

INF6

Algorithme Avancé

Manuele Kirsch Pinheiro

Contenu prévisionnel

- Piles et files
- Listes
- Récursivité
 - Récursivité dans le calcul
 - Récursivité structurelle
- Arbres binaires
 - Parcours en profondeur et en largeur
- Généralisation de la notion d'arbre
 - Insertion et suppression de nœuds
- Arbre de recherche
 - Recherche
 - Rééquilibrage

LISTES

10/01/16

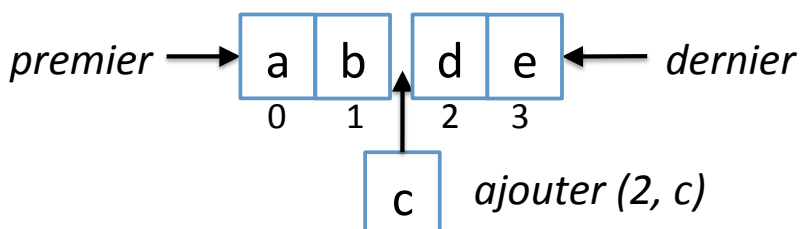
Manuele Kirsch Pinheiro - UP1 / CRI / EMS

3

Listes

- **Liste**

- Structure linéaire composée d'une **séquence** finie d'éléments, **repérés** selon leur **rang**



Le **rang** indique la **position** d'un élément dans la **séquence** d'éléments qui compose la liste (**n° de séquence**)

- Contrairement aux piles et files, les listes autorisent **l'ajout** et la **suppression n'importe où** **dans la séquence**

10/01/16

Manuele Kirsch Pinheiro - UP1 / CRI / EMS

4

• Opérations abstraites

Opérations abstraites définies pour la files

- **longueur** : renvoie le nombre d'éléments dans la liste
- **ième** : retourne l'élément de rang « r »
- **ajouter** : ajoute un élément à un rang « r » donné
- **supprimer** : supprime l'élément à un rang donné

[Element]

<<Interface>> Liste	
+longueur()	: int
+ième(r : int)	: Element
+ajouter(r : int, e : Element)	: booléen
+supprimer(r : int)	: void

10/01/16

Manuele Kirsch Pinheiro - l

• Opérations

- la **longueur** d'une **liste vide** est **0**

$$\text{longueur}(\text{listeVide}) = 0$$

- **ajouter** **incrmente** de **1** la **longueur** de la liste, alors que **supprimer** la **réduit** de **1**

soit **l** une **liste**,
r un **rang** et
e un **élément**

$$\forall r \ 0 \leq r \leq \text{longueur}(l) \text{ alors}$$

$$\text{longueur}(\text{ajouter}(l, r, e)) = \text{longueur}(l) + 1$$

$$\forall r \ 0 \leq r < \text{longueur}(l) \text{ alors}$$

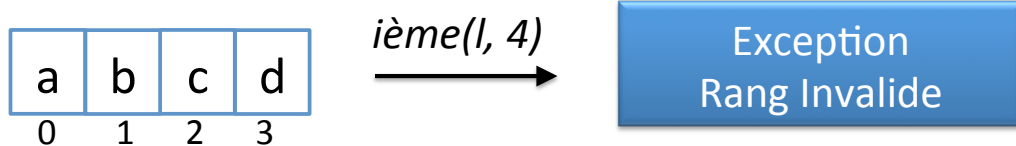
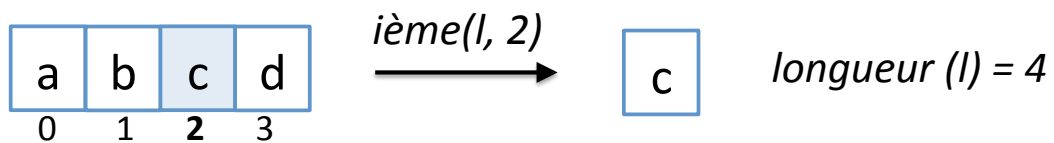
$$\text{longueur}(\text{supprimer}(l, r)) = \text{longueur}(l) - 1$$

Exception
Rang Invalide

• Opérations

- L'opération **ième** n'est définie que si le **rang** est **valide**

$$\forall r \ (r < 0) \vee (r \geq \text{longueur}(l)) \rightarrow \nexists e \ e = \text{ième}(r)$$

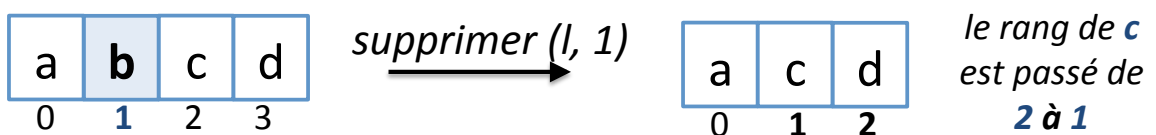
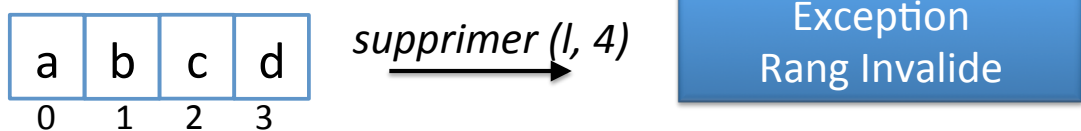


Exception
Rang Invalide

• Opérations

- Le même vaut pour **supprimer** et **ajouter**
- **supprimer** retire un élément qui **appartient à la liste**
 - rang valide $\Rightarrow 0 \leq r < \text{longueur}(l)$

$\text{longueur}(l) = 4$



Les **rangs** des **éléments à droite** de l'élément **supprimé** sont **décrémentés de 1**

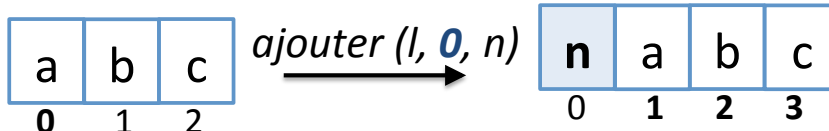
• Opérations

- **ajouter** insère un élément à un **rang** compris **entre 0 et la longueur** (dernier)

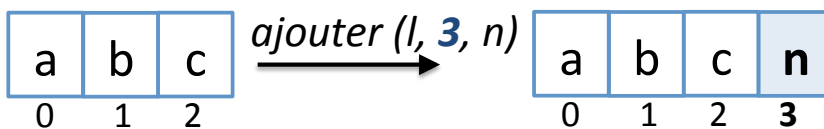
- rang valide $\Rightarrow 0 \leq r \leq \text{longueur}(l)$

*ajouter(l, 4, n) →
Exception Rang Invalide*

longueur(l) = 3



*le rang de c est passé de
2 à 3*



ajout à la fin de la liste

Les **rangs** des éléments à **droite** du **nouvel élément** sont **incrémentés de 1**

• Opérations

- Autres opérations sont possibles

Opérations de base :

- longueur (*size*)
- ajouter (*add*)
- supprimer (*remove*)
- ième (*get*)

Autres opérations utiles :

- estVide (*isEmpty*)
- estPleine (*isFull*)
- éléments (*elements*)
- dernier (*last*)
- premier (*first*)
- ajouterALaFin (*add*)
- recherche (*search*)

• Exceptions

– Rang Invalide

- Lorsque le **rang** utilisé n'est pas **acceptable** par l'opération

– Liste Pleine

- A l'opération **ajouter**, lorsque la **capacité** de la liste est fixe et **limitée**, et qu'elle a été **atteinte**

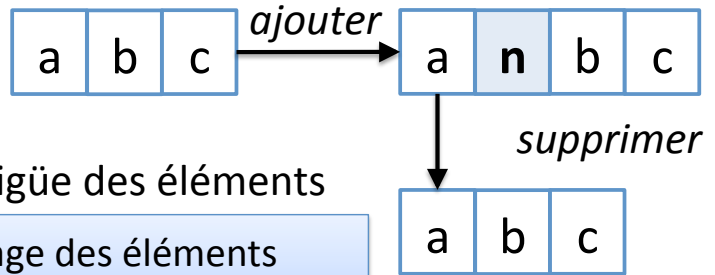
Listes

• Implémentations :

– Par tableau

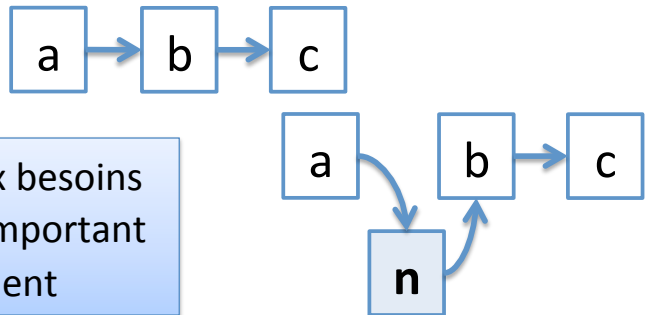
- Représentation contigüe des éléments

⊕ accès direct ⊖ décalage des éléments
⊙ gestion de la taille et taux d'occupation (gaspillage)



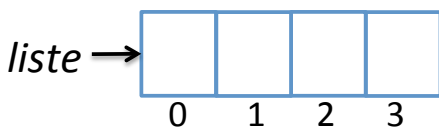
– Par structure chaînée

⊕ dynamique ⊕ taille s'adapte aux besoins
⊕ pas décalage ⊖ temps d'accès + important
⊙ consommation mémoire par élément

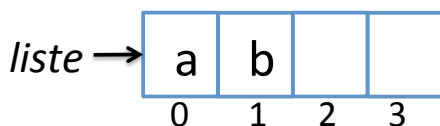


Listes

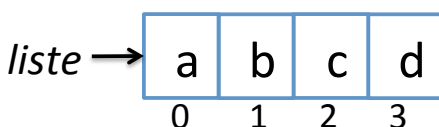
• Algorithmes : Liste sur tableau



longueur = 0 *liste vide*



longueur = 2



longueur = 4

longueur → nombre d'éléments

liste pleine
longueur == taille

ListePleineException
+ListePleineException()

RangInvalideException
+RangInvalideException()

extends
IndexOutOfBoundsException

```
<<Interface>>
Liste
+size() : int
+get(r : int) : T
+add(r : int, e : T) : boolean
+add(e : T) : boolean
+remove(r : int) : boolean
+isEmpty() : boolean
+isFull() : boolean
+first() : T
+last() : T
+elements() : T []
+search(e : T) : int
```

```
ListeTableau
-liste : T[]
-longueur : int = 0
-DEFAULT_SIZE : int = 10
+ListeTableau(taille : int)
+ListeTableau()
#resize(taille : int) : boolean
+size() : int
+get(r : int) : T
+add(r : int, e : T) : boolean
+add(e : T) : boolean
+remove(r : int) : boolean
+isEmpty() : boolean
+isFull() : boolean
+first() : T
+last() : T
+elements() : T []
+search(e : T) : int
```

Listes

• Algorithme **ième** : Liste sur tableau

ième (rang) : Élément

Entrée :

Entier rang

Sortie :

Élément e

Si $0 \leq \text{rang} < \text{longueur}$

alors

e = liste [rang]

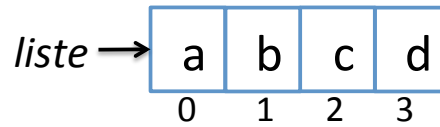
sinon

Lancer Exception **Rang Invalide**

fin si

retourner s

Fin ième



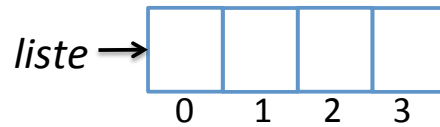
longueur = 4

ième (3)

↓

rang = 3

e = d

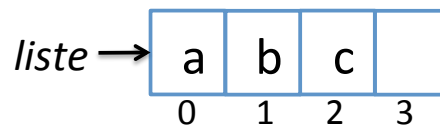


longueur = 0

ième (0)

↓

rang = 0



longueur = 3

Rang Invalide

rang = 3

↑

ième (3)

10/01/16

Manuele Kirsch Pinheiro - UP1 / CRI / EMS

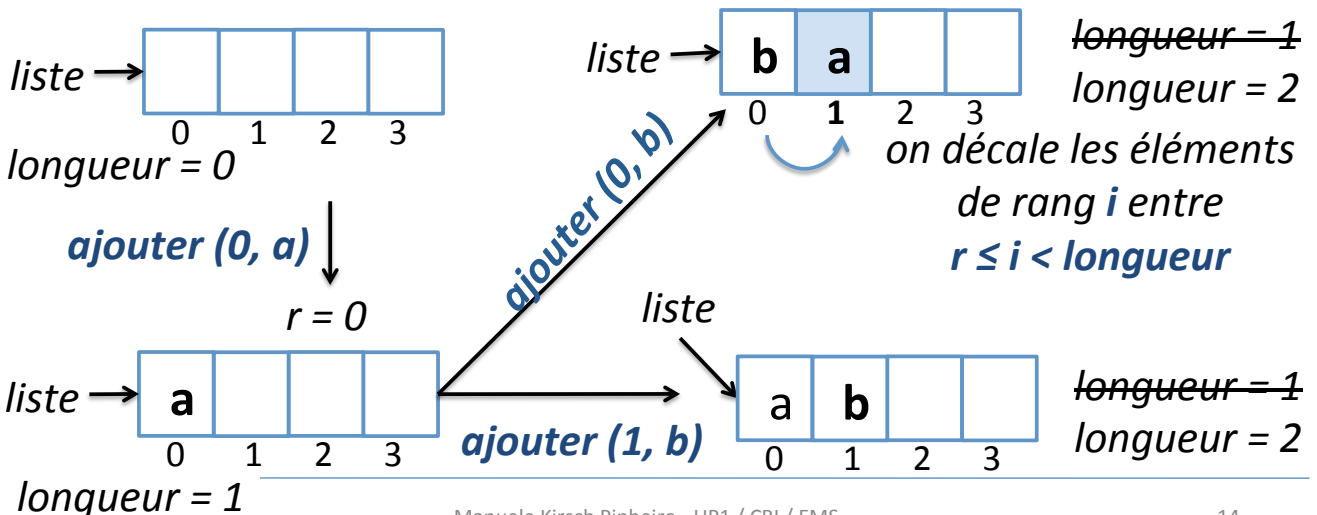
13

Listes

• Algorithme **ajouter (r,e)** : Liste sur tableau

– L'opération ajouter insère un nouvel élément **e** à un rang **r**. *Il ne s'agit pas d'un accès direct.*

– Le rang **r** doit donc être compris entre **$0 \leq r \leq \text{longueur}$**

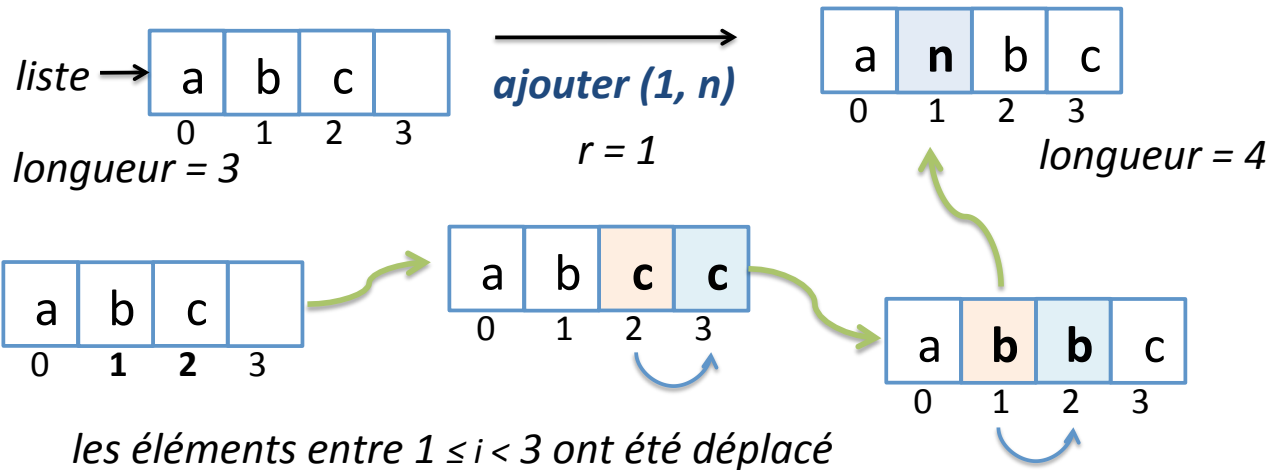


Manuele Kirsch Pinheiro - UP1 / CRI / EMS

14

Listes

- **Algorithme ajouter (r,e) : Liste sur tableau**
 - On va devoir décaler les éléments à droite du nouvel élément \rightarrow éléments rang $r \leq i < \text{longueur}$



10/01/16

attention au risque d'écrasement

15

Listes

ajouter(rang, e) : booléen

Entrée :

Entier rang

Élément e

Sortie :

booléen s

Si $0 \leq \text{rang} \leq \text{longueur}$

alors // rang valide

Si $\text{longueur} < \text{Taille}(\text{liste})$

alors // il reste de la place

Entier i

i = longueur

Tant que i > rang

faire

// on décale de fin vers le début

liste [i] = liste [i - 1]

i = i - 1

Fin tant

liste [rang] = e

longueur = longueur + 1

s = VRAI

sinon

s = FAUX

(ou lancer exception Liste Pleine)

fin si

sinon

Lancer Exception Rang Invalide

fin si

retourner s

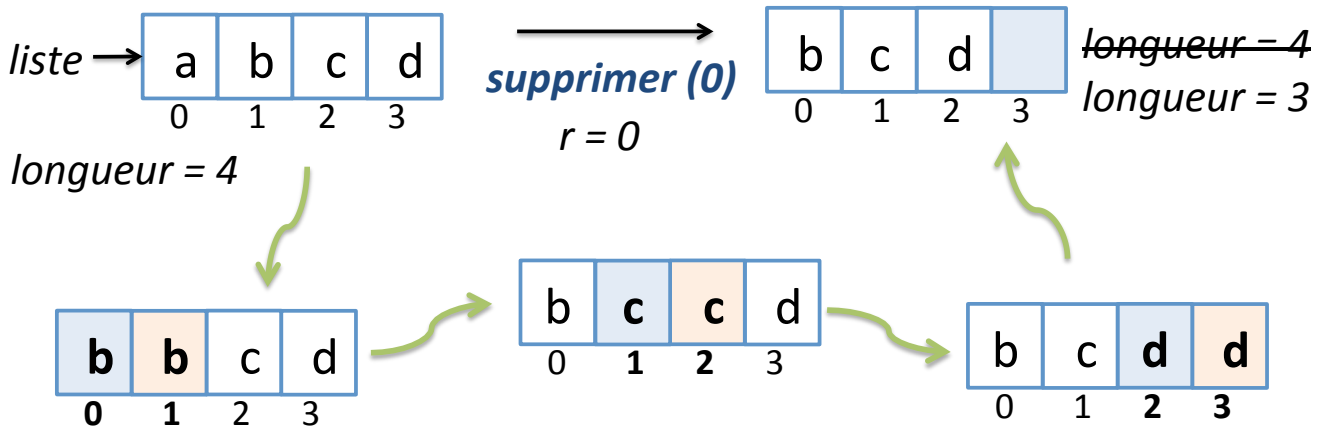
Fin ajouter

10/01/16

Manuele Kirsch Pin

Listes

- **Algorithme supprimer (r) : Liste sur tableau**
 - L'opération supprimer enlève l'élément au rang **r**.
 - Le rang **r** doit donc être compris entre $0 \leq r < \text{longueur}$



Listes

supprimer (rang) : booléen

Entrée :

Entier rang

Sortie :

booléen s

Si $0 \leq \text{rang} < \text{longueur}$

alors // rang valide

Entier i

i = rang

Tant que i < (longueur - 1)

faire // on décale vers le début

liste [i] = liste [i + 1]

i = i + 1

Fin tant

liste [i] = null

longueur = longueur - 1

s = VRAI

sinon

Lancer Exception **Rang Invalide**

(ou s = FALSE)

fin si

retourner s

Fin supprimer

- **Complexité des algorithmes :**

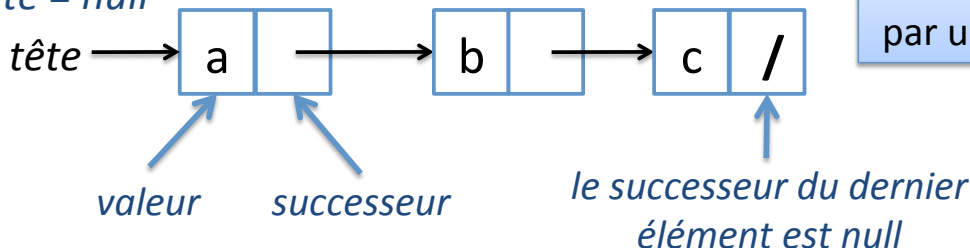
Liste sur tableau

- Les opérations **ajouter** et **supprimer** ont, sur une liste de **longueur n** , une complexité $\mathcal{O}(n)$
 - Il est nécessaire, au pire de cas, de parcourir toute la liste (pour le décalage des éléments)
 - Pire cas : **ajouter (0, e)** et **supprimer (0)**
- L'opération **ième**, en revanche, a, une complexité $\mathcal{O}(1)$
 - Aucune boucle n'est nécessaire

- **Liste chaînés (*linked list*)**

- Structure dynamique formée par des **nœuds** reliés par des **liens** (pointeurs)
- Ordre entre les éléments est déterminée par un pointeur dans chaque objet

liste vide
tête = null



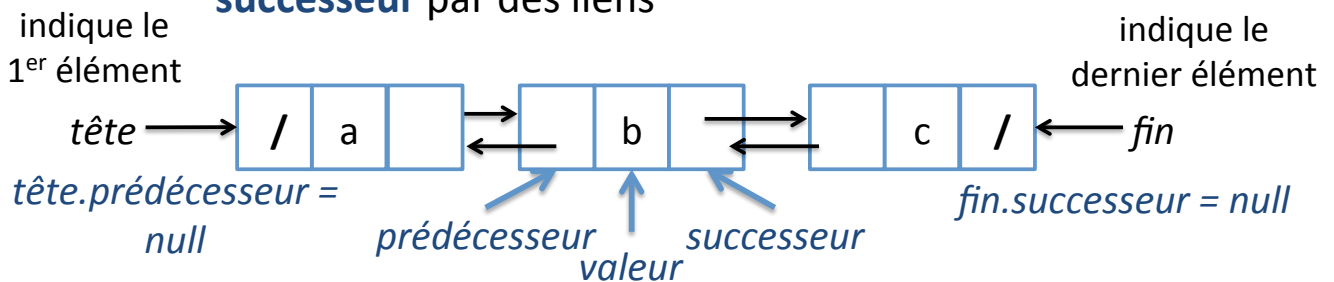
chaque élément est relié à son successeur par un lien (pointeur)

- **Variations & structure chaînée**

- Les multiples variations de la notion de liste sont faciles à mettre en œuvre par structure chaînée

- **Liste doublement chaînée**

- chaque élément est relié à son **prédécesseur** et à son **successeur** par des liens

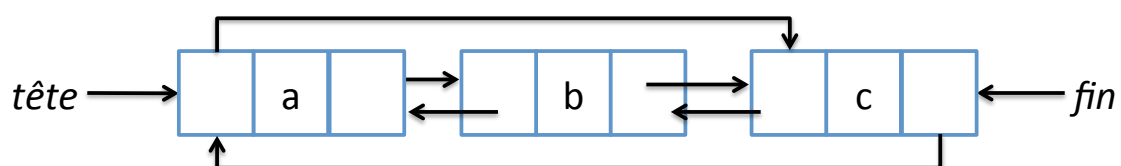


- on peut **parcourir** la liste dans les **2 sens** : du début vers la fin et vice-versa

- **Variations & structure chaînée**

- **Liste circulaire**

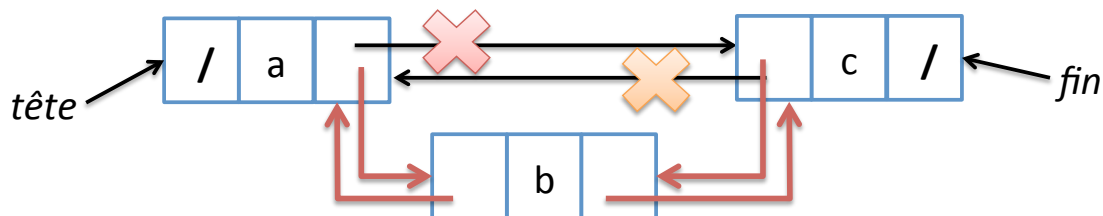
- les éléments sont organisés dans un **anneau**
- le **prédécesseur** du nœud **tête** est (pointe vers) le nœud **fin**
- le **successeur** du nœud **fin** est (pointe vers) le nœud **tête**



• Algorithmes : Listes doublement chaînées

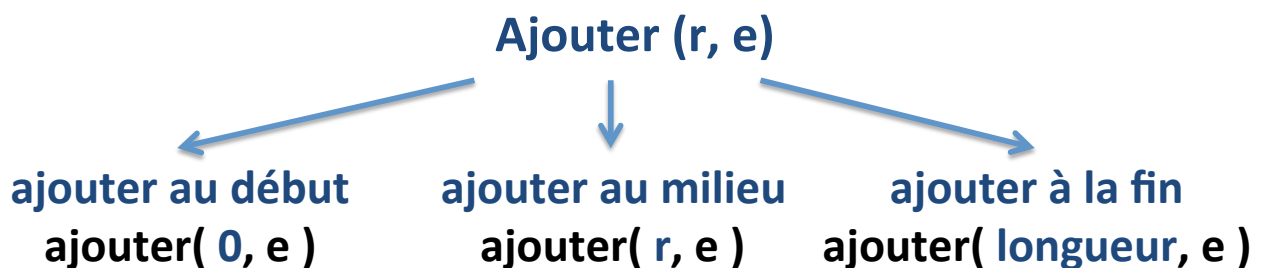
– Ajouter (r, e) :

- Trois étapes principales
 - 1) créer le nouveau nœud
 - 2) trouver sa position dans la liste
 - 3) mettre à jour les liens prédécesseur / successeur
- Cas particuliers : insertion à la tête et/ou à la fin



• Algorithmes : Listes doublement chaînées

– Trois scénarios d'usage



Listes

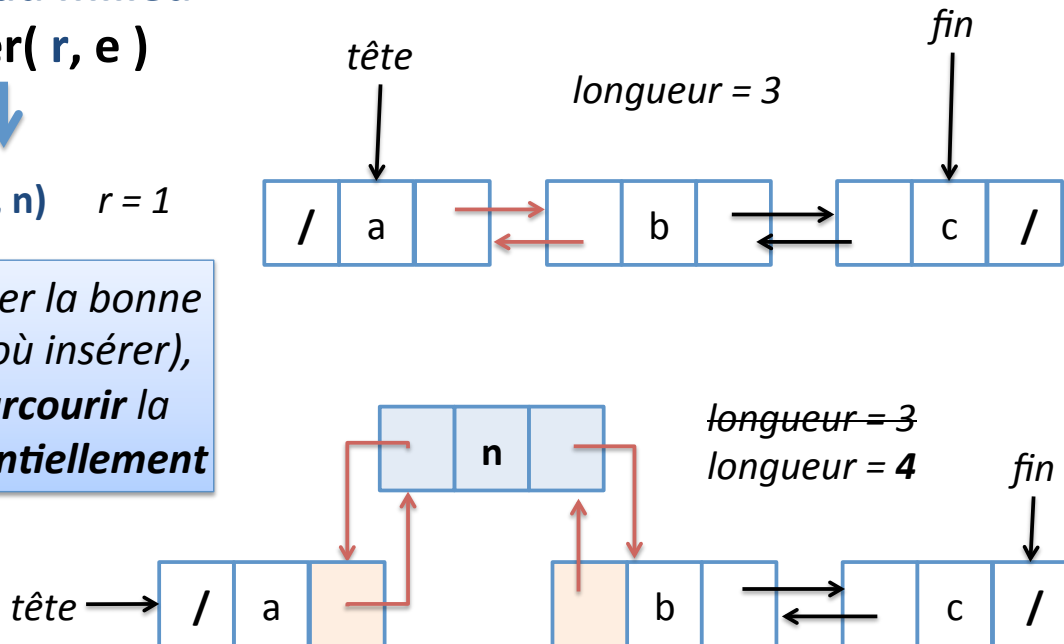
ajouter au milieu

ajouter(r, e)



ajouter(1, n) $r = 1$

Pour trouver la bonne **position** (où insérer),
il faut **parcourir** la
liste **séquentiellement**



Listes

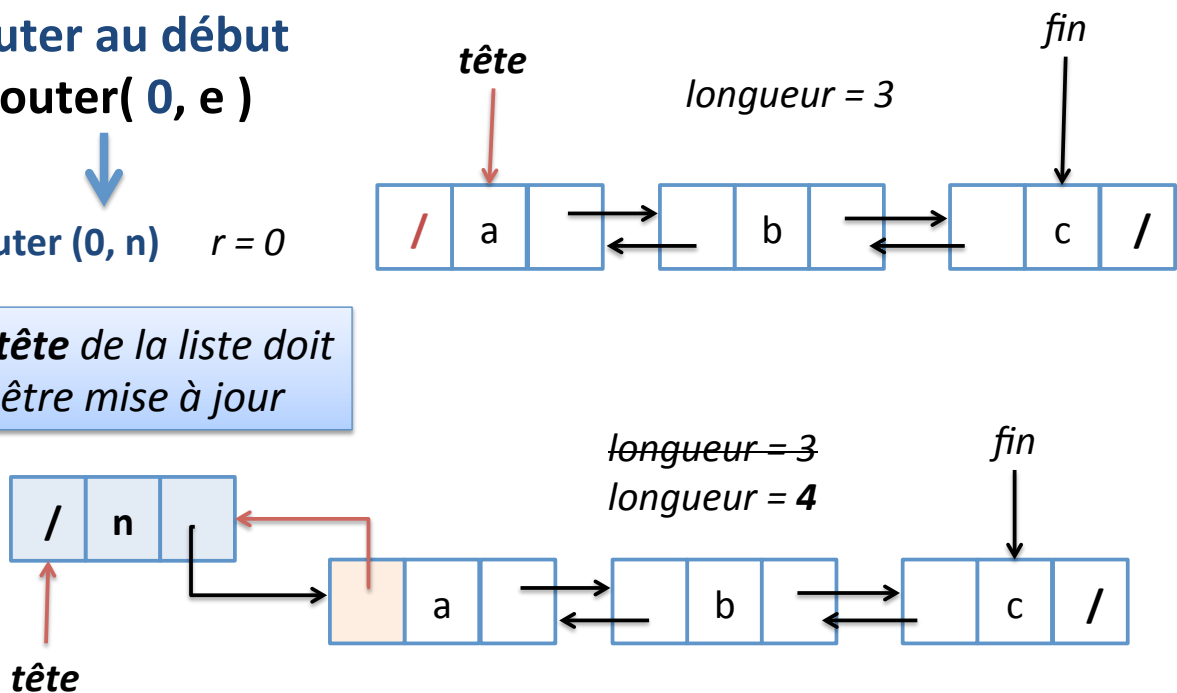
ajouter au début

ajouter(0, e)



ajouter(0, n) $r = 0$

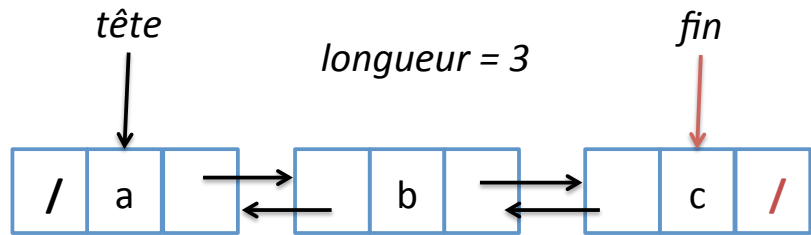
la **tête** de la liste doit
être mise à jour



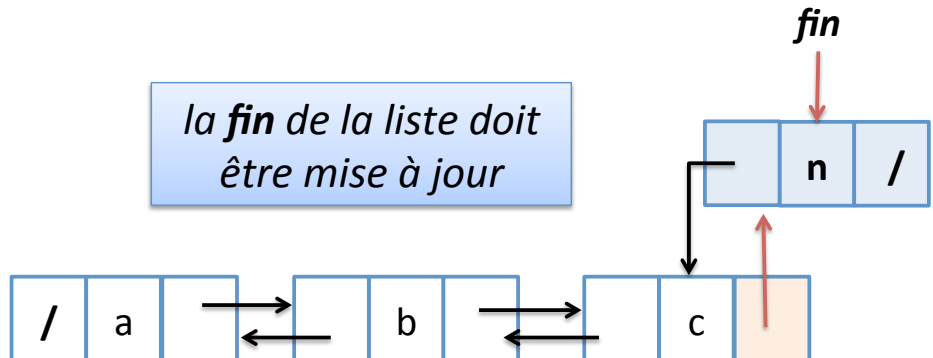
Listes

ajouter à la fin
ajouter(longueur, e)

↓
ajouter(3, n) $r = 3$
 (longueur)



~~longueur = 3~~
longueur = 4



Listes

ajouter : cas particuliers

Premier élément

↓
ajouter(0, n)

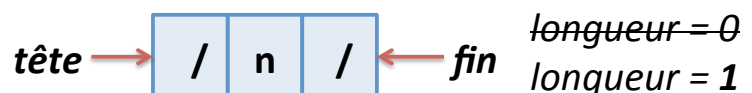
tête → / (null) longueur = 0

fin → / (null)

$r = 0$
(longueur)

ajouter(0, n)

la tête et la fin de la
liste doivent être
mises à jour



ajouter(r, e) : booléen

Entrée :

Entier r

Élément e

Sortie : booléen s

Si $0 \leq r \leq \text{longueur}$

alors // rang valide

// créer nouveau nœud

Nœud nv = **nouveau** Nœud

nv.valeur = e

// trouver le suivant p et précédent q

Nœud p, q

Si $0 < r < \text{longueur}$ alors

Entier i = 0

p = tête

Tant que $i < r$ faire

// on cherche la bonne position

q = p

p = p.suivant

i = i + 1

Fin tant

sinon

Si r = 0 alors **// insertion tête**

p = tête

q = null

tête = nv

fin si

Si r = longueur alors **// insertion fin**

p = null

q = fin

fin = nv

fin si

fin si

// mise à jour liens

nv.suivant = p

nv.précédent = q

Si p != null alors

p.précédent = nv

fin si

Si q != null alors

q.suivant = nv

fin si

longueur = longueur + 1

s = VRAI

sinon

Lancer Exception

Rang Invalide

(ou s = FAUX)

fin si

retourner s

Fin ajouter

9

Listes

- Exercices

- Simuler l'application de l'algorithme précédent pour les opérations suivantes

- ajouter (0, b)

- ajouter (0, a)

- ajouter (2, d)

- ajouter (2, c)

à partir d'une **liste vide**

tête \longrightarrow / (null)

longueur = 0

fin \longrightarrow / (null)

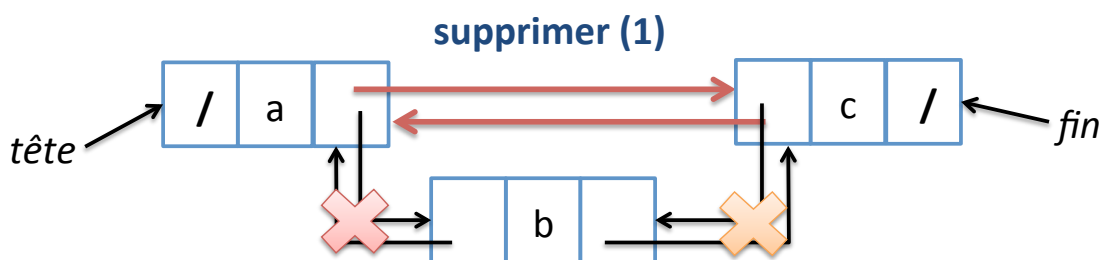


?

• Algorithmes : Listes doublement chaînées

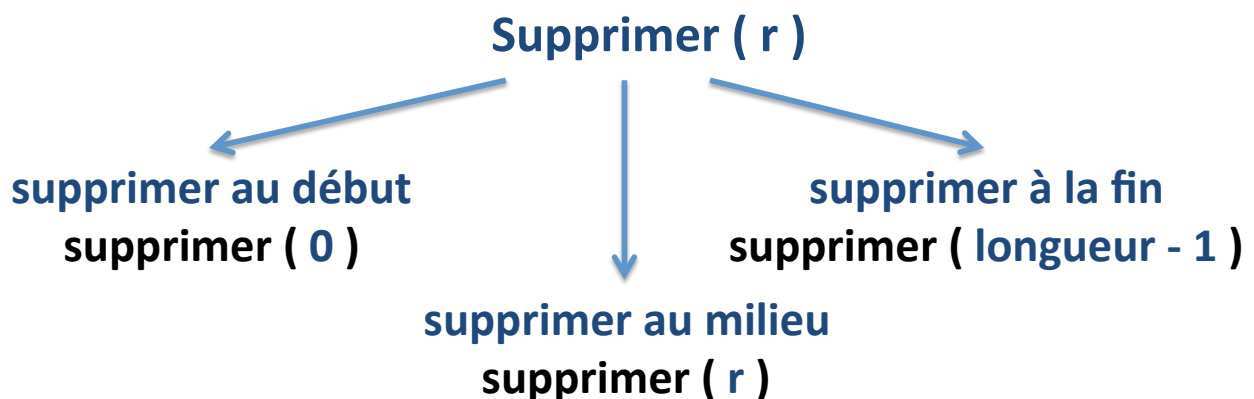
– Supprimer (r) :

- Deux étapes principales
 - 1) trouver le nœud à supprimer dans la liste
 - 2) mettre à jour les liens prédécesseur / successeur
- Cas particuliers : suppression en tête et/ou en fin de liste



• Algorithmes : Listes doublement chaînées

– Trois scénarios d'usage



Listes

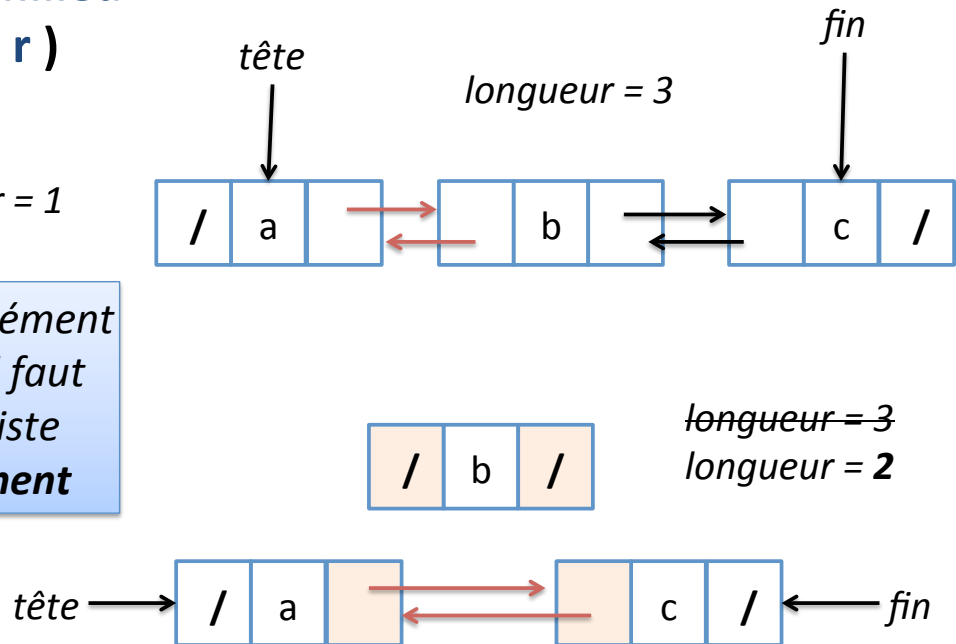
supprimer au milieu

supprimer (r)



supprimer (1) $r = 1$

Pour trouver l'élément à **supprimer**, il faut **parcourir** la liste **séquentiellement**



Listes

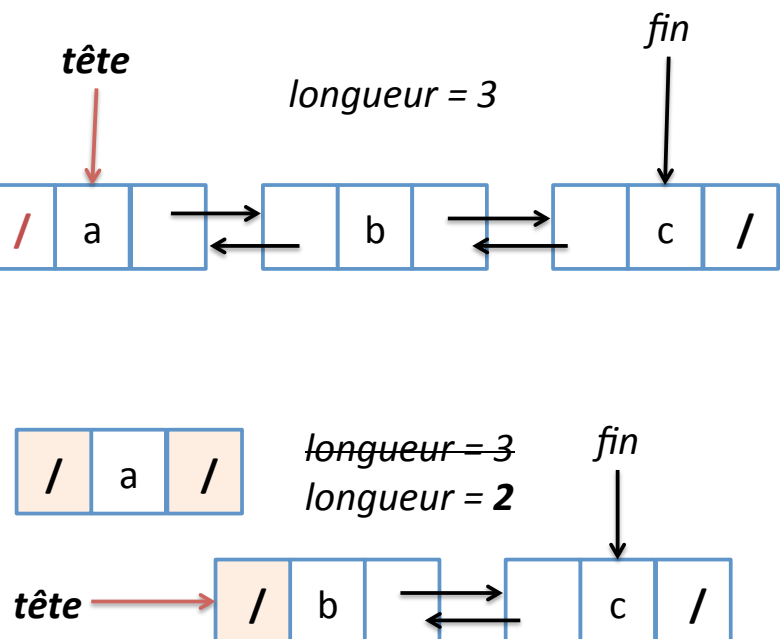
supprimer au début

supprimer (0)



supprimer (0) $r = 0$

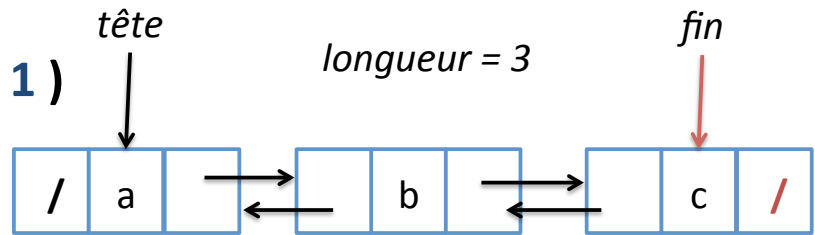
la **tête** de la liste doit être mise à jour



Listes

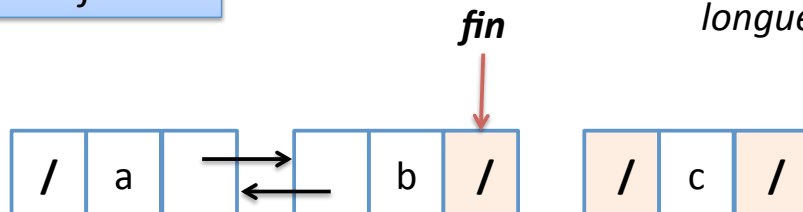
supprimer à la fin
supprimer (longueur - 1)

supprimer (2) $r = 2$
(longueur - 1)



la **fin** de la liste doit être mise à jour

longueur = 3
longueur = 2



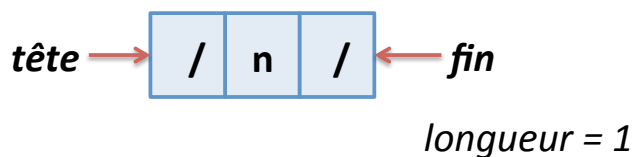
Listes

supprimer : cas particulier

Dernier élément

supprimer (0)

$r = 0$
(longueur - 1)



supprimer (0)

la **tête** et la **fin** de la liste doivent être mises à jour

tête \longrightarrow / (null)
fin \longrightarrow / (null)
longueur = 1
longueur = 0

supprimer (r) : booléen

Entrée :

Entier r

Sortie : booléen s

Si $0 \leq r < \text{longueur}$

alors // rang valide

// trouver le nœud p à supprimer

Nœud p

Si $0 < r < (\text{longueur} - 1)$ alors

Entier i = 0

p = tête

Tant que $i < r$ faire

// on cherche la bonne position

p = p.suivant

i = i + 1

Fin tant

sinon

Si $r = 0$ alors // suppression tête

p = tête

tête = tête.suivant

fin si

Si $r = \text{longueur}$ alors // suppression fin

p = fin

fin = fin.précédent

fin si

fin si

// mise à jour liens

Si p.suivant != null alors

p.suivant.précédent = p.précédent

fin si

Si p.précédent != null alors

p.précédent.suivant = p.suivant

fin si

p.suivant = null

q.précédent = null

longueur = longueur + 1

s = VRAI

sinon

Lancer Exception

Rang Invalide

(ou s = FAUX)

fin si

retourner s

Fin supprimer

10/01/16

Manue

Listes

• Exercices

– Simuler l'application de l'algorithme précédant pour les opérations suivantes

• **supprimer (2)**

• **supprimer (0)**

• **supprimer (1)**

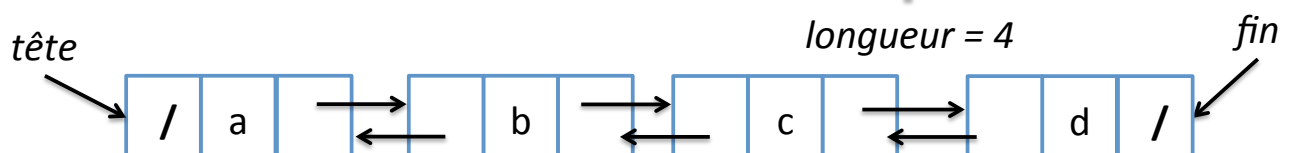
• **supprimer (0)**

tête → / (null)

longueur = 0

fin → / (null)

à partir de la **liste suivante**



• Complexité des algorithmes

- Dans une **liste chaînée**, la complexité des opérations **ajouter**, **supprimer** et **ième** est $\mathcal{O}(n)$, puisqu'il faut **parcourir la liste** pour trouver le rang
 - sauf pour les opérations concernant la tête, à la complexité $\mathcal{O}(1)$, car le **pointer vers la tête** assure un **accès direct**
- Dans une **liste doublement chaînée**, il est possible de réduire cette complexité à $\mathcal{O}(n/2)$
 - Comment ?? Illustration avec **ième**

ième (r) : Élément

Entrée : Entier r

Sortie : Élément e

Si $0 \leq r < \text{longueur}$

alors // trouver le nœud p au rang r

Nœud p

Si $0 < r < (\text{longueur} - 1)$ alors

Si $r < (\text{longueur} / 2)$ alors

Entier i = 0

p = tête

Tant que i < r faire

p = p.suivant

i = i + 1

Fin tant

sinon

Entier i = **longueur - 1**

p = fin

Tant que i > r faire

p = p.précédent

i = i - 1

Fin tant

fin Si

Listes

sinon

Si **r = 0** alors // on cherche la tête...

p = tête

fin si

Si **r = longueur** alors // ... ou la fin

p = fin

fin si

fin si

e = p.valeur

sinon

Lancer Exception **Rang Invalide**

fin si

retourner s

Fin ième

On ne parcourt que la moitié de la liste