Types abstraits de données

Types

Déf. Un type est un ensemble (possiblement infini) de valeurs et d'opérations sur celles-ci.

Déf. Un type abstrait est un type accessible *uniquement* à travers une interface.

client : le programme qui utilise le TA

implémentation : le programmme qui spécifie le TA

interface : contrat entre client et l'implémentation

En Java
: interface et implémentation souvent dans le même fichier
: interface défini par la signature des méthodes et variables non-privées
: «clients» de droits différents (sous-classe, package) : `interface` n'est pas exactement l'interface de notre définition (ne définit pas le syntaxe des constructeurs)

Collection d'éléments

Files généralisée : collection d'éléments avec deux opérations :

1. insertion d'un nouvel élément

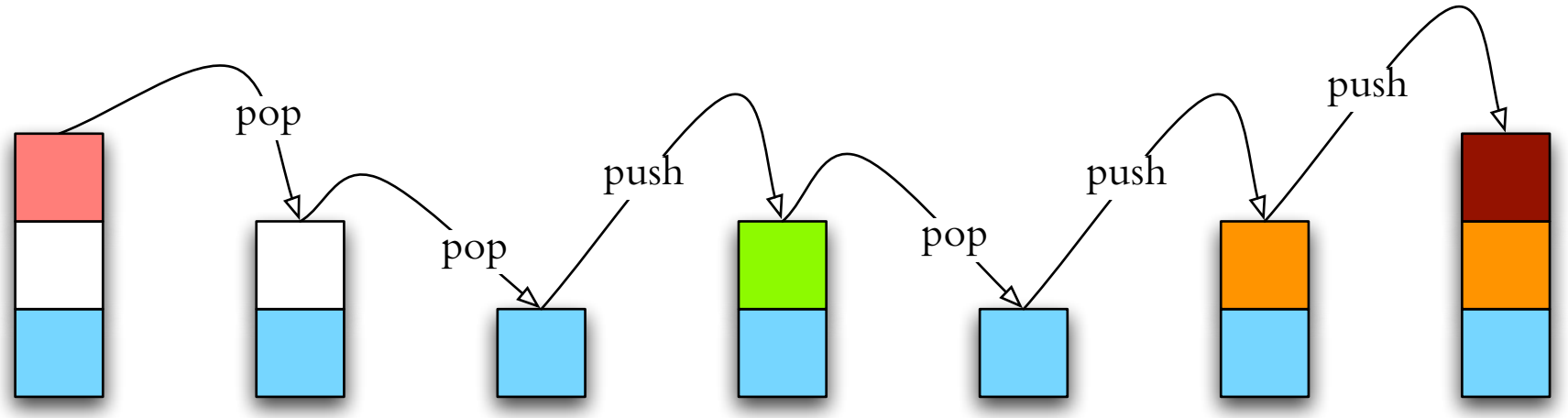
2. suppression d'un élément

il existe beaucoup de variantes selon la politique d'insertion et de suppression d'éléments, ainsi que l'interface pour le TA (comment spécifie-t-on l'«élément» ?)

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

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 ?
→ 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 --;
      Object retval = P[sommet]; P[sommet]=null;
      return retval;
   }
}
```

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

Pile — efficacité

Temps de calul par opération : $O(1)$

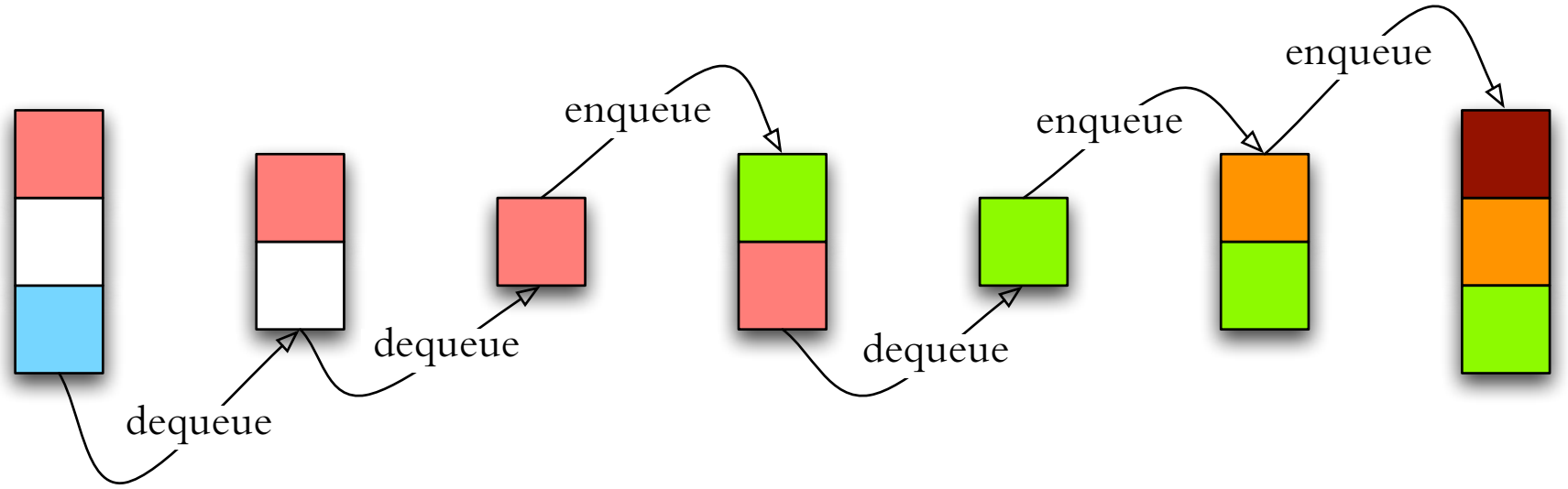
Accès à un élément quelconque : n'est pas possible directement, il faut utiliser une autre liste pour dépiler tous les éléments sur l'élément cherché ça prend $O(n)$ temps et une espace de travail $O(n)$ pour l'élément en position n

Queue — idée

(queue ou file FIFO, *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 :
$\mathsf{enqueue}: Q \times O \mapsto Q$; p.e., $\mathsf{enqueue}(Q, x)$
$\mathsf{dequeue}: Q \mapsto Q \times O$; p.e., $\mathsf{dequeue}(Q)$
$\mathsf{isEmpty}: Q \mapsto \{\mathsf{vrai}, \mathsf{faux}\}$

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

(Le livre de Sedgewick parle de put et get au lieu de enqueue et dequeue.)

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

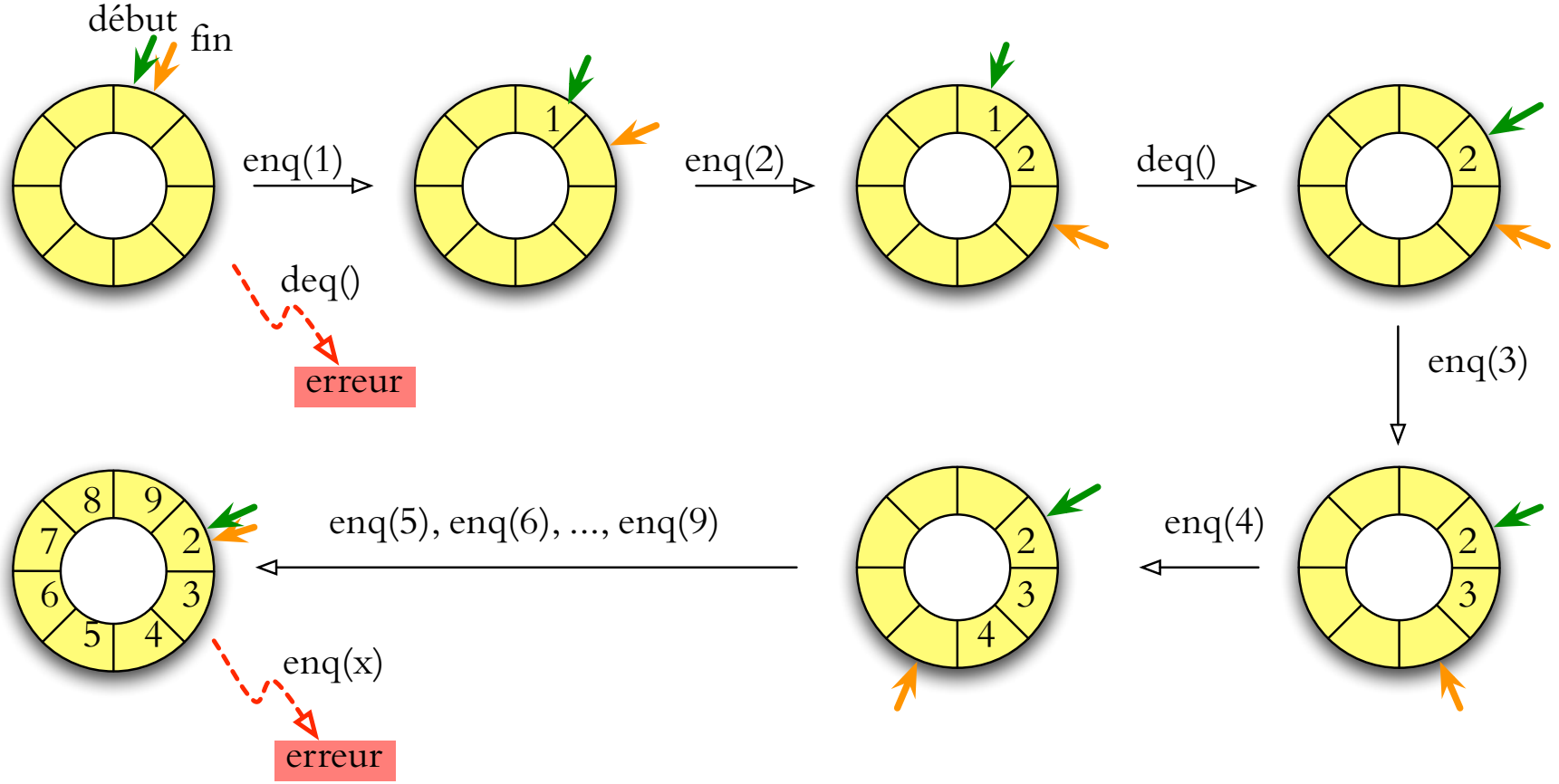
On peut exécuter enqueue en $O(1)$ si on maintient l'indice de la fin de la queue (comme le sommet de la pile)

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=2007;
   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 — cases vides

$k = (\texttt{debut} - \texttt{fin}) \bmod n$ peut prendre les valeurs $k = 0, 1, \ldots, n - 1$

Remarque : si on ne peut pas avoir un élément spécial pour dénoter les cases vides, alors on ne peut stocker que $(n - 1)$ éléments dans la queue avec un tableau de taille n

(c'est la version du livre)

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

Clients

Pile : notation polonaise postfixée, PostScript, analyse syntaxique de programmes

File : services dans systèmes d'exploitation, traitement d'événements dans simulation (ou jeus !)

Notation polonaise

Une opération arithmétique $a \,\mathtt{op}\, b$ est écrite en notation polonaise inverse ou notation «postfixée» par $a\, b\, \mathtt{op}$

Avantage : pas de parenthèses !

Exemples : $\mathtt{1 + 2} \rightarrow \mathtt{1\,2\,+}$, $\mathtt{(3 - 7) * 8} \rightarrow \mathtt{3\,7\,-\,8\,*}$

PostScript est un langage de programmation qui utilise une pile
Opérations en Postscript : `add`, `sub` `mul` `div`

P.e., la suite d'instructions `5 2 sub` place 5 et 2 sur la pile (dans cet ordre), et l'opérateur `sub` prend les deux éléments en haut de la pile pour les remplacer par le résultat de la soustraction. Dans ce cas-ci, la pile contiendra le seul élément 3 à la fin.