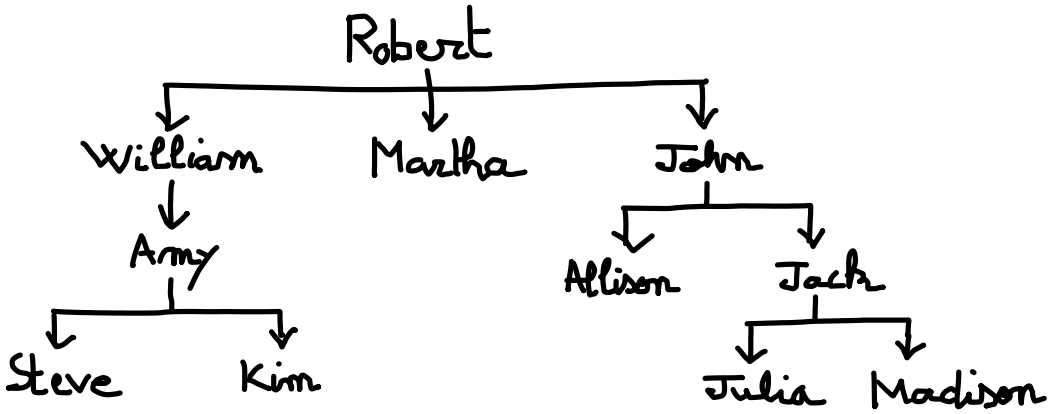
Cours 23 – Bases des graphes – Partie 2 – Principe de parcours d'un graphe

On a vu qu'il était possible d'implémenter un graphe $G = (S, A)$ sous la forme d'une liste d'adjacence ou d'une matrice d'adjacence. Dans un problème de graphe, on va vouloir déterminer l'ordre dans lequel les voisins d'un sommet S_i vont être visités. Dans des problèmes complexes, cet ordre peut être influencé par des probabilités, des pondérations, etc. Toutefois, on distingue ***deux grandes familles de parcours d'un graphe***, c'est-à-dire d'ordre de passage successif des sommets du graphe, de proche en proche à partir d'un sommet initial.

A / Exemple illustratif

Pour illustrer le principe de ces deux algorithmes, on utilise souvent un arbre (càd un graphe connexe sans cycle). Prenons l'arbre généalogique suivant :



Sur un arbre (ici orienté), on appelle ***feuille*** un sommet qui n'a qu'un seul voisin (donc de degré 1), qu'on représente par la fin de la ramification. Ici, c'est le cas de Steve, Kim, Martha, Allison, Julia et Madison.

Pour les algorithmes qui suivront, nous supposerons que les graphes sont implémentés sous la forme de **listes** d'adjacence, gérées par des dictionnaires :

```
Graphe = {Sommet1 :[Voisin_11, Voisin_12,…], Sommet2 : [Voisin_21, Voisin_22,…],…}
```

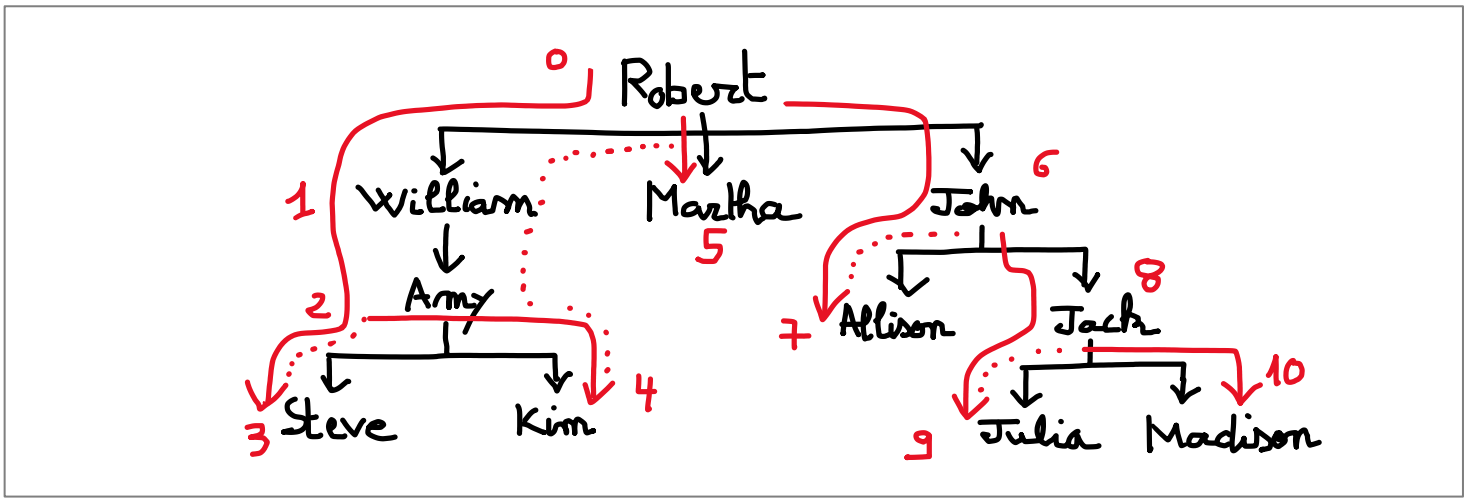
B / Parcours en profondeur

Le parcours en ***profondeur*** d'un graphe consiste à partir du sommet de départ, et le parcourir à la manière d'un labyrinthe, c'est-à-dire qu'on parcoure à fond un chemin, jusqu'à ce que parvienne à un « cul de sac », càd :

- soit un nœud de degré 1
- soit un nœud dont tous les voisins ont déjà été visités

On revient alors sur le dernier nœud qui avait des voisins pas encore visités.

Dans l'exemple ci-dessus (càd graphe non pondéré), on peut parcourir indistinctement un des sommets fils depuis un sommet père. Dans ce cas, comme dans un labyrinthe, il est conseillé de choisir une convention et de s'y tenir, prenons par exemple ici la convention qu'on visite prioritairement le fils le plus à gauche. L'ordre de parcours est alors :



B.2 / Implémentation récursive

La méthode la plus « intuitive » pour coder ce type de parcours est récursive.

```
def parcours_profondeur_r(Graphe, N_d, Visités = []):
    Visités.append(N_d) # Noeud départ ajouté à la liste des sommets visités
    for Voisin in Graphe[N_d]: # puis pour chaque voisin de ce sommet
        if Voisin not in Visités : # " " qui n'ait pas déjà été visité
            parcours_profondeur_r(Graphe, Voisin, Visités) # reproduire récursivement
    return Visités # on ne renvoit la liste que lorsqu'il n'y a plus aucun sommet à visiter.
```

dict ↔ liste adjacence ; noeud (ou sommet) de départ ; arg. opt. ne pas remplir pour appel initial ; valeur ; clé

On voit que la complexité de ce type d'algorithme est très importante (exponentielle ou factorielle). Or en *Python*, la gestion des ordres d'appels récursifs est gérée au moyen d'une pile de taille 1000 par défaut.

(Remarque : pour les experts, cette limite peut se modifier avec la commande :

```
import sys
nouvelle_lim = 10000
sys.setrecursionlimit(nouvelle_lim)
```

mais ceci est vraiment hors programme).

Il pourra donc, parfois, être avantageux de préférer l'écriture **itérative** de ce même parcours, pour de grands graphes.

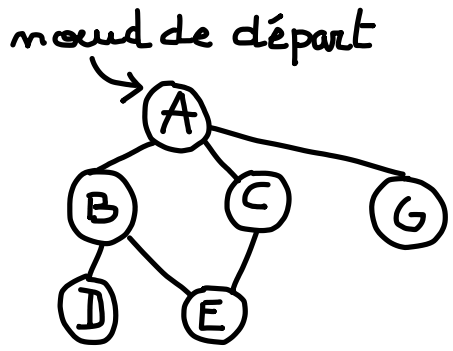
B.3 / Implémentation itérative

La méthode itérative de parcours en profondeur s'appuie sur une méthode de **pile** : nous allons tenir à jour une pile `A_visiter` des sommets qu'il nous reste à visiter, initialisée à `[N_d]` : 1 seul élément, le nœud de départ.

Puis, tant qu'il reste des éléments à visiter, de manière itérative :

- On dépile un sommet de `A_visiter`
- S'il n'y pas déjà été visité auparavant :
  - On le « visite » (s'il y a des actions à réaliser sur ce sommet)
  - Puis, pour chacun de ses voisins, (éventuellement, s'il n'y pas déjà été visité auparavant)
    - ➔ on l'ajoute à la pile `A_visiter` du même côté qu'on dépile

Exemple :



Prenons la convention qu'on empile et dépile par la droite.

Sommet visité → État de la pile `A_visiter` : (initialisation `[A]`)

A ⇒ [G] → [G,C] → [G,C,B]
B ⇒ [G,C,E] → [G,C,E,D]
→ [G,C,E,D,A]
ϕ (A déjà vu)
D ⇒ [G,C,E,B]
ϕ (B déjà vu)
E ⇒ [G,C,C] ⇒ [G,C,E,B]
ϕ (B déjà vu)
C ⇒ [G,C,E] ⇒ [ϕ,ϕ,E,A]
ϕ A vu
ϕ E vu
ϕ C vu
G ⇒ [A]
ϕ A vu [] → FIN ALGO

Cet algorithme itératif s'écrit :

```
def parcours_profondeur_ite(Graphe, N_d):
    A_visiter = [N_d]
    Deja_visites= []
    while A_visiter :  # éq. while len(A_visiter)>=0:
        Noeud_depart = A_visiter.pop()  # pop le dernier élement (droite) par défaut
        if Noeud_depart not in Deja_visites :
            Deja_visites.append(Noeud_depart)  # à droite car dans l'ordre de rencontre
            for Voisin in Graphe[Noeud_depart]:
                #if Voisin not in Deja_visites :  → ligne optionnelle, gain de temps
                    A_visiter.append(Voisin)  # pile → ajout/supression même côté
    return Deja_visites
```

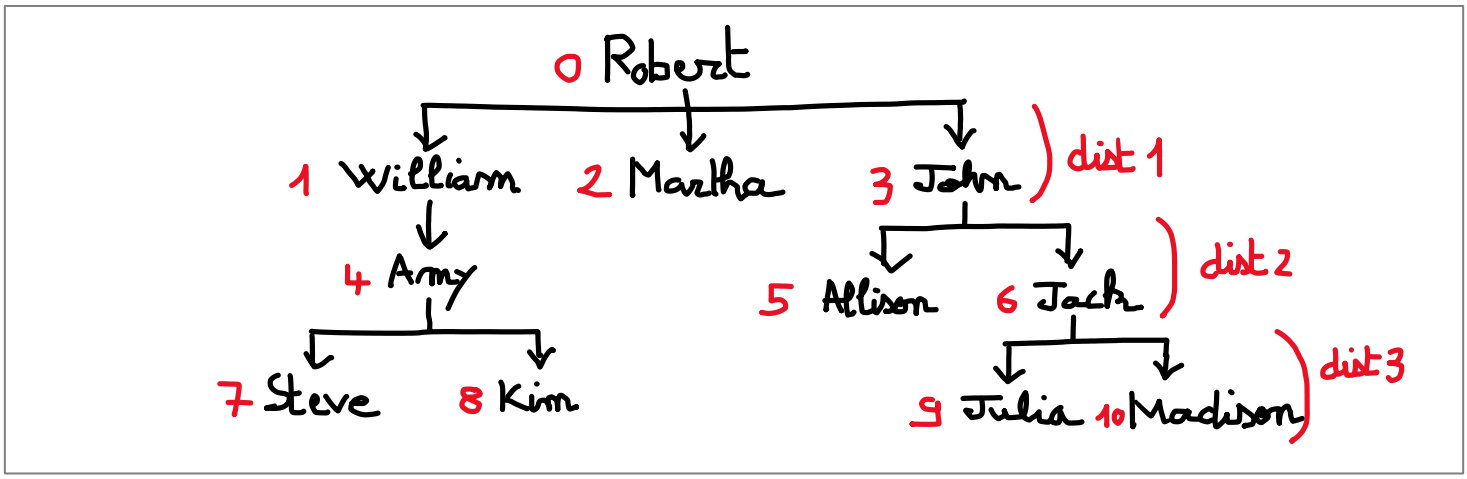
Cet algorithme de parcours d'un graphe est particulièrement adapté à certains problèmes (certains jeux, les labyrinthes, les parcours d'arbres probabilistes, la détection de cycles ou de certains motifs dans un graphe…). Entre autres, nous avons évoqué le problème du voyageur de commerce, et l'algorithme de Prim qui permet de le résoudre s'appuie sur un parcours de ce type.

En revanche, il n'est pas du tout adapté à d'autres problèmes, comme la recherche du plus court chemin entre deux sommets… Nous avons donc besoin d'un second algorithme de parcours, et vous verrez dans la suite de vos études que la quasi-totalité des algorithmes des graphes s'appuient, dans le fond, sur l'un ou l'autre des deux algorithmes.

C / Parcours en largeur

Le parcours en ***largeur*** d'un graphe consiste à partir du sommet de départ, et le parcourir de proche en proche, de façon « circulaire » ou « spiralaire », jusqu'à ne plus pouvoir visiter aucun nœud qui n'ait déjà été visité. Nous définissons que, par défaut, le sommet de départ est à une distance 0 de lui-même. Nous visitons donc d'abord ses voisins directs (à distance 1), puis leurs voisins directs qui n'ont pas déjà été vus (donc à distance 2 du départ), etc.

Revenons sur l'exemple de l'arbre généalogique. Dans l'ordre, seront visités (on part toujours de Robert) :



On voit immédiatement que ce type de parcours permettra très simplement d'évaluer des distances ou des meilleurs chemins entre deux nœuds d'un graphe. Entre autres, l'algorithme de recherche de plus court chemin dans un graphe pondéré, algorithme de Dijkstra évoqué au cours précédent (par exemple pour trouver le meilleur itinéraire entre deux villes) s'appuie sur un parcours de ce type. Certains algorithmes d'intelligence artificielle (historiquement, l'algorithme A* - prononcer *A star* – que vous verrez l'année prochaine) s'appuient également sur ce type de parcours de graphe.
Ce parcours n'est pas évident à coder récursivement, à cause des retours réguliers aux appels de rang supérieur, nous n'évoquerons donc que sa version itérative. Cette dernière ressemble beaucoup à la version itérative du parcours en profondeur, si ce n'est que la liste des éléments `A_visiter` est gérée comme une **file** et plus une pile !

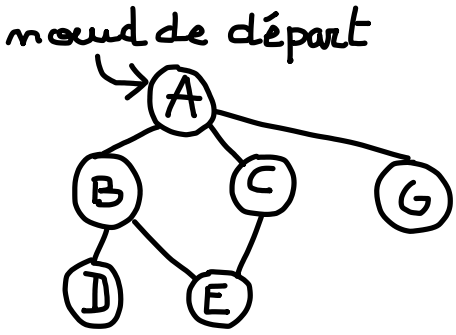
Ainsi, tant qu'il reste des éléments à visiter, de manière itérative :

- On dépile un sommet de `A_visiter`
- S'il n'y pas déjà été visité auparavant :
  - On le « visite » (s'il y a des actions à réaliser sur ce sommet)
  - Puis, pour chacun de ses voisins, (éventuellement, s'il n'y pas déjà été visité auparavant)
    ➔ on l'ajoute à la file `A_visiter` du côté opposé au dépilage.

Exemple :

Prenons la convention qu'on empile à gauche et dépile à droite.

Sommet visité → État de la pile `A_visiter` : (initialisation `[A]`)



A ⇒ [B] → [C, B] → [G, C, B]

B ⇒ [A, G, C] → [D, A, G, C]

→ [E, D, A, G, C]

C ⇒ [A, E, D, A, G] → [E, A, E, D, A, ϕ]

G ⇒ [A, E, A, E, ϕ, #]

ϕ (A déjà vu)

D ⇒ [B, A, E, A, #]

E ⇒ [B, B, A, E, A] → [#, B, B, #, E, A]

ϕ ⋮ ϕ } plus rien 6 prochaines étapes (déjà vus) jusqu'à [ ]

→ FIN ALGO

Le code *Python* correspondant serait :

```
def parcours_largeur_ite(Graphe, N_d):
    A_visiter = [N_d]
    Deja_visites = []
    while A_visiter :
        Noeud_depart = A_visiter.pop()
        if Noeud_depart not in Deja_visites :
            Deja_visites.append(Noeud_depart)
            for Voisin in Graphe[Noeud_depart]:
                #if Voisin not in Deja_visites:   → ligne optionnelle, gain de temps
                    A_visiter.insert(0,Voisin)       # SEULE ligne qui change
                                                     # par rapport à en «profondeur»
    return Deja_visites
```

D / Remarque sur une classe spécifique *Python* pour gérer les piles / files : les dèques

Le *dèque* est une classe spécifique (de la bibliothèque `collections`), qui ressemble aux listes mais avec des possibilités réduites afin d'optimiser le temps de calcul. Les 5 seules commandes au programme de prépa sont :

```
from collections import deque
```

- `D = deque()` équiv. L=[] rmq : si L est une liste, D = deque(L) est une copie sous forme de dèque
- `D.append(elem)` ↔ L.append(elem)
- `D.appendleft(elem)` ↔ L.insert(0, elem)
- `D.pop()` ↔ L.pop(-1) } ici aussi, la valeur de l'élément
- `D.popleft` ↔ L.pop(0) } dépilé est renvoyée ⇒ écrire elem = D.pop()