Listes

Alix Munier-Kordon et Maryse Pelletier

LIP6
Université P. et M. Curie
Paris

2I003 initiation à l'algorithmique

Plan du cours

Notion de liste

Definition

Une liste $L = (a_0, \cdots, a_n)$ est une succession d'éléments.

```
jour=['lundi', 'mardi', 'mercredi']
corbeille=[56, 'jeudi', 45, 67, 'coucou']
```

Quels points communs (différences) voyez-vous entre listes et ensembles ?

Definition

La liste L est *homogène* si tous ses éléments sont de même type.

Sous-liste

Definition

Soit $L = (a_0, \cdots, a_n)$ une liste. Pour tout couple d'entiers $(i, j) \in \{0, \cdots, n\}$ avec $i \leq j$, $L[i : j]$ est la sous-liste de L définie par :

$$L[i : j] = (a_i, \cdots, a_{j-1})$$

Généralisation de l'accès direct : $L[i] = a_i$.

```
»> corbeille=[56, 'jeudi', 45, 67, 'coucou', 56]
»> corbeille[0:2]
[56, 'jeudi']
»> corbeille[:3]
[56, 'jeudi', 45]
»> corbeille[2:]
[45, 67, 'coucou', 56]
```

Construction

Une liste est en général construite par insertions successives :

L.append(x): insertion de l'élément x en queue de la liste L;

```
»> jour=[ ]
»> jour.append('mardi')
»> jour.append('mercredi')
»> jour.append('vendredi')
»> jour
['mardi', 'mercredi', 'vendredi']
```

Construction (suite)

On peut également insérer dans une position donnée :

L. insert(*i*, *x*) : insertion de l'élément *x* en *i*-ème place (positions numérotées à partir de 0).

En tête....

```
»> jour= ['mardi', 'mercredi', 'vendredi']
»> jour.insert(0,'lundi')
»> jour
['lundi', 'mardi', 'mercredi', 'vendredi']
```

ou n'importe où....

```
»> jour.insert(3,'jeudi')
»> jour
['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi']
```

Destruction

L. pop(*i*) : renvoie l'élément en *i*-ème position (positions numérotées à partir de 0) et le supprime de la liste.

```
»> jour=['lundi', 'mardi', 'mercredi', 'jeudi',
'vendredi']
»> jour.pop(2)
'mercredi'
»> jour
['lundi', 'mardi', 'jeudi', 'vendredi']
```

Destruction (suite)

L.remove(*x*): détruit la première instance de l'élément *x* dans la liste *L*;

```
»> MaListe=[5, 8, 9, 4, 7, 4, 3, 6, 4, 2]
»> MaListe.remove(4)
»> MaListe
[5, 8, 9, 7, 4, 3, 6, 4, 2]
```

Comptage

len(L): renvoie le nombre d'éléments de L;

L.index(x): indice du premier élément de valeur x dans L;

L. count(x) : nombre d'occurrences de x dans L

```
»> MaListe=[5, 8, 9, 4, 7, 4, 3, 6, 4, 2]
»> len(MaListe)
10
»> MaListe.index(4)
3
»> MaListe.count(4)
3
```

Opérations sur les listes

$L_1 + L_2$: renvoie la concaténation des deux listes;

$L \star k$: crée une liste de k occurrences de L.

```
»> L1=[5, 8, 9, 4]
»> L2=[8, 2, 5]
»> L1+L2
[5, 8, 9, 4, 8, 2, 5]
»> L1*2
[5, 8, 9, 4, 5, 8, 9, 4]
```

Tableaux

Un tableau est une zone contiguë en mémoire qui permet de stocker un nombre fixé d'éléments de même type.

1. Alloué statiquement (dans la pile) ou dynamiquement (dans le tas);
2. L'accès au i-ème élément est en $\Theta(1)$ et correspond à un calcul d'adresse;
3. Difficile de rajouter des éléments. Le cas échéant, il est sur-dimensionné et le nombre d'éléments stockés est sauvegardé dans une variable.

Liste simplement chaînées

Une liste simplement chaînée est une structure allouée dynamiquement composée de cellules, chacune d'elles stockant une donnée et pointant vers la cellule suivante.

1. La dernière cellule pointe sur le vide;
2. On accède aux éléments de la liste les uns après les autres en partant du premier élément;
3. Bien adapté quand le nombre d'éléments à stocker n'est pas connu à l'avance.

Liste circulaire doublement chainée

Une liste circulaire doublement chaînée est une structure allouée dynamiquement composée de cellules, chacune d'elles pointant à la fois sur l'élément suivant et sur l'élément précédent dans la liste (avec deux champs différents).

1. La dernière (resp. première) cellule pointe sur la première (resp.dernière);
2. On accéde aux éléments de la liste les uns après les autres en partant du premier élément (dans les deux sens possibles);
3. Plus difficile à programmer et plus coûteux en place mémoire que les listes simplement chaînées;
4. Accès au dernier élément en $\Theta(1)$.

Complexité des primitives d'accès direct

	$L[i]$	affichage de $L[i:j]$
Tableau	$\Theta(1)$	$\Theta(j-i)$
Liste simpl. chaînée	$\Theta(i)$	$\Theta(j)$
Liste doubl. circulaire	$\Theta(i)$	$\Theta(j)$

Complexité des primitives de construction

	L.append(x)	L.insert(i,x)
Tableau	$\Theta(1)$	$\Theta(n-i)$
Liste simpl. chaînée	$\Theta(n)$	$\Theta(i)$
Liste doubl. circulaire	$\Theta(1)$	$\Theta(n-i)$

$n = |L|$

Complexité des primitives de destruction

	L.pop(i)	L.remove(x)
Tableau	$\Theta(n-i)$	$\Theta(n)$
Liste simpl. chaînée	$\Theta(i)$	$\mathcal{O}(n)$
Liste doubl. circulaire	$\Theta(i)$	$\mathcal{O}(n)$

$n = |L|$

Complexité des primitives de comptage

	L.index(x)	L.count(x)
Tableau	$\mathcal{O}(n)$	$\Theta(n)$
Liste simpl. chaînée	$\mathcal{O}(n)$	$\Theta(n)$
Liste doubl. circulaire	$\mathcal{O}(n)$	$\Theta(n)$

$n = |L|$

Comment peut-on faire pour accélérer ces primitives pour un tableau ?

Complexité des primitives de comptage : la primitive len

1. Le nombre d'éléments d'un tableau doit être stocké dans une variable qui est modifiée à chaque insertion/suppression d'éléments. Donc, la complexité de len(L) est $\Theta(1)$;
2. Pour une liste chaînée (circulaire ou non), la complexité de l'appel len(L) est $\Theta(n)$.

Complexité des opérations sur les listes

	$L_1 + L_2$	$L \star k$
Tableau	$\Theta(n_1 + n_2)$	$\Theta(n \times k)$
Liste simpl. chaînée	$\Theta(n_1)$	$\Theta(n \times k)$
Liste doubl. circulaire	$\Theta(1)$	$\Theta(n \times k)$

$n = |L|$, $n_1 = |L_1|$, et $n_2 = |L_2|$.

Exemple : incrémentation des éléments d'une liste

```
def incrementeListe (L):
        if (L == []):
                return []
        m = L.pop(0)
        L = incrementeListe(L)
        L.insert(0, m+1)
        return L
```

```
L = incrementeListe(L)
```

Complexité de incrementeListe en fonction de la représentation de L

	$pop(0)$	$insert(0, m)$	$incrementeListe(L)$
Tableau	$\Theta(n)$	$\Theta(n)$	$\Theta(n^2)$
Liste simpl. chaînée	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Liste doubl. circulaire	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$

$n = |L|$.

Parcours d'une liste

```
def imprimeListe(L):
   for a in L:
       print a
»> L=[2,8,9]
»> imprimeListe(L)
2
8
9
```

Parcourt un à un tous les éléments de la liste : accéder à l'élément suivant est alors en $\Theta(1)$ dans le cas d'une liste chaînée.

Exemple : fonction miroir

```
def swapp (tab, i, j):
    aux = tab[i]; tab[i]=tab[j]; tab[j]=aux

def miroir (tab):
    n = len(tab)
    j = n // 2
    if (n%2 == 0):   # Si n est pair
        i = n // 2 - 1
    else:
        i = n // 2
    while (j<n):
        swapp(tab, i, j)
        i = i -1;  j = j +1
```

Complexité de miroir en fonction de la représentation de *tab*

	$len(tab)$	$swapp(tab, i, j)$	$miroir(tab)$
Tableau	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Liste simpl. chaînée	$\Theta(n)$	$\Theta(max(i, j))$	$\Theta(n^2)$
Liste doubl. circulaire	$\Theta(n)$	$\Theta(max(i, j))$	$\Theta(n^2)$

$n = |tab|$.

Conclusion

- Plusieurs représentations possibles des listes avec des primitives d'accès et de gestion de complexité différentes ;
- Quand on évalue la complexité d'un algorithme, il faut connaître précisement la complexité de toutes les primitives associées aux structures de données ;
- Choisir la structure de données en fonction des traitements à réaliser.