TP 14

Travaux Pratiques

Exercices d'application sur les piles

Dans les exercices qui suivent, on utilisera des **piles**. Les piles sont des structures de données basées sur le principe LIFO (Last In First Out : le dernier rentré dans la pile sera le premier à en sortir).

Les opérations élémentaires qu'on peut réaliser sur les piles sont les suivantes :

- savoir si une pile est vide;
- empiler un nouvel élément sur la pile;
- récupérer l'élément au sommet de la pile tout en le supprimant. On dit que l'on dépile;
- accéder à l'élément situé au sommet de la pile sans le supprimer de la pile;
- on peut connaitre le nombre d'éléments présents dans la pile.

Pour implémenter les piles on utilisera le module `deque`. Chacun des éléments de la pile peut être un objet de type différent.

```
from collections import deque

# Créer une pile vide
pile = deque()

# Tester si une pile est vide
len(pile) == 0

# Ajouter l'élément Truc au sommet de la pile
pile.append("Truc")

# Supprimer (et renvoyer) le sommet d'une pile non vide
sommet = pile.pop()
```

Seules ces fonctions éléméntaires seront utilisées dans les exercices suivants.

Exercice 1 – La parenthèse inattendue Dans cet exercice, on souhaite savoir si une chaîne de caractères est bien parenthésée ou non. Une chaîne bien parenthésée est une chaîne vide ou la concaténation de chaînes bien parenthésées.

> ■ **Exemple** Chaînes bien parenthésées :
> - `"()"`,`"()()"`,`"(())"` et `"(()())"`.
>
> Chaînes mal parenthésées :
> - `")("`,`"(("`,`"(()"` et `"())"`.
>
> ■

Question 1 *Implémenter la fonction* `parentheses(s:str)-> list` *renvoie une liste de tuples correpondants aux indices des parenthèses ouvrantes et fermantes.*

Question 2 *Réaliser un programme permettant de savoir si une chaîne de caractères est bien parenthésée. La structure de pile est-elle nécessaire?*

Question 3 *Adapter le premier programme pour qu'il puisse traiter des chaînes constituées de parenthèses, de crochets, ou d'accolades. Un mot est alors bien parenthésé si la parenthèse fermante qui correspond à chaque parenthèse ouvrante est du même type.*

Question 4 *Adapter le programme pour qu'il puisse traiter des mots constitués de parenthèses et d'autres caractères, qui n'interfèrent pas avec les parenthèses.*

Question 5 *Écrire une version récursive de la fonction* `parentheses`.

Exercice 2 – Inversion

Question 6 *Écrire une fonction qui intervertit les deux éléments situés au sommet d'une pile de taille au moins égale à 2.*

Exercice 3 – Dépile le n^e^

Question 7 *Écrire une fonction qui dépile et renvoie le troisième élément d'une pile de taille au moins égale à 3. Les premier et deuxième éléments devront rester au sommet de la pile.*

Exercice 4 – Lire le n^e^

Question 8 *Écrire une fonction qui permet de lire (sans l'extraire) le n-ième élément d'une pile. On prévoira le cas où la pile n'est pas de taille suffisante pour qu'un tel élément existe.*

Exercice 5 – Inversion des extrêmes

Question 9 *Écrire une fonction qui prend une pile non vide en argument et place l'élément situé à son sommet tout au fond de la pile, en conservant l'ordre des autres éléments. Quelle est sa complexité en temps et en espace ?*

Exercice 6 – Inversion de la pile

Question 10 *Écrire une fonction similaire à* `reversed`*, qui prend une pile en argument et renvoie une autre pile constituée des mêmes éléments placés dans l'ordre inverse.*

Question 11 *Si l'on s'autorise à détruire la pile fournie, quelle est la complexité en temps et en espace de cette fonction ? Et si on ne s'y autorise pas ?*

Exercice 7 – Tu coupes ?

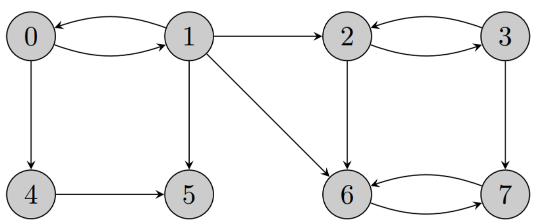
Question 12 *Écrire une fonction couper qui prend une pile et la coupe en enlevant de son sommet un certain nombre d'éléments (tirés au hasard) qui sont renvoyés dans une seconde pile.*

> ▪ **Exemple** Si la pile initiale est `[1, 2, 3, 4, 5]`, et que le nombre d'éléments retiré vaut 2, alors la pile ne contient plus que `[1, 2, 3]` et la pile renvoyée contient `[5,4]`. ▪

Exercice 8 – Parcours en largeur – Applications

D'après ressources Jules Svartz, Lycée Masséna.

Soit le graphe ci-dessous.



Question 1 *Modéliser le graphe ci-dessus en utilisant un dictionnaire d'adjacence nommé* `G1`*. On rappelle que dans ce cas, les clés seront des chaînes de caractère désignant le numéro du sommet et les valeurs seront constituées de la*

liste de chaînes de caractères où ces chaînes désignent les sommets successeurs.

Question 2 *Ecrire la procédure* `parcours_largeur(g,s) -> None` *réalisant le parcours en largeur d'un graphe, à partir du sommet* `s`.

On souhaite disposer de la fontion `parcours_largeur_distances(g,s) -> dict`. Cette fonction renverra le disctionnaire contenant toutes les distances à `s` en nombre de sommets. On va pour cela modifier la procédure précédente ainsi :

- les sommets visités seront stockés dans un dictionnaire `distances` où les clés seront les `n` sommets du graphe. Chacune des valeurs sera initialisée à l'infini (`float('inf')`), sauf la valeur associée au sommet de départ qui sera 0;
- à chaque fois que le voisin `v` d'un sommet `u` est ajouté à la file d'attente des sommets à traiter, si `distances[v]=inf` alors `distances[v]` prendra la valeur `1+distances[u]` .

Question 3 *Implémenter la fontion* `parcours_largeur_distances(g,s) -> dict`. *Dans le second exemple, comment interpréter les* `inf` *?*

Exemple

```
>>> parcours_largeur_distances(G1,"0")
        {'0': 0, '1': 1, '2': 2, '3': 3, '4': 1, '5': 2, '6': 2, '7': 3}
>>> parcours_largeur_distances(G1,"2")
        {'0': inf, '1': inf, '2': 0, '3': 1, '4': inf, '5': inf, '6': 1, '7': 2}
```

On souhaite maintenant connaître la liste des prédécesseurs de chaque sommet dans le cadre d'un parcours en largeur. On dit que u est le prédécesseur de v dans la parcours en largeur si v est découvert pour la première fois dans la liste d'adjacence de u. Dans ce cas, on a donc `distances[v]=distances[u]+1`. On pourra initialiser un dictionnaire `predecesseurs` de taille `n` ayant uniquement des `-1` comme valeur. À la fin du parcours on aura `predecesseurs[v]=-1` si `v` est le sommet de départ ou si `v` n'est pas accessible depuis le sommet de départ.

Question 4 *Implémenter la fontion* `parcours_largeur_predecesseurs(graphe,s) -> dict` *renvoyant le dictionnaire des prédécesseurs du sommet* `s`.

Exemple

```
>>> parcours_largeur_predecesseurs(G1,"0")
        {'0': -1, '1': '0', '2': '1', '3': '2', '4': '0', '5': '1', '6': '1', '7': '6'}
>>> parcours_largeur_predecesseurs(G1,"2")
        {'0': -1, '1': -1, '2': -1, '3': '2', '4': -1, '5': -1, '6': '2', '7': '3'}
```

À partir de la liste `predecesseurs` précédente, il est possible de calculer effectivement des plus courts chemins : pour calculer le plus court chemin de `s` à `v` (avec `v` accessible depuis `s`), il suffit de remonter de prédecesseur en prédecesseur de `v` jusqu'à `s` grâce à la liste `predecesseurs` : on obtient alors le chemin à l'envers.

Question 5 *Écrire une fonction* `chemin(g,s,v)` *prenant en entrée un graphe, les sommets* `s` *de départ et* `v` *d'arrivée, et renvoyant, dans l'ordre, un plus court chemin de* `s` *à* `v`. *La fonction renverra une liste vide si le chemin n'existe pas.*

Exemple

```
>>> chemin(G1,"0","6")
        ['0', '1', '6']
>>> chemin(G1,"2","1")
        []
```