Cours 5 : Implantation des structures linéaires

Jean-Stéphane Varré

Université Lille 1

jean-stephane.varre@lifl.fr

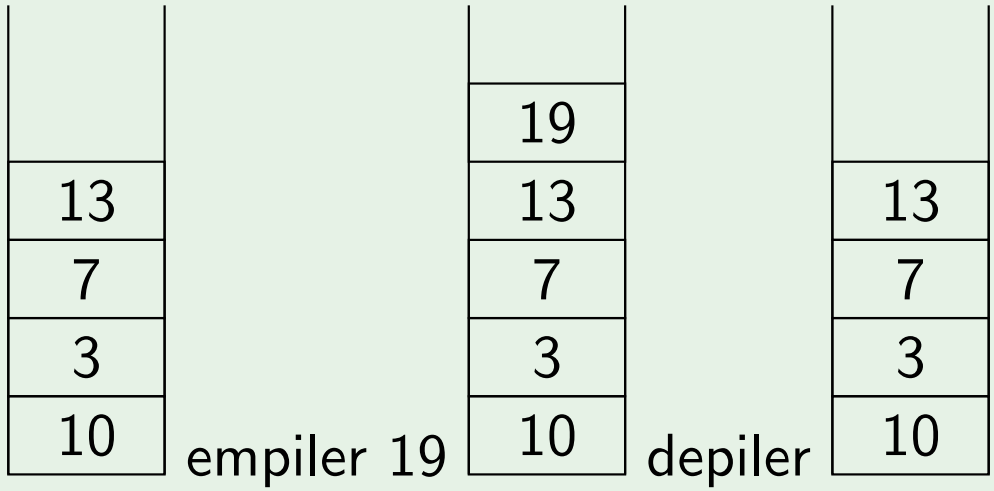
La structure de données LISTE

- suite ordonnée d'éléments (i.e. il existe un suivant et un précédent)
- de taille non bornée
- qui supporte les opérations :
  - rechercher
  - supprimer
  - insérer : en tête, après/avant un élément, en fin
  - $+$ les prédicats usuels

(5, 7, 9, 2, 11)
("oiseau", "chat", "chien", "pingouin")

La structure de données PILE

- empilement d'éléments (mémorisation)
- de taille non nécessairement 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 :
  - empiler : ajoute un élément
  - dépiler : enlève le dernier élément entré
  - sommet : accès au dernier élément entré
  - $+$ les prédicats usuels



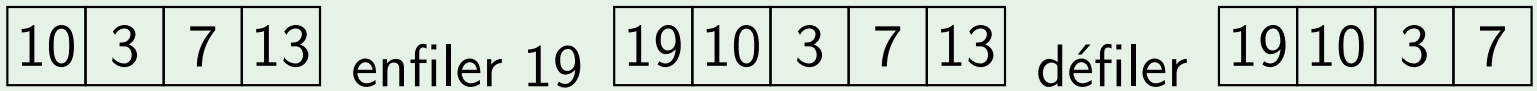
Cas d'utilisation

- calculatrice
- simulation de la pile d'appels récursifs (permet la mise en attente de résultats pour un traitement futur)
- parcours d'arbre en profondeur

Est utile dans tous les cas où il faut mémoriser une information avant taritement.

La structure de données FILE

- suite ordonnée d'éléments (i.e. il existe un suivant et un précédent)
- de taille non bornée
- premier élément entré, premier sorti (FIFO : First In First Out)
- qui supporte les opérations :
  - enfiler : ajoute un élément
  - défiler : enlève le premier élément entré
  - $+$ les prédicats usuels



Cas d'utilisation

- file d'impression
- attente téléphonique
- parcours d'arbre en largeur

Est utile dans tous les cas où des informations doivent être traitées par priorité.

La liste implantée avec un tableau

```
type LISTE = record
  longueur : CARDINAL; // position du dernier element de la liste
  laliste : array of ELEMENT;
end {record};
```

- insérer un élement en tête : nécessite de décaler tous les éléments du tableau , en $\Theta(n)$ (en $\Theta(1)$ si on est malin)
- chercher un élément : parcours du tableau, en $\Theta(n)$
- supprimer un élément : chercher l'élément, en $\Theta(n)$ + décaler tous les éléments du reste du tableau, en $\Theta(n)$

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 également en $\Theta(n)$

Avantages :

- l'accès au k-ième élément est en $\Theta(1)$
- peut permettre l'implantation de fonctions de recherche ou de tri efficaces

Rappel sur les pointeurs

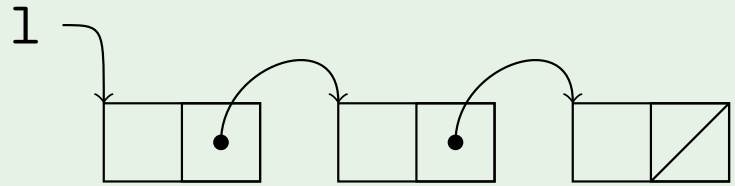
```
type ELEMENT = ...;
```

- déclaration d'un pointeur sur un type : `type PELEMENT = ^ELEMENT;`
- allocation de la mémoire : `var p : PELEMENT = new ELEMENT;`
- obtention de l'adresse d'une variable : `var e : ELEMENT; p := @e;`
- obtention de la valeur : `valeur = p^;`
- dans le cas d'un enregistrement : `valeur = p^.suivant;`
- desallocation de la mémoire : `dispose(p)`

La liste chaînée

Se base sur la définition récursive d'une liste comme une tête + une queue

```
type LISTE : ^CELLULE;
type CELLULE = record
  valeur : ELEMENT;
  suivant : LISTE;
end {record};
```



- insérer un élément en tête : créer un nouvelle cellule, puis chaîner `l := new CELLULE(e,l)`, en $\Theta(1)$
- recherche un élément : parcours de liste, en $\Theta(n)$
- suppression d'un élément : recherche de la cellule `ccour` à supprimer en se souvenant de la cellule précédente `cprec`, en $\Theta(n)$ + suppression par 'déchaînage' `cprec.suivant := ccour.suivant; dispose(ccour)` , en $\Theta(1)$

Implantation alternative

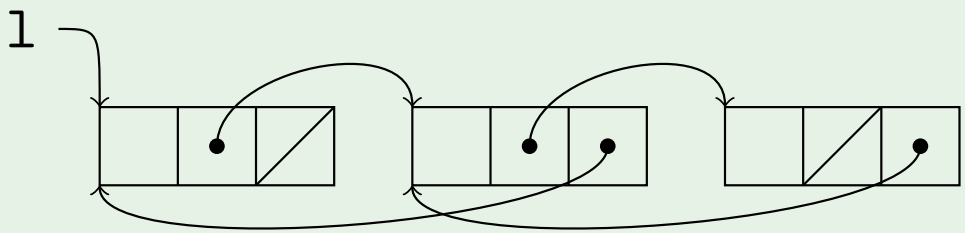
Si on a besoin d'accéder souvent à la longueur

```
type PCELLULE = ^CELLULE;
type CELLULE = record
  valeur  : ELEMENT;
  suivant : PCELLULE;
end {record};
```

```
type LISTE = record
  longueur : CARDINAL;
  tete     : PCELLULE;
end {record};
```

La liste doublement chaînée

```
type LISTE : ^CELLULE;
type CELLULE = record
  valeur    : ELEMENT;
  suivant   : LISTE;
  precedent : LISTE;
end {record};
```



- insérer un élément en tête : créer un nouvelle cellule, puis chaîner en $\Theta(1)$
- recherche un élément : parcours de liste, en $\Theta(n)$
- suppression d'un élément : recherche de la cellule `c` à supprimer (plus besoin de se souvenir du précédent), en $\Theta(n)$ + suppression par 'déchaînage' , en $\Theta(1)$

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

Suppression dans une liste doublement chaînée

```
if c.precedent <> NIL then
  c.precedent.suivant := c.suivant;
else // on est sur la tete de liste
  l := c.suivant;
if c.suivant <> NIL then
  c.suivant.precedent := c.precedent;
dispose(c);
```

Implantation alternative

Si on souhaite accéder facilement au dernier élément de la liste :

```
type PCELLULE = ^CELLULE;
type CELLULE = record
  valeur  : ELEMENT;
  suivant : PCELLULE;
end {record};
```

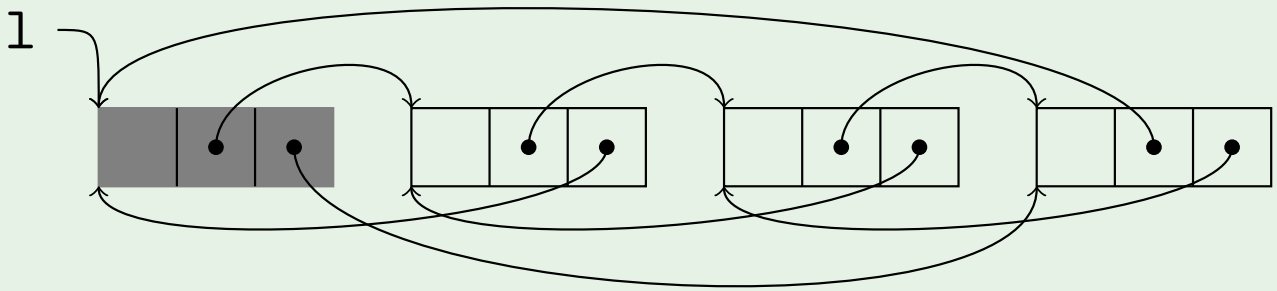
```
type LISTE = record
  longueur : CARDINAL;
  tete     : PCELLULE;
  queue    : PCELLULE;
end {record};
```

Liste avec sentinelle

Une astuce pour écrire (encore) plus simplement la suppression

Idée : ajouter une cellule vide (sans valeur) en début de liste telle que :

- le précédent est la cellule de fin de liste,
- le suivant 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 :

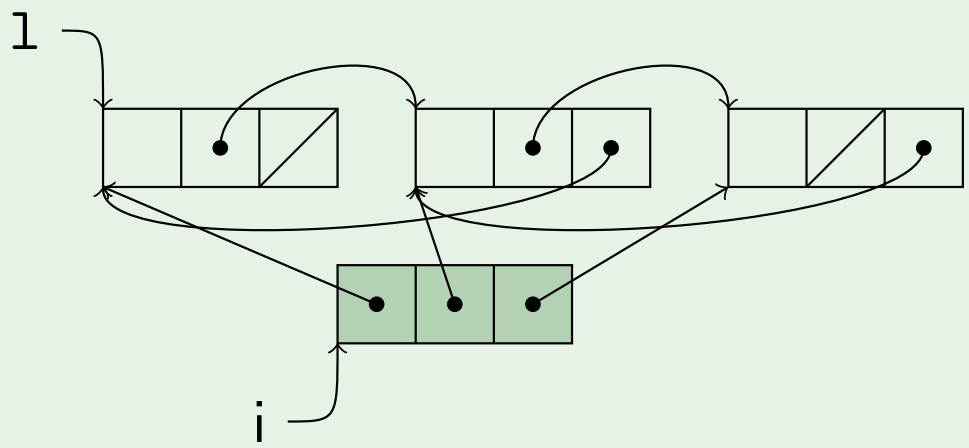
```
c.precedent.suivant := c.suivant; c.suivant.precedent := c.precedent;
```

il n'y a plus de tests à effectuer.

Abstraction pour les parcours de listes

Au lieu de parcourir la liste avec un pointeur de cellule, on va créer une structure de donnée permettant ce partours : un itérateur.
Un itérateur supporte, la plupart du temps, les opérations suivantes :

- avancer, reculer
- estEnFin, estEnDebut
- valeur
- insertion d'élément : insererApres, insererAvant
- supprimer un élément



(Rappel) sur les exceptions

- déclaration d'une exception

```
type MonException = class(SysUtils.Exception);
```

- levée d'une exception

```
raise MonException.create('Le message que je veux donner');
```

- capture d'une exception

```
try
   // les instructions a realiser pour le cas general
   ...
except
   // les instructions a realiser pour les cas particuliers
   ...
end;
```

Implantation sur des listes simplement chaînées

Déclaration des listes

```
type
  LISTE = ^CELLULE;
  CELLULE = record
    valeur : ELEMENT;
    suivant : LISTE;
  end {record};
```

Déclaration des itérateurs

```
type
  ITERATEUR = record
    precedent : ^CELLULE;
    courant   : ^CELLULE;
    suivant   : ^CELLULE;
  end {record};

  IterateurEnFin = class(SysUtils.Exception);
  IterateurEnDebut = class(SysUtils.Exception);
```

Opérations de base

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

```
procedure avancer (var i : ITERATEUR);
var
  ancien_suivant : ^CELULLE;
begin
  ancien_suivant := i.suivant;
  try
    i.suivant  := i.suivant.suivant;
    // leve une exception si i.suivant = NIL
    i.precedent := i.courant;
    i.courant  := ancien_suivant;
  except
    raise(IterateurEnFin.create('Impossible d avancer');
  end;
end {avancer};
```

Opérations de base

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

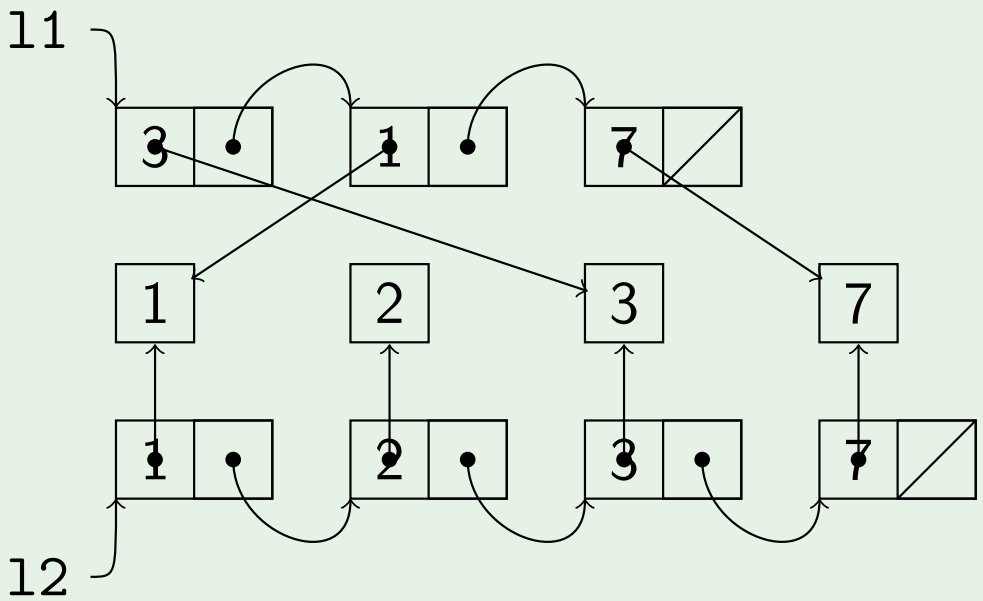
```
procedure reculer (var i : ITERATEUR);
var
  ancien_precedent : ^CELULLE;
begin
  ancien_precedent := i.precedent;
  try
    i.precedent := i.precedent.precedent;
    // leve une exception si i.precedent = NIL
    i.suivant  := i.courant;
    i.courant  := ancien_precedent;
  except
    raise(IterateurEnFin.create('Impossible de reculer');
  end;
end {avancer};
```

Parcours de liste avec iterateur

```
function estPresent (l : LISTE; e : ELEMENT) : BOOLEAN;
var
  it : ITERATEUR = auDebut(l);
  trouve : BOOLEAN = False;
begin
  try
    while not trouve do begin
      if valeur(it) == e then
        trouve := True;
      it := avancer(it);
    end {while};
  except
    // si on a trop avance
    // rien a faire puisque trouve vaut deja False
  end;
  estPresent := trouve;
end {estPresent};
```

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 un pointeur 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

Avec un tableau

```
type PILE = record
  index : CARDINAL; // position du sommet de la pile
  lapile : array of ELEMENT;
end {record};
```

Inconvénient majeur : nécessite une mise en oeuvre 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

Avec une liste

```
type PILE = record
  taille : CARDINAL;
  lapile : LISTE;
end {record};
```

- 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

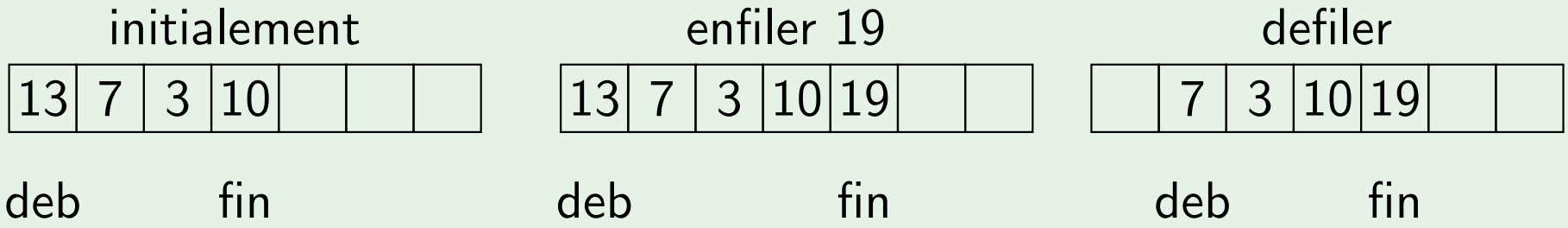
Comme toutes les opérations concernent la tête de la liste, une liste simplement chaînée suffit.

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

Avec un tableau




```
type FILE = record
  dernier : CARDINAL; // position du dernier element entre
  premier : CARDINAL; // position du premier element entre
  lafile  : array of ELEMENT;
end {record};
```

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

Implantation d'une file

Avec une liste

```
type FILE = record
  taille : CARDINAL;
  lafile : LISTE;
end {record};
```

- 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
- lorsqu'on développe un bibliothèque, il faut préciser à l'utilisateur la complexité des opérations