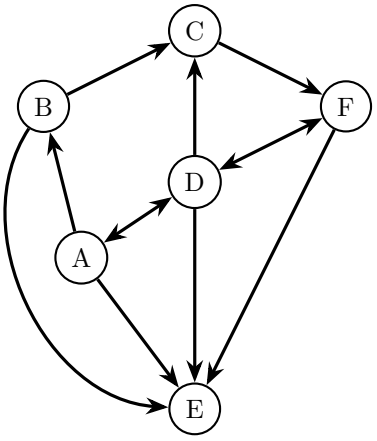
I Graphes et dictionnaire d'adjacence

On peut modéliser un graphe par un dictionnaire dont les clés sont les sommets et les valeurs associées sont les listes des sommets voisins

Un graphe orienté



Son dictionnaire d'adjacence

```
d = {'A':['B','D','E'],
     'B':['C','E'],
     'C':['F'],
     'D':['A','C','E','F'],
     'E':[],
     'F':['D','E']}
```

II Piles et Files

Pour parcourir un graphe efficacement, on utilise une structure de données linéaire appelée pile ou file.

- une pile suit le protocole *LIFO* (last in first out ou dernier entré premier sorti)
- une file suit le protocole *FIFO* (first in first out ou premier entré premier sorti)

L'intérêt des ces structures est de pouvoir être utilisées avec des faibles complexités : les opérations de création ou de modification sont à priori de coût constants ($O(1)$), donc indépendants de leur tailles.

1. Implémentation d'une pile à l'aide d'une liste

On peut définir les différentes opérations sur une pile à partir d'une liste et des méthodes `pop` et `append` qui agissent à coût constant sur le dernier élément de la liste

opération	code	coût
création	`P = []`	$O(1)$
tester si vide	`len(P) == 0`	$O(1)$
empiler un élément	`P.append(elt)`	$O(1)$
dépiler un élément	`P.pop()`	$O(1)$

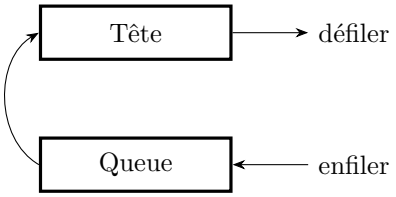
2. Implémentation d'une file

On peut tenter de définir les différentes opérations sur une file à partir d'une liste de la façon suivante :

opération	code	coût
création	`F = []`	$O(1)$
tester si vide	`len(F) == 0`	$O(1)$
enfiler un élément	`F.append(elt)`	$O(1)$
défiler un élément	`F = F[1:]`	$O(n)$

Le problème est alors l'opération de défilement dont le coût est linéaire à cause du slicing (qui demande de ré-indicer tous les éléments de la liste).

Une solution plus efficace est d'utiliser deux listes et les méthodes `pop` et `append` : une liste contient la queue de la file sur laquelle on va défiler les éléments, l'autre la tête de la liste sur laquelle on va les enfiler :



On peut alors définir les fonctions de base de la façon suivante :

```
def creer() :
    return ([],[])

def estVide(F) :
    Q,T = F
    return len(Q)+len(T) == 0

def enfiler(elt,F) :
    F[0].append(elt)
```

```
def defiler(F) :
    Q,T = F
    if len(T) == 0 :
        for _ in range(len(Q)) :
            a = Q.pop()
            T.append(a)
    return T.pop()
```

Le problème lors du défilement vient du cas où la tête est vide, on doit alors « reverser » la queue dans la tête avant de défiler la tête « normalement ». Avec une telle implémentation, l'opération de défilement est à coût constant, sauf lorsque la tête est vide (et le coût est alors linéaire)

3. Les objets `deque`

Il existe dans le module `collections` un objet appelé `deque` qui permet de simuler les piles et les files de façon efficace. Il faut commencer par importer cet objet :

```
from collections import deque
```

On peut alors agir à coût constant sur les deux « extrémités » de cet objet :

opération	code	coût
création	`L = deque()`	$O(1)$
tester si vide	`len(L) == 0`	$O(1)$
enfiler/empiler un élément	`L.append(elt)`	$O(1)$
dépiler un élément	`L.pop()`	$O(1)$
défiler un élément	`L.popleft()`	$O(1)$

III Parcours de graphes

1. Aller le plus loin possible – parcours en profondeur

Dans un parcours en profondeur, on part du sommet de départ, on choisit un arête et on essaie d'aller le plus loin possible dans le graphe ; lorsqu'on est bloqué, on remonte à la dernière bifurcation rencontrée et on explore une autre piste.
Le parcours en profondeur utilise une pile `P` et se déroule de la façon suivante :

- on empile le sommet de départ dans `P`
- tant que la pile `P` n'est pas vide, si le sommet de la pile `P` possède un voisin qui n'a pas encore été examiné, on l'empile et on recommence ; s'il n'a pas de voisin, on dépile la pile `P` et on recommence

Pour réaliser ce parcours, on a donc besoin, en plus de la pile `P`, d'un dictionnaire `vus` pour mémoriser les sommets que l'on a déjà examinés.

```
# le module permettant de gérer facilement les files
from collections import deque

def profondeur(G,depart) :
    # création d'une pile vide
    p = deque()
    # dictionnaire de booléens des sommets déjà vus
    vus = {x : False for x in G}
    # liste des sommets visités
    lst_visited = []
    # empilement du sommet de départ
    p.append(depart)
    # début du parcours
    while len(p) > 0 :
        # dépilement du sommet de q
        som = p.pop()
        # on n'examine som que s'il n'a pas déjà été examiné
        if not vus[som] :
            # som marqué comme vu
            vus[som] = True
```

```
            # ajout de som à la liste des sommets visités
            lst_visited.append(som)
        # parcours des voisins s de som
        for s in G[som] :
            # si s non déjà vu
            if not vus[s] :
                # empilement de s
                p.append(s)
    return lst_visited
```

Le parcours en profondeur peut assez naturellement se programmer de façon récursive.

2. Parcours par distances croissantes – parcours en largeur

Dans un parcours en largeur, on part du sommet de départ et on examine tous les sommets qui sont à une distance 1 du départ, puis ceux à distance 2, ...
Pour le parcours en largeur, on utilise une file et la démarche est la suivante :

- on enfile le sommet de départ.
- tant que la file n'est pas vide : si la tête de file possède des voisins qui n'ont pas encore été visités, on les enfile tous ; sinon on défile la tête de file.

```
# le module permettant de gérer facilement les files
from collections import deque

def largeur(G, depart) :
    # création d'une file vide
    q = deque()
    # dictionnaire de booléens des sommets déjà vus
    vus = {x : False for x in G}
    # liste des sommets visités
    lst_visited = []
    # enfilement du sommet de départ
    q.append(depart)
    # début du parcours
    while len(q) > 0 :
        # défilement du sommet de q
        som = q.popleft()
        # on n'examine som que s'il n'a pas déjà été examiné
        if not vus[som] :
            # som marqué comme vu
            vus[som] = True
            # ajout de som à la liste des sommets visités
            lst_visited.append(som)
        # parcours des voisins s de som
        for s in G[som] :
            # si s non déjà vu
            if not vus[s] :
                # enfilement de s
                q.append(s)
    return lst_visited
```

3. Comparaison des deux codes

Outre l'utilisation d'une pile ou d'une file, les deux codes sont assez similaires :

```
def profondeur(G,depart) :
    p = deque()
    vus = {x : False for x in G}
    lst_visited = []
    p.append(depart)
    while len(p) > 0 :
        som = p.pop()
        if not vus[som] :
            vus[som] = True
            lst_visited.append(
                som)
        for s in G[som] :
            if not vus[s] :
                p.append(s)
    return lst_visited
```

```
def largeur(G,depart) :
    q = deque()
    vus = {x : False for x in G}
    lst_visited = []
    q.append(depart)
    while len(q) > 0 :
        som = q.popleft()
        if not vus[som] :
            vus[som] = True
            lst_visited.append(
                som)
        for s in G[som] :
            if not vus[s] :
                q.append(s)
    return lst_visited
```

4. Adapter le code au problème

Les codes précédents sont des parcours du graphe en intégralité. Souvent on parcours le graphe jusqu'à trouver quelque chose dans ce graphe. Il faut donc modifier certaines lignes. Pour la parcours en largeur, le schéma est le suivant :

```
def largeur(G,depart) :
    q = deque()
    vus = {x : False for x in G}

    # liste des sommets visités
    lst_visited = []

    q.append(depart)

    # début du parcours
    while len(q) > 0 :

        som = q.popleft()
        if not vus[som] :
            vus[som] = True

            # ajout de som à la liste des sommets visités
            lst_visited.append(som)

        for s in G[som] :
            if not vus[s] :
                q.append(s)

    return lst_visited
```

- `# liste des sommets visités` / `lst_visited = []` : seulement pour voir dans quel ordre les sommets seront vus
- `# début du parcours` / `while len(q) > 0 :` : parcours intégral ; normalement jusqu'à trouver ce que l'on cherche
- `# ajout de som à la liste des sommets visités` / `lst_visited.append(som)` : à modifier selon ce que l'on doit faire avec som
- `return lst_visited` : on renvoie ce que l'on cherchait