ITC – cours n°13

Structures de données

Dans un programme informatique, il existe plusieurs façons de stocker des données. Certaines sont intuitives (par exemple, les listes Python sont des tableaux contigus : chaque élément est placé à côté du suivant dans la mémoire) et d'autres sont plus complexes. Chacune est plus rapide pour certaines opérations, et plus lente pour d'autres : le choix d'une structure est donc lié à des questions de complexité. Par exemple,

- accéder au n^{e} élément est bien plus rapide pour les tableaux contigus (il suffit d'aller à la case n, on sait où elle est en mémoire puisque chaque case fait la même taille) que pour les listes chaînées (il faut parcourir chaque élément depuis le début pour avoir la position du suivant, jusqu'au n^{e}) ;
- insérer un élément au milieu de la structure en position n est plus long pour les tableaux (il faut déplacer tous les éléments après n) que pour les listes chaînées (il suffit de dire que l'élément $n-1$ pointe vers le nouvel élément n, et que le nouvel élément n pointe vers l'ancien élément n).

Le choix de la structure de donnée impacte donc directement la complexité des opérations courantes sur les données. Pour choisir une structure adaptée à un problème, il faut donc bien identifier les besoins du problème en question.

Dans ce cours, nous allons nous concentrer sur des rappels concernant les algorithmes impliquant des listes ou des tableaux Numpy, puis sur deux nouvelles structures de données, les piles et les files, qui pousse à réfléchir l'algorithme d'une certaine manière.

I. Structures natives en Python

1.1. Listes Python

a) Présentation en mémoire

Les listes Python sont, en première approximation, des tableaux contigüs de références : pour une liste `l` donnée de taille n, `l[i]` ($i \in [\![0; n-1]\!]$) est une référence vers une autre variable en mémoire, et chaque référence est stockée à côté de la précédente.

b) Complexité des opérations élémentaires

La conséquence principale est la suivante : on peut calculer la position d'une étiquette en connaissant uniquement son index, en un temps $\mathcal{O}(1)$. Ainsi les opérations de lecture `l[i]` et d'affectation `l[i] = x` sont en $\mathcal{O}(1)$ pour les listes Python. Cette propriété est à la base de nombre de calculs de complexité que nous avons fait jusqu'à présent.

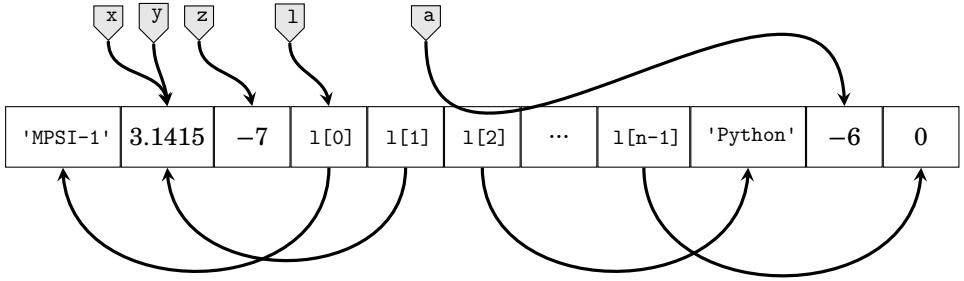


Figure 1 : Représentation mémoire des listes Python

Les opérations d'ajout et de retrait d'un élément en milieu de liste sont en revanche très coûteuses : si on veut maintenir la propriété de position calculable, il faut toujours avoir les références contigües en mémoire, ce qui implique de **déplacer tous les éléments trop à droite ou à gauche**. Le code suivant est équivalent à ce qui se passe en mémoire, en supposant que `append` coûte $\mathcal{O}(1)$:

```
# on suppose une liste l de taille n > 200
def insertion_équivalente(l, k, x):  # équivalent de l.insert(k, x)
    l.append(0)                      # O(1) dans ce modèle
    for i in range(n-1, k-1, -1):    # n-k répétitions
        l[i+1] = l[i]             # on décale chaque valeur vers la droite
    l[k] = x
def retrait_équivalent(l, k):        # équivalent de l.pop(k)
    x = l[k]
    for i in range(k, n):            # n-k répétitions
        l[i] = l[i+1]             # on décale chaque valeur vers la gauche
    return x
```

On constate donc que ces opérations sont en $\mathcal{O}(n)$. Il y a cependant un comportement différent pour la dernière position :

- il n'y a besoin de déplacer aucun élément pour renvoyer et supprimer le dernier élément, donc `pop` en dernière position est en temps $\mathcal{O}(1)$;
- pour ajouter un élément en dernière position, si on a réservé de la place supplémentaire, il suffit de placer la nouvelle référence, donc $\mathcal{O}(1)$; dans le cas contraire, comme dans l'exemple représenté figure 1, il faudra réserver assez de place ailleurs en mémoire et copier toutes les références, soit $\mathcal{O}(n)$. En réservant de la place supplémentaire à chaque fois que cette opération est nécessaire, on peut faire en sorte que `append` coûte presque toujours $\mathcal{O}(1)$. On parle de complexité amortie $\mathcal{O}(1)$.

c) *Exercices*

Proposer des algorithmes et justifier la complexité pour des listes Python de taille n :

- trouver l'indice de la première occurence d'une valeur x ($\mathcal{O}(n)$) ;
- compter le nombre d'occurences d'une valeur x ($\mathcal{O}(n)$) ;
- déterminer la présence ou non d'une valeur x en supposant la liste triée ($\mathcal{O}(\log_2 n)$) ;
- retirer les cinq derniers éléments ($\mathcal{O}(1)$) ;
- retirer le premier élément ($\mathcal{O}(n)$).

Sachant que les tranches font une copie de la liste, proposer un code équivalent aux commandes suivantes, et justifier la complexité annoncée :

- `l[i1:i2]` en $\mathcal{O}(i_2 - i_1)$;
- `l = [0]*n` en $\mathcal{O}(n)$;
- `l = [x**2 for x in li]` où `li` est une liste de taille n', en $\mathcal{O}(n')$.
- `val in l` en $\mathcal{O}(n)$;
- `l = l1 + l2` en $\mathcal{O}(n_1 + n_2)$.

1.2. Dictionnaires

Les dictionnaire sont des **tables de hachage**, dont le fonctionnement sera étudié en détail en 2e année ; pour comprendre les opérations élémentaires dessus, on peut l'imaginer comme un tableau contigü, mais dont l'utilisation des « cases » n'est pas faite dans l'ordre, mais à travers une fonction qui calcule la position (dite fonction de hachage). Ainsi, le fonctionnement est le même que le code équivalent suivant :

```
def insérer_dict(d, clé, val):      # équivalent de d[clé] = val
    indice = fonction_hachage(clé)  # par exemple 'MPSI-1' -> 756
    l = liste_sous_jacente(d)       # on récupère le tableau sous-jacent
    l[indice] = val                 # on écrit la valeur dans la case 756
```

Puisque les positions sont calculées selon les clés, on comprend qu'on peut « laisser des trous » dans le tableau, il n'est donc pas nécessaire de re-décaler les éléments un à un quand on en retire ou on en ajoute un. De cette manière :

- ajouter ou affecter un élément avec `d[clé] = valeur` est en $\mathcal{O}(1)$[1] ;
- supprimer un élément avec `del d[clé]` ou `d.pop(clé)` est en $\mathcal{O}(1)$;
- un test de présence `clé in d` est aussi en $\mathcal{O}(1)$: il suffit de calculer la case et de regarder si elle est occupée ;
- une copie de tout le dictionnaire nécessite de copier tous les élément, elle est donc en $\mathcal{O}(n)$.

[1] Sauf si le tableau sous-jacent est déjà plein : dans ce cas, il faut faire une copie comme avec `list.append` ; cependant, comme pour `list.append`, on prend plus de place que nécessaire lorsque cette opération est indispensable, ce qui amène à une complexité en $\mathcal{O}(1)$ amortie, $\mathcal{O}(n)$ dans le pire cas.

II. Liste chaînée

La liste chaînée est une structure qui garantit une insertion et une suppression en $\mathcal{O}(1)$ en première et dernière position. La logique en mémoire est la suivante : un élément est un nœud, composé à la fois de la valeur et des références sur les nœuds précédents et suivants.

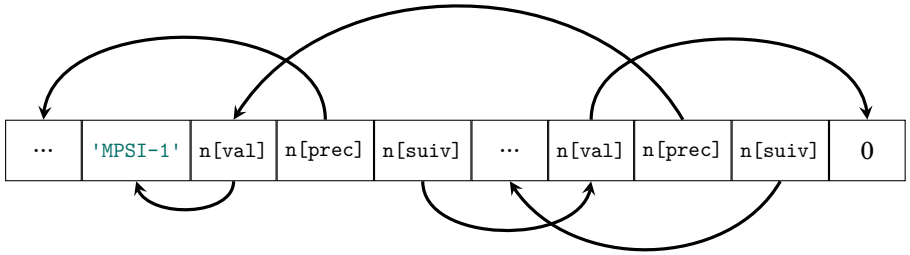


Figure 2 : Représentation mémoire des listes chaînées

Une conséquence de cela est qu'ajouter un élément en début ou en fin de liste chaînée coûte $\mathcal{O}(1)$ quoiqu'il arrive : il suffit de changer le nœud de début (ou de fin). De même, insérer un nouvel élément entre deux éléments coûte $\mathcal{O}(1)$, puisqu'il faut juste changer les références vers les nœuds précédents et suivants. Fusionner deux listes chaînées est donc également une opération en $\mathcal{O}(1)$, ce qui est intéressant dans des applications où on fusionne / sépare régulièrement des ensembles. En revanche, le simple fait de lire la valeur du k^{e} élément est en $\mathcal{O}(k)$, puisqu'il faut parcourir la liste depuis le début pour trouver la position en mémoire (et donc la valeur) du nœud correspondant.

Les listes chaînées sont souvent utilisées pour des jeux de données avec beaucoup de « trous » (par exemple, représenter un polynôme par une liste Python est très problématique pour $X^{1000} + 3$, avec 1000 cases pour seulement deux nombres à stocker), ou pour gérer des ordonnancement de tâches : c'est donc une structure très utilisée par les systèmes d'exploitation.

En python, on peut utiliser le type `deque` du module `collections`. Cependant, nous en aurons assez peu besoin dans le cadre du programme, sauf éventuellement pour simuler des piles ou des files.

```
li_ch = deque([0, 3, 5, 2])
li_ch.append(4)       # ajout de 4 en dernière position
li_ch.append_left(7)  # ajout de 7 en première position
x = li_ch.pop()       # x vaut 4, la dernière valeur est retirée de li_ch
y = li_ch.popleft()   # x vaut 7, la première valeur est retirée de li_ch
```

Exemple d'utilisation : on souhaite récupérer et enlever le plus petit élément d'un ensemble. Voici deux implémentations en utilisant les listes Python ou les listes chaînées :

```
def retrait_mini_liste(li):
    imin, mini = 0, li[0]
    for i, elt in enumerate(li):  # O(n)
        if elt < mini:
            imin, mini = i, elt
    li.pop(imin)                  # O(n)
    return mini

def retrait_mini_deque(de):
    imin, mini = 0, de[0]
    for i, elt in enumerate(de):  # O(n)
        if elt < mini:
            imin, mini = i, elt
    de.pop(imin)                  # O(1)
    return mini
```

Si ce type d'opérations est très demandé dans l'algorithme écrit, on peut ainsi gagner un facteur 2 environ sur le temps d'exécution.

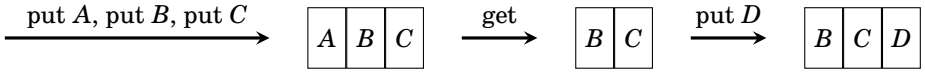
III. Piles et files

3.1. Files

Une file est une une collection dite FIFO, pour « First in, first out » ou « premier arrivé, premier sorti ». Elle se comporte comme une file d'attente : les éléments insérés en premier seront retirés en premier. Cette structure est donc adaptée à tout algorithme devant traiter des données dans un certain ordre.

Définition d'une file

Une file est une structure de données où les opérations autorisées sont
- *enfiler* un élément (« put ») c'est-à-dire ajouter un élément en fin de la file ;
- *défiler* un élément (« get ») c'est-à dire enlever l'élément situé en-avant de la file ;
- tester si la file est vide.



On peut simuler une telle structure en python de plusieurs manières :
- une façon maladroite utilise une liste Python avec `append` pour ajouter un élément à la file, et `pop(0)` pour l'enlever. Cela conduit à un coût $\mathcal{O}(n)$ pour l'extraction d'un élément... ;
- une meilleure possibilité est d'utiliser une liste chaînée en se restreignant aux opérateurs `append`, `popleft` et `len` pour savoir si la file est vide ;
- l'idéal est d'avoir un type dédié, par exemple `Queue` dans le module `queue`, par exemple (on suppose qu'il existe un dictionnaire `priorités`) :

```
from queue import Queue
file_fifo = Queue()
# ... des choses sont ajoutées dans la file d'attente
# on veut traiter tout ce qui a une priorité minimale d'abord, et
#   on ajuste le niveau de priorité au fur et à mesure
while file_fifo.empty() is False:
    elt = file_fifo.get()
    if priorités[elt] > prio_min:
        traitement(elt)
    else:
        file_fifo.put(elt)
    prio_min = mise_à_jour_prio_min()
```

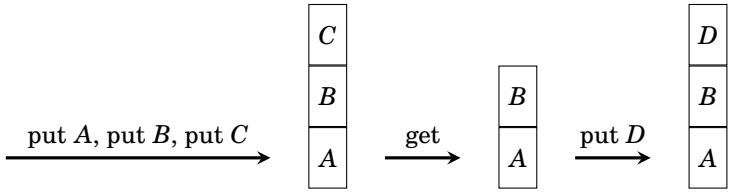
3.2. Piles

Une pile est un autre type de structure de données, comparable à une pile d'assiettes. Une pile obéit à la loi « dernier arrivé, premier sorti » ou Last In, First Out (abrégé LIFO) : la dernière assiette ajoutée à une pile d'assiette est la première prise.

Définition d'une pile

Une pile est une structure de données où les opérations autorisées sont
- *empiler* un élément (« put ») c'est-à-dire ajouter un élément en haut de la pile ;
- *dépiler* un élément (« get ») c'est-à dire enlever l'élément situé en haut de la pile ;
- tester si la pile est vide.

Si on empile A puis B puis C, si on dépile alors on enlève l'élément situé en haut de la pile qui se trouve être le dernier empilé.



Les piles sont particulièrement recommandées lorsque l'on peut se contenter d'accéder au dernier élément stocké, par exemple pour programmer :
- la fonction « page précédente » dans un navigateur ;
- la fonction « annuler action » dans un traitement de texte ;
- des algorithmes de parcours exhaustifs (graphes, labyrinthes...)

En revanche si l'on doit pouvoir accéder un n'importe quel élément stocké, alors on préférera utiliser une liste chaînée ou un tableau.

On peut simuler une telle structure en python de plusieurs manières :

- une façon maladroite utilise une liste Python avec `append` pour ajouter un élément à la file, et `pop()` pour l'enlever. Cela conduit à un coût $\mathcal{O}(1)$ amorti pour l'insertion d'un élément, mais pas un $\mathcal{O}(1)$ garanti, alors qu'on pourrait... ;
- une meilleure possibilité est d'utiliser une liste chaînée en se restreignant aux opérateurs `append`, `pop` et `len` pour savoir si la file est vide ;
- l'idéal est d'avoir un type dédié, par exemple `LifoQueue` dans le module `queue`.

3.3. Exercices : tours de cartes

- Écrire une fonction `créerPaquet(n, k)` qui créé une pile représentant un paquet de n cartes numérotées de 0 à $k-1$ (par exemple, un jeu de 52 cartes possède 13 numéros de cartes, donc $n = 52$, $k = 13$, et $n/k = 4$ couleurs), rangées par couleur et dans l'ordre.
- Écrire une fonction `couper(p)` qui prend un paquet de cartes, en enlève un nombre aléatoire sur le sommet et les renvoie dans une seconde pile, en ordre inverse. Par exemple, si la pile est `p = [0, 4, 3, 4, 1, 2, 3, 0, 2, 1]`, un résultat possible est de renvoyer `q = [1, 2, 0, 3]` et que la pile de départ soit devenue `p = [0, 4, 3, 4, 1, 2]`. On utilisera `random.randint(mini, maxi)` pour déterminer l'endroit aléatoire de la coupure (revoir la doc de cette fonction pour éviter les erreurs sur les bornes).
- Écrire une procédure `melange(p, q)` qui prend en entrée deux paquets et qui renvoie un paquet `r` formé du mélange suivant :
  - ▷ tant qu'aucun paquet n'est vide, on dépile aléatoirement `p` ou `q` et on empile l'élément sur `r` ;
  - ▷ si un des paquets est vide, on dépile les éléments restants de l'autre paquet sur `r`.
- Tour de magie de Gilbreath : créer un paquet formé de `m` répétitions de `k` cartes (par exemple, pour 4 répétitions de 3 cartes : `p = [0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2]`). Le couper puis le mélanger avec les fonctions précédentes. Vérifier que l'on retrouve `m` répétitions successives des `k` cartes, éventuellement mélangées au sein d'une répétition.
- Jeu de bataille : créer un jeu complet de 54 cartes (quatre ensembles de 13 cartes), le mélanger plusieurs fois. Séparer le paquet mélangé en deux paquets de taille égale. Deux joueurs découvrent alternativement le sommet de leur paquet ; celui qui a la carte la plus forte marque un point. Écrire une fonction qui déroule la partie et proclame le gagnant.

On pourra adapter l'ensemble de ces exercices (sauf le dernier) de cartes avec des files, ou encore avec des listes chaînées en utilisant aléatoirement l'insertion/retrait à gauche ou à droite.

3.4. Exercices : mots bien parenthésés

Un mot est dit bien parenthésé lorsqu'il y a autant de parenthèses ouvrantes que fermantes, et si à chaque parenthèse ouvrante on peut faire correspondre une unique parenthèse fermante située après. Par exemple, le mot « salut (dis-je) (enfin) (je crois (quoique)) » est bien parenthésé alors que le mot « salut (les MPSI-1)) ! (» est mal parenthésé. Lorsque l'on utilise un logiciel de programmation tel que emacs, VSCode ou Spyder, à chaque fois qu'une parenthèse fermante est tapée, elle est associée à la parenthèse ouvrante correspondante. Pour gérer les parenthèses dans une expression, ces logiciels utilisent une pile. Nous nous intéressons à ce problème dans cette partie.

Étant donné un paragraphe de code, on souhaite obtenir les couples de parenthèses ouvrante/fermante. De plus, si le mot est mal parenthésé on doit retourner la parenthèse qui n'est pas fermée et/ou ouverte.

La méthode proposée est la suivante : on créé une pile dans laquelle on empile les positions des parenthèses ouvrantes. Dès que l'on rencontre une parenthèse fermante, on dépile et on retourne le couple. Si on rencontre une parenthèse fermante et que la pile est vide, c'est que l'on a rencontré une parenthèse fermante en trop. Si à la fin du parcours du paragraphe, la pile n'est pas vide c'est que l'on a trop de parenthèses ouvrantes.

Écrire une fonction `verif(c)` qui prend en argument une chaîne de caractère `c` et renvoie une liste contenant trois listes :

- la première contient des paires d'indices correspondants aux couples de parenthèses ouvrantes / fermantes
- la deuxième contient les indices correspondant aux parenthèses ouvrantes en trop
- la troisième contient les indices correspondant aux parenthèses fermantes en trop