3 Files et tableaux

3.1 Tableaux

Au lieu d'une liste chaîné, on peut utiliser un tableau (*array*) pour représenter un arrangement séquentiel. W(fr)

⋆ **tableau** : accès facile (k-ème élement : $T[k]$), manipulation inefficace (taille fixe, allocation explicite)

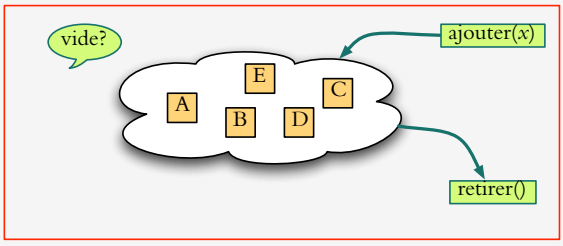
⋆ **liste chaînée** : manipulation efficace, accès difficile (p.e., parcours à partir de la tête pour chercher le k-ème élément)

Pour insérer ou supprimer un élément dans un tableau, il faut **décaler** les autres à côté.

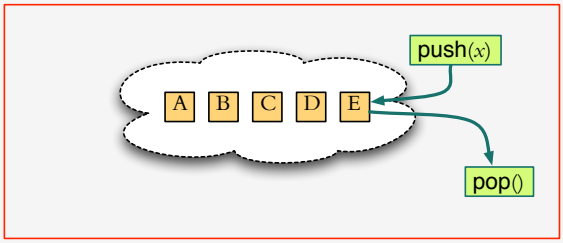
```
INSERT(T[0..n − 1], i, x)                                  // (insertion de l'élément x en position i)
1 for j ← n, n − 1, . . . , i + 1 do T[j] ← T[j − 1]       // (attention à l'ordre des déplacements !)
2 T[i] ← x

DELETE(T[0..n − 1], i)                                     // (suppression de l'élément en position i)
1 x ← T[i]
2 for j ← i + 1, i + 2, . . . , n − 1 do T[j − 1] ← T[j]   // (attention à l'ordre des déplacements !)
3 return x
```

Théorème 3.1. *L'insertion et la supression dans un tableau prennent un temps linéaire (au pire et en moyenne).*

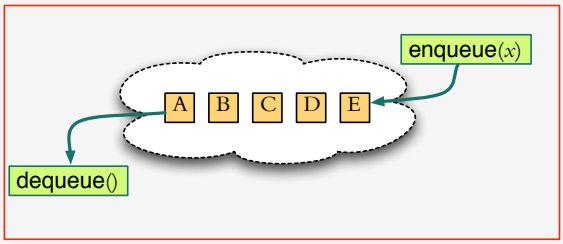
3.2 File (type abstrait)



Une **file** généralisée est un type abstrait pour une collection d'éléments avec deux opérations principales : une opération pour ajouter un élément, et une autre pour retirer un élément. (En général, il y a d'autres opérations auxiliaires comme tester si la file est vide.) La règle du choix de l'élément à retirer fait partie de la définition de l'interface : queue (first-in-first-out), pile (last-in-first-out), file de priorité (élément de priorité maximale).



Dans une **pile** (*stack*), l'élément le plus récemment ajouté est celui qui est retiré avant les autres (dernier entré, premier sorti). Les opérations de base s'appellent push («empiler») et pop («dépiler»). W(fr)



Dans une **queue** ou file FIFO, l'élément le plus ancient sera retiré avant les autres (premier entré, premier sorti). Les opérations de base s'appellent enqueue («enfiler») et dequeue («défiler»).

W(fr)

3.3 Pile par tableau

On peut implanter une pile par un tableau elements$[0 \ldots n-1]$: 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[0..n − 1] ← tableau de taille n ; capacity ← n
2 top ← 0                                          // (sommet de la pile)
Opération push(x)
1 elements[top] ← x
2 top ← top + 1
Opération pop
1 top ← top − 1 ; x ← elements[top]          // (débordement négatif avec pile vide !)
2 retourner x
```

3.4 Gestion dynamique de la taille d'un tableau

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

```
Opération pop
1 top ← top − 1 ; x ← elements[top]
2 if top < capacity/4
3 then REALLOC(⌈capacity/2⌉)
4 return x
Opération push(x)
1 if top = capacity
2 then REALLOC(2 · capacity)
3 elements[top] ← x ; top ← 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;
}
```

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

```
REALLOC(n)
R1 T[0..n − 1] ← nouveau tableau de taille n
R2 for i ← 0, . . . , top − 1 do a[i] ← elements[i]
R3 elements ← T ; size ← n
```

```
private void realloc(int n){
   Object[] T = new Object[n];
   for (int i=0; i<top; ++i) T[i]=elements[i];
   elements = T;
}
```

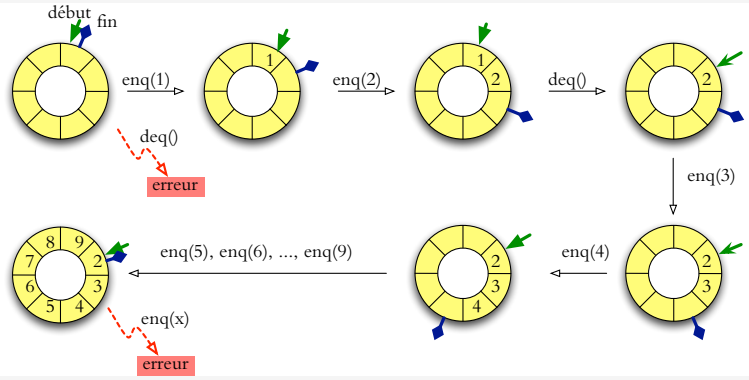
Dans le pire cas, une opération prend maintenant un temps linéaire en top qui est le nombre d'éléments à copier en ligne R2. Mais cela n'arrive pas trop fréquemment : on a un temps *amorti* constant.

Théorème 3.2. *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 W(en)

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 la séquence entière. ■

3.5 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_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);}}
```

3.6 Pile et queue par liste chaînée

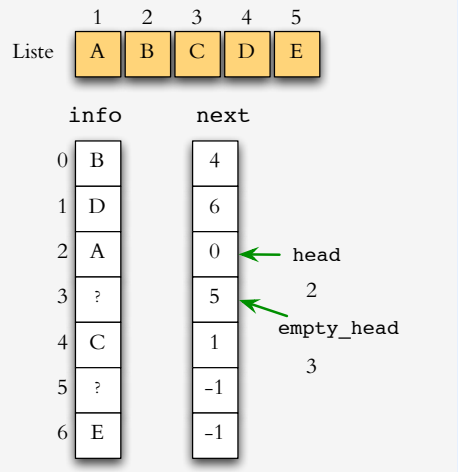
On peut également implanter la pile et la file FIFO en utilisant une liste chaînée. Les opérations de la pile s'implantent à l'aide d'insertion et suppression à la tête. Pour une implantation efficace de la file FIFO, on maintient un pointeur à la queue, à l'addition de la tête. Le pointeur à la tête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

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.

3.7 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 next d'un nœud est stocké comme indice dans un tableau elements.




```
public class Liste2
{
   private static int MAX_SIZE = 2015;
   private Object[] info;
   private int[] next;
   private int head;
   private int empty_head;
   public Liste2()
   {
      info = new Object[MAX_SIZE];
      next = new int[MAX_SIZE];
      head = -1;
      empty_head = 0;
      for (int i=0; i<MAX_SIZE-1; i++)
         next[i]=i+1;
      next[MAX_SIZE-1]=-1;
   }
   ...
}
```

Gestion de cases vides. On maintient alors une liste chaînée de cases vides (avec tête `empty_head`), 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 (supprimer un vrai élément) :

```
next[idx] = empty_head;
empty_head = idx;
```

Supprimer une case vide (insérer un élément) :

```
// utiliser info[empty_head] pour stocker la valeur
empty_head = next[empty_head];
```