Tableaux

Tableaux

- taille fixe, allocation explicite : `int[] T = new int[12];`

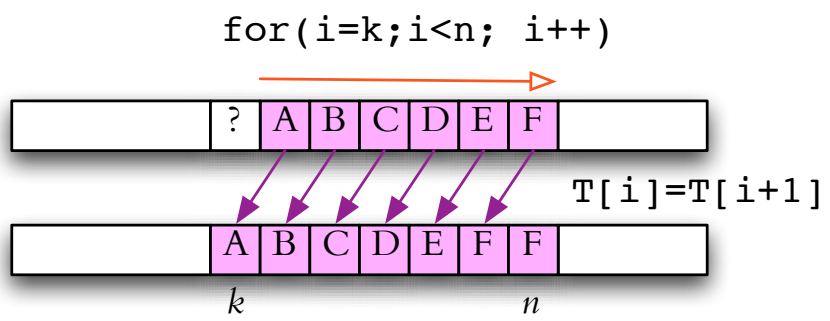
- acccès rapid au k-ème élément : `int z=2*T[k];`

- manipulation difficile : pour insérér ou supprimer un élément il faut décaler les éléments dans la reste du tableau

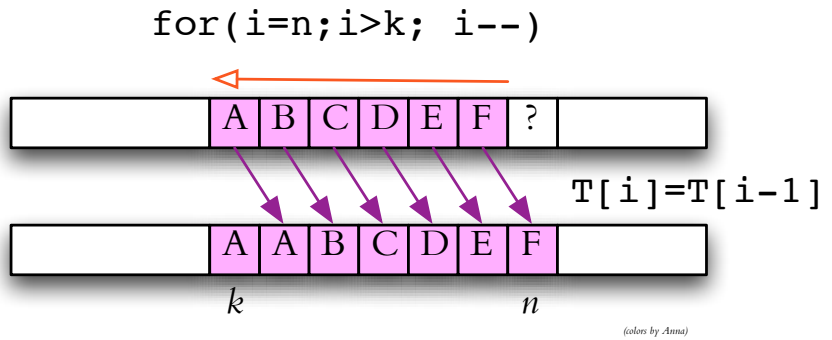
$\Rightarrow$ structure parfaite pour petite taille ou logique simple

Décalage

Vers la gauche



Vers la droite



INSERT$(T[0..n-1], i, x)$ // *(insertion de l'élément x en position i)*

1 **for** $j \leftarrow n-1, \ldots, i+1$ **do** $T[j] \leftarrow T[j-1]$ // *(attention à l'ordre !)*

2 $T[i] \leftarrow x$

DELETE$(T[0..n-1], i)$ // *(suppression de l'élément en position i)*

1 $x \leftarrow T[i]$

2 **for** $j \leftarrow i+1, i+2, \ldots, n-1$ **do** $T[j-1] \leftarrow T[j]$ // *(attention à l'ordre !)*

3 **return** x

Tri par insertion

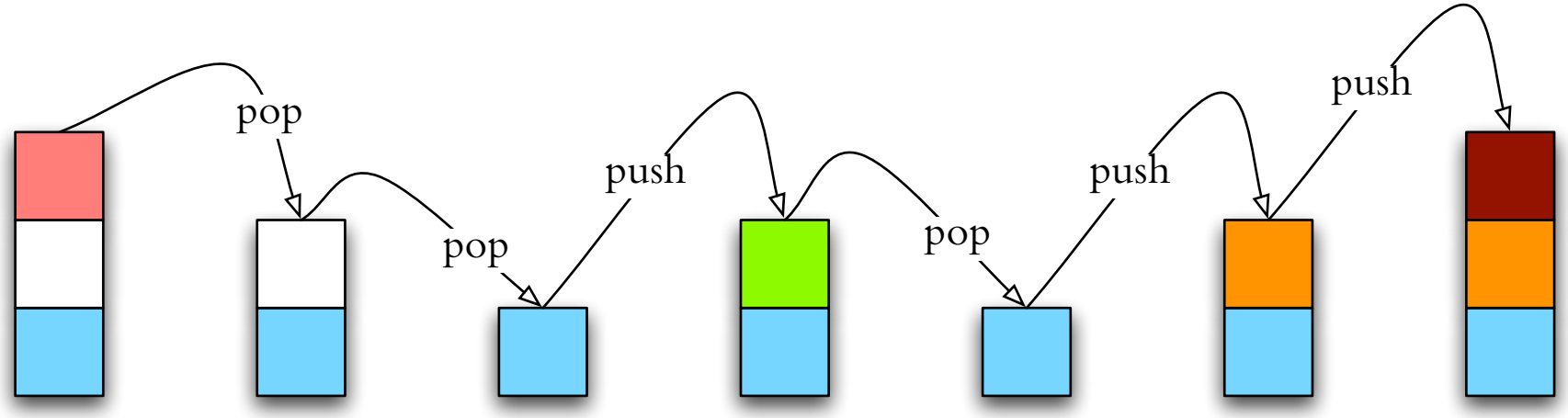
par l'insertion d'élément x dans le préfixe trié d'un tableau :

```
private void placer(double[] T, int n, double x)
{
    // T[0]<=T[1]<=...<=T[n-1]
    int i=n;                    // indice d'insertion
    while(i>0 && T[i-1]>x) {  // recherche + décalage
        T[i]=T[i-1];
        i--;
    }
    T[i]=x;
}

public void tri(int[] T){
    for (int n=0; n<T.length; n++)
        placer(T,n,T[n]);
}
```

Pile

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



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`

→ avertir le client (dans la documentation) que la capacité est fixée

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 informative

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

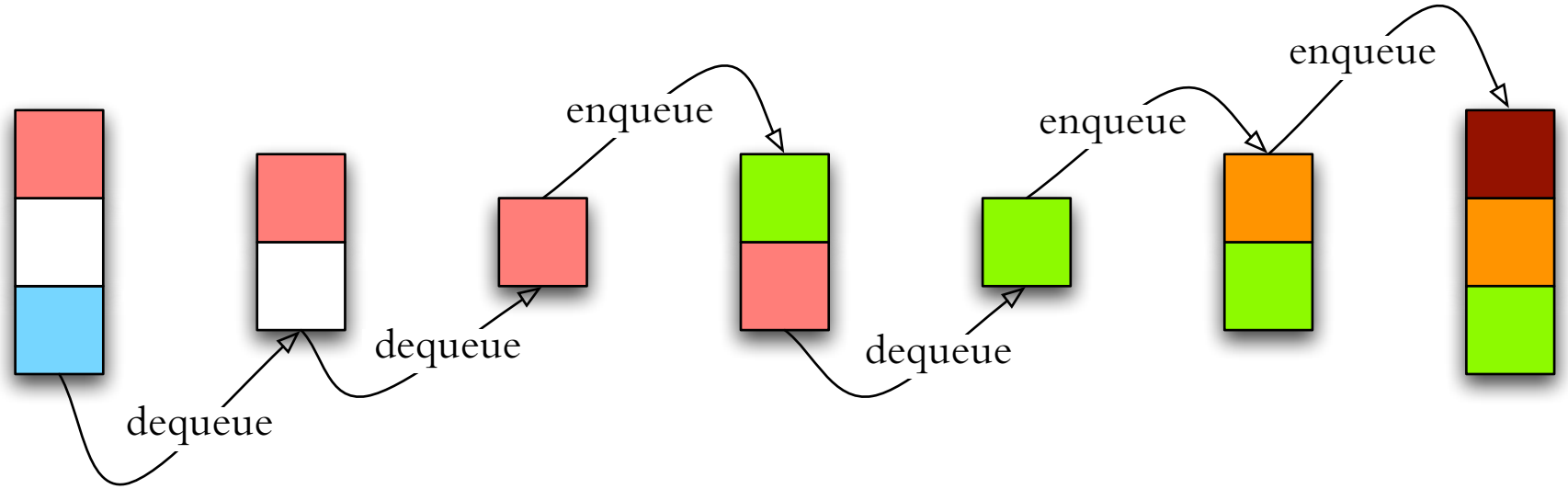
Temps de calul par opération : constant !

Accès à un élément quelconque : n'est pas possible directement.

(on peut utiliser une autre liste pour dépiler tous les éléments sur l'élément cherché, mais cela prend du temps et de l'espace linéaire dans la position n de l'élément)

Queue

opérations : «enfiler» (enqueue) et «défiler» (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 : temps linéaire dans la longueur

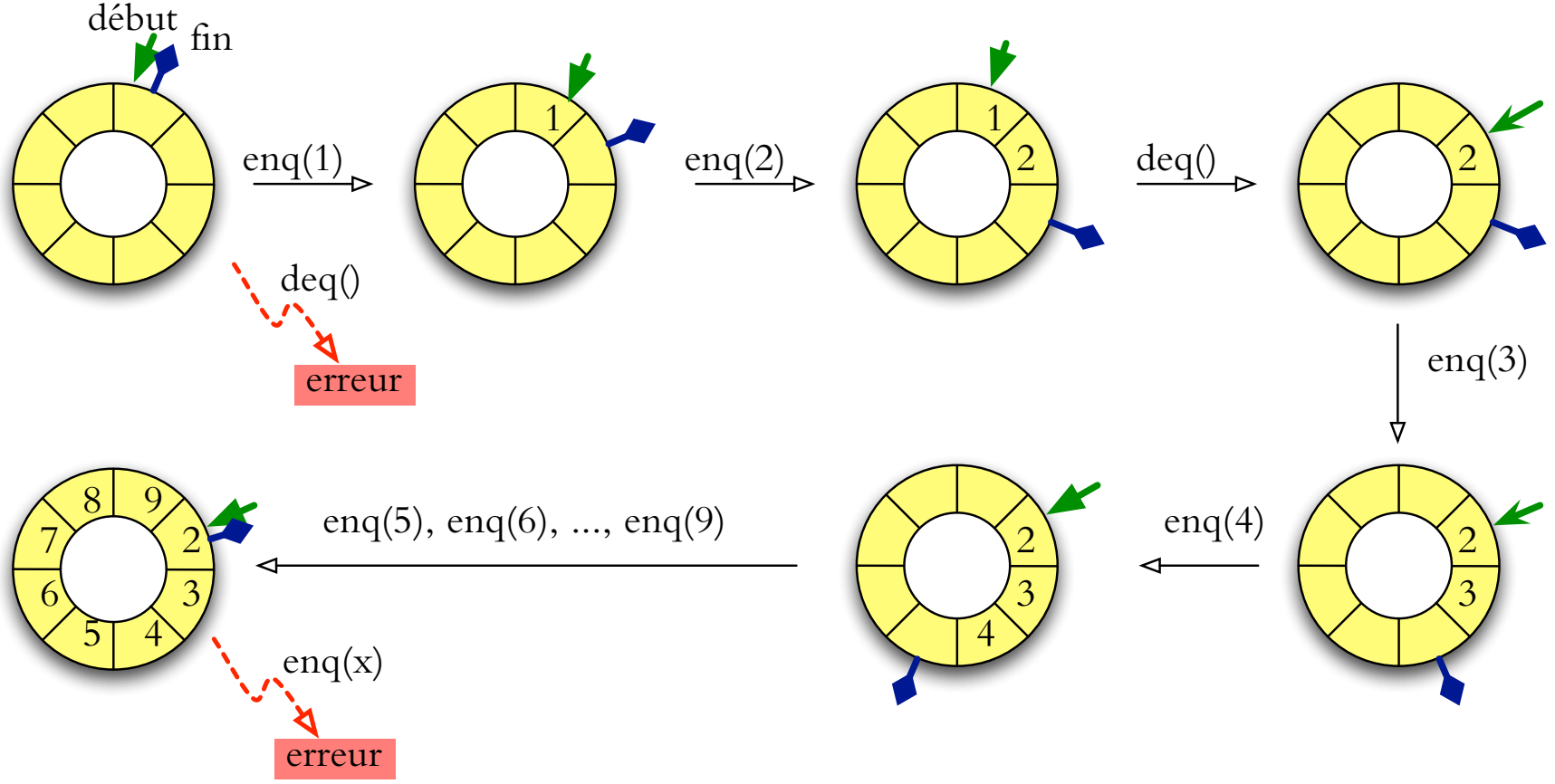
On peut exécuter enqueue en un temps constant si on maintient l'indice de la fin de la queue (comme le sommet de la pile)

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=2017;
   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(); // sentinelle
   public boolean isEmpty(){
      return (Q[debut]==VIDE);
   }
   ...
}
```

on ne veut pas utiliser `null` au lieu de `VIDE` car 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

Queue — efficacité

Temps de calcul par opération : constant

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»

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

Bag

```
public class Bag implements Iterable
{
    private Object[] elements;
    private int taille; // sac  à 0..taille-1
    private static final int CAPACITE_DEFAUT = 1;

    public Bag(){
        this(CAPACITE_DEFAUT);
    }
    public Bag(int capacite){
        elements = new Object[capacite];
        taille=0;
    }
```

Bag – expansion

expansion du tableau au besoin

```
public void add(Object o){
    if (taille == elements.length)
        reallocation(2*taille); // doubler
    elements[taille++] = o;
}
private void reallocation(int capacite){
    Object[] T = new Object[capacite]; // nouveau tableau
    for (int i=0; i<taille; ++i) T[i]=elements[i];
    elements = T; // remplacer l'ancien tableau
}
```

Itérateur

on doit implémenter la méthode iterator() de l'interface

```
    /**
     * @Override
     */
    public Iterator iterator(){
        class Iter implements Iterator { // classe locale
            private int pos=0;
            public boolean hasNext(){ return pos<taille;}
            public Object next(){ return elements[pos++];}
        }
        return new Iter();
    }
} // end of class
```

cela permet `for`-each

```
Bag B = new Bag();
...
for (Object o: B) { ... }
```

```
Iterator I = B.iterator();
while (I.hasNext()){
    Object o = I.next(); ...}
```

Gestion dynamique de la capacité

Opération pop

1 top $\leftarrow$ top $-$ 1 ; $x \leftarrow$ elements[top] ; elements[top] $\leftarrow$ null

2 **if** top $<$ capacity/4 **then** REALLOC($\lceil$capacity/2$\rceil$)

3 **return** x

Opération push(x)

1 **if** top $=$ capacity **then** REALLOC(2 $\cdot$ capacity)

2 elements[top] $\leftarrow x$; top $\leftarrow$ top $+$ 1

REALLOC(n)

R1 $T[0..n-1] \leftarrow$ nouveau tableau de taille n

R2 **for** $i \leftarrow 0, \ldots,$ top $-$ 1 **do** $a[i] \leftarrow$ elements[i]

R3 elements $\leftarrow T$; size $\leftarrow n$