Cours 3 : Structures linéaires usuelles et leur implantation

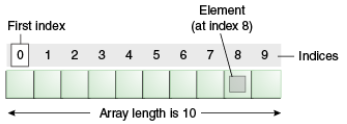
Jean-Stéphane Varré

Université Lille 1

jean-stephane.varre@univ-lille1.fr

La structure de données « tableau »

- suite ordonnée d'éléments (i.e. il existe un suivant et un précédent)
- de taille bornée
- chaque élément est associé à un indice
- qui supporte les opérations :
  - d'accès à un élément par son indice
  - d'accès au nombre d'éléments : la longueur
  - création étant donné sa longueur



tiré de la JavaDoc

En Java :

```
1  int[] t = new int[10];
```

En Python : cela n'existe pas (nativement)

La structure de données « liste »

- suite ordonnée d'éléments (i.e. il existe un suivant et un précédent)
- de taille non bornée
- qui supporte les opérations :
  - d'accès à la tête de la liste : le premier élément
  - d'accès au reste de la liste : tous les éléments sauf le premier
  - d'ajout en tête : ajoute un nouvel élément avant la tête
  - de test de la vacuité
- toutes les autres opérations sont une construites à partir de celles-ci
- c'est une structure de données intrinsèquement récursive

La structure de données « pile »

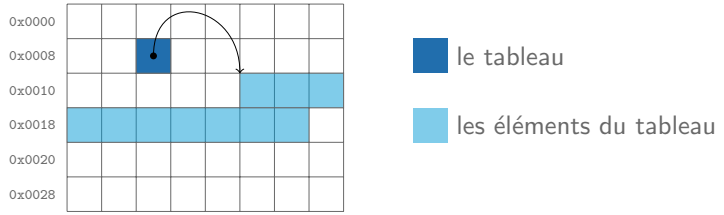
- suite ordonnée d'éléments
- de taille non bornée
- dont le dernier élément entré est le seul accessible immédiatement : dernier élément entré, premier sorti (LIFO : Last In First Out)
- qui supporte les opérations :
  - d'empilement : ajoute un élément
  - de dépilement : enlève le dernier élément entré
  - de lecture du sommet : accès au dernier élément entré
  - de test de vacuité

La structure de données « file »

- suite ordonnée d'éléments
- de taille non bornée
- premier élément entré, premier sorti (FIFO : First In First Out)
- qui supporte les opérations :
  - d'enfilage : ajoute un élément
  - de défilage : enlève le premier élément entré
  - de test de vacuité
- cas d'utilisation :
  - file d'impression
  - attente téléphonique
  - parcours d'arbre en largeur

Représentation mémoire d'un tableau

- un tableau est une structure de donnée statique : l'occupation en mémoire est fixée une fois pour toutes
- un tableau occupe une zone mémoire contigüe de taille sa longueur $\ell \times r$ avec r le nombre d'octets nécessaires pour représenter le type de ses éléments



- le décalage à la case suivante se fait en incrémentant de r l'adresse mémoire de la case courante

Représentation des listes

- une liste est une structure de donnée dynamique : l'occupation en mémoire peut varier au cours de son utilisation
- une liste occupe une zone mémoire non nécessairement contigüe
- l'occupation mémoire dépend de l'implantation réalisée
- les coûts des opérations primitives dépendent également de l'implantation réalisée

nous allons voir :

- représentation avec un tableau
- représentation avec des cellules (listes chaînées)
- représentation avec chaînage de tableaux

Implantation des listes - Extrait de la Javadoc

```
1 public class ArrayList<E>
2 extends AbstractList<E>
3 implements List<E>, RandomAccess, Cloneable, Serializable
```

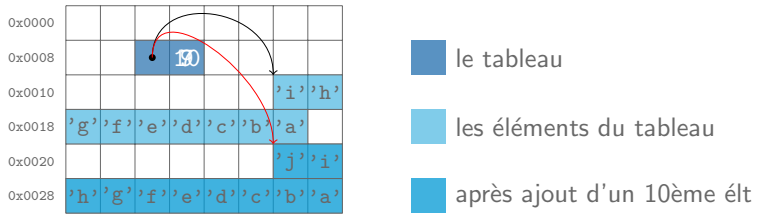
```
Resizable-array implementation of the List interface ...

The size, isEmpty, get, set, iterator, and listIterator operations run
in constant time. The add operation runs in amortized constant time,
that is, adding n elements requires O(n) time. All of the other
operations run in linear time (roughly speaking). The constant factor
is low compared to that for the LinkedList implementation.
```

```
1 public class LinkedList<E>
2 extends AbstractSequentialList<E>
3 implements List<E>, Deque<E>, Cloneable, Serializable
```

```
Doubly-linked list implementation of the List and Deque interfaces...
All of the operations perform as could be expected for a doubly-linked
list ...
```

La liste implantée avec un tableau

- ajouter un élement en tête :
  - si il reste de la place, nécessite de décaler tous les éléments du tableau
    en $\Theta(n)$ (en $\Theta(1)$ si on est malin)
  - et si le tableau est plein, alors nécessite de créer un nouveau tableau et de recopier les éléments
    en $\Theta(n)$
  - en conclusion, en $\Omega(1)$ et $\mathcal{O}(n)$



La liste implantée avec un tableau

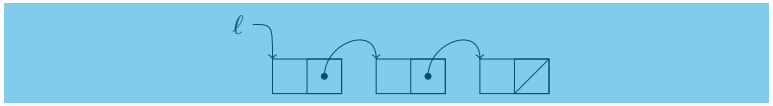
- inconvénients :
  - pas très efficace sur les opérations courantes
  - le dépassement de capacité peut être résolu en copiant le contenu dans un tableau plus grand, en $\Theta(n)$
  - la concaténation est en $\Theta(n+m)$
  - necessité de disposer d'un espace supplémentaire en $\Theta(n)$ pour ces deux dernières opérations
- avantages :
  - l'accès au k-ième élément est en $\Theta(1)$
  - peut permettre l'implantation de fonctions de recherche ou de tri efficaces

La liste chaînée, implantée avec des cellules

Definition (API2)

Une liste d'éléments d'un ensemble E est :

- soit la liste vide ;
- soit un couple (x, ℓ') constitué d'un élément $x \in E$ et d'une liste ℓ' d'éléments de E.



Implantations en Python des listes chaînées

deux implantations déjà rencontrées :

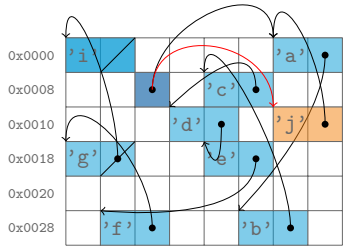
- non mutable, avec des couples, en utilisant `None` pour représenter la liste vide

```
1 l = (1, (2, (3, None))) # la liste [1,2,3]
```

- mutable, avec des dictionnaires, toujours avec `None` pour la liste vide

```
1 l = { "value": 1, "next" :
2         { "value" : 2, "next" :
3             { "value" : 3, "next" : None }}}
```

Représentation en mémoire d'une liste chaînée



- la liste
- les éléments de la liste
- ajout en queue d'un 8ème élt
- ajout en tête d'un 9ème élt

La liste chaînée mutable

- ajouter un élément en tête : créer un nouvelle cellule, puis chaîner = mettre à jour le suivant
  en $\Theta(1)$
- suppression d'un élément : recherche de la cellule `current_cell` à supprimer en se souvenant de la cellule précédente `previous_cell`
  + suppression par 'déchaînage'
  `previous_cell["next"] = current_cell["next"]`
  en $\mathcal{O}(n) + \Theta(1) = \mathcal{O}(n)$

Ajout en Python

```
1 def add(l,v):
2   c = { "value" : v, "next" : None}
3   c["next"] = l
4   return c
```

```
1 def add(l,v):
2   return { "value" : v; "next" : l}
```

```
1 >>> l = None
2 >>> l = add(l,3)
3 >>> l = add(l,2)
4 >>> l = add(l,1)
5 >>> print l
6 {'value': 1, 'next': {'value': 2, 'next': {'value': 3, 'next': None}}}
```


Implantation alternative de la liste chaînée

Si on a besoin d'accéder souvent à la longueur, on encapsule la liste récursive dans une structure de données.

```
1 # la liste vide
2 l = { "length" : 0, "head" : None }
3 # la liste [1,2,3]
4 l = { "length" : 3,
5   "head" : {'value': 1, 'next': {
6                 'value': 2, 'next': {
7                 'value': 3, 'next': None}}} }
```

L'ajout n'est plus une fonction mais une procédure.

```
1 def add(l,v):
2    l["length"] += 1
3    l["head"] = { "value" : v, "next" : l}
```

```
1 >>> l = empty_list()
2 >>> add(l,3)
3 >>> add(l,2)
4 >>> add(l,1)
5 >>> print l
6 {'length': 3, 'head': {'value': 1, 'next': {...}}}
```

des cellules + des tableaux

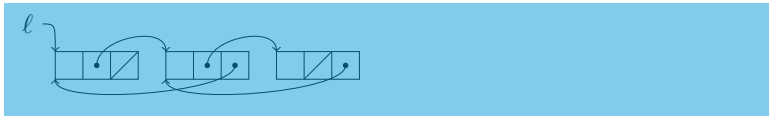
=

des cellules de tableaux


```
1  def delete (l,v):
2      r = l # result
3      c = r # current cell
4      while c != None and c["value"] != v:
5          c = tail(c)
6      # c is None or c contains v
7      if c != None:
8          if c["prev"] != None:
9              p = c["prev"]
10             p["next"] = c["next"]
11         else: # c is the head
12             r = c["next"]
13         if c["next"] != None:
14             n = c["next"]
15             n["prev"] = c["prev"]
16     return r
```

```
1  >>> print(l)
2  {'value': 1, 'prev': None, 'next': {'value': 2, 'prev': {...}, 'next': {'value'
3  >>> l = delete(l,2)
4  >>> print(l)
5  {'value': 1, 'prev': None, 'next': {'value': 3, 'prev': {...}, 'next': None}}
6  >>> l = delete(l,1)
7  >>> print(l)
8  {'value': 3, 'prev': None, 'next': None}
9  >>> l = delete(l,3)
10 >>> print(l)
11 None
```

La liste doublement chaînée




```
1  c1 = { "value" : 1, "next" : None, "prev" : "None" }
2  c2 = { "value" : 2, "next" : None, "prev" : c1 }
3  c1["next"] = c2
4  c3 = { "value" : 3, "next" : None, "prev" : c2 }
5  c2["next"] = c3
```

- ajouter en tête : créer un nouvelle cellule, puis chaîner, $\Theta(1)$
- recherche : parcours de liste, $\mathcal{O}(n)$
- suppression : recherche de la cellule à supprimer (inutile se souvenir du précédent) + suppression par 'déchaînage'
  `c["next"]["prev"] = c["prev"], c["prev"]["next"] = c["next"]`
  en $\mathcal{O}(n) + \Theta(1) = \mathcal{O}(n)$

Avantage : permet de se déplacer dans la liste,
Inconvénient : espace mémoire occupé par le second pointeur

Pourquoi est-il nécessaire de renvoyer la liste ?

```
1  >>> l = None
2  >>> l = add(l,3)
3  >>> l = add(l,2)
4  >>> l = add(l,1)
5  >>> print(head(l))
6  1
7  >>> print(head(tail(l)))
8  2
9  >>> ll = delete(l,1)
10 >>> print(ll)
11 {'value': 2, 'prev': None, 'next': {'value': 3, 'prev': {...}, 'next': None}}
12 >>> print(l)
13 {'value': 1, 'prev': None, 'next': {'value': 2, 'prev': None, 'next': {'value':
```

Si on veut que supprimer soit une procédure, il faut encapsuler la liste.

Implantation alternative

Comme on a une liste doublement chaînée, on peut souhaiter accéder facilement au dernier élément de la liste :

```
1 def empty_list ():
2     return { "begin" : None, "end" : None }
```

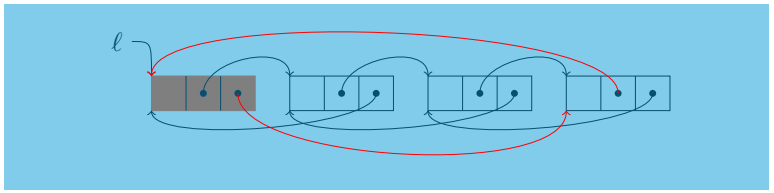
```
1 def add (l,v):
2     c = { "value" : v, "next" : l["begin"], "prev" : None }
3     if l["begin"] != None:
4         l["begin"]["prev"] = c
5     else:
6         l["end"] = c
7     l["begin"] = c
```

```
1 >>> l = empty_list()
2 >>> add(l,3)
3 >>> add(l,2)
4 >>> add(l,1)
5 >>> print(l["begin"])
6 {'value': 1, 'prev': None, 'next': {'value': 2, 'prev': {...}, 'next': {'value'
7 >>> print(l["end"])
8 {'value': 3, 'prev': {'value': 2, 'prev': {'value': 1, 'prev': None, 'next': {.
```


Liste avec sentinelle

ajouter une cellule sentinelle en tête de liste telle que :

- son précédent est la cellule de fin de liste,
- son suivant est la cellule de début de liste,
- le suivant de la dernière cellule est la sentinelle,
- et le précédent de la première cellule est la sentienelle.



La suppression s'écrit alors dans tous les cas :
`c["prev"]["next"] := c["next"]` puis `c["next"]["prev"] := c["prev"]`
il n'y a plus de tests à effectuer.