Listes (bis)

Skander Zannad et Judicaël Courant

Lycée La Martinière-Monplaisir

2013-11-05

1 Tableaux python

1.1 Des tableaux oui...

Déjà vu :

1. Construire un tableau en extension ;
2. Concaténer des tableaux ;
3. Accès à un élément ;
4. Modification d'un élément ;
5. Construire un tableau par tranchage (slicing).

(accès/modification : coût constant, concaténation/tranchage/construction : coût proportionnel à la taille de l'objet résultat)

1.2 ... mais des dynamiques !

On peut ajouter un élément en fin de tableau :

```
t = [ 42, 17 ]
t.append(10)
# t vaut maintenant [ 42, 17, 10 ]
```

On peut aussi enlever le dernier :

```
t = [ 42, 17, 33, 15]
x = t.pop()
# x vaut maintenant 15 et t [ 42, 17, 33 ]
```

NB : pour calculer `t.append(10)`, python «calcule» l'objet désigné par `t` (qui est une adresse mémoire) et lui applique la méthode `append`. Donc `t.append(10)` change l'objet désigné par la variable `t`, pas la variable `t` (qui pointe toujours vers la même adresse mémoire). Idem pour `t.pop()`

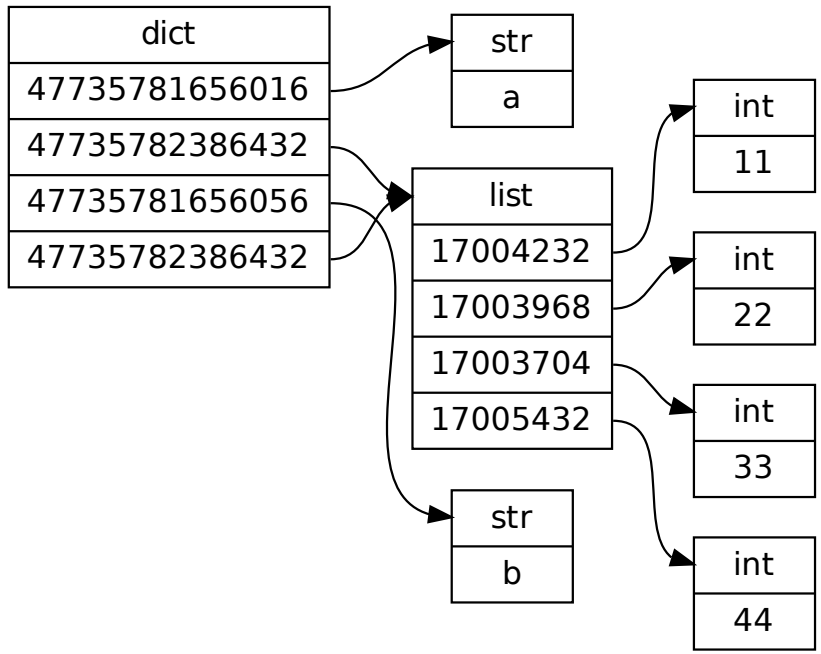
1.3 Comment ça marche

Résumé des épisodes précédents :

Après l'exécution de

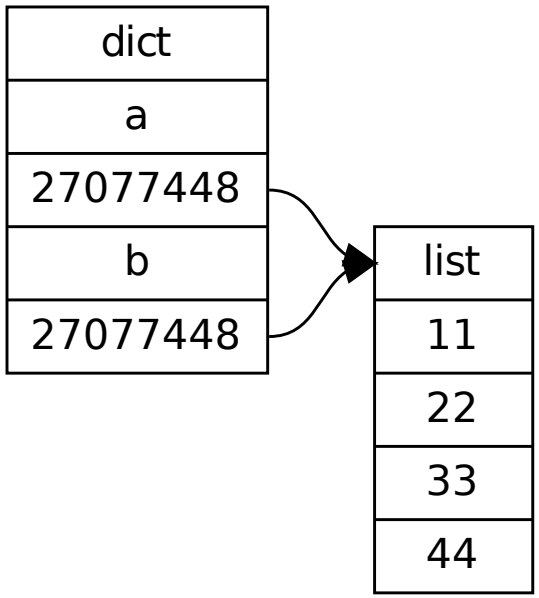
```
a = [11, 22, 33, 44]
b = a
```

on a en mémoire :



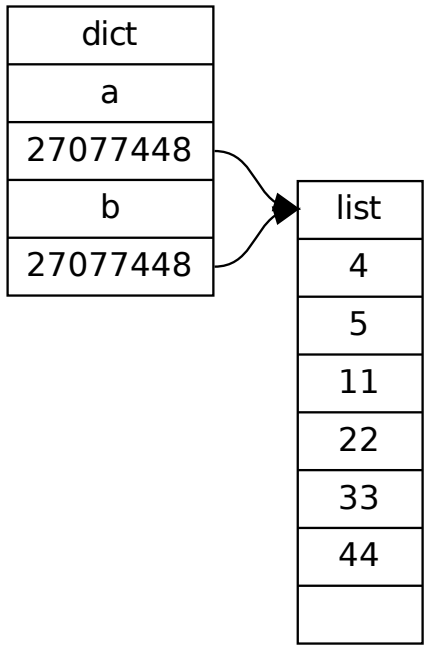
Pour alléger les dessins :

1. On prend uniquement des listes d'entiers
2. On abrège la représentation des éléments de la liste et du dictionnaire des variables :



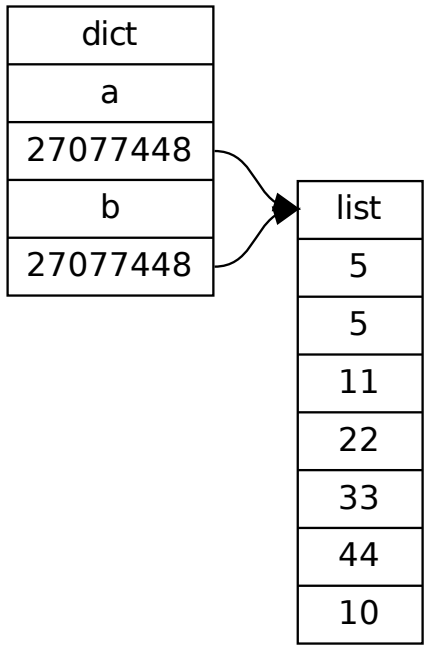
Idée pour ajouter un élément à une liste existante :

1. Garder des cases non utilisées dans le tableau
2. Et en tête du tableau, le nombre de cases réellement utilisées et le nombre de cases allouées pour le tableau.



Pour exécuter `a.append(10)`, python

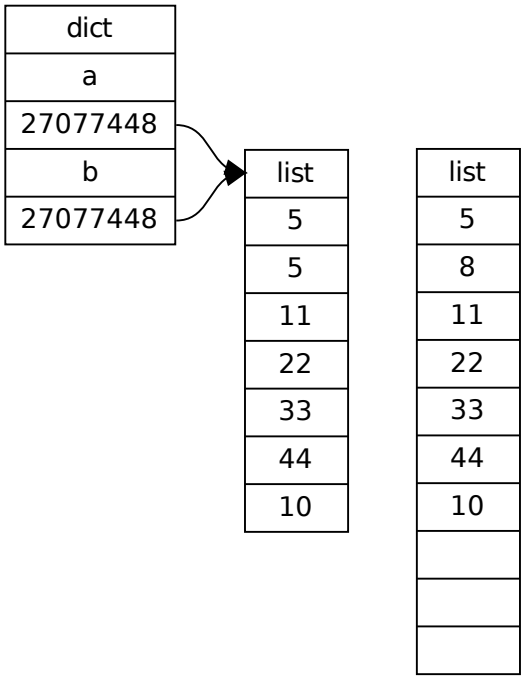
1. Met 10 dans la case suivant 44.
2. Augmente le compte du nombre de cases utilisées.



Problème : que faire quand le tableau est plein ?

1. On voudrait augmenter la taille du tableau.
   Mais a priori la case mémoire suivante peut être occupée.
2. On peut essayer de créer une nouvelle liste.

Comme ceci :



Rappel : En python la valeur d'un objet est son adresse.

1. On veut que `append` change l'objet.
2. Python n'a aucun moyen de changer l'adresse pointée par `a` et `b` à partir de l'objet.
3. Donc rendre un nouvel objet n'est pas adapté.

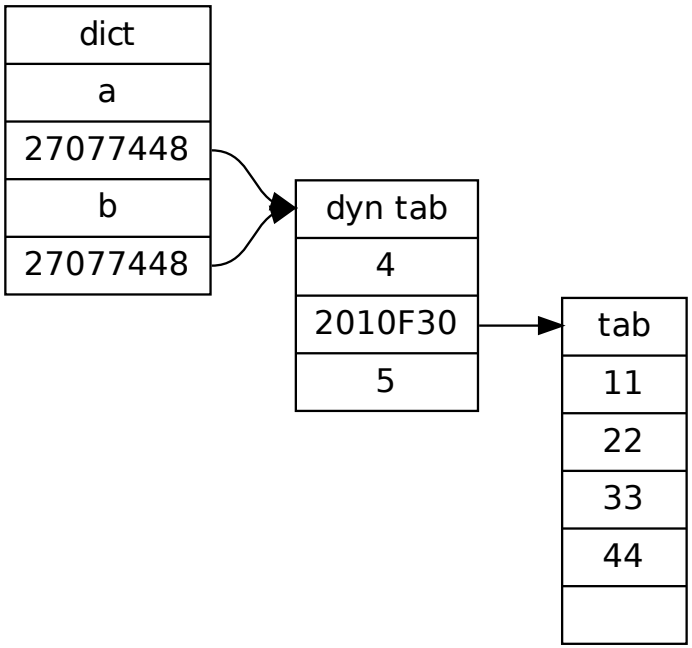
Solution possible pour réaliser des tableaux dynamiques : ajouter une indirection.

Tableau dynamique =

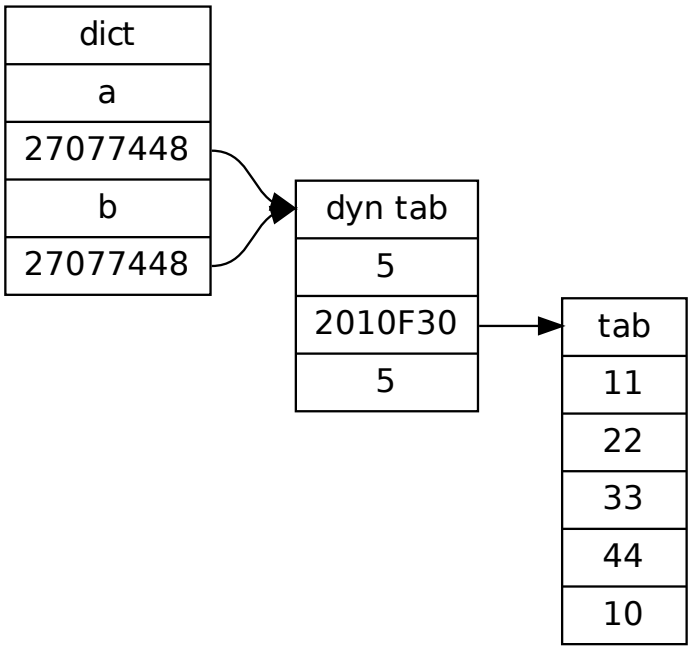
Tableau statique (zone mémoire dans laquelle seront stockées les données)

\+ Objet pointant sur le tableau statique et contenant également sa taille *physique* (nombre d'éléments) et sa taille *logique* (nombre d'éléments utilisés)

Listes (bis)

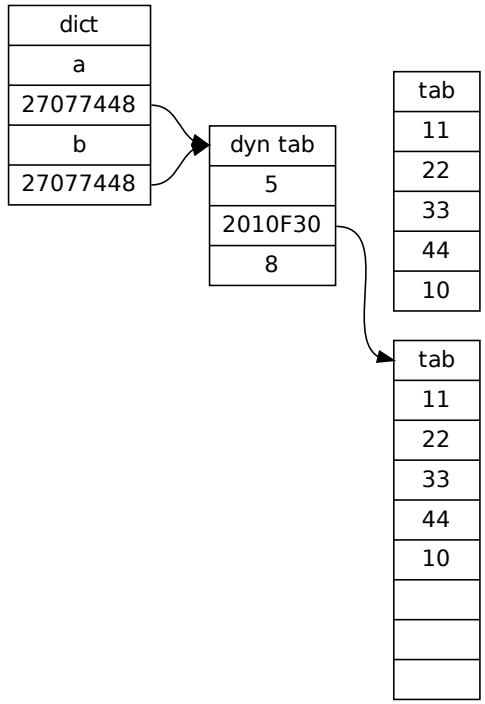


Exécution de `a.append(10)` : pas de surprise.



Pour ajouter un nouvel élément, il faut changer de tableau statique :

1. On alloue un tableau statique plus grand.
2. On recopie les données dans le tableau statique.
3. On fait pointer l'objet tableau dynamique vers le nouveau tableau statique et on met à jour la case donnant la taille physique du tableau statique.



C'est comme cela que les tableaux dynamiques sont implantés en Python.

2 Stratégie de redimensionnement

Lorsque le tableau statique de n cases physiques est plein et qu'on veut ajouter un élément il faut l'agrandir. De combien ?

On peut adopter plusieurs stratégies :

1. Ajouter une case.
2. Ajouter un nombre fixé de cases (par exemple 10 cases).
3. Doubler le nombre de cases.
4. Ajouter un nombre de cases proportionnel à n (par exemple $n/10$).

Quelle est la meilleure stratégie ?

Remarque :

1. Doubler le nombre de cases «gaspille» $n-1$ cases mémoires mais permet d'attendre un certain temps avant de redimensionner.
2. Ajouter une case à la fois ne gaspille pas de case mais oblige à redimensionner à chaque ajout, ce qui pourrait être coûteux en temps.

Conclusion provisoire :

1. Il faut regarder plus en détail.
2. Compromis espace/temps (situation fréquente en informatique).

3 Complexité de l'ajout

3.1 Cas le pire

Soit t un tableau dynamique de taille n.

Dans le cas le pire : On a besoin de redimensionner le tableau, donc d'allouer un tableau de taille k (où $k > n$).

Coûts en temps :

1. Création d'un nouveau tableau vide. Coût en temps : pas très clair mais raisonnablement compris entre une constante et k multiplié par une constante. Donc $O(k)$.
2. Copie des éléments de l'ancien tableau : $\Theta(n)$.
3. Modification de l'objet tableau dynamique : $O(1)$.

Coût total : $\Omega(n)$ et $O(n+k)$.

En faisant l'hypothèse que k est choisi dans $]n, 2n]$: $O(n+k) = O(n)$.

Donc coût total : $\Theta(n)$.

Coût en espace dans le cas le pire :

1. Temporairement : un tableau de k cases plus un tableau de n cases, soit $O(n+k) = O(n)$ cases (si $k = O(n)$).
2. Au final : k cases (plus constante) soit $O(n)$ cases (si $k = O(n)$).

3.2 Notion de coût amorti

Pour n fixé, quel est le temps $t(n)$ mis par le programme suivant ?

```
a = [ 0 ]
for i in range(n):
    a.append(i)
```

Cela dépend de la stratégie de redimensionnement :

– Avec la stratégie d'ajout systématique d'une case, à l'étape i, on a redimensionnement donc un coût $\Theta(i)$, c'est-à-dire compris entre $ai + b$ et $ci + d$ où a, b, c, d sont des constantes (avec $a > 0$ et $c > 0$). Or $\sum_{i=0}^{n-1}(ai + b) = an(n-1)/2 + bn$, donc

$$t(n) \leq a\frac{n^2}{2} + (b - a/2)n$$
$$t(n) \leq (a + \frac{b - a/2}{n})n^2$$
$$t(n) = O(n^2)$$

De même $t(n) = \Omega(n^2)$. Donc $t(n) = \Theta(n^2)$.

– Avec la stratégie de doublement de la taille :

- Aux étapes sans redimensionnement, coût constant C, donc au total, temps Cn pour l'ensemble des étapes sans redimensionnement.
- Pour chaque ajout i avec redimensionnement, coût au plus $\alpha i + \beta$.
- Étapes de redimensionnement : quand la liste a 1 élément, puis 2, puis 4, puis 8, ...
- Coût total pour les étapes avec redimensionnement :

$$\sum_{\substack{k=0 \\ 2^k \leq n}} (\alpha 2^k + \beta) = \alpha(2^{\lfloor \log_2 n \rfloor + 1} - 1) + \beta \lfloor \log_2 n \rfloor \leq 2\alpha n + \beta \lfloor \log_2 n \rfloor$$

– Coût total au final :

$$t(n) \leq Cn + 2\alpha n + \beta \lfloor \log_2 n \rfloor$$
$$t(n) = O(n)$$

Et par ailleurs, il a fallu faire n opérations, donc $t(n) = \Omega(n)$.

Donc $t(n) = \Theta(n)$.

Résumé :

- Dans les deux cas, au final l'espace mémoire utilisé est $\Omega(n)$.
- Pour la stratégie d'ajout d'une case à chaque redimensionnement : $\Theta(n^2)$.
- Pour la stratégie de doublement à chaque étape : $\Theta(n)$. C'est nettement mieux : n ajouts ont coûté des temps respectifs $\Omega(n)$. *Comme* si on avait un coût constant pour chaque ajout. On parle de *coût amorti* constant.

NB : attention, la notion de coût en *moyenne* désigne autre chose.

De manière générale :

- Ajout d'un nombre constant de cases : complexité amortie égale à la complexité du cas le pire ($\Theta(n)$).
- Ajout d'un nombre proportionnel : complexité amortie $O(1)$.

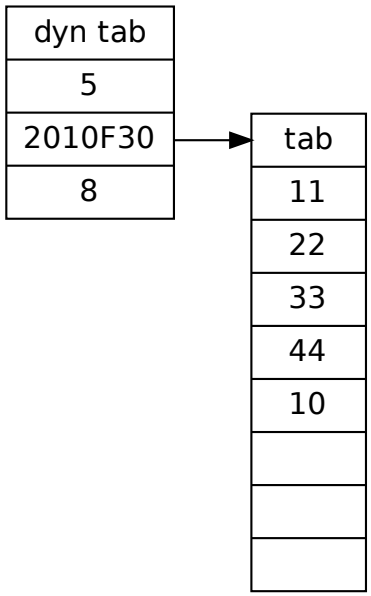
En python : stratégie légèrement plus complexe. Grosso modo : ajout d'un nombre constant de cases plus $n/8$ (coût amorti constant).

4 Autres opérations proposées par Python

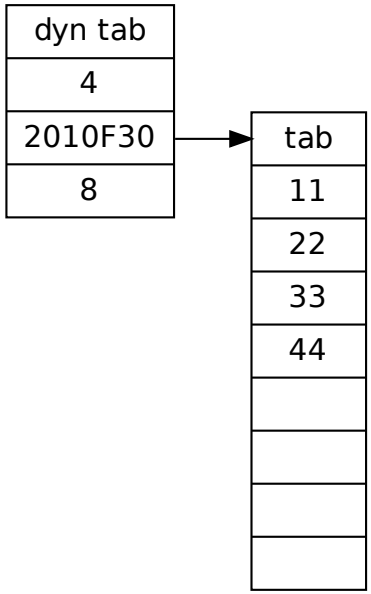
Soit `t` un tableau de taille n, `u` un tableau de taille k, et $i \in [\![0, n[\![$.

– `t.pop()` : enlève le dernier élément.

Avant :



Après :



Si le tableau statique est peu utilisé : redimensionnement. Complexité :

- cas le pire : $\Theta(n)$
- amortie : $O(1)$

- `t.pop(i)` : enlève l'élément de la case i. Pour cela :
  - Décale chacun des éléments situés après l'indice i sur le précédent (coût : $n - i - 1$ recopies).
  - Puis fait un `pop()` (coût amorti $O(1)$, coût dans le cas le pire $\Theta(n)$).
- `t.insert(i, x)` insère `x` juste avant l'indice i. Pour cela :
  - Redimensionne si nécessaire (cas le pire $\Theta(n)$, coût amorti $O(1)$).
  - Recopie tous les éléments du dernier jusqu'à l'indice i sur la case suivante (coût $n - i$ recopie).
  - Écrit x case i (coût unitaire).

- `t.extend(u)` ajoute successivement les éléments de `u` à `t`. Complexité dans le cas le pire : $O(n+k)$. Complexité amortie $O(k)$.
- `t.reverse()`. Complexité $O(n)$.
- `t.sort()`. Complexité $O(n \log n)$.
- `t.remove(x)`. Complexité $O(n)$.
- `t.count(x)`. Complexité $O(n)$.
- `t.index(x)`. Complexité $O(n)$.

5 Conclusion

1. Les listes python ne sont pas des tableaux de taille fixée mais des tableaux dynamiques.
2. Implantation des tableaux dynamiques plus compliquée qu'une implantation statique.
3. Il y a des opérations de coût « naturellement » constant : accès et modification d'un élément.
4. Les autres ont toutes une complexité plus élevée ($O(n)$ dans le cas le pire).
5. Certaines ont une complexité *amortie* constante.