Lycée Jean Moulin **Draguignan**
NSI *Terminale* Année .....

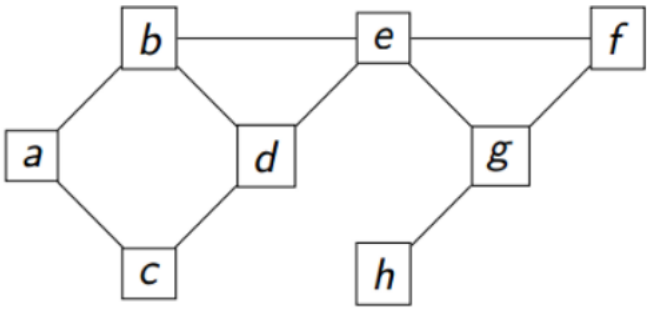
TD - Parcours en largeur d'un graphe - BFS

La méthode - BFS(Breadth First Search)

Parcourir un graphe en largeur à partir d'un sommet, consiste à visiter le sommet puis ses enfants, puis les enfants de ses enfants...

Comme on l'a déjà vu avec les arbres, il faut utiliser **une file** et **une liste** pour marquer les sommets visités..

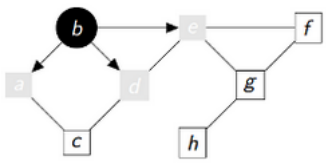
Prenons en exemple ce graphe :



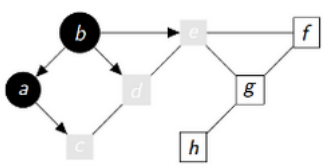
On dispose d'un graphe(G), d'une liste(sommet_visité) et d'une file(f)
Le sommet de départ est par exemple 'b', **on l'enfile**.
Puis **tant que la file n'est pas vide** :

- On **défile** f dans une variable par exemple tmp
- **Si tmp n'est pas dans sommet_visité**
  - On l'ajoute à sommet_visité
- **Pour chaque voisin de tmp**
  - **S'il n'est ni dans sommet_visité ni dans la file**
    - On **l'enfile**
- On renvoie sommet_visité

Voici les contenus des variables au premier tour de la boucle tant que :



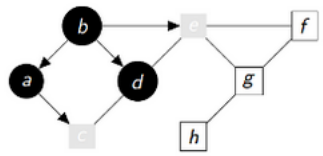
tmp='b'
sommet_visité=['b']
file='e' , 'd' , 'a'



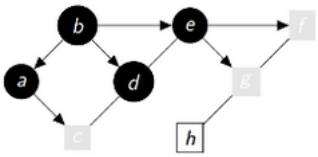
Au second tour
tmp='a'
sommet_visité=['b', 'a']
file='c' , 'e' , 'd'

À faire 1:

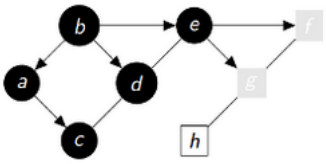
Compléter les contenus des variables tmp, sommet_visité et file aux tours suivants



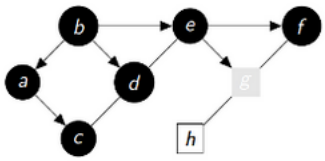
Au 3ème tour
tmp=.....
sommet_visité=.....
file=.....



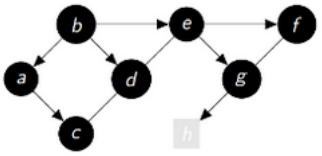
Au 4ème tour
tmp=.....
sommet_visité=.....
file=.....



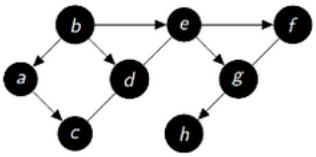
Au 5ème tour
tmp=.....
sommet_visité=.....
file=.....



Au 6ème tour
tmp=.....
sommet_visité=.....
file=.....



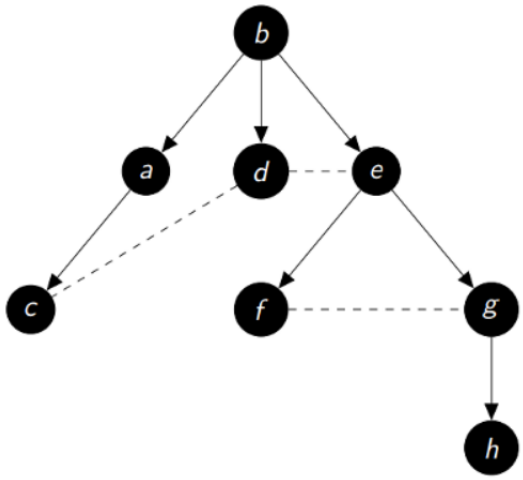
Au 7ème tour
tmp=.....
sommet_visité=.....
file=.....



Au 8ème tour
tmp=.....
sommet_visité=.....
file=.....

Au final l'arborescence associée au parcours peut donc être modélisée de la façon suivante :

['b', 'a', 'd', 'e', 'c', 'f', 'g', 'h']



Implémentation en Python du BFS

Voici une classe File dans laquelle on a rajouté une méthode `present(self,x)` qui renvoie vrai si x est dans la file.

```
class File:
    ''' classe File
    création d'une instance File avec une liste
    '''
    def __init__(self):
        self.L = []

    def vide(self):
        return self.L == []

    def defiler(self):
        assert not self.vide(), "file vide"
        return self.L.pop(0)

    def enfiler(self,x):
        self.L.append(x)

    def taille(self):
        return len(self.L)

    def sommet(self):
        return self.L[0]

    def present(self,x):
        return x in self.L
```

Vous pouvez également utiliser votre propre classe File.

Voici le code pour la création du dictionnaire qui représente le graphe G et une fonction qui renvoie les voisins d'un sommet

```
G = dict()
G['a'] = ['b','c']
G['b'] = ['a','d','e']
G['c'] = ['a','d']
G['d'] = ['b','c','e']
G['e'] = ['b','d','f','g']
G['f'] = ['e','g']
G['g'] = ['e','f','h']
G['h'] = ['g']

def voisins(G,sommet):
    return G[sommet]
```

L'algorithme du BFS

```
fonction parcours_largeur(G,sommet):
sommet_visite ← []
f ← File()
f ← sommet
Tant que  f n'est pas vide faire
    on défile f dans tmp
    on affiche tmp
    Si tmp n'est pas dans sommet_visite alors
        l'ajouter à sommet_visite
    Pour chaque voisin de tmp faire
        Si il n'est pas dans sommet_visite et pas dans la file alors
            l'enfiler
fin tant que
renvoyer sommet_visite
```

À FAIRE 2:

Implémenter cet algorithme en Python et tester le sur notre graphe G.

BFS - Version récursive

La présence d'une boucle **while** nous suggère la version récursive de cet algorithme.
On dispose d'un graphe, d'une File contenant le sommet de départ, d'une liste contenant le sommet de départ et qui nous servira à marquer les sommets visités.

Le processus :

1. on défile la File dans une variable (tmp) (on l'affiche)
2. pour chaque voisins non déjà visité de tmp
3. on le note comme visité
4. on l'enfile
5. on recommence du 1

Le processus s'arrête quand la File est vide

Voici le programme :

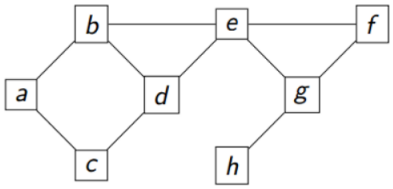
```
def bfs_recur(G,f,sommets_visites):
    if f.vide():
        return None
    tmp=f.defiler()
    print(tmp,end=" ")
    for u in voisins(G,tmp):
        if u not in sommets_visites:
            sommets_visites.append(u)
            f.enfiler(u)
    bfs_recur(G,f,sommets_visites)

f=File()
sommets_visites=[]
sommet='b'
f.enfiler(sommet)
sommets_visites.append(sommet)
bfs_recur(G,f,sommets_visites)
```

À FAIRE 3:
Faire fonctionner ce programme pour notre graphe.

Application : Un chemin entre deux sommets

L'objectif est de faire afficher un chemin entre deux sommets d'un graphe.

Par exemple :
a - b - e - g - h
est l'un des chemins possible entre a et h



La méthode consiste à mémoriser les sommets voisins du sommet visité comme clés d'un dictionnaire et ayant pour valeur son parent(le sommet visité).
Le sommet de départ n'aura bien entendu pas de parent (None)
À la fin notre dictionnaire parents sera:
{'a': None, 'b': 'a', 'c': 'a', 'd': 'b', 'e': 'b', 'f': 'e', 'g': 'e', 'h': 'g'}
Il nous faudra lire ce dictionnaire pour pouvoir établir le chemin entre 'a' et 'h'
h a pour parent g qui a pour parent e qui a pour parent b qui a pour parent a.
d'où le chemin : a - b - e - g - h

L'algorithme du BFS modifié

```
fonction parcours_largeur(G,depart):
parents ← dict()
sommet_visite ← []
f ← File()
f ← depart
parents[depart] ← None
Tant que  f n'est pas vide faire
    on défile f dans tmp
    Si tmp n'est pas dans sommet_visite alors
        l'ajouter à sommet_visite
    Pour chaque voisin de tmp faire
        Si il n'est pas dans sommet_visite et pas dans la file alors
            l'enfiler
            parents[el] ← tmp
fin tant que
renvoyer parents
```

Cette fonction renvoie un dictionnaire qui contient les sommets visités(clés) et leurs parents(valeurs).
Il faut maintenant exploiter ce dictionnaire pour faire afficher un chemin entre deux sommets C'est ce que réalise cette fonction qui prend en paramètre l'arrivée et le dictionnaire parents.

```python
def Solution(end, parents):
    chemin = []
    courant = end
    while courant != None:
        chemin = [courant] + chemin
        courant = parents[courant]
    return chemin
```

À FAIRE 4:

Implémenter l'algorithme en Python et la fonction Solution pour faire afficher un chemin entre les sommets b et h de notre graphe G.

À FAIRE 5:

Remplacer dans la fonction parcours_largeur la ligne:

.....
Si il n'est pas dans sommet_visite et pas dans la file **alors**
    ...

par la ligne :

.....
Si il n'est pas dans sommet_visite **alors**
    ...

Qu'en est-il des chemins proposés ?

À faire 6:

Reprendre ce travail en utilisant la version récursive du BFS