is a BinTree, or None; right is a BinTree, or None
        Requirement: If the value v is None, so must be both children.
        """

    # Getters
    def get_root(self):
        """
        Pre: It is assumed that the BinTree is NOT empty
        returns the value at the root of the BinTree
        """
    def get_left(self):
        """
        Pre: It is assumed that the BinTree is NOT empty
        returns the left child of the BinTree
        """
    def get_right(self):
        """
        Pre: It is assumed that the BinTree is NOT empty
        returns the right child of the BinTree
        """

    # Setters
    def set_root(self,v):
        """
        changes the value at the root of the BinTree
        """
    def set_left(self,left):
        """
        Pre: left is a BinTree and the BinTree is not empty
        changes the left child of the BinTree
        """
```

```
    def set_right(self,right):
        """
        Pre: right is a BinTree and the BinTree is not empty
        changes the right child of the BinTree
        """

    # Other operations
    def empty(self):
        """
        returns True if the BinTree is empty, False in other case
        """
    def leaf(self):
        """
        returns True if the BinTree is a leaf, False if not.
        """

    # Traversals
    def preorder(self):
        """
        returns a list with the elements of the BinTree, ordered
        as is specified in the definition of the pre-order traversal.
        """
    def postorder(self):
        """
        returns a list with the elements of the BinTree, ordered
        as is specified in the definition of the post-order traversal.
        """
    def inorder(self):
        """
        returns a list with the elements of the BinTree, ordered
        as is specified in the definition of the in-order traversal.
        """
    def levelorder(self):
        """
        returns a list with the elements of the BinTree, ordered
        as is specified in the definition of the levels-order traversal.
        """
```

Un cop tinguem implementada aquesta classe (en un fitxer anomenat, p.ex. **bintree.py**), ja la podem fer servir. Per exemple, podem construir l'arbre d'exemple de les transparències dels recorreguts:



i comprovar que els quatre recorreguts diferents de l'arbre són correctes:

```
>>> from bintree import *
>>> g = BinTree('G')
>>> h = BinTree('H')
>>> i = BinTree('I')
>>> j = BinTree('J')
>>> k = BinTree('K')
>>> d = BinTree('D',left=g,right=h)
>>> e = BinTree('E',right=i)
>>> f = BinTree('F',left=j,right=k)
>>> b = BinTree('B',left=d,right=e)
>>> c = BinTree('C',left=f)
>>> arbre = BinTree('A',left=b,right=c)
>>>
>>> arbre.preorder()
['A', 'B', 'D', 'G', 'H', 'E', 'I', 'C', 'F', 'J', 'K']
>>> arbre.postorder()
['G', 'H', 'D', 'I', 'E', 'B', 'J', 'K', 'F', 'C', 'A']
>>> arbre.inorder()
['G', 'D', 'H', 'B', 'E', 'I', 'A', 'J', 'F', 'K', 'C']
>>> arbre.levelorder()
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K']
```

Podeu mirar d'aprofitar aquesta classe per fer alguns dels exercicis de la col·lecció de problemes. De fet, l'exercici 12 ja l'heu fet si implementeu aquesta classe (els *setters*!)

2.- *Feu servir aquesta classe per programar una funció senzilla, no destructiva, que retorni la inversió d'un arbre binari. La inversió d'un arbre binari és un arbre amb la mateixa arrel i amb la inversió de tots dos fills canviada de lloc (la inversió del fill dret com a fill esquerre, i la inversió del fill esquerre com a fill dret). La capçalera de la funció pot ser* `def inversio(t):` *on* `t` *podem suposar que és un* **`BinTree`**.

Sessió 7: Classes i Objectes (III)

Ara li toca el torn als grafs. Mirarem de fer una classe amb la representació dels grafs que ja vam veure, amb mètodes que corresponguin a les funcions que ja vam estudiar.

1.- *Convertiu en classe la implementació de graf que vam estudiar, considerant que algunes operacions no tenen per què ser implementades igual. La classe ha de tenir les operacions públiques:*

```
from collections import deque     # El BFS necessita deque
from heap import Heap             # La classe que vam fer a la sessió 5!
from operator import lt

class GrafDirigitEtiquetat:
    # Feu una implementació simple (la més simple possible?)
    # amb llistes d'adjacència.
    # =====> Recordeu la que ja vam fer servir
    # Els vèrtexos es representen amb nombres: n vèrtexos => nombres 0...n-1

    def __init__(self, n = 0):
        """
        n és el nombre de vertexos: 0 <= n
        No deixem afegir vèrtexos un cop creat el graf
        """
        self._vertexos = [None] * n # Llista de vèrtexos
        for i in range(n):
            self._vertexos[i] = []  # Inicialment les llistes d'adjacència
                                    # estan buides.

    def afegir_aresta(self, src, dst, weight):
        """
        Pre: 0 <= src,dst < nombre de vèrtexos del graf
        Afegeix una aresta src --weight--> dst
        """

    def veins_vertex(self, u):
        """
        Pre: 0 <= u < nombre de vertexos del graf
        Retorna la llista de veins associada al vèrtex u
                (sense els costos de les arestes, només els veins)
        """

    def nombre_vertexos(self):
        """
        Retorna el nombre de vèrtexos del graf
        """

    def bfs_vertex(self,s):
        """
        Pre: 0 <= s < nombre de vèrtexos del graf
        Retorna la llista d'arestes, en l'ordre en que s'han recorregut.
        """
```

```
    def dfs_vertex(self,s):
        """
        Pre: 0 <= s < nombre de vèrtexos del graf
        Retorna la llista d'arestes, en l'ordre en que s'han recorregut.
        """

    def dijkstra(self,s):
        """
        Pre: 0 <= s < nombre de vèrtexos del graf
        Retorna una llista amb els costos d'anar al node des de s
        i una llista de llistes amb els camins corresponents
        """

    def cami_minim(self,u,v):
        """
        Pre: 0 <= u,v < nombre de vèrtexos del graf
        Retorna el cost d'anar des de u a v, i una llista amb el camí
        """

    def bellman_ford(self,s):
        """
        Pre: 0 <= s < nombre de vèrtexos del graf
        Retorna una llista amb els costos d'anar al node des de s
        i una llista de llistes amb els camins corresponents
        """

    def _calcul_camins(self,s,prev):
        """
        Funció auxiliar per a dijkstra i bellman_ford, que calcula els camins,
        donada la llista prev, els nodes previs per a cada node.
        """
```

Si poseu aquest codi (ja implementat) en un fitxer anomenat **graf.py**, ara podríem escriure els programes que ja vam veure a la sessió 3, per fer proves amb grafs, de la següent manera:

Fitxer **dijkstra_grafs_classes.py**:

```
import sys
from pytokr import make_tokr
from graf import GrafDirigitEtiquetat

# Lectura graf DIRIGIT amb llistes d'adjacència, i els nodes
# són nombres enters 0...N-1 (si el graf té N nodes)
def llegir_graf_etiquetat():... # Cal modificar la funció que vam veure...

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 = G.dijkstra(0)
print('--------------------')
```

```
for v in range(G.nombre_vertexos()):
    print('cost',v,'=',d[v])
    print('camí',v,'=',p[v])
    print('--------------------')
```

i de manera similar podríem fer el fitxer **bellman_ford_grafs_classes.py**. Fixem-nos que si volem provar el dfs i el bfs no és estrictament necessari tenir un graf dirigit *etiquetat*. Amb un graf dirigit en tenim prou:

Fitxer **dfs_bfs_grafs_classes.py**:

```
import sys
from pytokr import make_tokr
from graf import GrafDirigit

# Lectura graf DIRIGIT amb llistes d'adjacència, i els nodes
# són nombres enters 0...N-1 (si el graf té N nodes)
def llegir_graf():...  # Cal modificar la funció que vam veure...

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()

arestes_dfs = G.dfs_vertex(0)
print(arestes_dfs)
arestes_bfs = G.bfs_vertex(0)
print(arestes_bfs)
```

Cal revisar les funcions de lectura de grafs:

```
from graf import GrafDirigit

def llegir_graf():
    # nombre de vertexos |V|
    n = int(f_item())
    # nombre d'arestes |E|
    m = int(f_item())

    # Suposo que tot són grafs dirigits
    G = GrafDirigit(n)

    # m parelles u,v: aresta u -> v
    for _ in range(m):
        u = int(f_item())
        v = int(f_item())
        G.afegir_aresta(u,v)

    return G
```

```
from graf import GrafDirigitEtiquetat

def llegir_graf_etiquetat():
    # nombre de vertexos |V|
    n = int(f_item())
    # nombre d'arestes |E|
    m = int(f_item())

    # Suposo que tot són grafs dirigits
    G = GrafDirigitEtiquetat(n)

    # m ternes u,v,w: aresta u --w--> v
    for _ in range(m):
        u = int(f_item())
        v = int(f_item())
        w = int(f_item())
        G.afegir_aresta(u,v,w)

    return G
```

Un cop implementada la classe **GrafDirigitEtiquetat**, voldríem implementar **GrafDirigit** i **GrafNoDirigit**. I volem també reaprofitar tant de codi com sigui possible de la classe ja implementada. **Farem que aquestes classes siguin subclasses de GrafDirigitEtiquetat**.

Hi ha un petit detall que cal tenir en compte. Recordeu que vam explicar que s'hereta (quasi bé) tot, fins i tot funcions que ens poden NO interessar. Per exemple, els mètodes `dijkstra`, `cami_minim` i `bellman_ford` no tenen sentit en un **`GrafNoDirigit`**. Podem no heretar un atribut públic? NO, no podem. El que sí podem fer és sobreescriure els mètodes en qüestió per a que generin un error. Per exemple, si **`G`** és instància de **`GrafNoDirigit`**, invocar **`G.`**`dijkstra(u)`, **`G.`**`cami_minim(u,v)` o **`G.`**`bellman_ford(u)` hauria de generar un error. Hi ha diverses maneres de fer això en Python, nosaltres hem triat una manera que no afegeix conceptes nous a l'explicació.

2.- *Implementeu les classes* **`GrafDirigit`** *i* **`GrafNoDirigit`***. seguint l'especificació següent. Fixeu-vos que* ***només*** *cal sobreescriure* `afegir_aresta`*.*

```
class GrafDirigit(GrafDirigitEtiquetat):
    def __init__(self, n = 0):
        """
        n és el nombre de vertexos: 0 <= n
        No deixem afegir vèrtexos un cop creat el graf
        """
        super().__init__(n)

    def afegir_aresta(self, src, dst):
        """
        Pre: 0 <= src,dst < nombre de vèrtexos del graf
        Afegeix una aresta src ---> dst
        """

class GrafNoDirigit(GrafDirigitEtiquetat):
    def __init__(self, n = 0):
        """
        n és el nombre de vertexos: 0 <= n
        No deixem afegir vèrtexos un cop creat el graf
        """
        super().__init__(n)

    def afegir_aresta(self, src, dst):
        """
        Pre: 0 <= src,dst < nombre de vèrtexos del graf
        Afegeix dues arestes: src ---> dst i dst --> src
        """


    def dijkstra(self, s):
        assert False, "missatge d'error"

    def cami_minim(self,u,v):
        assert False, "missatge d'error"

    def bellman_ford(self,s):
        assert False, "missatge d'error"
```

Hem deixat obert el missatge d'error a ensenyar, però podria ser quelcom similar a: "*No té sentit fer un dijkstra d'un graf no dirigit. Prova a fer un BFS*".

Comentem uns quants detalls:

- A l'`__init__` de **`GrafDirigitEtiquetat`** fem servir la variable d'instància `_vertexos`, i no la fem privada (en aquest cas seria `__vertexos` , amb doble guió baix), com hem dit sovint a classe que calia fer. La raó és que en aquest cas hem volgut fer servir l'herència, i ja vam dir a classe de teoria que els atributs amb un nom que *comença* amb doble guió baix no s'hereten (excepte si el nom *acaba* amb doble guió baix).

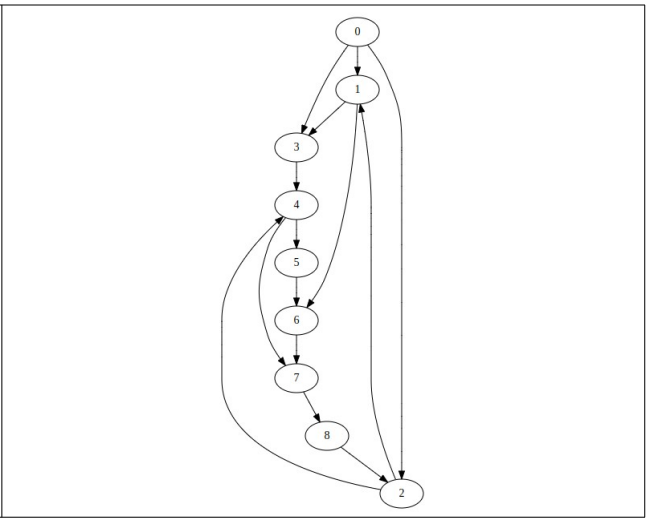
- Com és possible que només calgui sobreescriure `afegir_aresta`? Això és per a què puguem aprofitar la implementació dels grafs dirigits etiquetats per fer grafs dirigits i no dirigits. Ho podem fer senzillament assignant el mateix pes constant a totes les arestes, per exemple el nombre 1. Podem aleshores aprofitar la infraestructura dels grafs dirigits etiquetats sense més problemes.

3.- *Amb els fitxers de prova que ja us vam passar (recordeu el fitxer* **`Sessio3_Lab_PA2.zip`***), podeu comparar el resultat d'executar els programes que fan servir* **NetworkX** *amb els programes, que acabem de veure, que fan servir les classes que hem implementat. Òbviament, el resultat hauria de ser idèntic.*

Sessió 8: Dibuixant Grafs

Hi ha utilitats que, a partir d'una descripció purament textual d'un graf, generen un dibuix (en algun dels formats típics: png, jpg, etc). Un d'aquests formats de descripció textual d'un graf és el format **.dot**. Aquest format és interpretat després per un programa anomenat **graphviz** per fer una representació gràfica dels grafs (`graphviz.org`). És un format molt senzill, i s'entén de seguida amb exemples. Recordeu el fitxer **`exemple_dijkstra.inp`**? Doncs en format .dot és com veieu al costat esquerre de la figura, i el que dibuixa el graphviz és el que veieu (en petit) a la dreta. Fixeu-vos que els costos a les arestes no hi són.

```
digraph "" {
      0 -> 1      [weight=20.0];
      0 -> 2      [weight=20.0];
      0 -> 3      [weight=1.0];
      1 -> 3      [weight=4.0];
      1 -> 6      [weight=4.0];
      2 -> 1      [weight=3.0];
      2 -> 4      [weight=3.0];
      3 -> 4      [weight=5.0];
      6 -> 7      [weight=4.0];
      4 -> 5      [weight=9.0];
      4 -> 7      [weight=2.0];
      5 -> 6      [weight=10.0];
      7 -> 8      [weight=1.0];
      8 -> 2      [weight=7.0];
}
```



Donada una descripció d'un graf en format .dot, teniu diverses maneres de fer servir el graphviz. La més senzilla seurament és fer servir la web `webgraphviz.com`. També podeu fer servir Xdot (després d'instal·lar-lo, **`pip3 install xdot`**) des de la línia de comandes: **`xdot <nom fitxer de text amb la descripció .dot>`**.

Ara, donada una descripció del graf en el *nostre* format, que ja hem vist en altres sessions de laboratori, podem fer servir **NetworkX** (sessió 3) per generar automàticament descripcions .dot dels grafs. El llegim en una instància de `networkx.DiGraph()`, anomenem-lo `gr`, i invoquem la funció:

`nx.drawing.nx_agraph.write_dot``(gr, <nom del fitxer.dot>)`

Podem tornar a l'exemple de fer servir **NetworkX** per calcular els camins mínims amb l'algorisme de Dijkstra, i acabar generant un fitxer .dot amb la representació del graf (fixem-nos que no fem servir `pytokr` per llegir les dades):

```
gr = nx.DiGraph()
with open("exemple_dijkstra.inp") as in_gr:
    in_gr.readline() # skip number of nodes
    in_gr.readline() # skip number of edges
    for line in in_gr:
        s, t, w = line.split() # read edges in and add to graph
        gr.add_edge(int(s), int(t), weight = float(w))
```

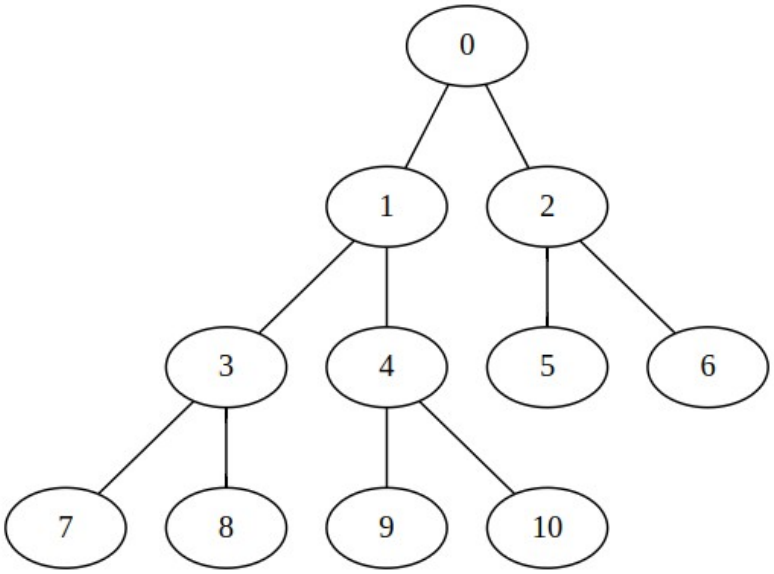
```
# Compute shortest paths and print them
sh_paths = nx.shortest_path(gr, source = 0, weight = "weight")
for v in sh_paths:
    print("From 0 to", v, ":", sh_paths[v])
# Save the graph in .dot format
nx.drawing.nx_agraph.write_dot(gr, "exemple_dijkstra.dot")
```

Hem fet **import** networkx **as** nx. El fitxer generat **exemple_dijkstra.dot** el podem obrir amb qualsevol editor de text. El seu contingut és el de la figura de més amunt, i s'ha fet servir a webgraphviz.com per generar el dibuix del graf.

Aquesta utilitat del mòdul **NetworkX** la podem fer servir per visualitzar els grafs que aquest pot generar per defecte. Una col·lecció, força nombrosa, dels grafs que **NetworkX** és capaç de generar la podeu trobar aquí:
networkx.org/documentation/stable/reference/generators.html
Per exemple, podem generar un *full rary tree* (arbres semi-complets amb un cert nombre de nodes) i visualitzar-lo fent simplement:

```
g = nx.full_rary_tree(2, 11)  # semi-complete binary tree with 11 nodes
nx.drawing.nx_agraph.write_dot(g, "rary_2_11.dot")
```

i el resultat, fitxer .dot, es pot visualitzar:



Malgrat això, la interfície de **NetworkX** pel format .dot és bastant senzilla. Podem utilitzar graphviz des de Python mateix amb el mòdul **PyGraphviz** (mireu com instal·lar-lo a pygraphviz.github.io) i manipular grafs directament des d'aquest mòdul. Hi ha altres mòduls similars, **PyGraphviz** no és l'únic (veieu graphviz.org/resources/#python).

Veiem-ne un exemple: Aquest programa llegeix directament el fitxer .dot i crea el graf corresponent. Després demanem informació sobre el graf i la proporciona correctament:

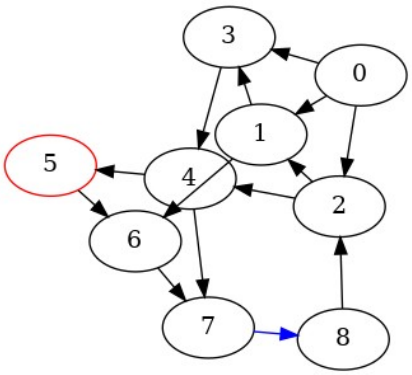
```
import pygraphviz as pgv

gr = pgv.AGraph("exemple_dijkstra.dot")

print("Directed? ", gr.is_directed())
for u in gr.nodes():
    'both nodes and edges (and also graphs) have attribute dicts'
    for v in gr.out_neighbors(u):
        e = gr.get_edge(u, v)
        for a in e.attr:
            print(u, v, a, e.attr[a])
```

En aquest bocinet de codi demanem que s'escriguin els *atributs* que tenen les arestes del graf, tot i que en aquest exemple només hi ha els pesos (*weights*). Tant les arestes com el pesos surten llistats pel terminal.

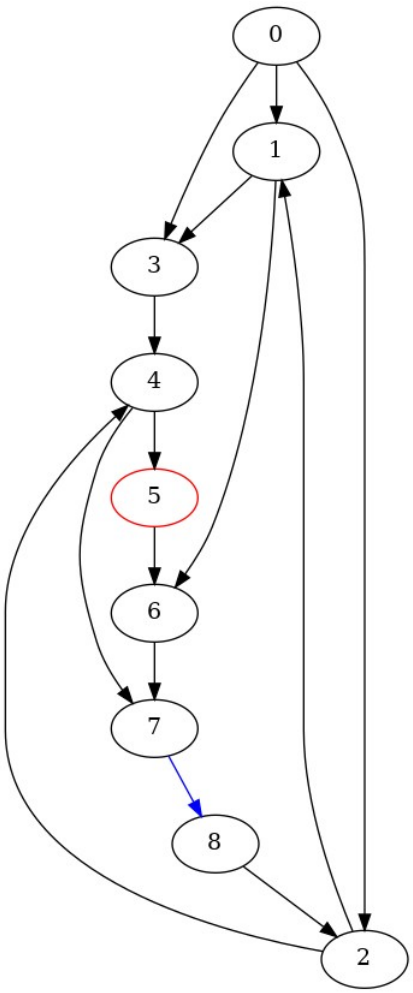
Veiem aquest altre exemple, on s'assignen colors a un vèrtex i a una aresta, i a més es dibuixa el graf de tres maneres diferents, seguint tres *layouts* diferents (*neato*, *dot*, *circo*, respectivament).

```
gr = pgv.AGraph("exemple_dijkstra.dot")

n = gr.get_node('5')
n.attr["color"] = "red"
e = gr.get_edge('7', '8')
e.attr["color"] = "blue"

for alg in ("dot", "neato", "circo"):
    'different layout algorithms to decide node positions'
    gr.layout(prog = alg) # default is neato
    gr.write("ex_" + alg + ".dot")
    gr.draw("ex_" + alg + ".png")
```

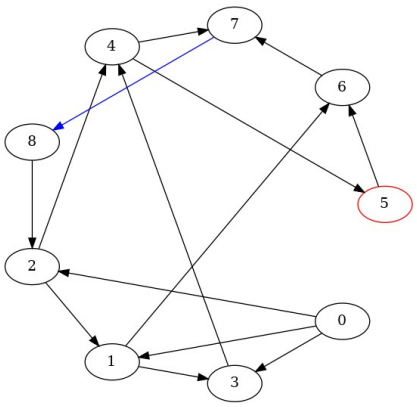
Amb els atributs es pot jugar força, tant de vèrtexos, arestes com del mateix graf. Podem assignar-hi colors, formes, etc. Per dibuixar el graf tenim el *layout*, que determina la forma que tindrà el graf final. En aquest exemple, fixeu-vos que per a cada *layout* es generen un fitxer .dot i un altre fitxer .png. Si obriu amb un editor de text els fitxers .dot veureu que apareixen les coordenades pels vertexos, i una representació numérica que descriu com es dibuixen les arestes (*splines*).



Neato



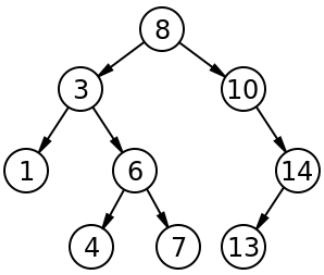
Dot



Circo

Recomanació: per defecte fer servir sempre el *layout* dot. Podeu obtenir la informació completa sobre els mètodes de la classe **`AGraph`** de **`PyGraphviz`** en aquesta adreça: `pygraphviz.github.io/documentation/stable/reference/agraph.html`

1.- Mireu de dibuixar **`BinTree`**'s. Feu un programa que tingui una instància de **`BinTree`** com a paràmetre i que generi un fitxer **`.png`** amb el dibuix de l'arbre. Proveu-ho amb el següent problema: Un **Arbre Binari de Cerca** (**BST**, de *Binary Search Tree*) és un arbre binari tal que tots els elements del fill esquerre d'un node són més petits que el node, i tots els elements del fill dret del node són més grans que el node. I això es verifica *per a tot node* de l'arbre. Per exemple, aquest arbre binari:



és un BST.

Fes una funció que, donada una llista amb el recorregut en pre-ordre d'un BST, retorni el BST reconstruït i generi un fitxer png amb un dibuix de l'arbre[3]:

```
def bst_a_partir_de_preordre(lst):
    """
    Pre: lst és el recorregut en pre-ordre d'un BST.
         Suposem que tots els elements d'lst són diferents
    Retorna: una instància de BinTree i genera un fitxer amb un
             dibuix de l'arbre
    """
```

2.- Mireu de fer un programa que llegeixi un graf dirigit etiquetat i el dibuixi. Ara, donats dos vertexos qualsevol, fer que el programa calculi el camí mínim entre aquests dos vertexos i dibuixi el graf pintant d'un altre color els nodes i les arestes del camí.

3 Aquest problema (sense generar el fitxer amb el dibuix) és el primer problema del parcial del curs 2021-22, que teniu resolt al Racó.

Sessió 9: Estructures de Dades Dinàmiques: Cues

En aquesta sessió de laboratori posarem en pràctica l'ús de les Estructures de Dades Dinàmiques (EDD), en concret, en aquesta sessió farem problemes sobre cues. Teniu els fitxers amb el codi de les EDD, ja que us els vam fer arribar via Racó.

Problema 1 (*Jutge P90861*):

Simuleu el comportament de les cues d'un supermercat: Inicialment, hi ha n cues (1, 2, ..., n), cadascuna amb els seus clients. Després, es poden produir dos successos diferents:

- Un client arriba a una cua: Si la cua està entre 1 i n, el client entra a la cua. Altrament, el succés s'ignora.
- Un client surt d'una cua: Si la cua està entre 1 i n, i la cua no és buida, el *primer* client de la cua surt. Altrament, el succés s'ignora.

Entrada: L'entrada (a l'*standard input*) comença amb el nombre de cues n (un natural estrictament positiu). Segueixen n línies, una per cua, cadascuna amb els seus clients (una paraula) en l'ordre en què han arribat a la cua. Després ve una línia en blanc i la descripció d'una sèrie de successos, un per línia: la paraula "ENTRA" seguida del client (una paraula) i de la cua (un enter); o bé la paraula "SURT" seguida de la cua (un enter).

Sortida: Primer, escriviu els noms dels clients que surten de les cues, en l'ordre en què ho fan. Després, escriviu el contingut final de les n cues, cadascuna en l'ordre en què sortirien els clients. Seguiu el format de l'exemple:

Entrada:	**Sortida**:
4 Cristina Tomas Francesc Damia Domenec Teresa Toni Carles SURT 1 SURT 1 ENTRA Amalia 4 SURT 2 SURT 1 ENTRA Leo 1 ENTRA Maria 3 SURT 4 SURT 4 SURT 3 ENTRA Carme 4 SURT 2 SURT -1 SURT 2	SORTIDES -------- Cristina Tomas Francesc Teresa Toni Maria Damia Domenec CONTINGUTS FINALS ----------------- cua 1: Leo cua 2: cua 3: cua 4: Carles Amalia Carme

1.- Resoleu el problema 1. Feu servir el fitxer **`cua.py`** amb la implementació de les cues que vam veure a classe.

Problema 2 (*Jutge P83396*):

Simuleu el comportament de les cues d'un supermercat: Inicialment, hi ha n cues (1, 2, ..., n), cadascuna amb els seus clients. Després, es poden produir dos successos diferents:

- Un client arriba a una cua: Si la cua està entre 1 i n, el client entra a la cua. Altrament, el succés s'ignora.
- Un client surt d'una cua: Si la cua està entre 1 i n, i la cua no és buida, el *client de més edat* surt de la cua. Altrament, el succés s'ignora.

Entrada: L'entrada (a l'*standard input*) comença amb el nombre de cues n (un natural estrictament positiu). Segueixen n línies, una per cua, cadascuna amb els seus clients (una paraula) i les seves edats (un nombre real). Després ve una línia en blanc i la descripció d'una sèrie de successos, un per línia: la paraula "ENTRA" seguida del client (una paraula), de la seva edat, i de la cua (un enter); o bé la paraula "SURT" seguida de la cua (un enter).

Sortida: Primer, escriviu els noms dels clients que surten de les cues, en l'ordre en què ho fan. Després, escriviu el contingut final de les n cues, cadascuna en l'ordre en què sortirien els clients. Seguiu el format de l'exemple:

Entrada:	**Sortida**:

`4`
`Cristina 10 Tomas 27`
`Francesc 70 Damia 25.5 Domenec 80`
 	
`Teresa 19 Toni 83 Carles 24`
 	
`SURT 1`
`SURT 1`
`ENTRA Amalia 30 4`
`SURT 2`
`SURT 1`
`ENTRA Leo 22 1`
`ENTRA Maria 20 3`
`SURT 4`
`SURT 4`
`SURT 3`
`ENTRA Carme 18 4`
`SURT 2`
`SURT -1`
`SURT 2`
	`SORTIDES`
	`--------`
	`Tomas`
	`Cristina`
	`Domenec`
	`Toni`
	`Amalia`
	`Maria`
	`Francesc`
	`Damia`
	
	`CONTINGUTS FINALS`
	`-----------------`
	`cua 1: Leo`
	`cua 2:`
	`cua 3:`
	`cua 4: Carles Teresa Carme`

2.- Resoleu el problema 2. FIxeu-vos que en aquest problema l'edat és una prioritat amb la que col·locar-se a la cua. En definitiva, està clar que ens calen *cues amb prioritat*. AIxí, us convé fer servir un *heap* abans que una cua (fitxer **`heap.py`**).
Pista: Fixeu-vos bé què poseu al *heap*. Recomanem que siguin tuples `(edat,nom)`, ja que la comparació entre elements del *heap* es farà així en funció de l'edat, i en cas d'empat decidirà el nom (on l'ordre de les *strings* és el lexicogràfic).

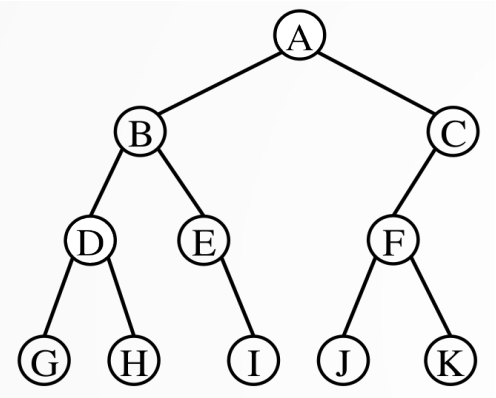
Sessió 10: La Pràctica... i altres problemes

Dedicarem aquesta sessió de laboratori a la pràctica. Proposarem alguns problemes per si de cas la discussió de la pràctica acaba abans d'hora.

La pràctica proposada enguany és una pràctica que requereix l'ús d'algunes de les noves estructures de dades que hem après en aquest curs, i de la programació orientada a objectes. Aquest darrer apartat s'ha mantingut a un nivell senzill i no ens caldrà fer servir l'herència.

Qüestions que cal discutir:

- Lectura d'arbres binaris: Aquesta es fa a partir d'una notació on els subarbres buits es fan explícits utilitzant una marca determinada. Per exemple, si tenim l'arbre:



sabem que el recorregut en pre-ordre: `A,B,D,G,H,E,I,C,F,J,K` no ens proporciona prou informació per reconstruir-lo. Si afegim marques (un '`0`', per exemple) per indicar els subarbres buits: `A,B,D,G,0,0,H,0,0,E,0,I,0,0,C,F,J,0,0,K,0,0,0`, ara ja podem saber qui és fulla i qui no, qui no té fill esquerre o dret, i per tant podem reconstruir l'arbre. La funció de lectura i reconstrucció d'un **`BinTree`** a partir del pre-ordre d'un arbre binari en aquesta notació és molt senzilla (suposem que fem servir el mòdul **`pytokr`**). Suposarem que l'arbre està en un fitxer que es passa com a paràmetre:

```
def llegeix_bintree_int(marca):
    x = int(f_item())
    if x != marca:
        l = llegeix_bintree_int(marca)
        r = llegeix_bintree_int(marca)
        return BinTree(x,l,r)
    else:
        return BinTree()
```

Fixeu-vos que aquesta funció suposa que estem llegint nombres enters. Si els elements emmagatzemats en els nodes de l'arbre són elements d'un altre tipus, caldrà modificar convenientment aquesta funció. De cara a la pràctica, ja ens està bé aquesta funció.

Finalment, és possible que, a la nostra pràctica, aquesta funció acabi sent un mètode privat d'alguna classe. Les modificacions en aquest cas són trivials:

```
def llegeix_bintree_int(self,marca):
    x = int(f_item())
    if x != marca:
        l = self.llegeix_bintree_int(marca)
        r = self.llegeix_bintree_int(marca)
        return BinTree(x,l,r)
    else:
        return BinTree()
```

Fixeu-vos també que aquesta funció, quan es llegeix la marca, retorna l'arbre buit.

- Programa principal: Tal com està especificat a l'enunciat, el fitxer d'entrada (per exemple, el fitxer **.inp** que us hem passat) consta d'una col·lecció d'instruccions que cal processar, fins arribar a la instruccio **fi**. Cada instrucció implicarà que cal llegir una sèrie de dades addicionals, i fer el procés que calgui. Podríem proposar-vos un esquema de programa principal:

```
from pytokr import items, item
# ...importar les classes que calgui, fetes per vosaltres

# ...inicialitzacions

instruccio = item()
while instruccio != 'fi':

    if instruccio == 'experiment':
        # ...llegir dades addicionals i processar 'experiment'

    elif instruccio == 'afegir_tret':
        # ...llegir dades addicionals i processar 'afegir_tret'

    elif instruccio == 'treure_tret': # només grups 3 persones
        # ...llegir dades addicionals i processar 'treure_tret'

    elif instruccio == 'consulta_tret':
        # ...llegir dades addicionals i processar 'consulta_tret'

    elif instruccio == 'consulta_individu':
        # ...llegir dades addicionals i processar 'consulta_individu'

    elif instruccio == 'distribucio_tret':
        # ...llegir dades addicionals i processar 'distribucio_tret'

    instruccio = item()
```

- Tractament d'errors: Els jocs de prova que es faran servir per corregir aquesta pràctica seran *correctes*, des d'un punt de vista sintàctic, d'acord al que s'ha mencionat a l'enunciat. Així doncs, excepte els errors que cal tenir en compte, i que s'han fet explícits a l'enunciat, no hi haurà cap altre error. En definitiva, només cal que considereu els errors que es mencionen a l'enunciat, *no cal que feu cap altre tractament d'errors*.

- Disseny: En aquesta assignatura no us hem explicat res respecte al disseny orientat a objectes. Vau veure amb detall l'exemple, una mica substancial, del joc *Snakes and Ladders*, i ja està. Així, pensem que cal que us donem algunes pistes respecte a quines classes us faran falta. Òbviament aquest problema admet diverses solucions que serien essencialment correctes, per tant aquestes indicacions que us donem no són l'*única* manera de resoldre correctament la pràctica.

Considerarem les classes següents (l'expressió "Conjunt de..." no és literal):

- **`Conjunt d'Individus`**: L'estat (privat!) de les instàncies d'aquesta classe (en realitat només en caldrà una per experiment) consistirà en l'arbre binari que defineix la relació entre individus i una col·lecció amb les instàncies d'`Individu` que hagin aparegut. Quines operacions us fan falta (els mètodes públics) és quelcom que heu de decidir vosaltres.

- **`Conjunt de Trets`**: L'estat (privat!) de les instàncies d'aquesta classe (en realitat només en caldrà una per experiment) consistirà en un diccionari de parelles `(nom_tret, info_tret)`, on `nom_tret` és una *string* i `info_tret` pot ser una tupla de dos elements, una instància de `Parell de Cromosomes` i un *conjunt* d'instàncies d'`Individu` (aquells que tenen el tret). Aquí, l'expressió *conjunt* es refereix a l'estructura de dades **`set`**, que Python ens proporciona[4]. Quines operacions us fan falta (els mètodes públics) és quelcom que heu de decidir vosaltres.

- **`Individu`**: L'estat (privat!) de les instàncies d'aquesta classe consistirà en una instància de `Parell de Cromosomes` i un *conjunt* d'identificadors de tret, aquells que l'individu posseeix. Aquí, l'expressió *conjunt* es refereix a l'estructura de dades **`set`**, que Python ens proporciona. Quines operacions us fan falta (els mètodes públics) és quelcom que heu de decidir vosaltres.

- **`Parell de Cromosomes`**: L'estat (privat!) de les instàncies d'aquesta classe pot consistir en un (o més) contenidors amb els *bits* dels cromosomes. Quines operacions us fan falta (els mètodes públics) és quelcom que heu de decidir vosaltres.

A més de les classes mencionades us farà falta la classe **`BinTree`**, i podeu fer servir la implementació que vulgueu (n'hem vist dues). Com conseqüència de necessitar **`BinTree`**, també haureu d'incloure **`Cua`**.

4 L'explicarem breument durant la sessió

Hi ha un parell de regles, conseqüència de tractar l'estat de les instàncies de les classes com a quelcom privat, que són inquebrantables:

a.- No es pot accedir a l'estat privat d'un objecte, ni aquest pot retornar cap de les seves estructures internes per a que hi accedeixi cap altre objecte. Per exemple, els arbres binaris només els poden fer servir, internament, les instàncies de `Conjunt d'Individus`.

b.- Cap mètode fa *input/output*, excepte els mètodes públics de lectura/escriptura expressament associats a cada classe. Per exemple, si cal llegir una instància d'`Individu` caldrà crear-la (`inv = Individu()`) i invocar el mètode corresponent (`inv.llegir_individu(m)` on `m` és la mida dels cromosomes). Penseu que si l'`Individu` té com a estat intern una instància de `Parell de Cromosomes`, caldrà que invoqui la corresponent funció de lectura d'aquesta instància.

Trencar aquestes regles implica una penalització molt important en la puntuació de la pràctica.

Us aconsellem fer un fitxer per classe, i utilitzar els `import` quan calgui.

1.- Si ens sobra temps, dedicarem la resta de la sessió a resoldre problemes de la col·lecció. Els triarem en funció de les vostres peticions, o a criteri del professor.

Sessió 11: Diccionaris i Taules de Dispersió

A classe de teoria us hem ensenyat la implementació d'un diccionari amb BSTs, però, tot i explicar-vos les taules de dispersió, no us hem mostrat cap implementació. Això és així ja que volem que ho feu vosaltres en aquesta sessió de laboratori.

Ja coneixeu l'interficie d'un Diccionari, és a dir, els seus mètodes públics:

```
from collections import namedtuple
Element = namedtuple("Element", ["clau", "valor"])

class Diccionari:
    def __init__(self):
        """
        Inicialització del diccionari amb taula de mida M (constant interna)
        """
    def buit(self):
        """
        retorna True si el diccionari és buit, False en altre cas.
        """
    def assigna(self, clau, info):
        """
        Assigna informació a una clau. Si la clau ja hi és dins
        el diccionari, la informació és modificada.
        cas pitjor: Theta(n). cas mitjà: Theta(1+n/M).
        """
    def valor(self, clau):
        """
        retorna el valor associat a una clau, None si la clau no hi és
        cas pitjor: Theta(n). cas mitjà: Theta(1+n/M).
        """
    def elimina(self, clau):
        """
        Elimina la parella (clau, valor) del diccionari.
        Si la clau no pertany al diccionari, res canvia.
        cas pitjor: Theta(n). cas mitjà: Theta(1+n/M).
        """
    def mida(self):
        """
        retorna el nombre de parelles (clau,valor) del diccionari
        """
    def troba_minim(self):
        """
        retorna la parella (clau,valor) amb la clau més petita.
        suposem que les claus són comparables.
        """
```

```
    def troba_maxim(self):
        """
        retorna la parella (clau,valor) amb la clau més gran.
        suposem que les claus són comparables.
        """
    def elements(self):
        """
        retorna una llista amb les parelles (clau,valor) del diccionari
        """
```

1.- Implementeu la classe considerant que resoldrem les col·lisions amb *encadenament separat* i que no cal fer *rehashing*:

```
    def __init__(self):
        """
        Inicialització del diccionari amb taula de mida M (constant interna)
        """
        self.__M = 1099
        # Taula de dispersió: cada item en aquesta taula serà una
        # llista d'elements (clau, valor)
        self.__taula = self.__M * [None]
        for i in range(self.__M):
            self.__taula[i] = []
        self.__n = 0    # nombre d'elements emmagatzemats
```

Hi ha un detall que cal tenir en compte en el mètode `assigna`, i és que estem emmagatzemant les parelles `(clau, valor)` en `namedtuple`. Això són tuples, i per tant immutables. Com podem canviar el valor d'una tupla `(clau, valor)`? Caldrà que penseu com canviar el valor d'una parella `(clau, valor)` quan la `clau` ja existeix.

Aquesta implementació que us demanem no té en compte el fet que el factor de càrrega pot acabar creixent massa...

2.- Modifiqueu el mètode `assigna` de manera que si el factor de càrrega sobrepassa un cert llindar (per exemple 1), faci *rehashing*. és a dir, canviï la constant que defineix la funció de dispersió, construeixi una taula nova (més gran) i re-col·loqui els elements. Us caldrà definir mètodes auxiliars.

Sessions 12 i 13: Problemes

En aquestes dues sessions us proposem una sèrie de problemes variats, que pretenen revisar part del que s'ha fet durant el curs. Alguns problemes seran de la col·lecció, i alguns seran nous.

1.- (*piles, Jutge P69546*) Hi ha n nens, cadascun amb una pila de cromos inicial. De tant en tant, dos nens x i y es juguen c cromos (tant x, com y, com c, van variant). El nen que perd dóna, d'un en un, els c cromos de dalt de la seva pila a l'altre nen, que els va posant a dalt de la seva pila en l'ordre d'arribada. Si un nen perd més dels cromos que li queden, només dóna els que té, però ha de pagar la diferència, en caramels, a una bossa comuna per a tots els nens.

Donades les n piles inicials, i els resultats de les partides jugades, digueu el contigut final de cada pila, i quants caramels ha perdut cada nen. Suposeu que els nens tenen inicialment un nombre arbitràriament gran de caramels.

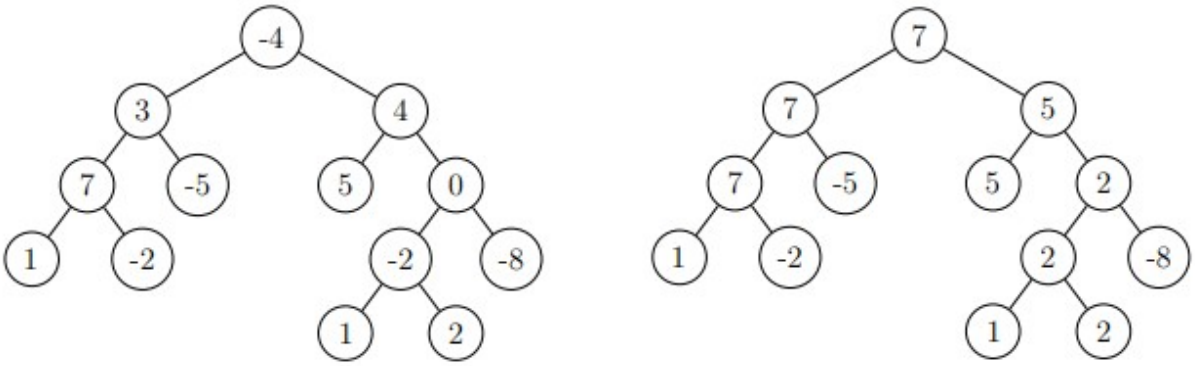
Entrada: L'entrada consisteix en diversos casos. Cada cas comença amb n, seguit del contingut de les n piles, de baix a dalt: una seqüència de paraules minúscules acabada en "FI". Segueix la informació de les partides: triplets amb x, y i c, que indiquen que x ha guanyat c cromos a y. Un triplet especial amb x = y = c = 0 marca el final de les partides.
Sortida: Per a cada cas, escriviu una línia per a cada nen, amb el nombre de caramels que ha perdut, seguit del contigut final de la seva pila de cromos, de dalt a baix. Escriviu una línia amb 10 guions al final de cada cas.
Exemple:

Entrada	**Sortida**
3 gandalf frodo bilbo FI aragorn legolas gimli FI arwen FI 1 2 1 1 3 2 3 2 1 0 0 0 2 FI sauron saruman gollum FI 1 2 1000 0 0 0	0 arwen gimli bilbo frodo gandalf 0 aragorn 1 legolas ---------- 0 sauron saruman gollum 997 ----------

2.- (*cues*) Resol el problema **38**, afegir a la classe **Cua** el mètode per trenar cues.

3.- (*llistes*) Resol el problema **44**, afegir a la classe **Llista** el mètode **splice**.

4.- (*arbres binaris*) Implementa un nou mètode a la classe **BinTree**, amb la següent capçalera **`def arbre_maxims(self)`** que torna un **BinTree** *nou* amb idèntica estructura que **self** (el paràmetre implícit) i on cada node conté el màxim dels nodes del subarbre corresponent en l'arbre original. Per exemple, si **a** és l'arbre a la part esquerra de la figura, aleshores **`a.arbre_maxims()`** retorna l'arbre de la part dreta



Cal que el programa que feu tingui un cost *lineal* en el nombre de nodes de l'arbre original. Cal suposar que els elements emmagatzemats en els nodes de l'arbre són comparables. Us passem un fitxer **`prova_bintree.py`** amb l'arbre de la figura ja codificat:

```
>>> from prova_bintree import *
>>> arbre2 = arbre.arbre_maxims()   # arbre està definit a prova_bintree.py
>>> arbre2
BinTree(7, left=BinTree(7, left=BinTree(7, left=BinTree(1), right=BinTree(-2)),
right=BinTree(-5)),    right=BinTree(5,    left=BinTree(5),    right=BinTree(2,
left=BinTree(2, left=BinTree(1), right=BinTree(2)), right=BinTree(-8))))
>>> arbre2.preorder()
[7, 7, 7, 1, -2, -5, 5, 5, 2, 2, 1, 2, -8]
>>> arbre2.inorder()
[1, 7, -2, 7, -5, 7, 5, 5, 1, 2, 2, 2, -8]
```

5.- (*heaps, Jutge P62266*) Suposeu que en l'escriptura d'un determinat llenguatge s'utilitzen n símbols diferents. Una manera simple de codificar-hi un text consisteix a assignar $\lceil \log_2 n \rceil$ bits a cada símbol. Per exemple, considereu les cinc vocals i les tres consonants més freqüents en llengua catalana. Segons la taula següent, una codificació "normal" de la paraula RES seria 101000010:

Lletra	Codificació normal	Freqüència relativa	Codificació Huffman
E	000	21	01
A	001	19	111
S	010	12.7	101
L	011	11.6	100
I	100	10.6	001
R	101	10.2	000
O	110	8.6	1101
U	111	6.3	1100

Suposeu ara que coneixem les probabilitats (o freqüències relatives) de cadascun dels n símbols (per exemple, segons la taula, de cada 100 símbols entre els vuit escollits, 21 són e's, 19 són a's, etcètera). Es pot aconseguir una codificació més eficient *en mitjana*, si a cada lletra se li assignen menys bits com més freqüent és. Segons la taula, la codificació de Huffman de la paraula RES seria 00001101, amb només vuit bits en lloc de nou.

La construcció d'una codificació de Huffman és relativament senzilla: *repetidament, s'escullen els dos símbols menys freqüents, se'ls assigna arbitràriament un bit (0 o 1) a cadascun per diferenciar-los, i a partir d'aquell moment se'ls considera un únic símbol, amb freqüència (acumulada) la suma de les freqüències dels símbols escollits. L'algorisme acaba quan només queda un símbol. La freqüència acumulada d'aquest símbol dividida per 100 és la longitud mitjana de la codificació*.

En aquest exemple, la longitud mitjana de la codificació de Huffman d'un símbol és només 0.21*2 + 0.19*3 + 0.127*3 + 0.116*3 + 0.106*3 + 0.102*3 + 0.086*4 + 0.063*4 ≃ 2.9390, és a dir, més petita que la longitud mitjana d'una codificació "normal", la qual és evidentment 3. Per a distribucions de probabilitats més esbiaxades, s'aconsegueixen estalvis molt més significatius.

Feu un programa que llegeixi les freqüències relatives d'unes quantes lletres, i en calculi la longitud mitjana de la seva codificació de Huffman. No cal construir l'arbre del codi de Huffman.

Entrada: L'entrada consisteix en el nombre de símbols n ≥ 2, seguit de les freqüències relatives dels n símbols. Aquestes freqüències són totes no negatives, i sumen 100.

Sortida: Escriviu amb quatre decimals el nombre esperat de bits per lletra.

Exemple:

Entrada	**Sortida**
8 19 21 10.6 8.6 6.3 12.7 11.6 10.2	nombre esperat de bits per lletra: 2.9390
4 5 2 3 90	nombre esperat de bits per lletra: 1.1500

Nota: Podeu llegir nombres reals fent **`float(item())`**

6.- (*Jutge, P78721*) Feu un programa que, donada una seqüència quasi ordenada de paraules que té exactament una paraula que apareix massa aviat, escrigui la seqüència totalment ordenada. L'entrada consisteix en dues o més paraules només amb lletres minúscules. Les paraules són totes diferents. La seqüència estaria ordenada alfabèticament si no fos per una paraula que apareix abans d'hora. El final de l'entrada

es marca amb la paraula especial “END”. El programa ha d’escriure la seqüència totalment ordenada. *Observació: No podeu usar llistes o similars.*

7.- (*Jutge, P12061*) Feu un programa que, donada una seqüència de paraules, escrigui quantes paraules hi ha entre la paraula “principi” i la paraula “final”. L’entrada consisteix en diverses paraules. Tant la paraula “principi” com la paraula “final” hi apareixen, com a molt, una vegada. Cal escriure el nombre de paraules entre la paraula “principi” i la paraula “final”, si aquestes apareixen en aquest ordre. Altrament, cal escriure “sequencia incorrecta”: si falta la paraula “principi”, o falta la paraula “final”, o ambdues, o si apareixen en ordre invers.

8.- (*Jutge, P14130*) Feu un programa que llegeixi una seqüència no buida d’enters, i que escrigui quants són iguals a l’últim. L’entrada consisteix en un natural n > 0, seguit de n enters. La sortida és el nombre d’elements que són iguals a l’últim, aquest exclòs.

9.- (*Jutge, X77867*) Donada una seqüència de zero o més línies, on cada línia consisteix en zero o més paraules formades per lletres minúscules, volem esbrinar si hi ha cap línia en la qual totes les paraules son pentavocàliques. Una paraula és pentavocàlica si cadascú dels caràcters ’a’, ’e’, ’i’, ’o’, ’u’ apareix si més no una vegada a la paraula. El programa ha d’escriure la primera línia, si n’hi han, a la qual totes les paraules són pentavocàliques; si no hi ha cap línia així, ha d’escriure “No s’ha trobat”.

10.- (*Jutge, X43847*) Representem existències de productes amb la seguent notació, que n'és un exemple:

```
stock_alice = {'orange':5, 'lemon':2, 'tangerine':1}
stock_bob = {'apple':3, 'orange':6, 'tangerine':1}
```

Com veiem, són diccionaris. Volem una funció **`total_stock(stk1, stk2)`** que construeixi les existències totals a dos diccionaris com aquests. Amb l’example, si es demana la igualtat

```
(total_stock(stock_alice, stock_bob) ==
        {'tangerine': 2, 'orange':11, 'lemon':2, 'apple':3})
```

el resultat ha de ser **`True`**.