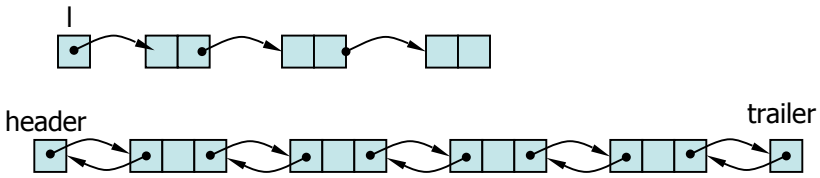
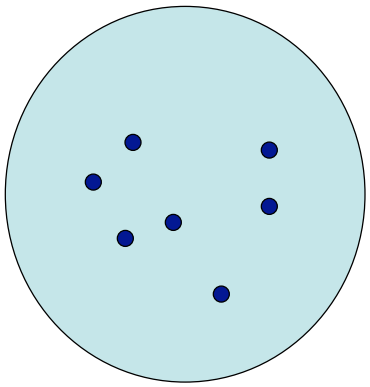
Les structures de données de base
(structures de données « concrètes »)

Tableaux

Listes chaînées



Types abstraits de données (TAD)

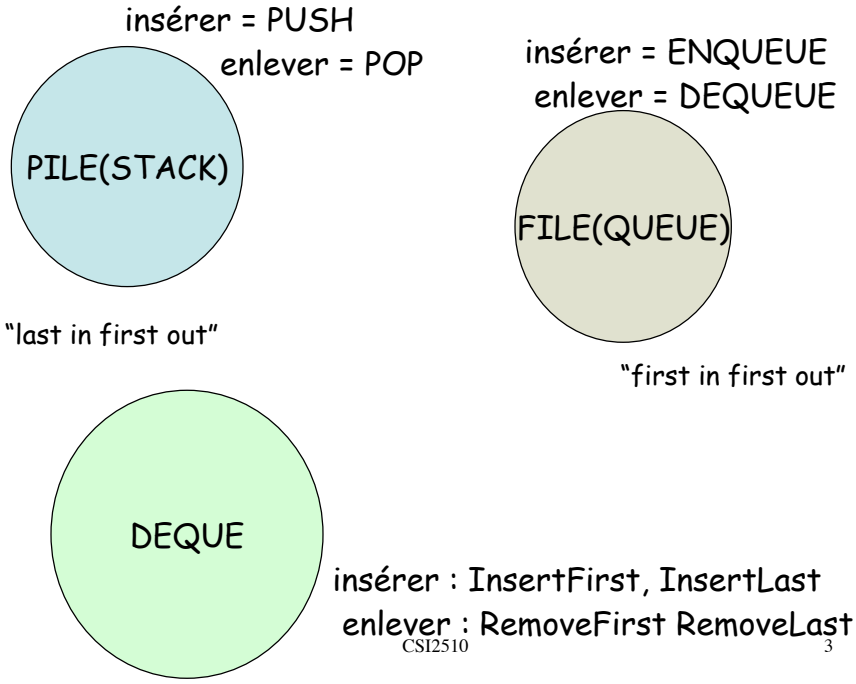


Contient des objets

On peut INSÉRER

On peut ENLEVER

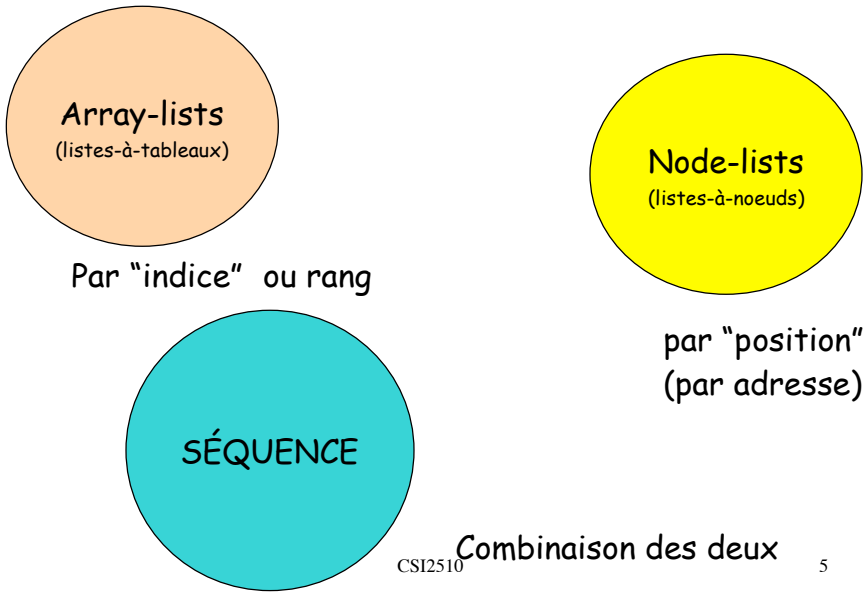
On peut .....



Ce que nous allons voir maintenant

Généralisation

LISTES = collection d'éléments
ordonnés d'une façon linéaire



Par "indice" ou rang

par "position"
(par adresse)

Combinaison des deux

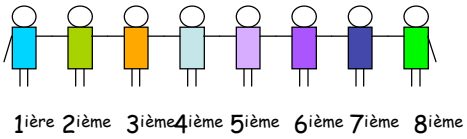
Listes et Séquences

- Array-Lists
- Node-Lists
- Séquences

Array-Lists

- Allocation séquentielle
- Les éléments sont identifiés par leur indice/rang
- Pas de relation spatiale entre les éléments (que par le rang)
- On peut accéder a n'importe quel élément directement (pas seulement le premier ou dernier)
- Tout élément est accessible par son indice/rang = nombre des éléments qui le précédent (dont les identités sont inconnues pour l'élément)
- Ex.: Liste d'étudiants organisée suivant les numéros d'id

Array-lists



Questions comme:

Qui est le 5ième ?

Le TAD Array-List

- Une séquence S (avec n éléments) qui supporte les méthodes suivantes:

-get(i):	Retourne l'élément de *S* au index i; une erreur survient si *i < 0* ou *i > n -1*
-set(i,e):	Remplace l'élément au rang *i* avec *e* et retourne l'ancien élément; un erreur survient si *i < 0* ou *i > n - 1*
-add(i,e):	Insère un nouvel élément dans *S* qui aura le rang *i*; un erreur survient si *i< 0* ou *i > n*
-remove(i):	Retire de *S* l'élément au rang *i*; une erreur survient si *i< 0* ou *i > n - 1*

Observation … Adapter Pattern

- Deux structures de données (classes) A et B avec des fonctionnalités similaires
- Adapter la structure B pour être utilisée comme A
- Créer une "wrapper class" A qui contient B

Exemples:

- ✓ Tableau -Array-list
- ✓ Array-list- Deque

Deque	Array-List
getFirst(), getLast()	get(0), get(size()-1)
addFirst(e), addLast(e)	add(0,e), add(size(),e)
removeFirst(), removeLast()	remove(0), remove(size()-1)

Implémentation intuitive: avec un tableau

- Intuitivement avec un tableau V de taille N

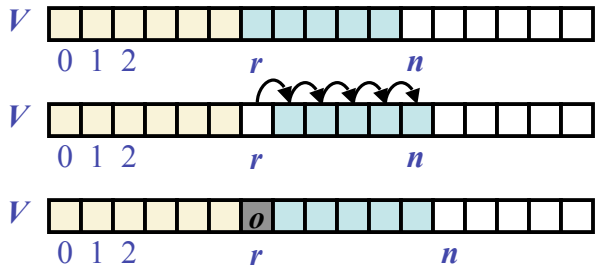
- Une variable n indique la taille du de l'Array-List (nombre d'éléments stockés)

- La méthode get(i) est exécuté en temps O(1) (retourne V[i])

Insertion

- Dans l'opération ***add(r, o)***, nous avons besoin de créer l'espace pour le nouvel élément en déplaçant avant les ***n - r*** éléments ***V[r]***, …, ***V[n - 1]***
- Dans pire des cas (***r*** = 0), le temps d'exécution est ***O(n)***

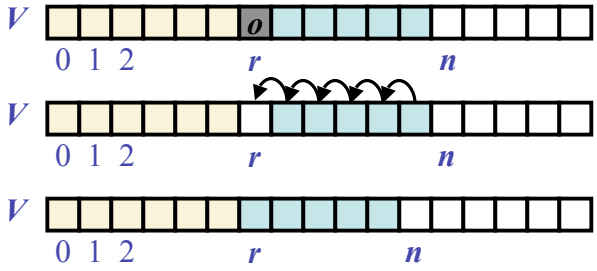
```
add(r,e):
    for i = n - 1, n - 2, ... , r do
            S[i+1] ← s[i]
    S[r] ← e
    n ← n + 1
```



Suppression

- Dans l'opération ***remove(r)***, nous avons besoin de remplir le trou laissé par l'élément enlevé en déplaçant en arrière les ***n - r - 1*** éléments ***V[r + 1], …, V[n - 1]***
- Dans pire des cas (***r*** = 0), le temps d'exécution est ***O(n)***

```
remove(r):
    e ← S[r]
    for i = r, r + 1, ... , n - 2 do
        S[i] ← S[i + 1]
    n ← n - 1
return
```



Performance

- L'espace utilisé par la structure de données est O(n)
- size, isEmpty, get et set sont exécutées en un temps O(1)
- add et remove sont exécutées en un temps O(n)

Pour la méthode add, quand le tableau est plein (n=N), au lieu d'avoir une erreur, nous pouvons remplacer le tableau avec un tableau plus grand

Performance

Complexité des diverses méthodes:

size	***O***(1)
isEmpty	***O***(1)
get	***O***(1)
replace	***O***(1)
insert	***O***(*n*)
remove	***O***(*n*)

La Classe java.util.ArrayList<E>

- Inherits from
  - java.util.AbstractCollection<E>
  - java.util.AbstractList<E>
- Implements
  - Iterable<E>
  - Collection<E>
  - List<E>
  - RandomAccess

Implémentation avec tableaux extensibles

- Les methodes
  - size(), isEmpty(), get(int) et set(int,E) en temps O(1)
  - add(int,E) et remove(int) en temps O(n)

--- Tableaux extensibles ---

Utilisés en Java pour TAD Pile, File, Liste, etc…

Implémentation d'un pile avec Tableaux extensibles

Idée:

Quand le tableau S est plein, nous pouvons remplacer le tableau par un plus grand tableau et continuer à traiter les opération *push*

```
Algorithm push(obj):
  if size() = N then
        A ← new array of length f(N)
        for i ← 0 to N - 1
            A[i] ← S[i]
        S ← A
  t ← t + 1
  S[t] ← obj
```

Tableaux extensibles

- Quelle devrait être la dimension du nouveau tableau ?
  - **Stratégie « tight »** (ajouter une constante): f(N) = N + c
  - **Stratégie de croissance** (doubler): f(N) = 2N

Tableaux extensibles- Strategie de croissance

A plein

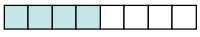
Créer B

B 

copier A dans B

B 

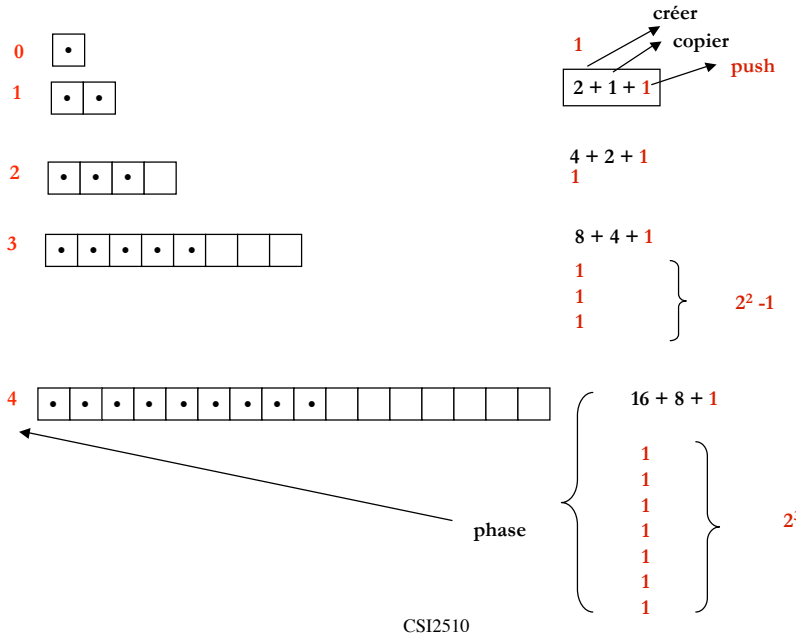
Réassigner la référence A
au nouveau tableau

A 

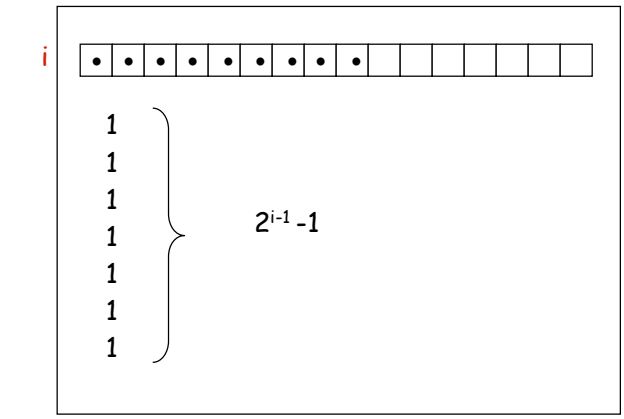
Tableaux extensibles-
Stratégie de croissance

OPÉRATION	Temps exécuté
Opération *push* régulière: ajouter un élément	1
Opération *push* spéciale: créer un tableau de taille 2N, copier N éléments, et ajouter un élément	2N+N+1

Tableaux extensibles



Tableaux extensibles



2^{i-1} push dans la phase i,
(1 special parmi eux)

coût d'un push spécial i:

$2^i + 2^{i-1} + 1 = 3\ 2^{i-1} + 1$

creer copier push

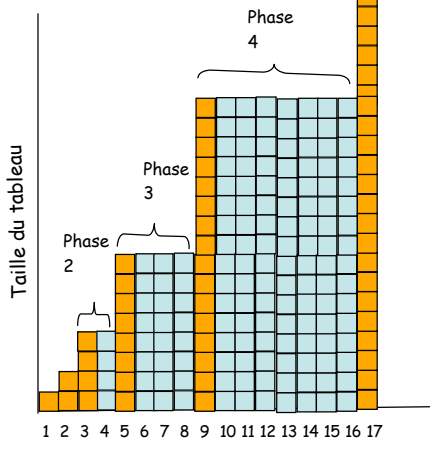
spécial normal

Totale Phase i: $3\ 2^{i-1} + 1$ + $2^{i-1} - 1$

Tableaux extensibles
Stratégie de croissance

coût total Phase i:

$3\ 2^{i-1} + 1 + 2^{i-1} - 1 = 2^{i+1}$



COMBIEN DE PHASES POUR
effectuer n push ?

Si n est une puissance de 2:
La taille du tableau devient n
quand $n = 2^i$

$i = \log n$

Quand j'ai effectué n push
je me trouve dans la phase $\lfloor \log n \rfloor$

Tableaux extensibles

Donc, il y a log n phases:
(⌊log n ⌋ pour être précis)
Chaque phase coût: 2^{i+1}

$$\sum_{0}^{\log n} 2^{i+1} = 2\left(\sum_{0}^{\log n} 2^{i}\right)$$

$$= 2(2^{\log n+1} - 1)$$

$$= 2(2\ 2^{\log n} - 1)$$

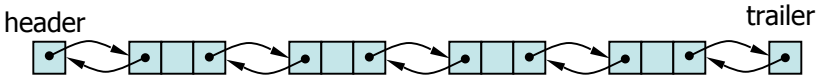
$$= 2(2\ n - 1) = O(n)$$

N'OUBLIEZ PAS:

$$S = \sum_{i=0}^{n} 2^{i} = 2^{n+1}-1$$

Le TAD Array List – Implémentation avec une liste doublement chaînée

Est-ce que c'est efficace d'implémenter une « array-list » avec une liste doublement chaînée ?



Non

get(i) ???

```
Algorithm get(rank)
        if (rank <= size()/2) { //scan forward from head
                                node ← header.next
                                for (int i=0; i < rank; i++)
                                node ← node.next
    }else { // scan backward from the tail
                                node ← trailer.prev
                                for (int i=0; i < size()-rank-1 ; i++)
                                node ← node.prev
     }
     return node;
```


Performance …

avec une liste doublement chaînée

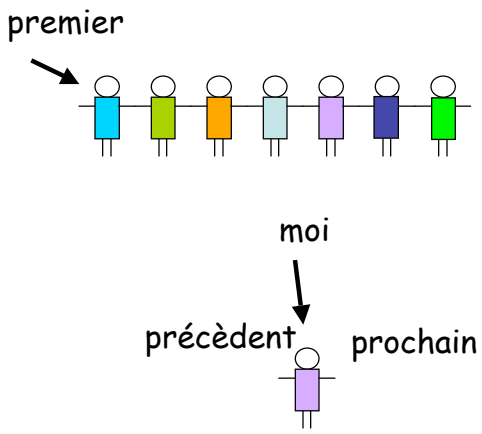
size	***O***(1)
isEmpty	***O***(1)
get	***O***(n)
replace	***O***(n)
insert	***O***(***n***)
remove	***O***(***n***)

Node-List

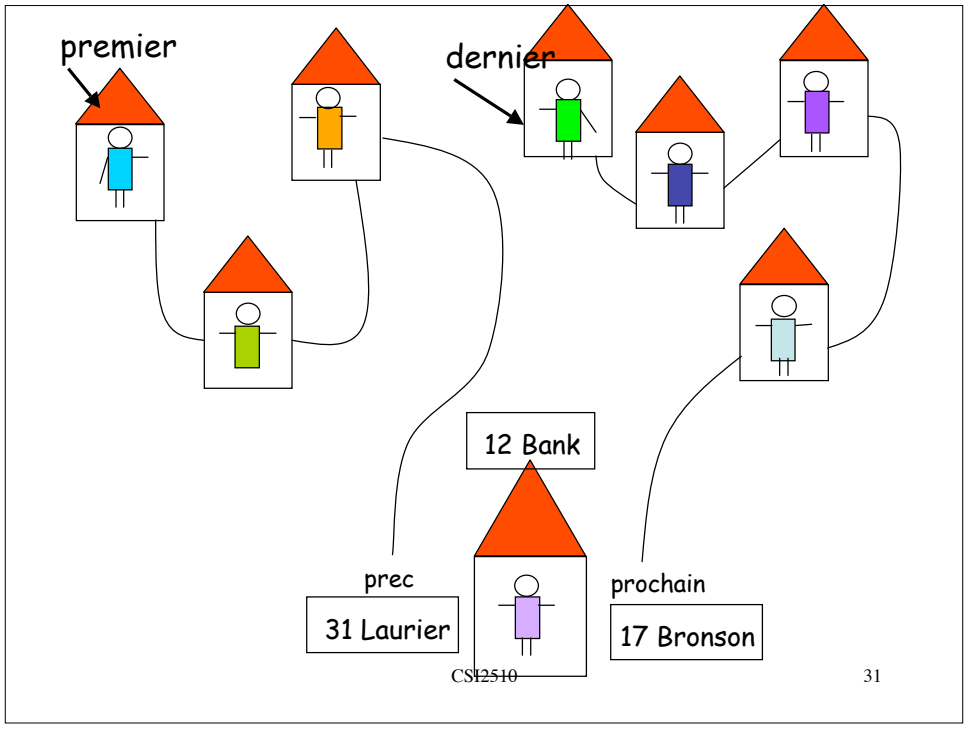
Contenant d'éléments où chaque élément est stocké à une **position** donnée;
Les positions sont rangées dans un ordre linéaire

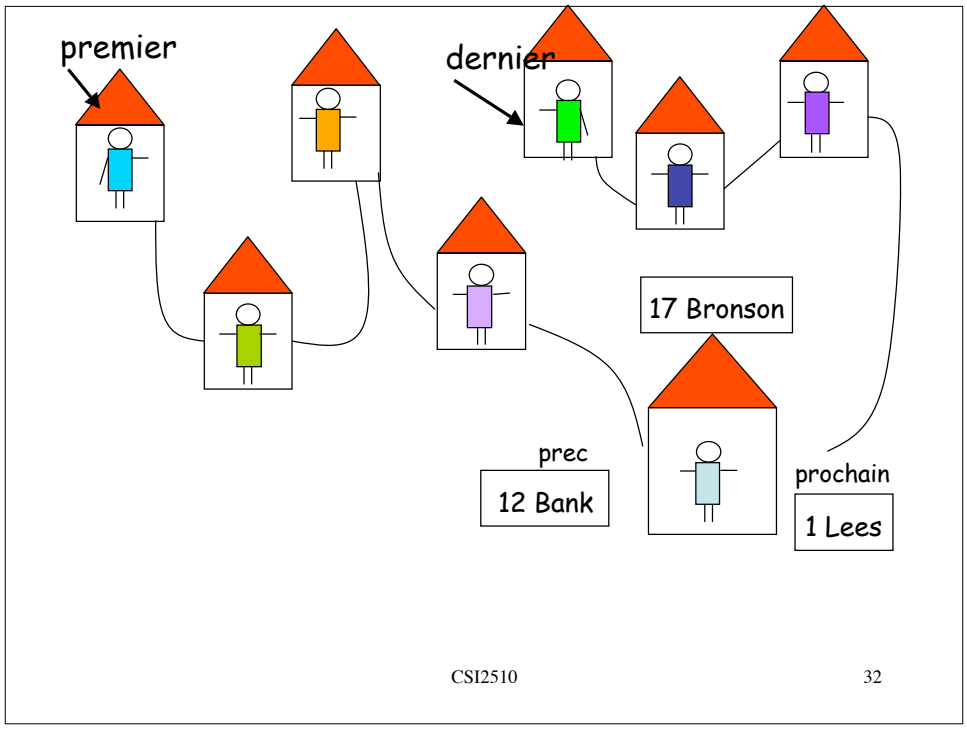
Chaque élément fait référence aux éléments **suivant** et **précèdent** et il est accessible à partir de ses voisins

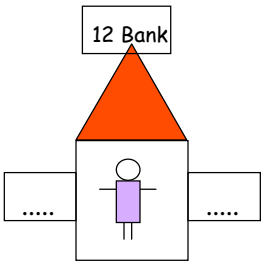
- On ne peut pas accéder à un élément directement, on peut accéder juste au premier ou au dernier.
- Les éléments sont accédés par leur position.
  - Les positions sont définies relatives aux autres positions (adresse/place). Les positions sont liées par les relations spatiales avant/après



Je ne connais pas mon indice - la notion de indice ou n'existe pas.
Je sais seulement qui est le prochain et le précèdent.


premier
dernier
12 Bank
prec
31 Laurier
prochain
17 Bronson
CSI2510
31


premier
dernier
17 Bronson
prec
12 Bank
prochain
1 Lees
CSI2510
32



Si j'ai connais une addresse
Je peux poser une question
comme:

Qui est après 12 Bank ?

Qui est premier ?
Qui est dernier ?

Le TAD Node List

- Une séquence S de n éléments qui supporte les méthodes suivantes:
  - ✓ first(): Retourne la position du premier élément de S; une erreur survient si S est vide
  - ✓ last(): Retourne la position du dernier élément de S; une erreur survient si S est vide
  - ✓ prev(p):Retourne la position de l'élément de S qui précède celui qui est à la position p; une erreur survient si p est le premier élément de S
  - ✓ next(p):Retourne la position de l'élément de S qui suit celui qui est à la position p; une erreur survient si p est le dernier élément de S

Le TAD Node List

✓set(p,e): Remplace l'élément à la position p par e et retourne l'élément se situant à p auparavant

✓addFirst(e): Insère un élément e dans S en tant que premier élément

✓addLast(e): Insère un élément e dans S en tant que dernier élément

✓addBefore(p,e): Insère un élément e dans S avant la position p

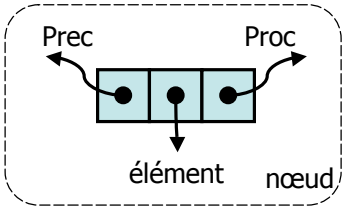
✓addAfter(p,e): Insère un élément e dans S après la position p

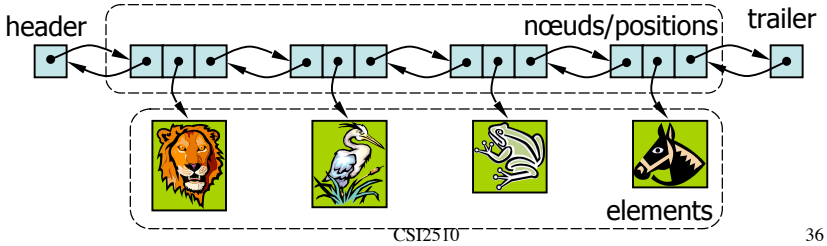
✓remove(p): Supprime et retourne l'élément e de S se trouvant à la position p et invalide cette position dans S

Une erreur survient si p null ou déjà supprimé ou est une position d'une autre liste etc.

Implementation intuitive: avec une liste doublement chaînée

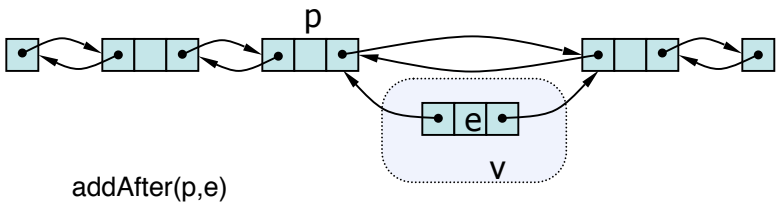
- Une liste doublement chaînée est l'implémentation intuitive de le TAD Liste
- Nœuds implémente Position et conserve:
  - élément
  - Link à le nœud précèdent
  - Link à le nœud prochain
- Nœuds spécial : trailer et header





Insertion

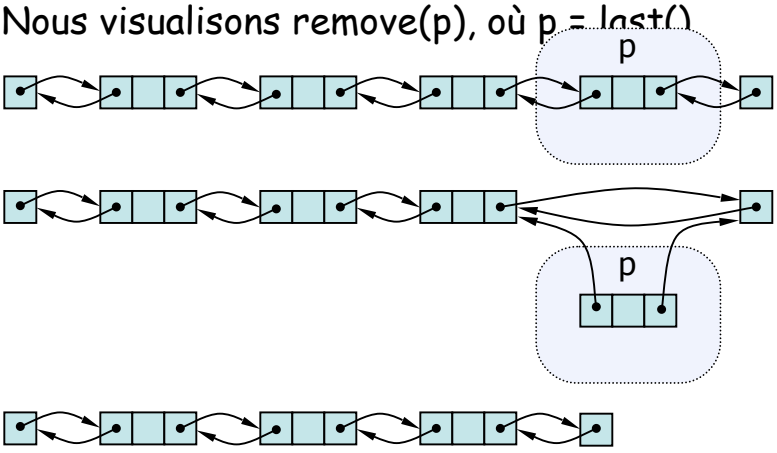
- Nous visualisons l'opération addAfter(p, X), qui retourne le position q

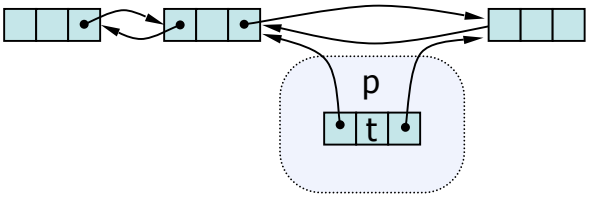


addAfter(p,e)
Create a new node v
v.setElement(e)
v.setPrev(p)
v.setNext(p.getNext())
(p.getNext()).setPrev(v)
p.setNext(v)

Suppression

- Nous visualisons remove(p), où p = last()





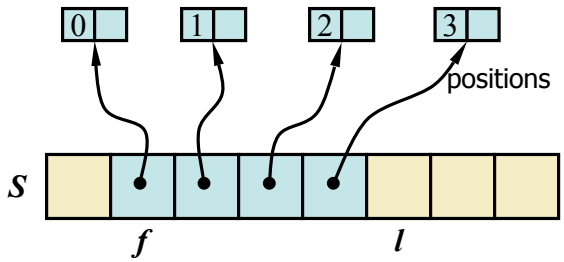

```
remove(p)
        t ← p.element
        (p.getPrev()).setNext(p.getNext())
        (p.getNext()).setPrev(p.getPrev())
        p.setPrev(null)
        p.setNext(null)
        return t
```

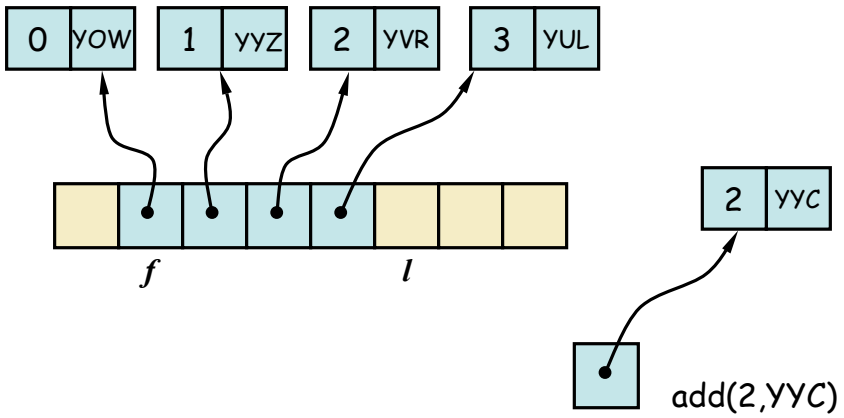

Le TAD Séquence

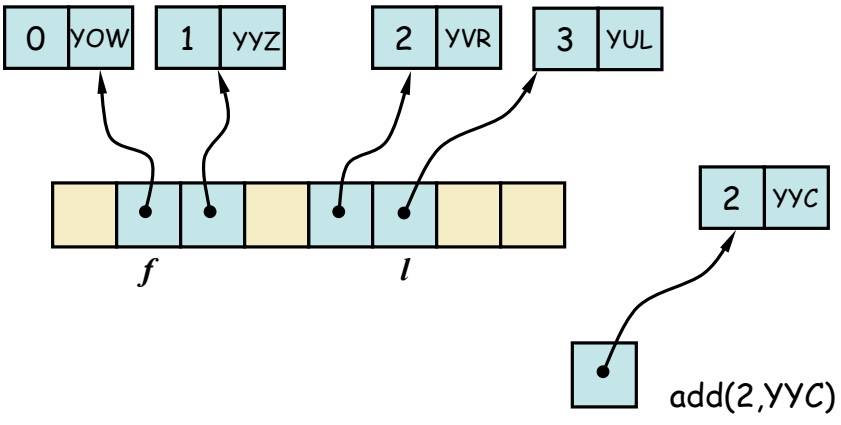
- **Un plus général TAD**
- Combine les TAD Array-list et Node-List (héritage multiple)
- Ajoute des méthodes qui font le pont entre rangs et positions
  - -atIndex(i) retourne une position
  - -indexOf(p) retourne un indice (entier)

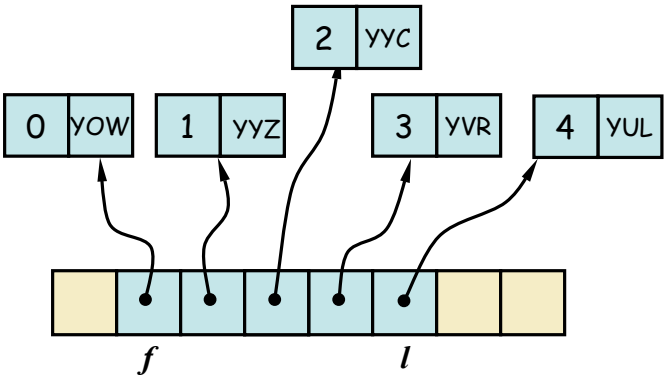
Implémentation à base de tableau

- Le tableau circulaire conserve des positions
- Un objet position conserve:
  - Élément
  - Indice
- *f* et *l* gardent la première et la dernière postions






0
YOW
1
YYZ
2
YVR
3
YUL
2
YYC
f
l
add(2,YYC)


2
YYC
0
YOW
1
YYZ
3
YVR
4
YUL
f
l

Implémentation à base de tableau

addFirst, addBefore, addAfter, remove

O(n)

Aussi: add, remove basé sur indices

O(n)

Autres methodes

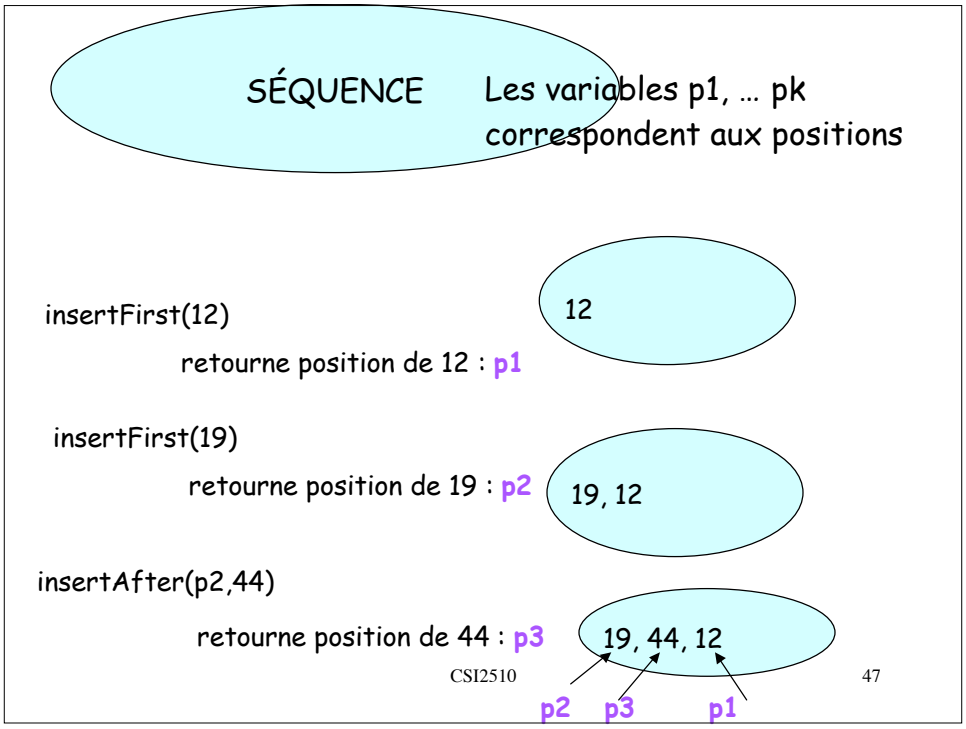
O(1)

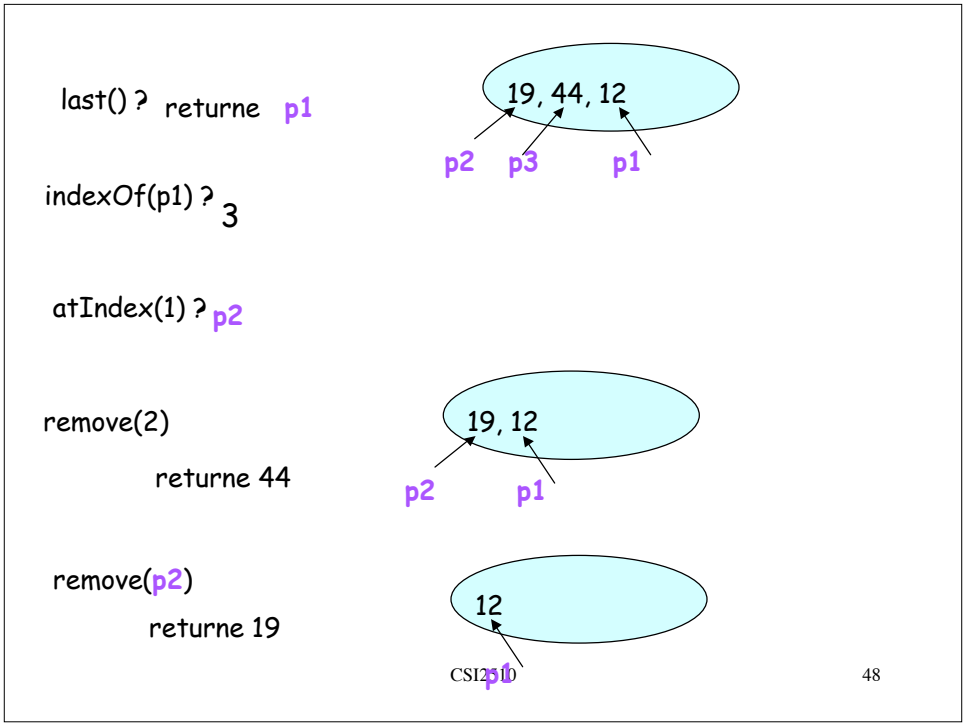
Implémentation avec Liste doublement chaînée

Toutes les méthodes sont héritées

Ponts:

atIndex(i), indexOf(p): O(n)


SÉQUENCE
Les variables p1, … pk correspondent aux positions
insertFirst(12)
retourne position de 12 : p1
12
insertFirst(19)
retourne position de 19 : p2
19, 12
insertAfter(p2,44)
retourne position de 44 : p3
19, 44, 12
p2
p3
p1
CSI2510
47


last() ? returne p1
19, 44, 12
p2
p3
p1
indexOf(p1) ? 3
atIndex(1) ? p2
remove(2)
returne 44
19, 12
p2
p1
remove(p2)
returne 19
12
p1
CSI2510
48

Séquences et Java Collections Framework

java.util.List<E> est un interface qui est réalisé par

- java.util.LinkedList<E>
- java.util.ArrayList<E>
  - Aside: List works with indicies/rank

Fait partie du "Java Collections Framework"

- Structure du framework
  - interfaces, e.g., List, Queue, Map, Set
  - implementations, e.g., ArrayList, IndexList, PriorityQueue, HashSet
- Toujours:
  - iterator(), size(), isEmpty()

Performance

- Dans le réalisation de le TAD Liste avec une liste doublement chaînée
  - L'espace utilisé par une liste avec n éléments est $O(n)$
  - L'espace utilisé par chaque position de la liste est $O(1)$
  - Chaque méthode de le TAD Liste est exécutée en temps $O(1)$
  - Opération element() de le TAD est exécutée en temps $O(1)$

Liste Doublement chaînée dans Java Collections Framework

Réalisation alternative d'une séquence en Java

- Inherents from
  - java.util.AbstractCollection<E>
  - java.util.AbstractList<E>
  - java.util.AbstractSequentialList<E>
- Implements
  - Iterable<E>
  - Collection<E>
  - Deque<E>, List<E>, Queue<E>

Methods

- size(), isEmpty(), addFirst(E) and addLast(E) in O(1) time
- Il n y a pas: prev, next, addBefore ou addAfter qui sont nécessaires pour nodeList ADT

Observation … "Iterators"

Iterators passent a travers une collection d'éléments

- Ils sont associes a une séquence
- Il y a la notion d'élément actuel
- Il y a accès au prochain
- On peux avancer au prochain

généralisation d'une boucle "FOR" sur les élément de la collection

Iterator ADT

- hasNext()
- next()

Collections support Iterators in Java

- interface Iterable<E>
- Une méthode: Iterator<E> iterator()

java.util.LinkedList<E>

Ce n'est pas un nodeList; seulement a liste doublement chaînée

Stratégie:

- utilisez java.util.LinkedList
- Avec l' Iterator java.util.ListIterator
- Regardez l'iterator comme une sous-classe d'un ADT position.

SEQUENCES: CONCLUSION

Séquences avec Tableaux :

Il faut deplacer des elements

addFirst,addBefore,addAfter,add(i,e) ---- O(n)

remove(position) remove(index) ---- O(n)

Ponts: atIndex(i), indexOf(p): ---- O(1)

get(i), set(i,e) ----- O(1)

Parce que la position garde aussi l'info sur l'index

Séquences avec listes doublement chaînes:

addFirst,addBefore,addAfter, remove(position) ---- O(1)

add(i,e)

remove(index) ---- O(n)

Bridges: atIndex(i), indexOf(p): ---- O(n)

Il faut traverser pour trouver l'index

get(i), set(i,e) ----- O(n)