3. *Tableaux*

UN TABLEAU contient n $(= 0, 1, 2, \ldots)$ variables de même type : on a accès aux variables groupées par les indices $i = 0, \ldots, n-1$. L'arrangement séquentiel se traduit en une structure économique pour stocker les éléments d'une collection. Pour des TAs de logique simple (sac, pile, queue), et pour n'importe quelle collection de petite taille (disons, $n < 20..100$), la meilleure implantation est souvent basé sur un seul tableau.

Notation. Tableau $x[0..n-1]$ de longueur n contient les variables $x[0]$, $x[1]$, $\ldots$, $x[n-1]$.

Tableaux en Java. En Java, les tableaux sont des objets, créés dynamiquement par instanciation explicite. La **longueur** (ou capacité) doit être de type `int` et non-négative. Elle est stockée dans la variable publique `.length`, et ne change jamais après la création du tableau. La longueur ne fait pas partie du type. L'**allocation** de tableau se fait par **new** $\mathsf{E}[n]$ qui crée un tableau de longueur n dont les composants sont de type E. On peut **initialiser** les variables du tableau en mettant les valeurs en accolades $\{\cdots\}$.

```
int[] t1 = new int[10];          // tableau d'entiers
double[] t2 = {0.1, 0.2, 0.3};   // tableau de flottants, initialisation
String S[][] = new String[10][]; // tableau 2D: tableau de String[]
int[] t3, t4[];                  // int[] t3 et int[][] t4
t4 = new int[20][30];            // initialisation d'un tableau 2D
double d = t2[1]/3.0;            // accès à élément 1
int tlen = t1.length;            // accès à la taille
```

FIG. 1: Usage de tableaux en Java.

Manipulation de tableaux

Parcours. Boucle sur les éléments dans leur ordre :

```
double[] t;
// ...
double max = Double.NEGATIVE_INFINITY;
for (double x: t) if (x>max) max=x;
    // maintenant max est la valeur maximale dans t[]
double min = Double.POSITIVE_INFINITY;
for (int i=0; i<t.length; i++) if (t[i]<min) min=t[i];
    // maintenant min est la valeur minimale dans t[]
```

Décalage. Une technique fondamentale est de **décaler** les éléments vers la gauche ou la droite (Fig. 2). Pour insérer ou supprimer un élément dans un tableau, il faut décaler les autres à côté.

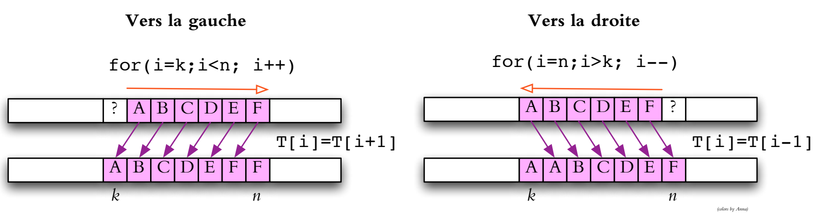

FIG. 2: Décalage d'éléments vers la gauche ou la droite. Attention au sens du parcours.

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

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

$T[i] \leftarrow x$

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

$x \leftarrow T[i]$

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

return x

Tri par insertion. Le code suivant insère l'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) {T[i]=T[i-1]; i--;} // recherche + décalage
    T[i]=x;
}
```

On peut trier le tableau entier en appellant `placer` en chaque position.

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

Cette méthode de tri s'appelle **tri par insertion**. Il existe de méthodes de tri plus rapides dans le cas général, mais quand $T[]$ est «presque» trié au début, on doit décaler juste un petit nombre d'éléments lors de chaque appel de `placer`, et le calcul ainsi finit dans un temps linéaire.

3.1 *Sac par tableau.*

Implémenter le TA sac n'est pas teop difficile. Pour accommoder la taille variable, on peut se servir d'un tableau «généreusement» alloué dont les premières cases donnent les vrais éléments (Fig. 3). Si le nombre des éléments atteint la capacité du tableau sous-jacent, on le remplace par un tableau élargi : (1) on crée un nouveau tableau de taille maximale plus grande, (2) on copie tous les éléments dans le nouveau tableau, et (3) on remplace le tableau original par le nouveau.

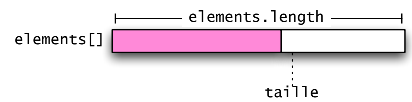

FIG. 3: Implémentation d'un sac — code minimaliste

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

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

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

L'exécution de l'opération `add` peut prendre un temps linéaire avec cette solution : lors du n-ème appel, si $n = 2^k + 1$ avec $k = 0, 1, \ldots$, on double la capacité du tableau à 2^{k+1} en copiant $(n-1)$ éléments.

Théorème 3.1. *Soit $A(n)$ le nombre total des affectations de cellules en une série de n appels à* `add`*. Pour tout $n = 1, 2, \ldots$,*

$$\underbrace{2n-1}_{\text{égalité à } n=2^k} \leq A(n) \leq \underbrace{3n-3}_{\text{égalité à } n=2^k+1} \tag{3.1}$$

Classe ArrayList

Java inclut la classe `java.util.ArrayList`[1] qui utilise les techniques décrites ci-dessus pour implanter la fonctionnalité complète d'une liste dynamique. La classe est **paramétrique** : on utilise le syntaxe `ArrayList`$\langle$`E`$\rangle$ pour spécifier le type `E` des éléments. `E` ne peut pas être un type primitif, mais tout type agrégé est permis. Le compilateur émet un message d'alarme quand le type des éléments n'est pas spécifié, mais le code est toujours exécutable (avec interprétation `ArrayList<Object>`).

[1] API doc :`java.util.ArrayList`
source code : `java.util.ArrayList`

```
java.util.ArrayList L0 = new java.util.ArrayList();
    // liste de type non-spécifié (Object)
java.util.ArrayList<String> L
    = new java.util.ArrayList<>();        // une liste de Strings

L.add("IFT1025");               // ajout d'un String à la fin
L.add("un autre");
String s = L.get(1);            // get retourne String ici
int taille = L.size();          // taille de la liste
L.add(new Object());            // erreur de compilation: type String enforcé
for (String s: L){ System.out.println(s);}      // boucle sur les éléments
```

3.2 Pile par tableau

On peut implanter une pile par un tableau $\mathsf{elements}[0\ldots n-1]$: on doit maintenir l'indice du sommet séparamment (Fig. 4). Sans gestion de taille, la pile déborde (*overflow*) quand on dépasse l'allocation initiale.

Initialisation(n) // *(capacité n)*
$\mathsf{elements}[0..n-1] \leftarrow$ tableau de taille n ; $\mathsf{capacity} \leftarrow n$
$\mathsf{top} \leftarrow 0$ // *(sommet de la pile)*

Opération $\mathsf{push}(x)$
$\mathsf{elements}[\mathsf{top}] \leftarrow x$
$\mathsf{top} \leftarrow \mathsf{top}+1$

Opération $\mathsf{pop}()$
$\mathsf{top} \leftarrow \mathsf{top}-1$; $x \leftarrow \mathsf{elements}[\mathsf{top}]$ // *(débordement négatif si $\mathsf{top}=0$!)*
$\mathsf{elements}[\mathsf{top}] \leftarrow \mathsf{null}$ // *destruction explicite de référence pour ramasse-miette*
retourner x

FIG. 4: Implémentation de pile basée sur un tableaude capacité fixe.

Gestion dynamique de la capacité

On utilise la technique suivante : si le nombre d'éléments sur la pile atteint la capacité allouée, on fait une réallocation en redoublant la longueur. Si le nombre d'éléments tombe en-dessous de $1/4$ de la capacité, on réduite la capacité à moitié.

Dans le pire cas, une opération prend maintenant un temps linéaire en top

Opération pop
top $\leftarrow$ top -1 ; $x \leftarrow$ elements[top]
if top $<$ capacity$/4$
then REALLOC($\lceil$capacity$/2\rceil$)
return x

Opération push(x)
if top $=$ capacity
then REALLOC($2 \cdot$ capacity)
elements[top] $\leftarrow x$; top $\leftarrow$ top $+1$

```
public Object pop()
{
   --top; Object x = elements[top]; elements[top]=null;
   if (4*top<elements.length)
      { realloc((elements.length+1)/2); }
   return x;
}
public void push(Object x)
{
   if (top==elements.length)
      { realloc(2*elements.length); }
   elements[top++]=x;
}
```

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 $\leftarrow n$

```
private void realloc(int n)
{
  Object[] T = new Object[n];
  System.arraycopy(elements,0,T,0,top); // copiage rapide
  elements = T;
}
```

FIG. 5: Implémentation de pile avec réallocation au besoin.

qui est le nombre d'éléments à copier en ligne 5. Mais cela n'arrive pas trop fréquemment : on a un temps *amorti* constant.

Théorème 3.2. *Une séquence de m opérations de* push *et* pop *prend un temps linéaire en m.*

Démonstration. On démontre le théorème par la méthode de **débit-crédit** :[2] on impose qu'on doit payer pour le temps CPU, et on montre qu'une séquence de m opérations push et pop coûte un montant linéaire en m. Pour cela, on se sert d'un *compte d'épargne*. Supposons que l'exécution d'une itération de la boucle de la ligne 5 coûte r. Chaque fois qu'on exécute push, on dépose $2r$, et lors de chaque pop on dépose r. Ainsi, on aura assez d'argent «épargné» entre deux exécutions de REALLOC pour copier les éléments. On voit que le compte d'épargne n'est jamais en négatif, et on doit dépenser tout au plus $(2r + c)m$ si c est le coût d'une opération sans réallocation. ■

[2] W(en):Accounting method (credit-debit)

3.3 Queue par tableau

L'idée principale pour une queue FIFO est de circulariser (virtuellement) le tableau par deux indices qui dénotent le début et la fin de la queue. La circularisation veut dire qu'on avance les indices par $j \leftarrow j \bmod n$: après $j = n-1$ on retombe à $j = 0$.

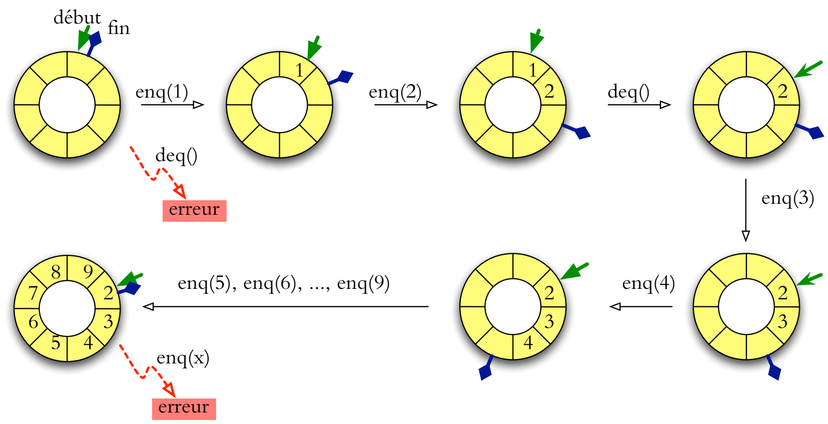

FIG. 6: Opérations dans un tableau cirularisé

```
public class Queue
{
   private int debut;
   private int fin;
   private Object[] Q;
   private static final int MAX_SIZE=2015;
   private static final Object EMPTY=new Object();
   public Queue()
   {
      debut=fin=0;
      Q=new Object[MAX_SIZE];
      for (int i=0; i<MAX_SIZE; i++)
         Q[i] = EMPTY;
   }
   public boolean isEmpty()
   {
      return (Q[debut]==EMPTY);
   }
```

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

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

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