TD : ALGORITHME DE DIJKSTRA AVEC LES TAS

L'objectif de ce TD est de mettre en œuvre l'algorithme de Dijkstra en utilisant les tas comme structure de données.

I) UTILISATION DES TAS EN PYTHON

Nous allons commencer par utiliser la structure de données tas binaire avec la bibliothèque Python heapq, en manipulant ses opérations principales :

- Insertion
- Extraction du minimum
- Transformation d'une liste en tas
- Recherche des plus petits éléments

Les différentes fonctions disponibles sont les suivantes :

Opération	Fonction	Complexité	Description
Créer un tas	`heapq.heapify(liste)`	O(n)	Transforme une liste en tas
Ajouter un élément	`heapq.heappush(heap, item)`	O(log n)	Insère un élément
Extraire le min	`heapq.heappop(heap)`	O(log n)	Retire le plus petit élément
Consulter le min	`heap[0]`	O(1)	Donne le plus petit élément sans le retirer
Insertion + extraction	`heapq.heappushpop(heap, item)`	O(log n)	Ajoute et retire le min en une étape
Extraction + insertion	`heapq.heapreplace(heap, item)`	O(log n)	Retire le min, puis ajoute un élément
k plus petits	`heapq.nsmallest(k, iterable)`	O(n log k)	Renvoie les k plus petits éléments
k plus grands	`heapq.nlargest(k, iterable)`	O(n log k)	Renvoie les k plus grands éléments

Exercice 1 — Création et première manipulation d'un tas

1. Créer la fonction `CreationListe(n,min,max)` permettant de créer et de retourner une liste python de n valeurs entières aléatoires, comprises entre [min,max].

2. Créer une liste python nommée `data` constituée de 10 valeurs entières aléatoires comprises entre 1 et 100 et afficher cette liste.

3. Utiliser ensuite la fonction `heapify()` pour transformer la liste en tas avec la commande `heapq.heapify(data)`. Afficher le tas avec un print().

On rappelle que les tas gèrent des objets associés à des clés, de manière à respecter la propriété fondamentale du tas : pour chaque paire parent-enfant, la clé du parent est inférieure ou égale à celle de l'enfant. Les clés dupliquées sont autorisées.

Si les positions du tableau sont numérotées 1, 2, …, n (où n est le nombre d'objets), alors les enfants de l'objet en position i se trouvent en positions 2i et 2i + 1 (s'ils existent) ; et inversement, pour un objet non racine en position i ≥ 2, son parent se trouve à la position ⌊i/2⌋.

Soit un objet en position i dans le tableau :	
Position du parent	⌊i/2⌋ (si i ≥ 2)
Position de l'enfant à gauche	2i (si 2i ≤ n)
Position de l'enfant à droite	2i + 1 (si 2i + 1 ≤ n)

Tableau 3 : Relations parent-enfant des objets dans un tableau

4. La structure attendue du tas est-elle respectée ? Dessiner ce tas sous forme d'un arbre binaire.

Exercice 2 — Insertion et extraction

1. Ajouter l'élément 0 au tas en utilisant la fonction `heappush()`. Afficher le tas et vérifier la cohérence de l'insertion de la valeur.

2. Extraire le plus petit élément du tas avec la fonction `heappop()`, afficher la valeur extraite puis afficher de nouveau le tas.

3. Quelles sont les complexités de ces opérations ? Quelle serait la complexité avec une liste triée classique ?

4. Quelle est la complexité de consultation du plus petit élément avec `heap[0]` ? Comparer cette complexité avec celle d'une liste triée.

Exercice 3 — Comparaison des opérations d'insertion avec une liste python non triée

1. Définir une fonction `test_insertion_liste(n)` qui, à partir d'une liste python vide, ajoute n valeurs entières aléatoires comprises entre 0 et n en utilisant l'opération `append()` et qui permet de mesurer le temps d'exécution nécessaire pour réaliser cette opération.

2. Appeler cette fonction pour n=1000000.

3. Définir une fonction `test_insertion_tas(n)` qui, à partir d'un tas vide, ajouter n valeurs entières aléatoires comprises entre 0 et n en utilisant l'opération `heappush()` et qui permet de mesurer le temps d'exécution nécessaire pour réaliser cette opération.

4. Appeler la fonction précédente pour n=1000000. Comparer les durées d'exécution et expliquer.

Exercice 4 — Comparaison des opérations d'insertion avec une liste python triée

1. Définir une fonction `test_insertion_liste_avec_tri(n)` qui, à partir d'une liste python vide, ajoute n valeurs entières aléatoires comprises entre 0 et n en utilisant

l'opération `append()` et en les triant après chaque insertion avec `sort()` et qui permet mesurer le temps d'exécution nécessaire pour réaliser cette opération.

2. Appeler cette fonction pour n = 10000 et comparer le temps d'exécution avec la fonction `test_insertion_tas(n)`.

3. Comparer les résultats obtenus. Quelle est la complexité dans les deux cas ?

Exercice 5 — Comparaison d'extraction du minimum

1. Créer une liste non triée avec 1000000 de valeurs entières aléatoires comprises entre 0 et 1000000 à l'aide de la fonction `CreationListe(n,min,max)`.

2. Réaliser l'extraction de la valeur du minimum avec la fonction `min(list)` et mesurer le temps nécessaire pour réaliser cette opération.

3. Transformer cette liste en tas et réaliser l'opération d'extraction du minimum. Mesurer le temps nécessaire pour réaliser l'ensemble de ces deux opérations.

4. Comparer les résultats obtenus. Quelles sont les complexités d'extraction du minium sur une liste non trié et sur un tas ?

Exercice 6 — Utilisation des tuples

Le tas peut contenir des couples `(clé, valeur)` : la comparaison se fait sur la clé.

Exemple :

```
heap = []

heapq.heappush(heap, (5, 'A'))
heapq.heappush(heap, (3, 'B'))
heapq.heappush(heap, (8, 'C'))
heapq.heappush(heap, (1, 'D'))

print(heap)
print(heapq.heappop(heap))
```

1. Exécuter l'exemple ci-dessus.

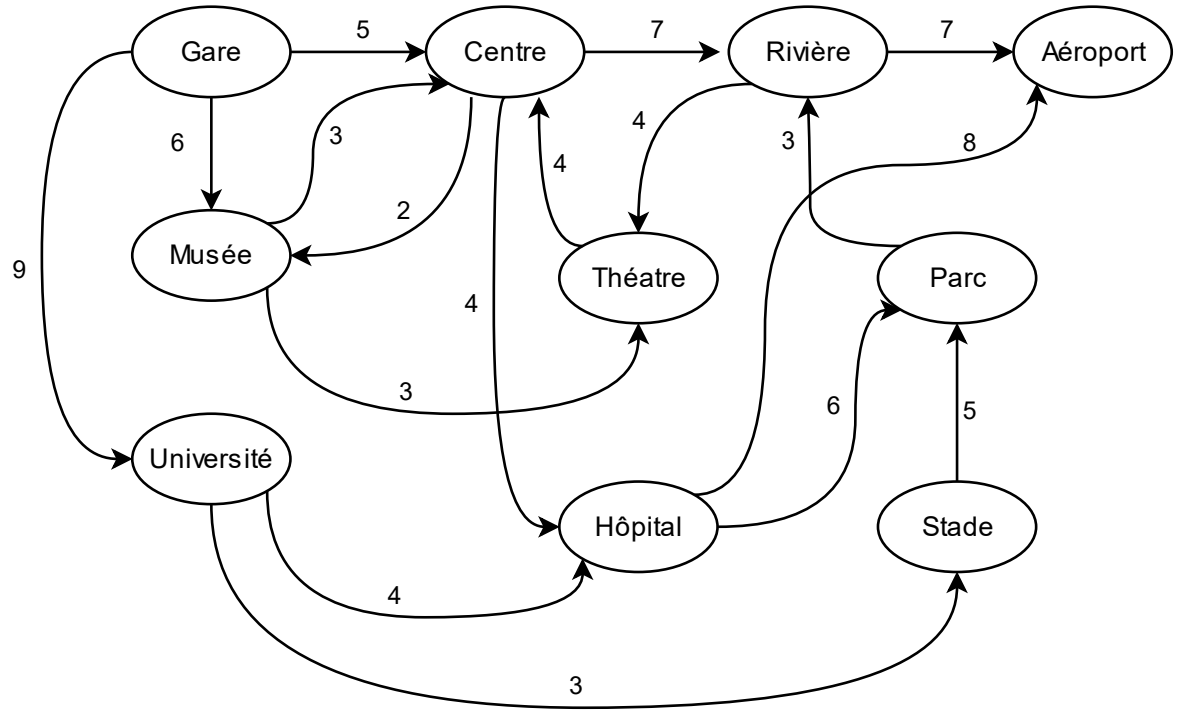
2. Dessiner le tas sous forme d'un arbre binaire.

3. Quel est l'élément extrait ? Pourquoi ?

4. Comment cette logique sera-t-elle utile pour l'algorithme de Dijkstra ? Rappeler à quoi correspondent les valeurs des clés du tas dans l'algorithme de Dijkstra.

II) EXEMPLE D'APPLICATION

Prenons l'exemple d'un mini réseau urbain où chaque arrête correspond au temps de déplacement :



On donne le dictionnaire permettant de décrire le réseau dans le fichier « TD_DIJKSTRA_RAPIDE_PARTIE2.PY » :

```
# Chaque arête est dirigée et pèse des minutes
dict_reseau = {
    "Gare":       {"Centre": 5, "Musée": 6, "Université": 9},
    "Centre":     {"Musée": 2, "Hôpital": 4, "Rivière": 7},
    "Musée":      {"Théâtre": 3, "Centre": 3},
    "Université":{"Stade": 3, "Hôpital": 4},
    "Stade":      {"Parc": 5},
    "Hôpital":    {"Aéroport": 8, "Parc": 6},
    "Parc":       {"Rivière": 3},
    "Rivière":    {"Aéroport": 7, "Théâtre": 4},
    "Théâtre":    {"Centre": 4},
    "Aéroport":   {}
}
```

L'objectif est de calculer le chemin le plus court pour aller de la Gare vers le Parc.

1. Écrire la fonction `Initialisation(reseau,depart)` qui prend en paramètre le réseau sous forme de dictionnaire (`dict_reseau`) et la station de départ puis initialise le tas selon les lignes 1 à 6 de l'algorithme donné dans le cours. Le tas sera structuré de la même manière que dans l'exemple de l'exercice n°6 de la partie précédente.

Vérifier :

```
>>> H
[(0, 'Gare'), (inf, 'Aéroport'), (inf, 'Hôpital'), (inf, 'Rivière'), (inf,
'Centre'), (inf, 'Musée'), (inf, 'Parc'), (inf, 'Université'), (inf,
'Théâtre'), (inf, 'Stade')]

>>> cles
{'Gare': 0, 'Centre': inf, 'Musée': inf, 'Université': inf, 'Stade':
inf, 'Hôpital': inf, 'Parc': inf, 'Rivière': inf, 'Théâtre': inf,
'Aéroport': inf}
```

Dans l'algorithme du cours, la mise à jour du tas en maintenant la condition sur la clé se fait de la manière suivante :

```
                // Mise à jour du tas en maintenant la condition sur la clé
11              POUR chaque arête (w*, y) :
12                      supprimer l'élément y du tas H
13                      clé(y) := min {clé(y), len(w*) + ℓw*,y}
14                      Insérer y dans le tas H
```

Cependant, la librairie heapq est une implémentation simple et rapide d'un tas binaire minimal, mais elle ne garde aucune information de position sur les éléments. heapq sait ajouter (heappush) et retirer le minimum (heappop), mais il ne sait pas où se trouve un élément particulier dans le tas.

On ne peut donc pas faire quelque chose comme :

```
heapq.remove((distance, sommet))
```

La technique la plus propre et rapide, utilisée dans tous les codes Dijkstra efficaces est la suivante : au lieu de supprimer l'élément, on laisse l'ancien dans le tas et on ajoute le nouveau avec sa nouvelle valeur. Puis on ignore les entrées périmées quand on les rencontre avec heappop en vérifiant si l'élément possède bien la bonne distance :

```python
import heapq

H = []
distances = {"A": 0, "B": 3, "C": 2}
heapq.heappush(H, (0, "A"))
heapq.heappush(H, (3, "B"))
heapq.heappush(H, (2, "C"))

# Mettons à jour la distance de B (3 → 1)
distances["B"] = 1
heapq.heappush(H, (1, "B"))  # on réinsère la nouvelle valeur

# Lors du pop, on vérifie la validité :
while H:
    d, u = heapq.heappop(H)
    if d != distances[u]:
        continue  # entrée obsolète, on l'ignore
    print(f"Sommet {u} traité avec distance {d}")
```

2. Écrire la fonction `Dijkstra(reseau)` qui prend en paramètre le réseau sous forme de dictionnaire (`dict_reseau`) et calcule les distances minimales depuis la gare vers les différentes stations.

Vérifier :

```
>>> distances
{'Gare': 0, 'Centre': 5, 'Musée': 6, 'Hôpital': 9, 'Théâtre': 9,
'Université': 9, 'Rivière': 12, 'Stade': 12, 'Parc': 15, 'Aéroport':
17}
```