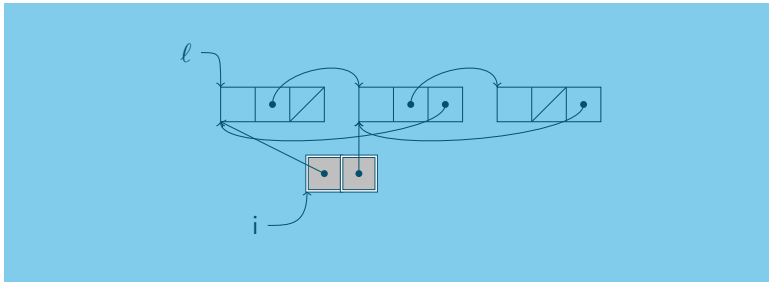
Abstraction pour les parcours de listes

- un itérateur est une structure de donnée permettant le parcours
- opérations supportées :
  - avancer, reculer
  - est_en_fin, est_en_debut
  - valeur
  - inserer_apres, inserer_avant, supprimer



Exemple en Java

```
1  public interface ListIterator<E>
2  extends Iterator<E>
```

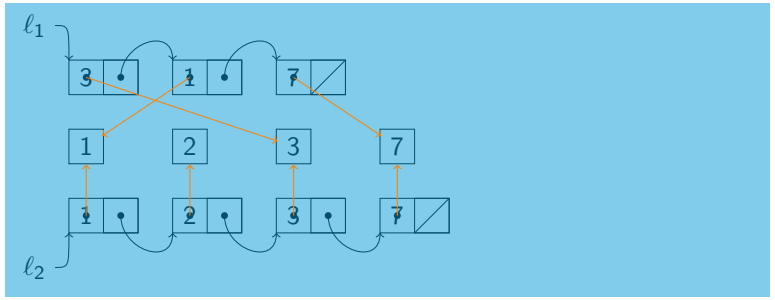
```
An iterator for lists that allows the programmer to traverse the list
in either direction, modify the list during iteration, and obtain the
iterator's current position in the list. A ListIterator has no current
element; its cursor position always lies between the element that
would be returned by a call to previous() and the element that would
be returned by a call to next(). An iterator for a list of length n
has n+1 possible cursor positions, as illustrated by the carets (^)
below:

                      Element(0)   Element(1)   Element(2)   ... Element(n-1)
 cursor positions:  ^            ^            ^            ^                  ^

Note that the remove() and set(Object) methods are not defined in
terms of the cursor position; they are defined to operate on the last
element returned by a call to next() or previous().
```


Parcours de liste avec iterateur

```
1  # la bibliotheque developpee en TP
2  import listiterator as list
3
4  def print_with_iterator (l):
5      """
6      Print elements of a list using an iterator.
7
8      :param l: The list to be printed
9      :type l: listiterator
10     """
11     it = list.get_listiterator(l)
12     while list.hasNext(it):
13         print(list.next(it),end=' ')
14     print()
```


Note sur le stockage des éléments

Si on a plusieurs listes, organisées différemment mais avec les mêmes éléments, il est inutile de dupliquer les éléments. Chaque cellule de la liste ne contient plus l'élément mais une référence vers l'élément.



Attention à la suppression : on ne supprime pas l'élément

Résumé des complexités des opérations sur les listes

	Tableau	Listes SC	Listes DC	avec sentinelle
inserer en tête	$\mathcal{O}(n)$	$\Theta(1)$	$\Theta(1)$	
chercher	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	
supprimer [1]	$\mathcal{O}(n)$	$\Theta(1)$	$\Theta(1)$	
accès au premier	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	
accès au dernier	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
accès au suivant	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	
accès au précédent	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$	
inserer après/avant[1]	$\mathcal{O}(n)$	$\Theta(1)$	$\Theta(1)$	

1. une fois l'élément trouvé

Implantation d'une pile

```
1 def empty_stack ():
2     return {
3       "index" : 0, # position du sommet de la pile
4       "stack" : XXX; # un tableau
5     }
```

Inconvénient majeur : nécessite une mise en œuvre particulière lorsque la taille du tableau représentant la pile est atteinte

- il faut recopier le tableau dans un nouveau, plus grand, en $\Theta(n)$
- il faut changer la longueur du tableau si celui-ci est dynamique, en $\mathcal{O}(n)$

Implantation d'une pile

```
1 def empty_stack ():
2     return {
3       "size" : 0,
4       "stack" : empty_list();
5     }
```

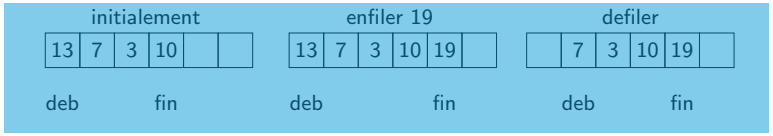
- pas de limitation en taille
- l'élément au sommet se trouve en tête de liste
- peut-être intéressant de stocker la taille : le calcul de la longueur de liste étant coûteux

Comme toutes les opérations concernent la tête de la liste, une liste simplement chaînée suffit.

Complexité des primitives

	Tableau	Liste SC
avec taille bornée		
empiler	$\Theta(1)$	$\Theta(1)$
depiler	$\Theta(1)$	$\Theta(1)$
sommet	$\Theta(1)$	$\Theta(1)$
avec taille non bornée		
empiler	$\mathcal{O}(n)$	$\Theta(1)$
depiler	$\Theta(1)$	$\Theta(1)$
sommet	$\Theta(1)$	$\Theta(1)$

Implantation d'une file




```
1 def empty_queue ():
2     return {
3       "beg"   : 0, # position du dernier element
4       "end"   : 0, # position du premier element
5       "queue" : XXX; # un tableau
6     }
```

- nécessite la gestion de la plage de cases occupée dans le tableau
- comme pour la pile, nécessite une mise en œuvre particulière lorsque le nombre d'éléments atteint la taille du tableau

Implantation d'une file

```
1 def empty_queue ():
2     return {
3       "size" : 0;
4       "queue" : empty_list();
5     }
```

- pas de limitation en taille
- les insertions se font en tête et les suppressions en queue $\mapsto$ besoin d'accès au dernier élément
- une liste avec sentinelle ou avec accès à la queue pour être efficace

Complexité des primitives

	Tableau	Liste SC	Liste DC	Liste SC avec accès à la queue
enfiler	$\mathcal{O}(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
défiler	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$

En conclusion

- le choix de la structure de données dépend de ce à quoi elle va être utilisée
- son implantation dépend de la manière dont on va l'utiliser
- il est donc primordial de bien analyser les besoins avant de faire un choix
- lorsqu'on utilise une bibliothèque, il faut connaître la façon dont sont réalisées les structures de données
  n'utiliser que des APIs bien documentées
- lorsqu'on développe une bibliothèque, il faut préciser à l'utilisateur la complexité des opérations
  ne créer que des APIs bien documentées