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

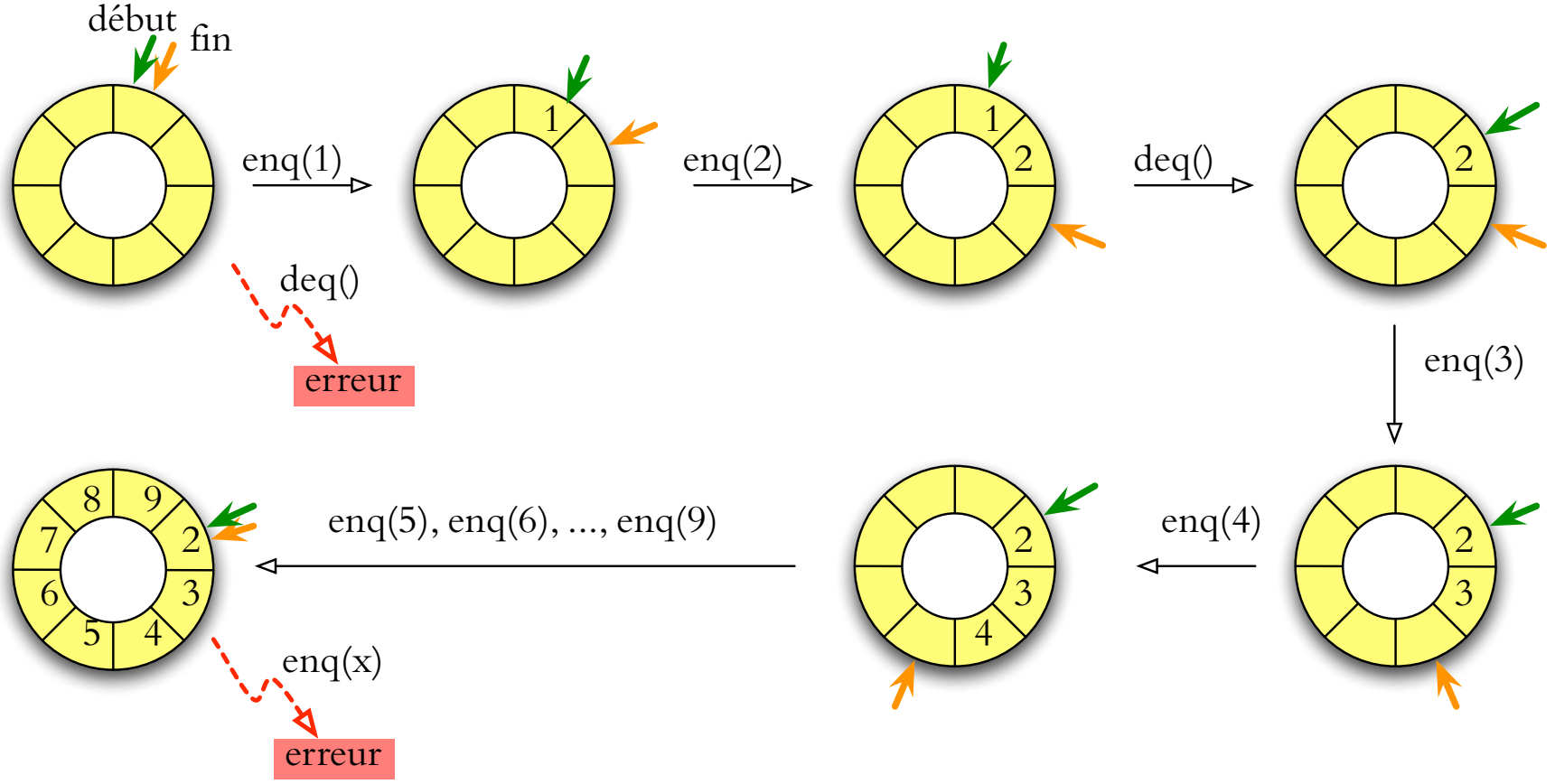
On peut exécuter enqueue en $O(1)$ si on maintient l'indice de la fin de la queue (comme le sommet de la pile)

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;
   }
   ...
}
```

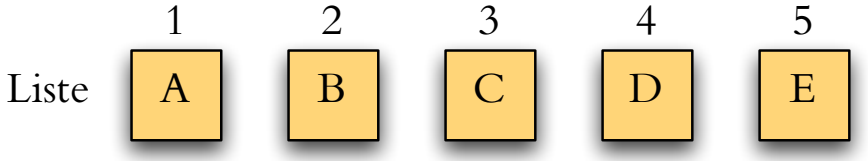
Liste — implantation 2

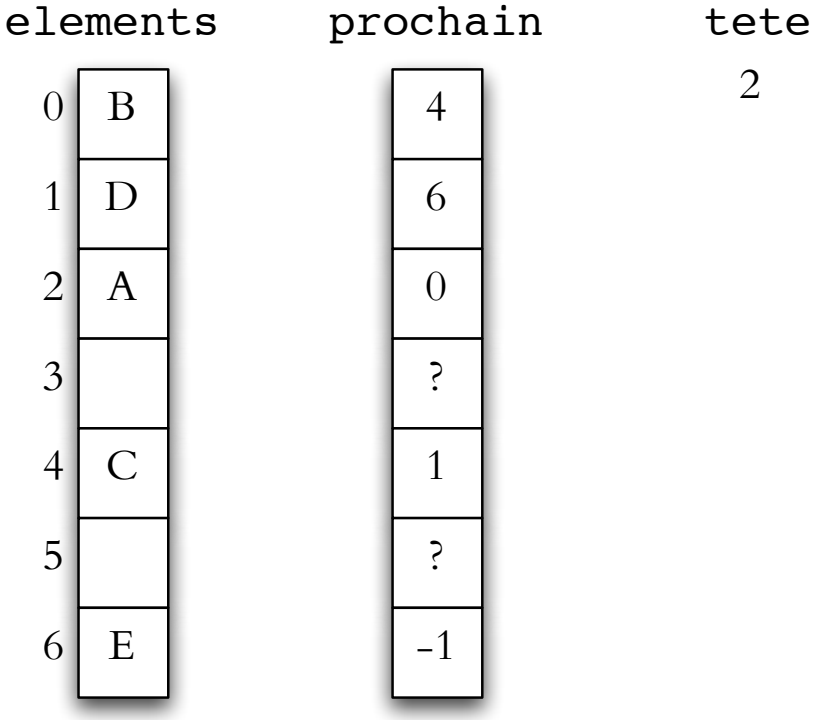
Est-ce que'on peut éviter l'allocation d'un nouveau tableau lors de chaque opération ?

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.)

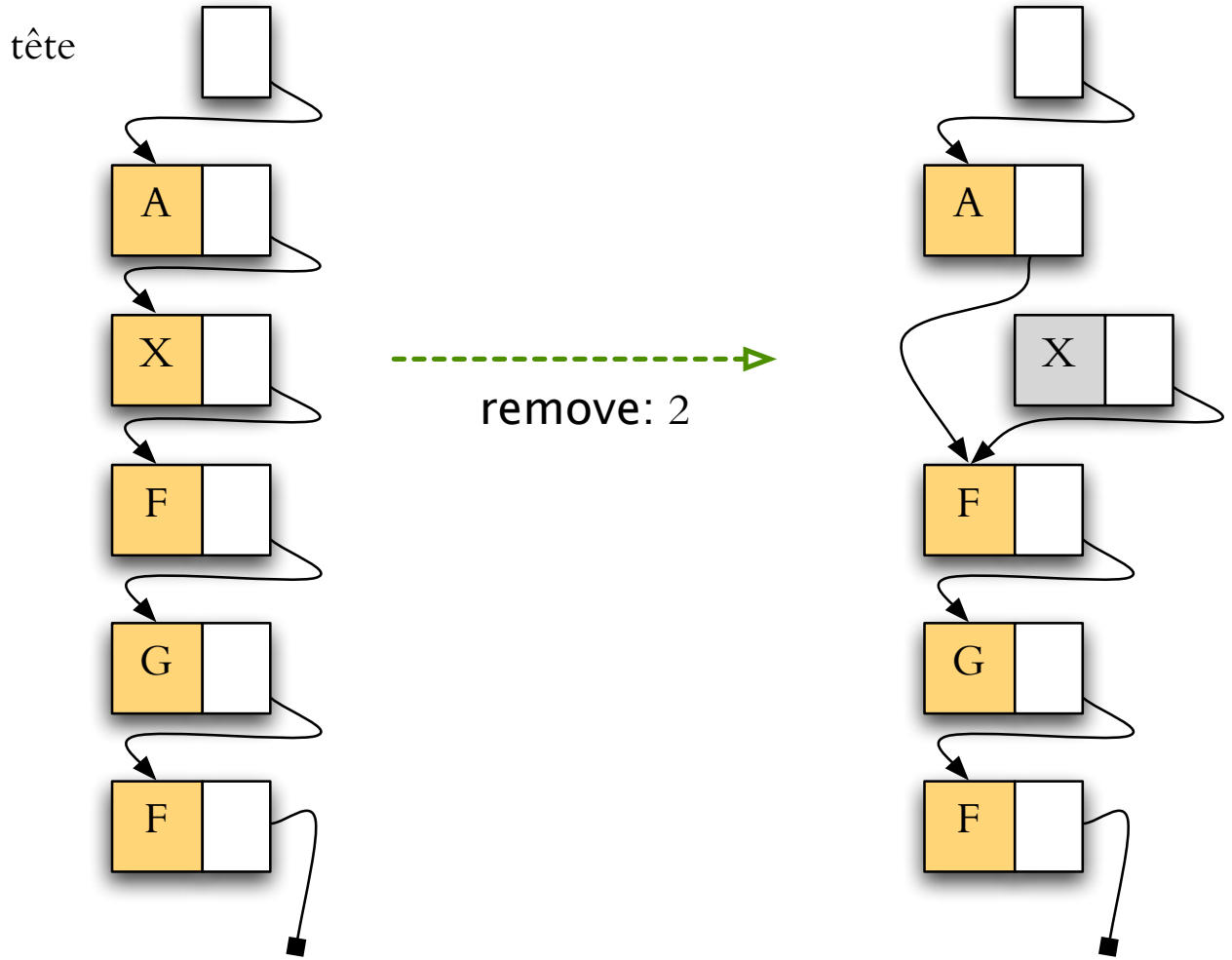

1
2
3
4
5
Liste
A
B
C
D
E


elements
prochain
tete
2
0
B
4
1
D
6
2
A
0
3
?
4
C
1
5
?
6
E
-1

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

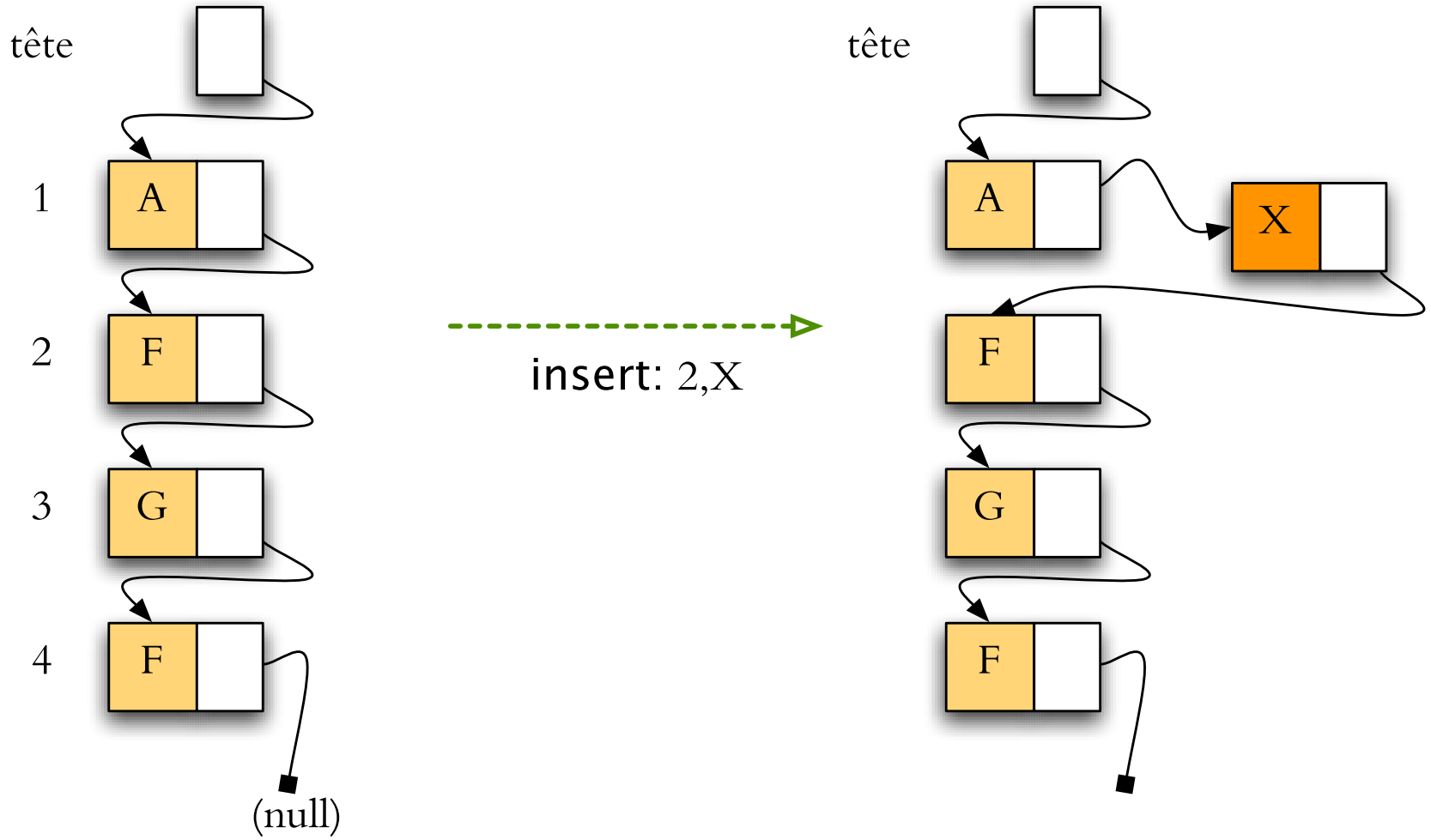
Deux cas :

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 $\Rightarrow O(M)$ 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);
    }
}
```