Tableau statique et dynamique

1 - Tableau statique

1.1 Tab. statique voir Construit 5.3.1

1.2 RAM

Mémoire vive : La mémoire vive est la zone dans laquelle on place les programmes en cours d'exécution et les données sur lesquelles ils travaillent.

RAM est l'acronyme de **Random Access Memory**, "mémoire à accès aléatoire" ? L'accès aléatoire veut dire qu'on peut atteindre directement une case mémoire et en lire le contenu pourvu qu'on connaisse son adresse.

1.3 Principe du tableau

Propriétés générales d'organisation : En informatique, un tableau désigne un conteneur de données qui a les propriétés suivantes :

- Les octets sont consécutifs en mémoire à partir d'une première adresse A.
- Le tableau occupe une largeur de L octets.
- Chaque case encode le même type de données, et comporte C octets.

Exemple : pour 1000 integers sur 4 octets.

- Imaginons que le premier octet du tableau soit stocké à l'adresse A = 500 dans la mémoire.
- Chaque case contient C = 4 octets.
- Le tableau occupe donc L = 1000 * 4 = 4000 octets consécutifs allant de l'adresse 500 à l'adresse 4499.
- Chaque octet est bien entendu représentable comme un entier compris entre 0 et 255.
- Les 4 octets 500 à 503 représentent collectivement le premier entier.

Indice des cases

- On identifie chaque case par un simple numéro qu'on appelle indice. Attention : l'indice de la première case n'est pas 1 mais 0
- t[0] veut donc dire d'aller décoder le contenu DES CASES qui correspondent à l'indice 0 : les cases 500 - 501 - 502 - 503 ici.
- t[1] veut donc dire d'aller décoder le contenu DES CASES qui correspondent à l'indice 1 : les cases 504 - 505 - 506 - 507 ici.

Espace des noms : L'espace des noms fait le lien entre le nom d'une variable (ici t) et les données du tableau mais... la variable ne référence pas les données, uniquement l'adresse à partir de laqulle elles sont stockées (ici 500).

Exemple visuel : des données, stockées à partir de l'adresse 200, ont toutes une taille de 4 octets.

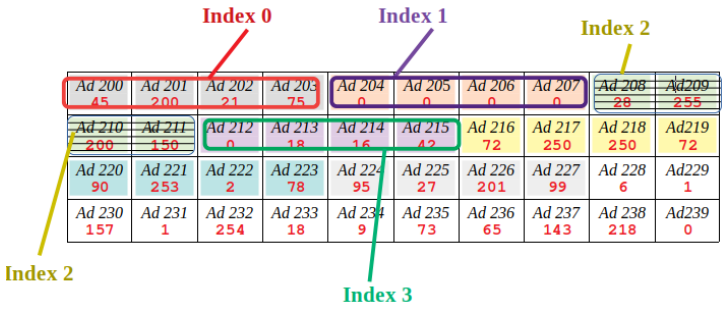

Figure 15.1 – Principe du tableau

1.4 Accès à coût CONSTANT

Accès à Coût CONSTANT : L'accès à **n'importe quelle case du tableau** se fait à coût constant : quelque soit la taille du tableau, atteindre une case particulière prend le même temps. Voyons pourquoi.

Explications

- **1 - RAM** L'accès à une case-mémoire dont on connait l'adresse est à coût CONSTANT.
- **2 - Espace des noms** L'espace des noms fait le lien entre le nom d'un tableau et l'adresse A de son premier octet, à coût CONSTANT.
- **3 - Calcul de l'adresse d'une case du tableau** Connaissant l'adresse A du début du tableau (500 puis 200 sur les exemples), la largeur C des cases du tableau (4 sur les exemples), et l'indice i de la case voulue, on peut calculer à coût CONSTANT l'adresse du premier octet à lire :

```
Adresse du 1er octet à lire = A + i * C
```

Avec A=500 et C=4 :

`t[0]` : commence à 500 + 0*4 = 500

`t[1]` : commence à 500 + 1*4 = 504

`t[2]` : commence à 500 + 2*4 = 508

`t[3]` : commence à 500 + 0*3 = 512

— **4 - Conclusion** Lorsqu'on tape t[2], Python obtient (via l'espace des noms) l'adresse 500 (action à coût CONSTANT), calcule l'adresse de la case 2 (500 + 2*4 = 508, action à coût CONSTANT), lit le contenu des cases 508, 509, 510 et 511 (4 actions à coût CONSTANT) et décode le contenu et renvoie la réponse (action à coût CONSTANT). L'ensemble forme donc une expression à coût CONSTANT.

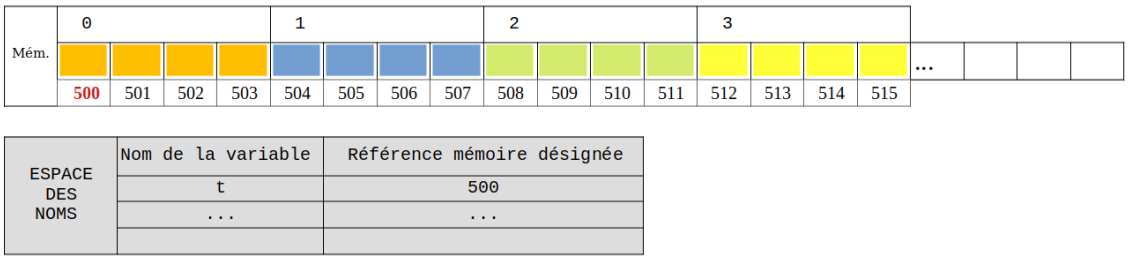

FIGURE 15.2 – Espace des noms et tableau

1.5 Type `list` de Python : un tableau non homogène

Le type natif **`list`** de Python permet de créer des tableaux en apparence non-homogène. Comment est-ce possible ?

En réalité, un type list référence un **tableau d'adresses** (8 octets sur les systèmes 64 bits). Dans les cases du tableau, on stocke donc **l'adresse du contenu** qu'on veut mettre à cette position et pas directement le contenu.

Grace à cette stratégie, on obtient des tableaux qui semblent ne pas avoir exactement le même type dans chaque case.

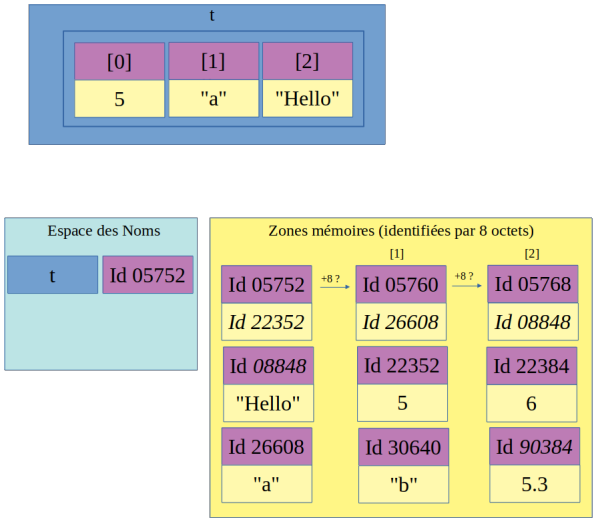

FIGURE 15.3 – Espace des noms et list-Python

2 - Tableaux dynamiques

2.1 Présentation rapide

Tableau statique avec le type `list`

Dans un tableau statique :

— les cases contiennent le même type de données

— le nombre de cases est constant : après création, on ne peut pas en supprimer ou en rajouter.

Les tableaux statiques s'implémentent en Python à l'aide du type **`list`**. D'ailleurs, ontrairement à un tableau statique standard, Python accepte des contenus de types différents dans les différentes cases.

Type `list` : tableau dynamique

Le type **`list`** de Python est encore plus flexible que cela : il s'agit d'un **tableau dynamique** (on peut ajouter et supprimer des cases).

On pourrait donc créer une list Python contenant trois cases comme celle-ci : **`exemple = [1, 2, 5]`**

Et plus tard dans le programme, le tableau pourrait faire référence à 6 cases : **`[1, 2, 5, 10, 20, -10]`**

Ou même simplement 1 case : **`[5]`**

2.2 `list.append()` (tab.dyn.)

Principe et syntaxe

La méthode **`append()`** rajoute une case contenant la valeur voulue à la fin du tableau. Toujours à la fin. La syntaxe est la suivante :

```
nom_du_tableau.append(contenu_de_la_nouvelle_case)
```

Insertion en fin : coût **CONSTANT**, c'est donc une opération efficace qu'on peut utiliser sans dégrader les performances du programme.

Exemple

```
>>> tab = []
>>> tab.append(10)  # Ajout de 10 en fin
>>> tab
[10]

>>> tab.append(20)  # Ajout de 20 en fin
>>> tab
[10, 20]

>>> tab.append(30)  # Ajout de 30 en fin
>>> tab
[10, 20, 30]

>>> tab.append(0)  # Ajout de 0 en fin
>>> tab
[10, 20, 30, 0]
```

L'élément est toujours ajouté à la fin du tableau.

Modifie en place, aucune réponse renvoyée

La méthode ne renvoie aucune réponse : elle modifie en place le tableau sur lequel elle doit agir.

Une **erreur très courante**, qu'il ne faut donc jamais faire : réaliser une affectation de ce type :

```
>>> tab = [10, 20, 30]
>>> tab = tab.append(100)
>>> tab
```

Ici, on place dans **`tab`** le résultat renvoyé par append(), à savoir... rien. On a donc involontairement "vidé" **`tab`**. Notez que cela ne provoque pas d'erreur, c'est une erreur sémantique.

2.3 `list.pop()` sans arg. (tab.dyn.)

Principe et syntaxe

La méthode **`pop()`** supprime la dernière case d'un tableau ET renvoie la valeur supprimée : cela permet de la stocker si on le désire. La syntaxe est la suivante :

```
nom_du_tableau.pop()
```

Suppression en fin : coût **CONSTANT**.

Exemple

```
>>> tab = [10, 20, 30]
>>> tab.pop()      # supprimer sans mémoriser
>>> tab
[10, 20]

>>> a = tab.pop()  # on stocke le 20 qu'on va supprimer
>>> tab
[10]

>>> tab.pop()   # supprimer sans mémoriser
>>> tab
[]

>>> a
20

>>> tab.pop()   # supprimer une case d'un tableau vide ?
IndexError: pop from empty list
```

Gérer le cas problématique

Attention : vouloir supprimer une case d'un tableau vide lève une exception. Si vous n'êtes pas certain qu'il reste une case à supprimer, il faut donc tester avant d'agir :

```
>>> tab = [10]
>>> tab.pop()  # supprimer la case de fin
>>> tab
[]

>>> if tab: tab.pop()  # si tab n'est pas vide

>>> if len(tab) > 0: tab.pop()  # si tab n'est pas vide
```

Il n'y aura d'activation de la méthode que si le tableau contient un élément au moins.

Modifie en place et renvoie la valeur supprimée

La méthode renvoie la valeur supprimée. C'est le développeur qui décide ce qu'il veut en faire. Ne pas réaliser d'affectation permet donc de supprimer la valeur.

2.4 `list.pop(i)` avec arg. (tab.dyn.)

Principe et syntaxe

La méthode **`pop(i)`** supprime la case d'indice i du tableau ET renvoie la valeur supprimée : cela permet de la stocker si on le désire. La syntaxe est la suivante :

```
nom_du_tableau.pop(i)
```

Suppression sur une position quelconque : coût **LINEAIRE**, c'est donc une opération à utiliser avec précaution car elle peut dégrader les performances du programme si le tableau possède beaucoup de cases.

Exemple

```
>>> tab = [10, 20, 30]
>>> tab.pop(1) # supp. la case 1 sans mémoriser
>>> tab
[10, 30]

>>> a = tab.pop(0) # on stocke le 10 qu'on supp.
>>> tab
[30]

>>> a
10
```

2.5 `list.remove(x)` (tab.dyn.)

Principe et syntaxe

La méthode **`remove(x)`** supprime la première case contenant **`x`** dans le tableau. Attention, elle ne renvoie rien. La syntaxe est la suivante :

```
nom_du_tableau.remove(valeur_a_supprimer)
```

Suppression d'un élément quelconque : coût **LINEAIRE**.

Exemple

```
>>> tab = [10, 20, 30, 10, 40]
>>> tab.remove(20)  # supp. la case qui réf. 20
>>> tab
[10, 30, 10, 40]

>>> tab.remove(10)  # supp. la 1er case qui réf. 10
>>> tab
[30, 10, 40]

>>> tab.remove(10)  # supp. la case qui réf. 10
>>> tab
[30, 40]

>>> tab.remove(10)  # supp. une case qui réf. 10 ?
ValueError: list.remove(x): x not in list
```

Gérer le cas problématique

Attention, tenter de supprimer une valeur qui n'existe pas lève une exception. Si vous n'êtes pas certain qu'un élément que vous voulez supprimer est présent, il faut donc tester avant d'agir :

```
>>> tab = [10, 20, 30, 10, 40]
>>> tab.remove(20)  # supp. la case qui réf. 20
>>> tab
[10, 30, 10, 40]

>>> if 50 in tab: tab.remove(50)
```

Il n'y aura pas d'activation de la méthode puisque 50 n'est pas dans le tableau. Et donc pas d'erreur.

Modifie en place, aucune réponse renvoyée

La méthode ne renvoie aucune réponse : elle modifie en place le tableau sur lequel elle doit agir. Une **erreur très courante**, qu'il ne faut donc jamais faire : réaliser une affectation de ce type :

```
>>> tab = [10, 20, 30]
>>> tab = tab.remove(20)
>>> tab
```

Ici, on place dans **`tab`** le résultat renvoyé par **`remove()`**, à savoir... rien. On a donc involontairement vidé **`tab`** de tout contenu. C'est une erreur sémantique.

3 - RECAP des coûts

ACCES à la case **`i`** : CONSTANT.

`append()` : CONSTANT.

`pop()` : CONSTANT.

`pop(i)` : LINEAIRE.

`remove(x)` : LINEAIRE.