Types abstraits

Types abstraits

Abstract Data Type — ADT

Type abstrait : un ensemble d'objets et un ensemble d'opérations

But : comparaison d'implantations qui offrent la même fonctionnalité

Pensez, p.e., à des `interfaces` de Java : signature des méthodes+sémantique (en Javadoc) mais plusieurs implantations possibles

Notez que `interface` ne définit que le syntaxe des opérations donc il ne correspond pas à la notion de l'ADT

Java interface

Overview Package Class Use Tree Deprecated Index Help

Java™ 2 Platform Std. Ed. v1.4.2

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

java.util

Interface Collection

All Known Subinterfaces:
BeanContext, BeanContextServices, List, Set, SortedSet

All Known Implementing Classes:
AbstractCollection, AbstractList, AbstractSet, ArrayList, BeanContextServicesSupport, BeanContextSupport, HashSet, LinkedHashSet, LinkedList, TreeSet, Vector

....

Method Summary	
boolean	`add(Object o)` Ensures that this collection contains the specified element (optional operation).
boolean	`addAll(Collection c)` Adds all of the elements in the specified collection to this collection (optional operation).
void	`clear()` Removes all of the elements from this collection (optional operation).
boolean	`contains(Object o)` Returns `true` if this collection contains the specified element.
boolean	`containsAll(Collection c)` Returns `true` if this collection contains all of the elements in the specified collection.
boolean	`equals(Object o)` Compares the specified object with this collection for equality.
int	`hashCode()` Returns the hash code value for this collection.
boolean	`isEmpty()` Returns `true` if this collection contains no elements.

pas un ADT mais une abstraction . . .

Nombres naturels

Objets : $N = \{0, 1, \ldots\}$

Opérations :
add: $N \times N \mapsto N$;
succ: $N \mapsto N$;
eq ?: $N \times N \mapsto \{\text{vrai}, \text{faux}\}$;
less: $N \times N \mapsto \{\text{vrai}, \text{faux}\}$

Beaucoup d'implantations possibles :
chaînes de chiffres (base 10, `String`), factorisation en primes (`int[]`), $i \in N$ representé par un `char[]` de longueur i

Grande différence dans l'efficacité de l'exécution d'opérations

Nombres naturels — implantation 1

```
public class N {
   private int valeur ;  // implantation basé sur le type int
   public N(int v){this.valeur = v;}
   public N add(N a, N b){
      return new N(a.valeur+b.valeur);
   }
   ...
}
```

implantation facile

problème de représentation (si $v \geq 2^{31}$)

Nombres naturels — implantation 2

```
public class N {
   private String valeur ;  // chaîne de chiffres
   public N(String v){this.valeur = v;}
   public N add(N a, N b){
      // ... l'«algorithme» d'addition sur papier
   }
   ...
}
```

implantation un peu plus compliqué

pas de problème de représentation (au moins jusqu'à $10^{2^{31}}$ — longueur max de `String` est `Integer.MAX_VALUE`)

Nombres naturels — implantation 3

factorisation : $312 = 2^3 \cdot 3 \cdot 13 = 2^3 \cdot 3^1 \cdot 5^0 \cdot 7^0 \cdot 11^0 \cdot 13^1$ representé par `int[] fact = {3,1,0,0,0,1}`

```
public class N {
   private int[] fact ;   //  factorisation en primes
   public N(String in){
    // factorisation
   }
   public N add(N a, N b){
      // ... algorithme d'addition + factorisation du résultat
   }
   ...
}
```

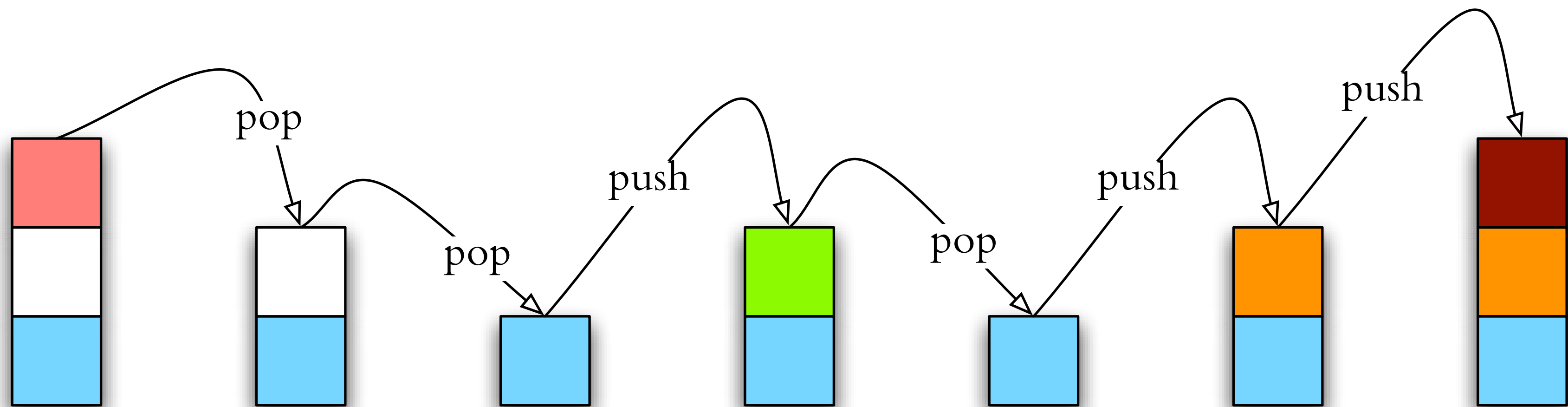
implantation assez compliqué

par contre, pgcd est facile à calculer sans l'algo d'Euclid :
si $a = \sum_i p_i^{a_i}$ et $b = \sum_i p_i^{b_i}$, alors $\mathrm{pgcd}(a,b) = \sum_i p_i^{\min\{a_i,b_i\}}$.
(avec les nombres premiers $p_1 = 2, p_2 = 3, p_3 = 5, \ldots$)

Pile — idée

Idée de pile : objets empilés un sur l'autre, on ne peut accéder qu'à l'élément supérieur

opérations : «empiler» (push) et «dépiler» (pop)



Pile

Objets : listes de style $A_1, A_2, \ldots, A_n$ et la liste nulle

Opérations :
$\mathsf{push}\colon P \times O \mapsto P$; p.e., $\mathsf{push}(P, x)$
$\mathsf{pop}\colon P \mapsto P \times O$; p.e., $\mathsf{pop}(P)$
$\mathsf{isEmpty}\colon P \mapsto \{\mathsf{vrai}, \mathsf{faux}\}$

En fait, `push` et `pop` sont implantés en modifiant leur argument de pile (au lieu de retourner une nouvelle pile)

Pile — implantation avec un tableau

Idée : on utilise un tableau avec un indice pour le sommet de la pile

(Un petit problème : la taille de la pile est bornée dans cette implantation)

```
public class Pile {
   private int sommet;
   private Object[] P;
   private static final int MAX_TAILLE = 100;

   public Pile(){
      P = new Object[MAX_TAILLE];
      sommet = 0;
   }
   public boolean isEmpty(){return (sommet==0);}
   ...
}
```

`sommet` indique où mettre le prochain objet de `push`

Pile — cont.

```
public void push(Object O){
   P[sommet] = O;
   sommet++;
}
```

Qu'est-ce qui se passe si on a trop d'éléments sur la pile ?

→ la pile déborde (*overflow*)

dans cette implantation : `ArrayIndexOutOfBoundsException`

c'est OK

Pile — cont.

```
public Object pop(){
    sommet --;
    return P[sommet];
}
```

Qu'est-ce qui se passe si on essaie de dépiler d'une pile vide ?
$\longrightarrow$ la pile déborde négativement (*underflow*)

dans cette implantation : `ArrayIndexOutOfBoundsException`
n'est pas trop élégant

Pile — cont.

Introduisons une exception spécifique à notre pile : `StackUnderflowException`

```
public class Pile {
    ...
    static class StackUnderflowException extends RuntimeException {
        private StackUnderflowException(String message){
            super(message);
        }
    }
    public Object pop(){
        if (sommet == 0)
            throw new StackUnderflowException("Rien ici.");
        sommet --;
        return P[sommet];
    }
}
```

(comme c'est une sous-classe de `RuntimeException`, on ne doit pas la déclarer par `throws`)

Pile — efficacité

Temps de calul par opération : $O(1)$

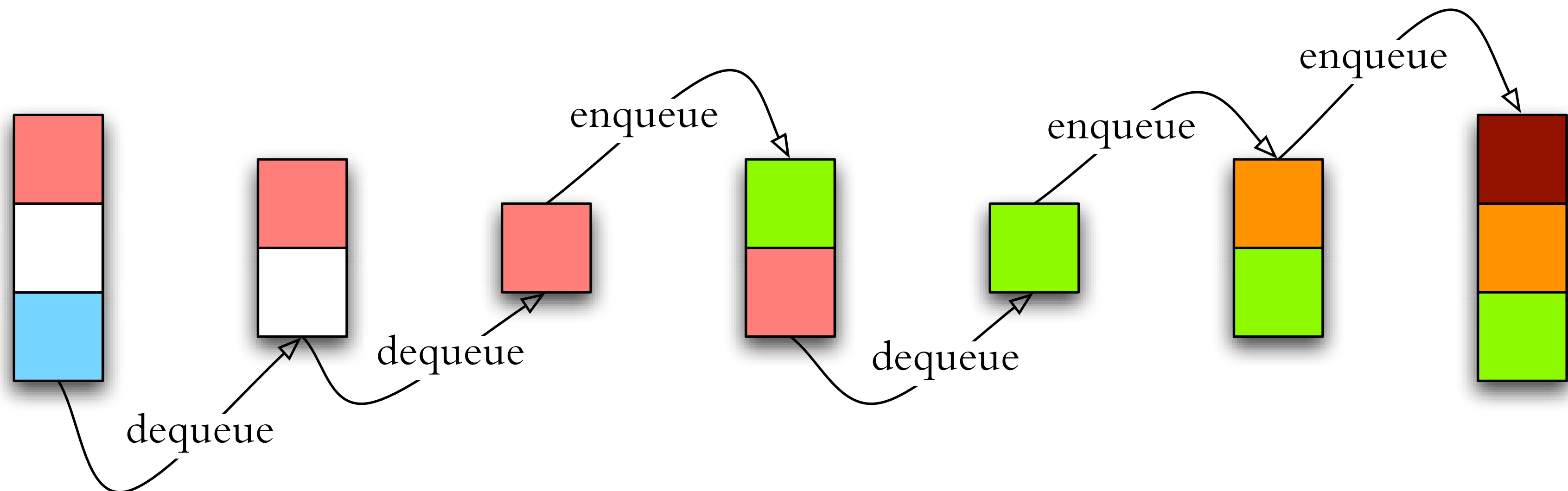
Accès à un élément quelconque : n'est pas possible directement, il faut utiliser une autre liste pour dépiler tous les éléments sur l'élément cherché ça prend $O(n)$ temps et une espace de travail $O(n)$ pour l'élément en position n

Queue — idée

(queue ou file, *queue* en anglais)

Idée de queue : comme une file d'attente, objets rangés un après l'autre, on peut enfiler à la fin ou défiler au début

opérations : «enfiler» (enqueue) et «défiler» (dequeue)



Queue

Objets : listes de style $A_1, A_2, \ldots, A_n$ et la liste nulle

Opérations :
enqueue: $Q \times O \mapsto Q$; p.e., enqueue(Q, x)
dequeue: $Q \mapsto Q \times O$; p.e., dequeue(Q)
isEmpty: $Q \mapsto \{\text{vrai}, \text{faux}\}$

En fait, `enqueue` et `dequeue` sont implantés en modifiant leur argument de queue (au lieu de retourner une nouvelle queue)

Queue — implantation inefficace

avec `Object[] Q` pour stocker les éléments de la queue

implantation inefficace — comme à la caisse du supermarché :
1. dequeue décale tous les éléments vers le début de `Q` et retourne l'ancien élément `Q[0]`
2. enqueue(x) cherche la première case vide sur `Q` et y met x

Efficacité : $O(\ell)$ sur une queue de longueur ℓ

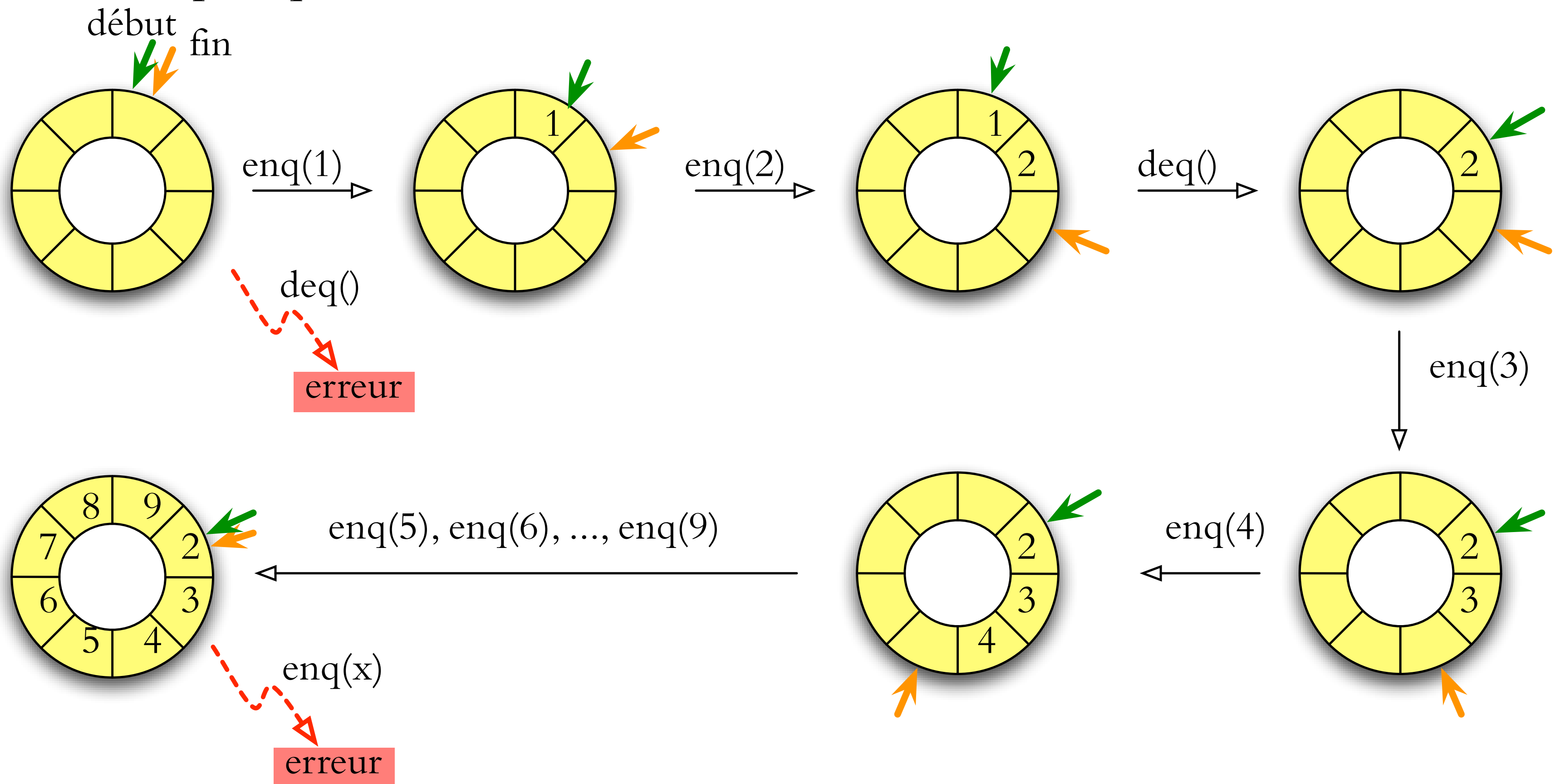
On peut exécuter enqueue en $O(1)$ si on maintient l'indice de la fin de la queue (comme le sommet de la pile)

Qu'est-ce qu'on fait pour dequeue ?

Queue — implantation avec un «anneau»

Idée : utiliser un tableau circulaire («anneau») avec deux indices pour le début et fin de la queue

anneau en pratique : en utilisant $\mathsf{mod}\, n$ avec un tableau de taille n



Queue — cont.

(la taille de la queue est bornée dans cette implantation)

```
public class Queue {
   private int debut;
   private int fin;
   private Object[] Q;
   private static final int MAX_TAILLE=2006;
   public Queue(){
      debut=fin=0;
      Q=new Object[MAX_TAILLE];
      for (int i=0; i<MAX_TAILLE; i++)
         Q[i] = VIDE;
   }
   private static final Object VIDE=new Object();
   public boolean isEmpty(){
      return (Q[debut]==VIDE);
   }
   ...
}
```

on ne veut pas utiliser `null` au lieu de `VIDE` parce qu'on veut permettre `enqueue(null)`...

Queue — cont.

```
public Object dequeue(){
   Object retval = Q[debut];
   if (retval==VIDE)
      throw new UnderflowException("Rien ici.");
   Q[debut]=VIDE;
   debut = (debut + 1) % MAX_TAILLE;
   return retval;
}
static class UnderflowException extends RuntimeException {
   private UnderflowException(String msg){
      super(msg);
   }
}
```

Queue — cont.

```
public void enqueue(Object O){
   if (Q[fin]!=VIDE)
      throw new OverflowException("Queue trop longue.");
   Q[fin]=O;
   fin = (fin+1) % MAX_TAILLE;
}
static class OverflowException extends RuntimeException {
   private OverflowException(String msg){
      super(msg);
   }
}
```

Queue — efficacité

Temps de calcul par opération : $O(1)$

Accès à un élément quelconque : n'est pas possible directement, il faut utiliser une autre liste pour défiler tous les éléments arrivés avant l'élément cherché

queue et pile : les éléments au milieu ne sont pas «visibles»

(si on voulait, on pourrait définir l'ADT avec une opération pour accéder au k-ème élément [k-ème par arrivée] ; les implantations présentées pourraient la supporter en $O(1)$ mais l'ADT de la pile ou la queue ne définissent pas une telle op)

queue : FIFO (*first-in first-out*) — premier entré, premier sorti
pile : LIFO (*last-in first-out*) — dernier entré, premier sorti

Liste

Objets : listes de style $A_1, A_2, \ldots, A_n$ et la liste nulle

(Il faut spécifier aussi le type des A_i — pour nos buts on permet n'importe quel type : $A_i \in O$)

Opérations :
insert: $L \times \mathbb{N}^+ \times O \mapsto L$; p.e., insert(L,2,x)
remove: $L \times \mathbb{N}^+ \mapsto L$; p.e., remove(L,2)
makeEmpty: $L \mapsto L$;
find: $L \times O \mapsto \{1, 2, \ldots, \}$; p.e., find(L,x)
Kth: $L \times \mathbb{N}^+ \mapsto O$; p.e., Kth(L,3)
...

$\rightarrow$ accès à tous les éléments !

En pratique, insert, remove et makeEmpty modifient leur argument au lieu de retourner une nouvelle liste

Liste — implantation 1

Utiliser `Object[]` pour stocker les éléments (variable `private` dans la classe)

Il faut copier tous les éléments à un nouveau tableau pour chaque insert et remove.

```
public class Liste1 {
   private Object[] elements;
   ...
   private void insert(int pos, Object O){
      Object[] newElements = new Object[elements.length+1];
      for (int i=0; i<pos-1; i++)
         newElements[i]=elements[i];
      elements[pos-1]=O;
      for (int i=pos-1; i<elements.length; i++)
         newElements[i+1]=elements[i];
      elements = newElements;
   }
   ...
}
```

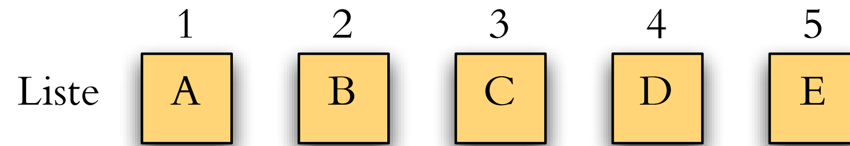
Liste — implantation 2

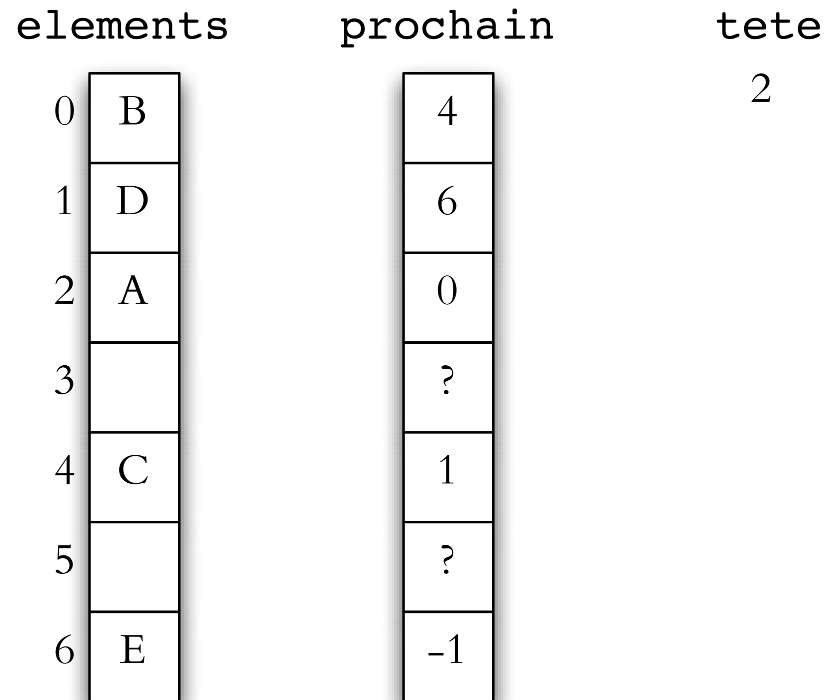
Est-ce que'on peut éviter l'allocation d'un nouveau tableau lors de chaque opération ?

Idée : détacher l'ordre dans le tableau `elements[]` de celui sur la liste
→ utiliser un autre tableau `int[] prochain` qui définit l'ordre des éléments dans la liste

```
public class Liste2 {
   private Object[] elements;
   private int[] prochain;
   private int tete;
   private static int MAX_TAILLE = 5555;
   public Liste2(){
      elements = new Object[MAX_TAILLE];
      prochain = new int[MAX_TAILLE];
      tete = -1; // liste vide
   }
   ...
}
```

Liste2 (cont.)

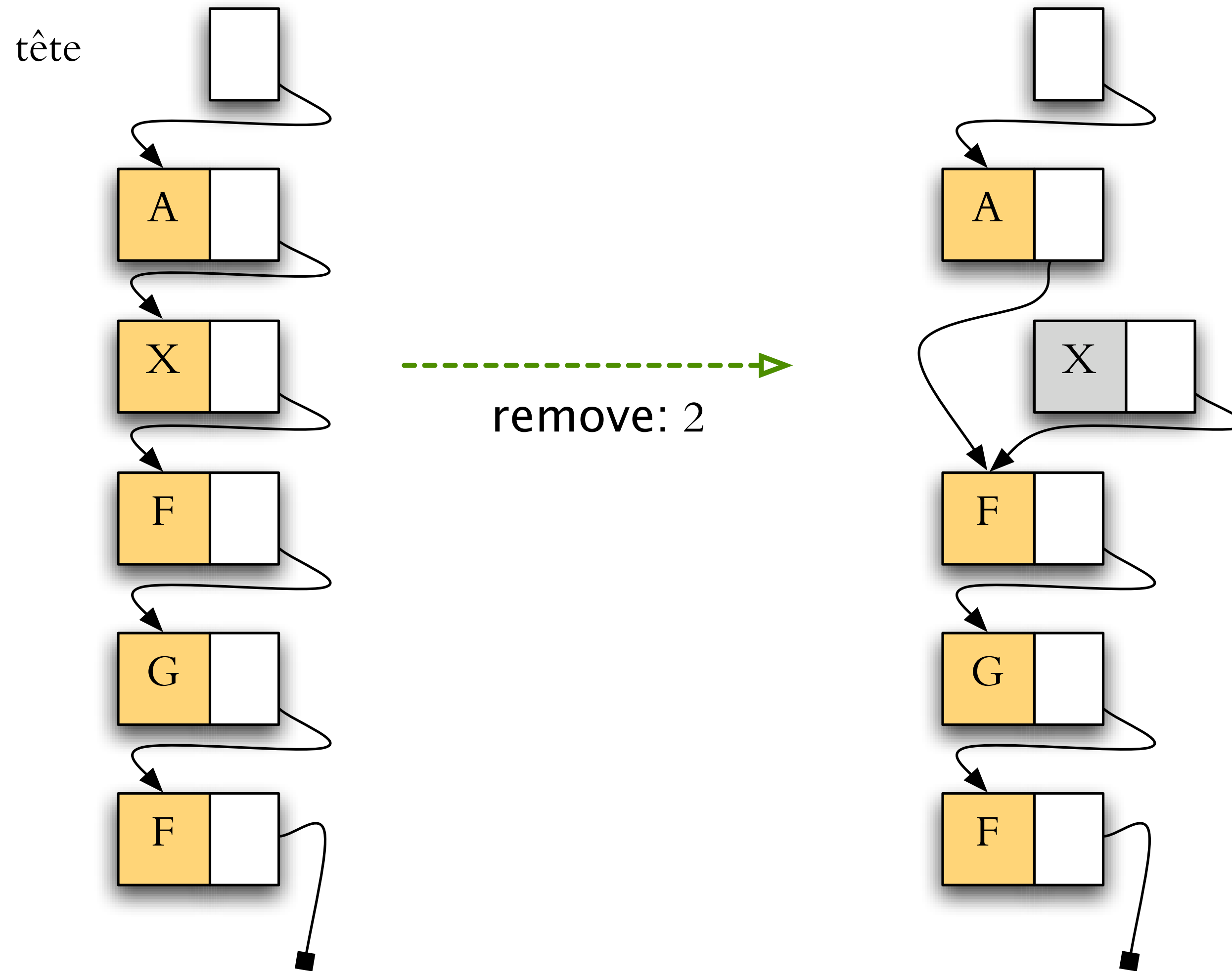




Liste2 — K-ème

```
public class Liste2 {
    ...
    public Object Kth(int pos){
        if (pos<=0)
            throw new IndexOutOfBoundsException("pos<=0");
        int emt_idx=indiceDe(pos);
        if (emt_idx != -1)
            return elements[emt_idx];
        else
            throw new IndexOutOfBoundsException("pos est trop grand");
    }
    private int indiceDe(int pos){
        int emt_idx=tete;
        int k=1;
        while (emt_idx != -1 && k<pos){
            emt_idx = prochain[emt_idx];
            k++;
        }
        return emt_idx;
    }
}
```

Liste — supprimer



Liste2 — supprimer

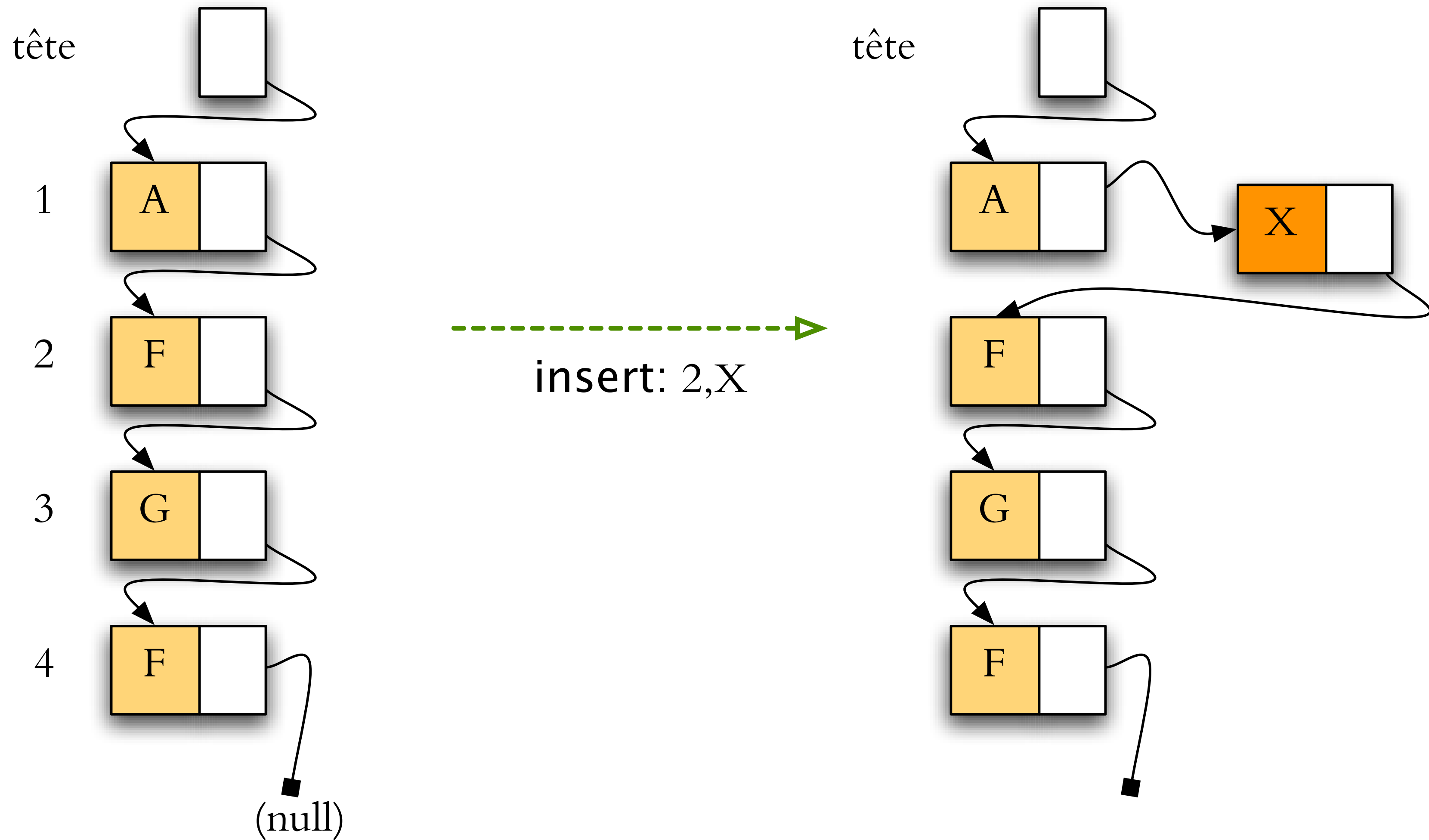
Deux cas :

1. `remove(1)` modifie `tete`
2. `remove(n)` avec `n` plus grand que 1 modifie le `prochain` de l'élément à $(n-1)$

Cas 1 : `tete=prochain[tete]`

Cas 2 : `prochain[precedent]=prochain[indice]`

Liste — insérer



Liste2 — cases vides

Problème : où placer le nouvel élément ?

Solution 1 : objet `VIDE` pour dénoter une case vide ⇒ $O(M)$ pour trouver une case libre dans un tableau de taille max. M

Solution 2 : maintenir la liste de cases libres
on peut utiliser `prochain[]` pour ceci avec une variable `tete_vide`

```
public class Liste2 {
   private int tete_vide;
   ...
   public Liste2(){
      elements = new Object[MAX_TAILLE];
      prochain = new int[MAX_TAILLE];
      tete = -1;
      tete_vide = 0;
      for (int i=0; i<MAX_TAILLE-1; i++)
         prochain[i]=i+1;
      prochain[MAX_TAILLE-1]=-1;
   }
   ...
}
```

Liste2 — cases vides

Ajouter une case vide (remove)

```
prochain[case_idx] = tete_vide;
tete_vide = case_idx;
```

Supprimer une case vide (insert)

```
// use elements[tete_vide] pour l'insertion
tete_vide = prochain[tete_vide];
```

(Notez que la liste de cases vides fonctionne comme une pile — une pile entrelacée avec une liste !)

Liste — implantation 3

On veut une liste de taille illimitée. . .

Chaque élément de la liste est stocké comme une paire de (valeur, prochain élément)

En Java :
(1) une classe `List.Element`
(2) classe `List` doit maintenir une variable pour la tête de la liste (de type `Element`)

```
public class Liste {
   public class Element {
      private Object valeur;
      private Element prochain;
      private Element(Object valeur) {
         this.valeur=valeur; this.prochain=null;
      }
      public Object getValue(){ return valeur;}
      public Element getNext(){ return prochain;}
      public void setNext(Element prochain){
         this.prochain = prochain;}
   }
   ...
}
```

Liste3 — K-ème

```
private Object tete;
public Liste(){
   tete = null;
}
public Object Kth(int pos){
    if (pos<=0)
        throw new IndexOutOfBoundsException("pos<=0");
    Element E = surLaListe(pos);
    if (E == null)
        throw new IndexOutOfBoundsException("pos trop grand");
    return E.getValue();
}
/** retourne l'élément dans cette position (sans vérifier pos) */
private Element surLaListe(int pos){
    int k=1;
    Element E = tete;
    while (E != null && k<pos){
        k++;
        E = E.getNext();
    }
    return E;
}
```

Liste3 — supprimer

```
public void remove(int pos){
    if (pos<=0 || tete==null)
        throw new IndexOutOfBoundsException();
    if (pos == 1){
        tete = tete.getNext();
    } else {
        Element E_prec = tete;
        int k = 2;
        while (E_prec.getNext() != null && k<pos){
           E_prec = E_prec.getNext();
           k++;
        }
        if (E_prec.getNext() == null)
            throw new IndexOutOfBoundsException();
        E_prec.setNext(E_prec.getNext().getNext());
    }

}
```

Liste3 — insérer

```
public void insert(int pos, Object O){
    Element E_inserer = new Element(O);
    if (pos<=0)
        throw new IndexOutOfBoundsException();
    if (pos == 1){
        E_inserer.setNext(tete);
        tete = E_inserer;
    } else {
        Element E_prec = tete;
        int k = 2;
        while (E_prec != null  && k<pos){
           E_prec = E_prec.getNext();
           k++;
        }
        if (E_prec == null)
            throw new IndexOutOfBoundsException();
        E_inserer.setNext(E_prec.getNext());
        E_prec.setNext(E_inserer);
    }
}
```

Sentinelles

sentinelle : élément «virtuel» sur la liste pour dénoter la tête

«virtuel» : n'est pas vu dehors de l'implantation

Donc on initialise avec `tete=new Element(null)`
et `insert` est implanté sans le test pour `pos=1`.

C'est la solution dans le livre

Avantage : code plus clair, exécution un peu plus rapide (mais pas en asymptotique)
Désavantage : espace pour un élément de plus
$\rightarrow n$ listes de longueur totale ℓ nécessitent un espace de $O(n+\ell)$ au lieu de $O(\ell)$
... problème si on a beaucoup de listes courtes

Liste — efficacité

Liste 2 (tableau)

Longueur de la liste : ℓ, taille max. M

makeEmpty : $O(M)$ [gestion de cases vides]

remove(i), insert(i, x) : $O(i) = O(\ell)$

Kth(i) : $O(i) = O(\ell)$

find(x) : $O(\ell)$

Liste 3 (pointeurs)

Longueur de la liste : ℓ

makeEmpty : $O(1)$

remove(i), insert(i, x) : $O(i) = O(\ell)$

Kth(i) : $O(i) = O(\ell)$

find(x) : $O(\ell)$

Pile par liste

On peut implanter une pile avec n'importe quelle implantation de liste :

```
private Liste L;
public Object pop(){
   Object retval = L.Kth(1);
   L.remove(1)
   return retval;
}
public void push(Object O){
   L.insert(1,O);
}
```

Les opérations $\mathsf{remove}(1)$, $\mathsf{Kth}(1)$ et $\mathsf{insert}(1, x)$ prennent $O(1)$ temps.

Variantes

1. liste circulaire : le dernier élément de la liste pointe vers le premier

2. Liste doublement chaînée : les éléments pointent vers tous les deux voisins (champs `prochain` et `precedent`)
avec ou sans sentinelle

Liste doublement chaînée

Avantage : on peut supprimer et insérer dans le milieu de la liste sans traverser tous les éléments pour y arriver

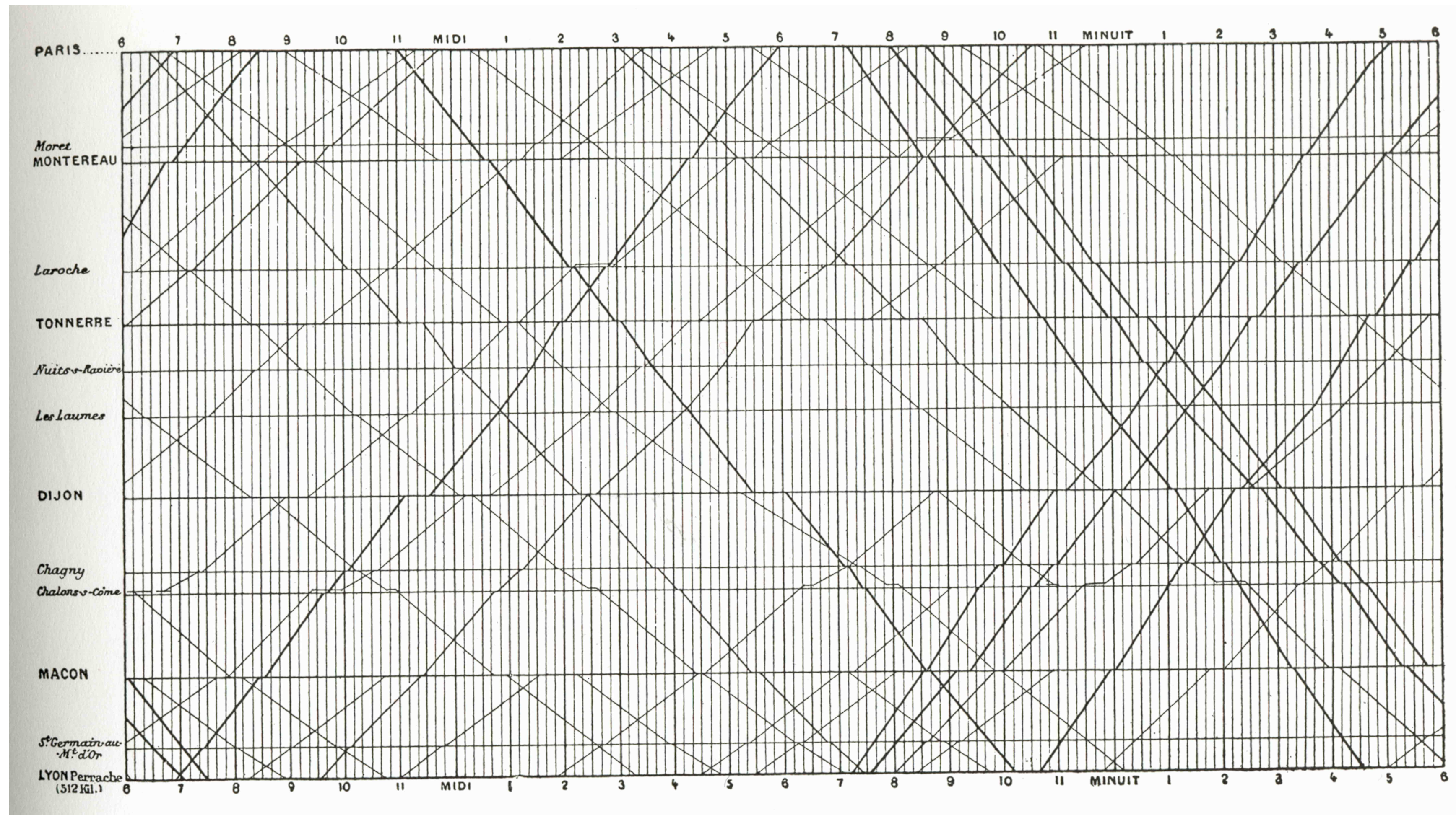
remove(i) et insert(i, x) : il faut changer tous les pointeurs affectés

Kth(i) : si on stocke aussi la taille de la liste, on peut rapidement trouver les éléments vers la fin aussi

Efficacité : en asymptotique comme liste simple, mais
- remove et insert prennent un peu plus de temps (plus de pointeurs)
- pire temps de Kth est à peu près la moitié de celui de la liste simple

Liste doublement chaînée — exemple

Exemple : horaire de trains — deux listes (stations et trains)



Tufte *The Visual Display of Quantitative Information*, 1983 après Marey *La méthode graphique*, 1878

Exemple (cont)

un objet pour chaque arrivée/départure d'un train à une station

4 champs : `prochainTrain`, `precedentTrain` (pour une liste de trains passant à une station),
`prochainstation`, `precedentStation` (pour une liste de stations où un train s'arrête)

opérations : supprimer/insérer un arrêt pour un train, ajouter/supprimer tout le train, insérer/supprimer une station, horaire par trains, etc.