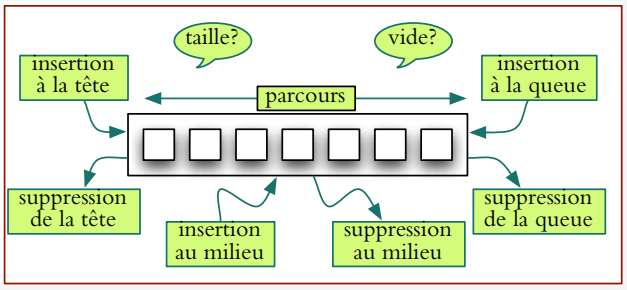
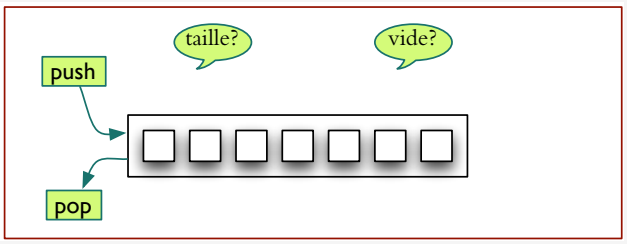
# 2 Listes et tableaux

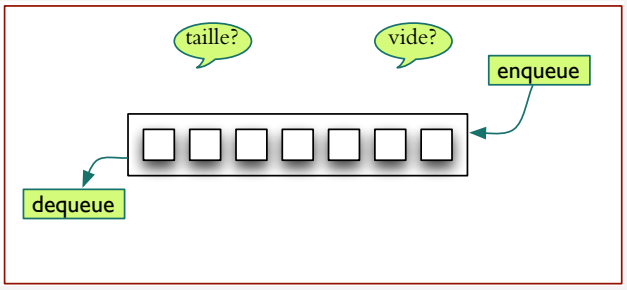
## 2.1 Liste (TA)




Une **liste** (comme type) est un ensemble d'éléments dans un arrangement séquentiel. Opérations typiques incluent l'insertion et suppression aux deux extremités ou au milieu, ainsi que le parcours, est des quêtes de taille.




Une **pile** (*stack*) permet l'accès à la tête mais pas ailleurs. Les opérations de base sont push («empiler») et pop («dépiler»). W(fr)




Une **queue** ou file FIFO permet l'insertion à la queue et la suppression à la tête. Les opérations de base sont enqueue («enfiler») et dequeue («défiler»).

W(fr)

## 2.2 Pile par tableau

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

**Initialisation**$(N)$
1 elements $\leftarrow$ tableau de taille $N$ ; taille $\leftarrow N$
2 sommet $\leftarrow 0$

**Opération** push$(x)$
1 elements[sommet] $\leftarrow x$
2 sommet $\leftarrow$ sommet $+ 1$

**Opération** pop
1 sommet $\leftarrow$ sommet $- 1$ ; $x \leftarrow$ elements[sommet] // *(débordement négatif avec pile vide !)*
2 elements[sommet] $\leftarrow$ null // *destruction explicite de référence pour ramasse-miette*
3 **retourner** $x$

## 2.3 Gestion dynamique de la taille d'un tableau

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

```
Opération pop
1 sommet ← sommet − 1 ; x ← elements[sommet] ; elements[sommet] ← null
2 si sommet < taille/4 alors faire REALLOCATION(⌈taille/2⌉)
3 retourner x

Opération push(x)
1 si sommet = taille alors faire REALLOCATION(2 · taille)
2 elements[sommet] ← x ; sommet ← sommet + 1
```

Pour la réallocation, on doit créer un tableau de la taille donnée, et copier les éléments.

```
REALLOCATION(t)
R1 a[] ← nouveau tableau de taille t
R2 pour i ← 0, . . . , sommet − 1 faire a[i] ← elements[i]
R3 elements ← a ; taille ← t
```

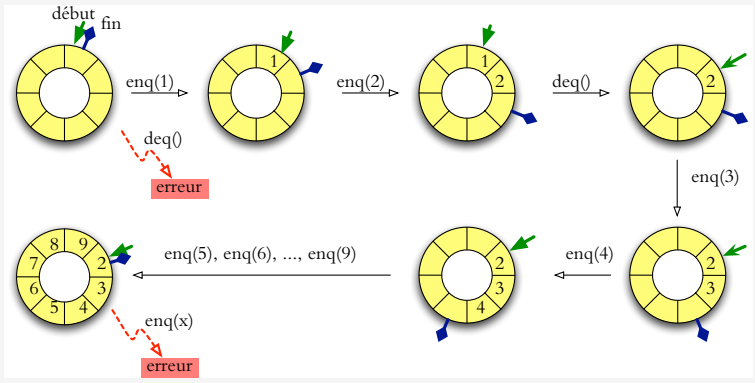
Dans le pire cas, une opération de pop ou push prend maintenant un temps linéaire en sommet qui est le nombre d'éléments à copier en ligne R2. Mais cela n'arrive pas trop fréquemment : on a un temps *amorti* constant.

**Théorème 2.1.** *Une séquence de $m$ opérations de* push *et* pop *prend un temps linéaire en $m$.*

*Démonstration.* Le temps d'exécuter une telle séquence est borné par $c \cdot (m + r) + b$ où $r$ est le nombre de fois on exécute la ligne R2 et $c, b$ sont des constantes quelconques caractérisant l'éxécution du code sans gestion de taille. On utilise un système de débits-crédits dans la preuve pour démontrer que $2m \geq r$. Lors d'un push, on met \$2 sur un compte. Lors d'un pop, on y met \$1. On utilise l'argent pour payer la réallocation : on paie \$1 pour copier un élément. À chaque réallocation, on remet la solde à \$0 même s'il reste de l'argent après le copiage. On peut voir que la solde n'est jamais négative. En conséquence, $c(m + r) + b \leq c(m + 2m) + b = 3cm + b$ temps suffit pour exécuter toute la séquence d'opérations. ■

W(en)

## 2.4 Queue par tableau




Idée : utiliser un tableau circulaire («anneau») avec deux indices pour le début et fin de la queue. Anneau en pratique : on utilise $\pmod n$ avec un tableau de taille $n$.

```
public class Queue
{
   private int debut;
   private int fin;
   private Object[] Q;
   private static final int MAX_TAILLE=2010;
   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)`...

```
   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);}}
```

```
   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 $k = 0, 1, \ldots, n-1$ Si on ne fait rien spécial (valeur spéciale pour cases vides, ou stocker la taille du tableau), alors on peut stocker tout au plus $(n-1)$ éléments avec un tableau de taille $n$.

## 2.5 Pile et queue par liste chaînée

On peut également implanter la pile et la file FIFO en utilisant une liste chaînée. Les opérations de la pile s'implantent à l'aide d'insertion et suppression à la tête. Pour une implantation efficace de la file FIFO, on maintient un pointeur à la queue, à l'addition de la tête. Le pointeur à la tête permet l'insertion à la queue pour enfiler et suppression à la tête pour défiler, en un temps constant.

**Exercice 2.1.** *Montrer le code pour une liste simplement chaînée qui permet l'insertion à la queue et à la tête, ainsi que la suppression à la tête : toutes en un temps constant.*

W(en)

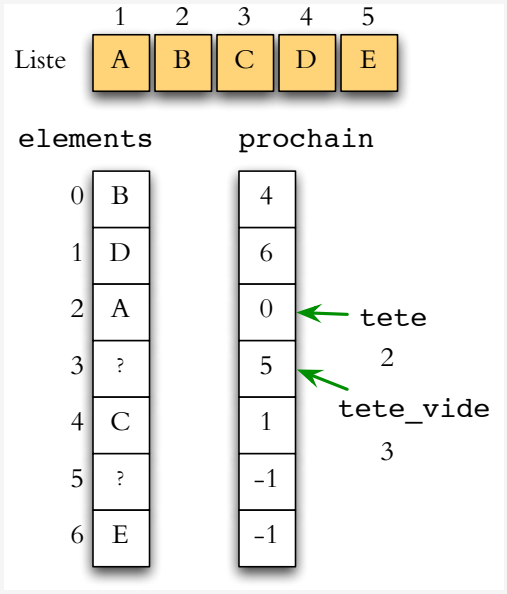
**Dèque.** Pour permettre la suppression à la tête et à la queue en même temps, on a besoin d'une liste doublement chaînée. Une telle structure s'appelle **dèque** (*double-ended queue*) : elle permet l'insertion et la suppression aux deux extremités. On peut renverser une liste doublement chaînée en un temps constant : on utilise un bit définissant l'intérpretation des deux pointeurs d'un nœud comme précédent-suivant.

## 2.6 Structure exogène pour liste chaînée

Normalement, l'insertion ou suppression dans le milieu d'un tableau nécessite le décalage des éléments. Mais on peut aussi créer une structure par deux tableaux pour implanter une liste chaînée : le champs prochain d'un nœud est stocké comme indice dans un tableau elements.

Dans la solution de la liste chaînée (Notes de cours 01), l'information primaire (`info`) est stockée avec l'information sur la *structure*, utilisée par l'implantation seulement. Il s'agit d'une structure **endogène** de données. On peut aussi stocker la structure séparamment des éléments. Il s'agit alors d'une structure **exogène** de données.

Avantage pour endogène : moins d'espace ; avantage pour exogène : même élément peut participer dans plusieurs structures.




```
public class Liste2
{
   private static int MAX_TAILLE = 5555;
   private Object[] elements;
   private int[] prochain;
   private int tete;
   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;
   }
   ...
}
```

**Gestion de cases vides.** On maintient alors une liste chaînée de cases vides (avec tête `tete_vide`), entrelacée avec la liste chaînée de vrais éléments. En fait, il s'agit d'une *pile* de cases vides : on insère et enlève toujours à la tête.

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];
```