DS de mi-semestre

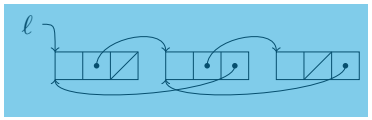
vendredi 14/03/2014 de 14h à 16h

M1 amphi GALOIS et PAINLEVE

documents de cours/TD/TP autorisés

La liste doublement chaînée

```
1  type 'a liste =
2    | Vide
3    | Cons of 'a cellule
4  and 'a cellule = {
5    valeur : 'a;
6    mutable suivant : 'a liste;
7    mutable precedent : 'a liste;
8  }
```



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

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


```
1  (* supprimer : 'a liste -> 'a -> 'a liste *)
2  let supprimer l x =
3    let r = ref l (* la liste resultat *)
4    and p = ref l in
5    while !p <> Vide && tete !p <> x do
6      p := reste !p
7    done;
8    let c = la_cellule !p
9    in
10   if c.precedent <> Vide then
11     let cp = (la_cellule c.precedent) in
12     cp.suivant <- c.suivant
13   else (* on est sur la tete de liste *)
14     r := c.suivant;
15   if c.suivant <> Vide then begin
16     let cs = (la_cellule c.suivant) in
17     cs.precedent <- c.precedent;
18   end;
19   !r
```

```
# let l1 = ajouter_en_tete Vide 10;;
# let l2 = ajouter_en_tete l1 20;;
# let l = ajouter_en_tete l2 30;;
# let l' = supprimer l 20;;
# tete l';;
# - : int = 30
# tete (reste l');;
# - : int = 10
```

Pourquoi est-il nécessaire de renvoyer la liste ?

```
# tete l;;
# - : int = 30
# tete (reste l);;
# - : int = 10
# let l'' = supprimer l 30;;
# val l'' : int liste =
    Cons {valeur = 10; suivant = Vide; precedent = Vide}
# l;;
# - : int liste =
Cons
 {valeur = 30;
  suivant = Cons {valeur = 10; suivant = Vide; precedent = Vide};
  precedent = Vide}
```

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

```
1  type 'a liste_encapsulee = { mutable la_liste : 'a liste }
2
3  val supprimer 'a liste_encapsulee -> 'a -> unit
```

Implantation alternative

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

```
1  type 'a liste = {
2    mutable longueur : int;
3    mutable tete    : 'a liste_interne;
4    mutable queue   : 'a liste_interne }
5  and 'a liste_interne =
6    | Vide
7    | Cons of 'a cellule
8  and 'a cellule = {
9    valeur : 'a;
10   mutable suivant : 'a liste_interne;
11   mutable precedent : 'a liste_interne
12 };;
```



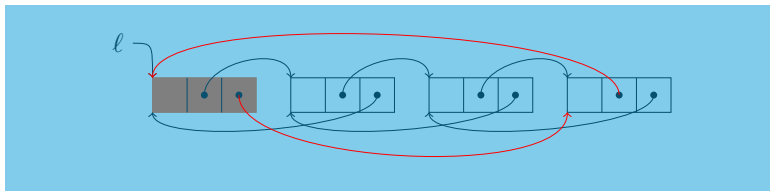
```
1  let liste_vide = { longueur = 0; tete = Vide; queue = Vide }
2
3  (* ajouter_en_tete : 'a liste -> 'a -> unit *)
4  let ajouter_en_tete ll v =
5    let r = ll.tete
6    and l = Cons { valeur = v; suivant = Vide; precedent = Vide} in
7    let c = la_cellule l in
8    c.suivant <- r;
9    if r <> Vide then begin
10     let cr = la_cellule r in
11     cr.precedent <- l;
12   end else begin
13     ll.queue <- l
14   end;
15   ll.tete <- l
```

```
# let l = liste_vide;;
# ajouter_en_tete l 10;;
# val l : int liste =
#  {longueur = 0;
#   tete = Cons {valeur = 10; suivant = Vide; precedent = Vide};
#   queue = Cons {valeur = 10; suivant = Vide; precedent = Vide}}
# - : unit = ()
# ajouter_en_tete l 20;;
# tete l;;
# - : int = 20
# queue l;;
# - : int = 10
```

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.



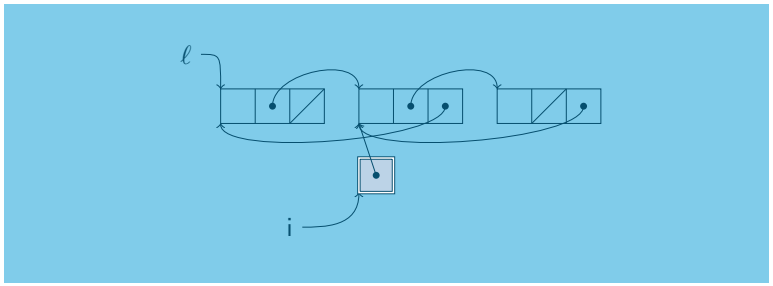
La suppression s'écrit alors dans tous les cas :

```
c.precedent.suivant := c.suivant; c.suivant.precedent := c.precedent;
```

il n'y a plus de tests à effectuer.

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



Implantation sur des listes doublement chaînées

```
1  exception ListeVide
2  exception IterateurEnDebut
3  exception IterateurEnFin
4
5  type 'a liste =
6    | Vide
7    | Cons of 'a cellule
8  and 'a cellule = {
9    valeur : 'a;
10   mutable suivant : 'a liste;
11   mutable precedent : 'a liste
12 }
13 and 'a iterateur = {
14   mutable c : 'a liste;
15 }
```

On s'arrangera pour qu'un itérateur soit toujours défini, c'est-à-dire que `it.c` ne pointe pas vers `Vide`.

Opérations de base

Avancer : passe à l'élément suivant dans la liste si, et seulement si, il existe.

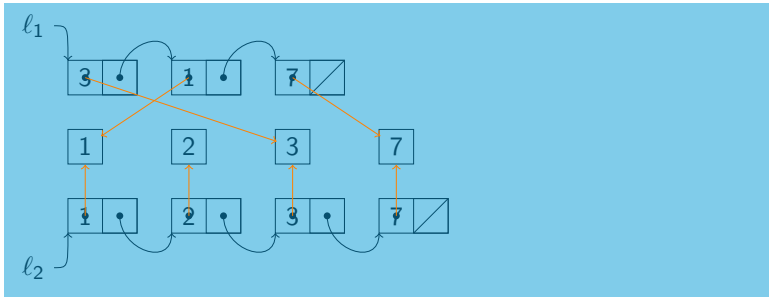
```
1  /* CU: it ne pointe pas sur Vide */
2  let avancer it =
3    if est_en_fin it then
4      raise IterateurEnFin
5    else
6      let cell = (la_cellule it.c).suivant
7      in
8      it.c <- cell
```


Parcours de liste avec iterateur

```
1  let est_present l e =
2    let it = iterateur_en_debut l
3    and trouve = ref false
4    in
5    try
6      while not !trouve do
7        if valeur it = e then trouve := true;
8        avancer it
9      done;
10     !trouve;
11   with IterateurEnFin ->
12     !trouve
```


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 type pile = {
2   mutable index : int; (* position du sommet de la pile *)
3   lapile : 'a array;
4 }
```

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 type pile = {
2   mutable taille : int;
3   mutable lapile : 'a liste;
4 }
```

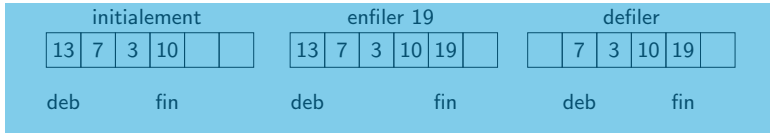
- pas de limitation en taille
- l'élément au sommet se trouve en tête de liste
- peut-être intéressant de stocker la taille : le calcul de la longueur de liste étant coûteux

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  type 'a file = {
2    mutable deb   : int; (* position du dernier element *)
3    mutable fin   : int; (* position du premier element *)
4    lafile : 'a array;
5  }
```

- 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  type 'a file = {
2    mutable taille : int;
3    mutable lafile : 'a liste;
4  }
```

- 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