Plan du cours

1/ Rappels et compléments Java.

2/ Tableaux, boucles et invariants.

3/ Notions élémentaires de complexité.

4/ Récursion.

5/ Structures de données et introduction aux types abstraits de données. **ICI**

6/ Quelques compléments Java.

Le type de données abstrait PILE (rappel)

Définition du type Pile de X (ici X est un type quelconque)

- **`Nom du type`** : Pile[X]
- **`Opérateurs`** (signature des méthodes)
  - Création. Pile :-> Pile[X]
  - Test. estVide: Pile[X] -> Booléen
  - empiler: XxPile[X] -> Pile[X]
    dépiler: Pile[X] -> XxPile[X]
    sommetPile: Pile[X] -> X
- **`Propriétés des opérateurs:`**
  - préconditions:
    dépiler(X): X ne doit pas être vide (utiliser le test au préalable!)
  - axiomes:
    - Pour tout x de X pour tout s de Pile[X]
    - estVide(Pile()) (une pile créée est initialement vide)
    - non estVide(empiler(x,s))
    - dépiler(empiler(x,s))=(x,s)

Implémentation des Piles avec des Listes

Interface Java (rappel)

Une interface en Java est composée de déclarations de méthodes mais **`sans définition de méthodes`** ni **`présence de variables d'instance`**.

Interface Java pour une Pile

On va utiliser la **`généricité de Java`** et donc le type Object (qui peut contenir n'importe quel objet du type référence). On va construire un type Pile de Object.

Implémentation des Piles avec des Listes

Interface Java (rappel)

Une interface en Java est composée de déclarations de méthodes mais **sans définition de méthodes** ni **présence de variables d'instance**.

Interface Java pour une Pile

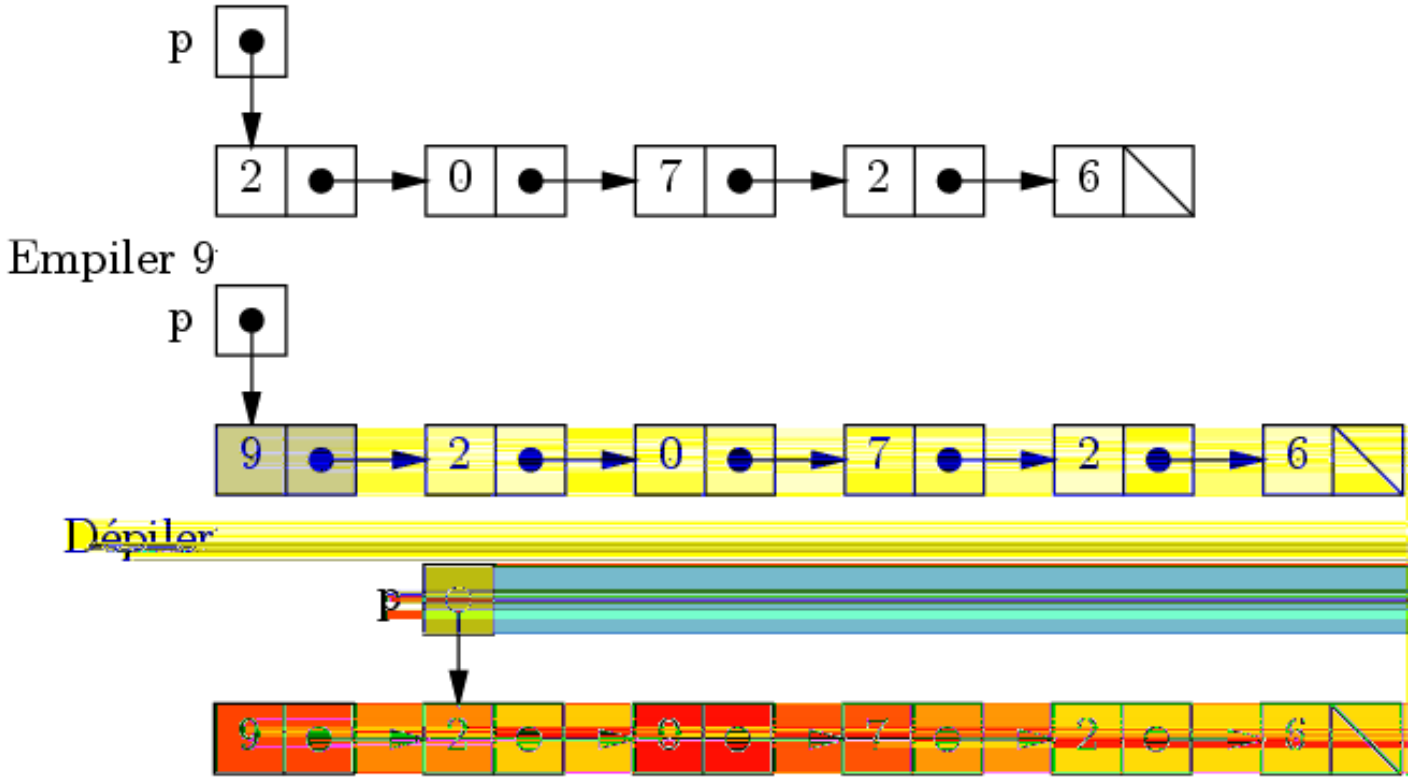
On va utiliser la **généricité de Java** et donc le type Object (qui peut contenir n'importe quel objet du type référence). On va construire un type Pile de Object.

On a donc pour notre pile **générique** :

Interface Pile simple

```
interface Pile {
        public Object depiler();
        public void empiler(Object o);
        public boolean estVide();
}
```

Implémentation d'une Pile avec des Listes (schéma)



Implémentation d'une Pile avec des Listes

Comme pour l'implémentation avec des tableaux, on utilise la **généricité** de Java (**listes d'Objects**)

Nœud

```
class Noeud {
        Object item;
        Noeud suivant;
        public Noeud(Object o) { item=o;suivant=null;}
        public Noeud(Object o, Noeud n) {item=o;suivant=n;}
}
```

Implémentation d'une Pile avec des Listes

Comme pour l'implémentation avec des tableaux, on utilise la **généricité** de Java (**listes d'Objects**)

Nœud

```
class Noeud {
        Object item;
        Noeud suivant;
        public Noeud(Object o) { item=o;suivant=null;}
        public Noeud(Object o, Noeud n) {item=o;suivant=n;}
}
```

Implémentation

```
class PileListe implements Pile {
        private Noeud sommet;
        public PileListe(){sommet=null;}
        public boolean estVide(){ return (sommet==null);}
        public void empiler(Object o) {sommet=new Noeud(o,sommet);}
        public Object depiler() {Object tmp=sommet.item;
sommet=sommet.suivant; return tmp; }
}
```

Remarques

- Les classes `PileListe` et `PileTable` déclarent dans leur entête qu'elles **`implémentent`** l'**`interface`** `Pile` .

Remarques

- Les classes `PileListe` et `PileTable` déclarent dans leur entête qu'elles **`implémentent`** l'**`interface`** `Pile` .
- A **`chaque déclaration`** de méthodes de l'interface correspond une **`définition`** de méthode de même **signature**.

Remarques

- Les classes `PileListe` et `PileTable` déclarent dans leur entête qu'elles **`implémentent`** l'**`interface Pile`** .
- A **`chaque déclaration`** de méthodes de l'interface correspond une **`définition`** de méthode de même **signature**.
- L'implémentation est alors déclarée comme **`private`** : l'utilisateur ne doit avoir accès qu'à ce qui concerne la Pile **abstraite**.

Remarques

- Les classes `PileListe` et `PileTable` déclarent dans leur entête qu'elles **`implémentent`** l'**`interface Pile`** .
- A **`chaque déclaration`** de méthodes de l'interface correspond une **`définition`** de méthode de même **signature**.
- L'implémentation est alors déclarée comme **`private`** : l'utilisateur ne doit avoir accès qu'à ce qui concerne la Pile **abstraite**.
- Chacune des implémentations a ses propres constructeurs.

Remarques

- Les classes `PileListe` et `PileTable` déclarent dans leur entête qu'elles **`implémentent`** l'**`interface Pile`** .
- A **`chaque déclaration`** de méthodes de l'interface correspond une **`définition`** de méthode de même **signature**.
- L'implémentation est alors déclarée comme **`private`** : l'utilisateur ne doit avoir accès qu'à ce qui concerne la Pile **abstraite**.
- Chacune des implémentations a ses propres constructeurs.
- La classe PileTable travaillant avec des tableaux qui ont nécessairement une taille maximale limite est **restrictive**.

Remarques

- Les classes `PileListe` et `PileTable` déclarent dans leur entête qu'elles **`implémentent`** l'**`interface Pile`** .
- A **`chaque déclaration`** de méthodes de l'interface correspond une **`définition`** de méthode de même **signature**.
- L'implémentation est alors déclarée comme **`private`** : l'utilisateur ne doit avoir accès qu'à ce qui concerne la Pile **abstraite**.
- Chacune des implémentations a ses propres constructeurs.
- La classe PileTable travaillant avec des tableaux qui ont nécessairement une taille maximale limite est **restrictive**.
- L'interface peut être vue comme une **`abstraction`** des implémentations (qui sont des **`concrétisations`** de l'interface).

Piles et appels récursifs

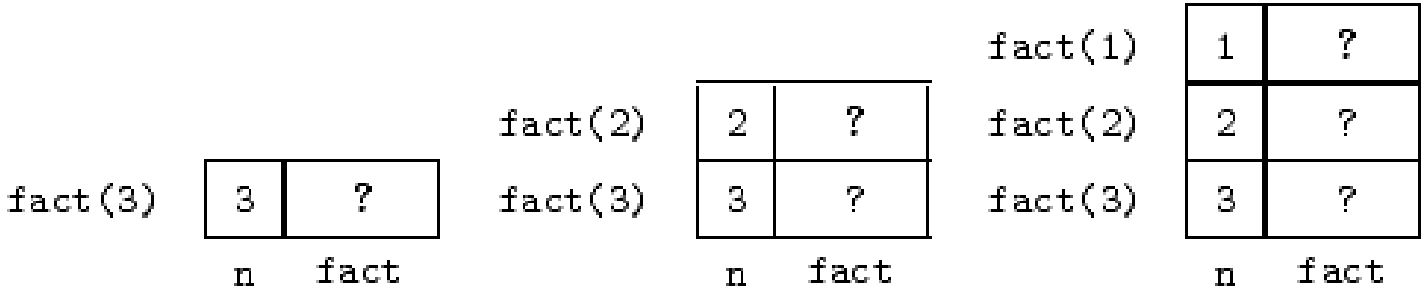
Les piles sont utilisés lors d' **appels récursifs** .

- Le système dispose d'une **pile**.

- Lorsqu'on appelle une **fonction** ou **méthode**:
  - on empile l' **adresse de retour** de la fonction
  - on empile les **arguments** de la fonction
  - on **exécute** le code de la fonction (en utilisant à son adresse)
- Le **fonction** ou la **méthode**:
  - alloue éventuellement des **variables locales** dans la pile
  - à la " **fin** " (par exemple lors de l'exécution de l'instruction **return**)
    - on dépile les ***variables locales*** et les ***arguments***
    - on dépile l' ***adresse de retour***
    - on empile (éventuellement) la ***valeur de retour***
    - on se rend à l' ***adresse de la fonction***

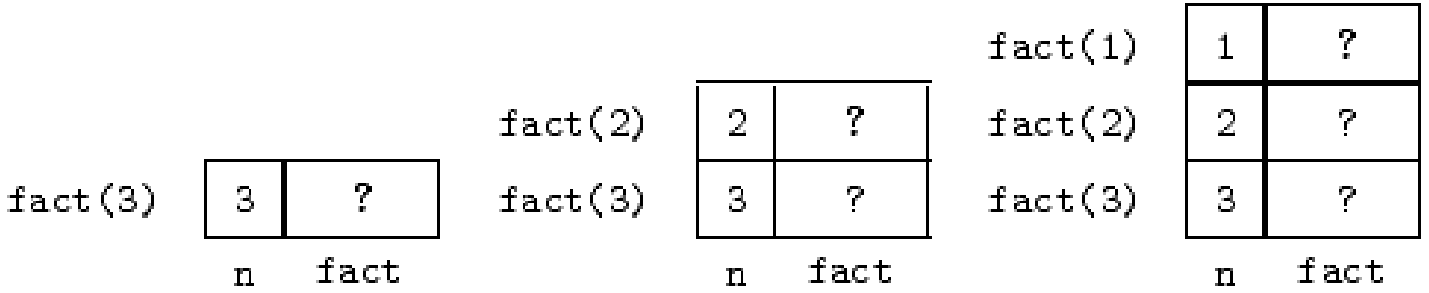
Un exemple: factoriel

Calcul de `fact(3)` : empilement des appels récursifs.

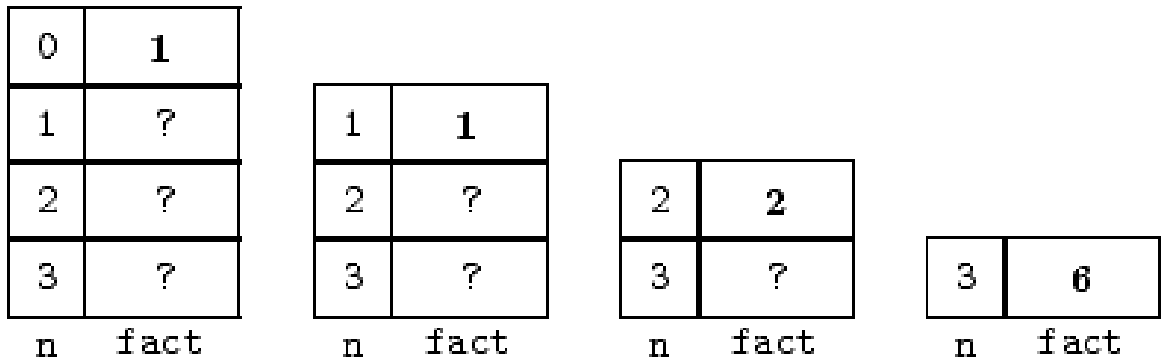


Un exemple: factoriel

Calcul de `fact(3)` : empilement des appels récursifs.



Dépilement



Spécifications

- Une ***file*** une **`structure de données`** du type: FIFO (**`Fist In, First Out`**).
- Dans une file, il faut pouvoir réaliser des opérations d' **`ajout`** et de **`suppression`** d'un ***élément*** en sachant
  1. qu'un ajout a toujours lieu à la **`fin de l'ensemble de données`**
  2. qu'une suppression a toujours lieu au **`début`**

Les files

Spécifications

- Une ***file*** une **`structure de données`** du type: FIFO (**`Fist In, First Out`**).
- Dans une file, il faut pouvoir réaliser des opérations d' **`ajout`** et de **`suppression`** d'un ***élément*** en sachant
  1. qu'un ajout a toujours lieu à la **`fin de l'ensemble de données`**
  2. qu'une suppression a toujours lieu au **`début`**

On va voir

- le **`type abstrait`** de la file et
- **`deux représentations`** de la file : par les ***tableaux*** et par les ***listes***.

File: le type abstrait

Définition du type File de X (ici X est un type quelconque)

- **`Nom du type`** : File[X]
- **`Opérateurs`** (signature des méthodes)
  - Création. File :-> File[X]
  - Test. fileVide: File[X] -> Booléen
  - ajouter : X,File[X] -> File[X]
    supprimer: File[X] -> File[X]
    sommet: File[X] -> X
    longueur: File[X] -> Entier
- **`Propriétés des opérateurs:`**
  - préconditions:
    longueur(FILE[X])=0 implique sommet(FILE[X]) = null
    longueur(FILE[X])=0 implique supprimer(FILE[X]) = null
  - axiomes: Pour tout x de X pour tout s de Pile[X]
    a) fileVide(File()) (une file créée est initialement vide) **b)** non fileVide(ajouter(x,s)) = vrai et **c)** nous avons supprimer(ajouter(x,s))=(x,s)

File et tableau

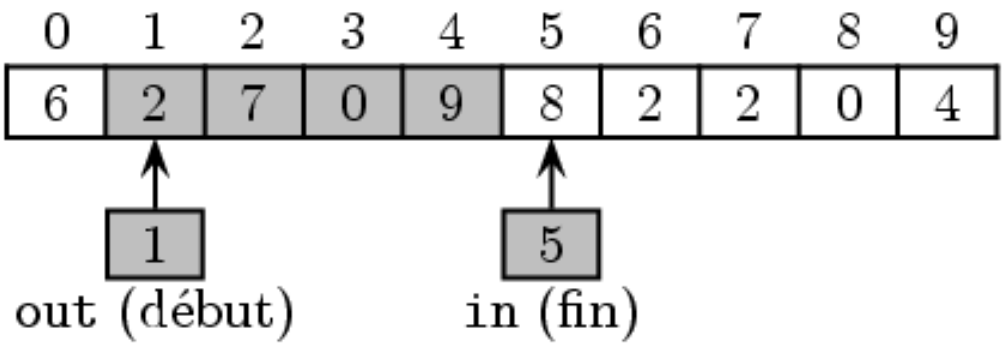
L'idée principale

Il s'agît de gérer deux indices dans un tableau: `in` qui marque la position où ajouter le prochain élément et `out` qui marque la position où on va enlever le prochain élément. La **`file est contenu dans le tableau`** en partant de l'indice `out` à l'indice `in`

File et tableau

L'idée principale

Il s'agît de gérer deux indices dans un tableau: `in` qui marque la position où ajouter le prochain élément et `out` qui marque la position où on va enlever le prochain élément. La **`file est contenu dans le tableau`** en partant de l'indice `out` à l'indice `in`



Le grand tour

A chaque **ajout**, on incrémente `in` . De même à chaque **suppression**, on incrémente `out` . Mais lorsqu'un indice atteint la fin du tableau, ***il fait le tour et repart à zéro***. On peut donc avoir out $<$ in.

Le grand tour

A chaque **`ajout`**, on incrémente **`in`** . De même à chaque **`suppression`**, on incrémente **`out`** . Mais lorsqu'un indice atteint la fin du tableau, ***il fait le tour et repart à zéro***. On peut donc avoir out $<$ in.

Le même tableau mais contenant 2,0,4,6,2,7

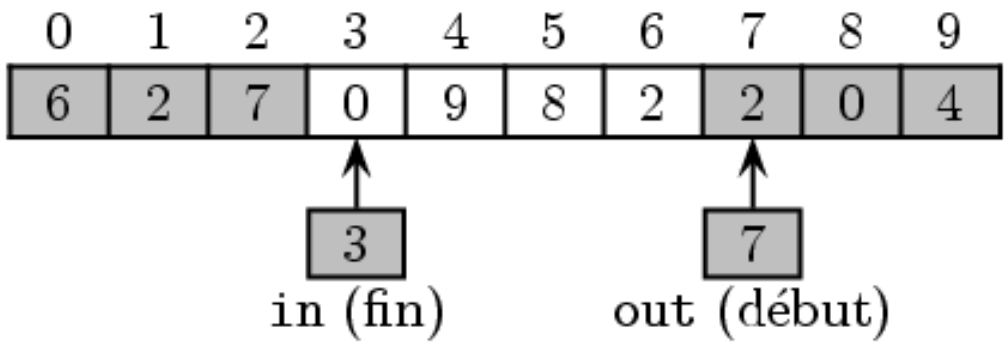


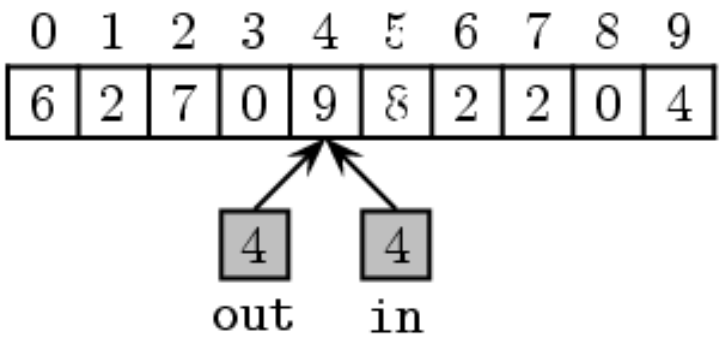
Tableau circulaire

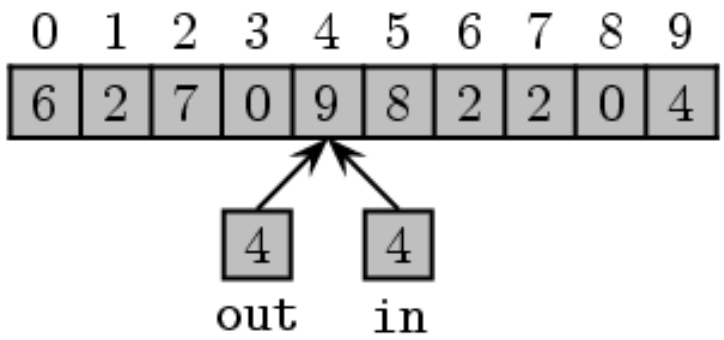
Le tableau d'une file est donc ici un tableau **`circulaire`** . Les incrémentations se feront ***modulo N*** où N est la **`taille du tableau`**. Mais cela engendre une difficulté pour distinguer entre FILE VIDE et FILE PLEINE.

Tableau circulaire

Le tableau d'une file est donc ici un tableau **`circulaire`** . Les incrémentations se feront ***modulo N*** où N est la **`taille du tableau`**. Mais cela engendre une difficulté pour distinguer entre FILE VIDE et FILE PLEINE.

Par exemple,



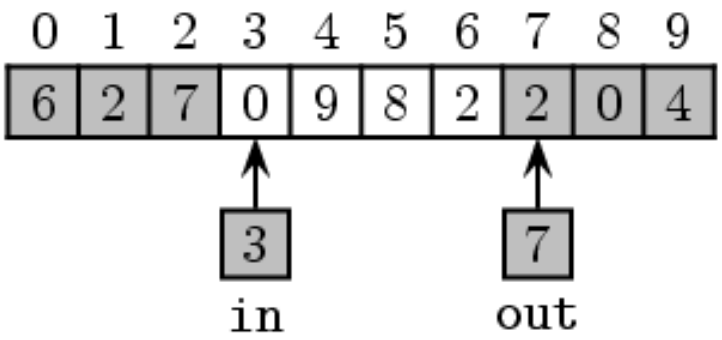


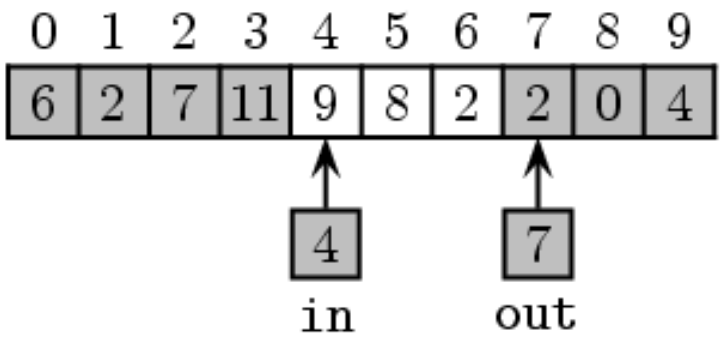
Solutions

Il existe deux solutions classiques à ce problème:

1. une variable `nb` représentant le `nombre d'éléments dans la file` peut être créée.
2. on peut aussi décréter qu'une file contenant N-1 éléments est pleine (tester si in+1 est congru à out modulo N).

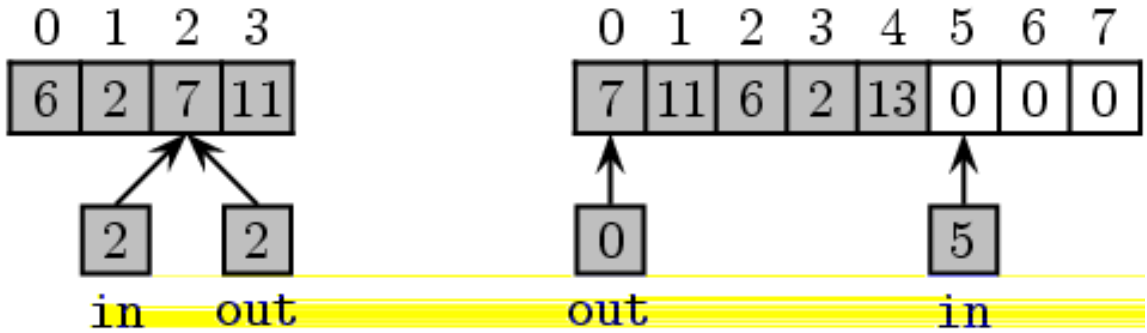
File et tableau : ajout d'un élément





File et tableau : ajout d'un élément avec redimensionnement

Avant ajout la file contient 7, 11, 6, 2, après ajout elle contient 7, 11, 6, 2, 13.



File et tableau : le code

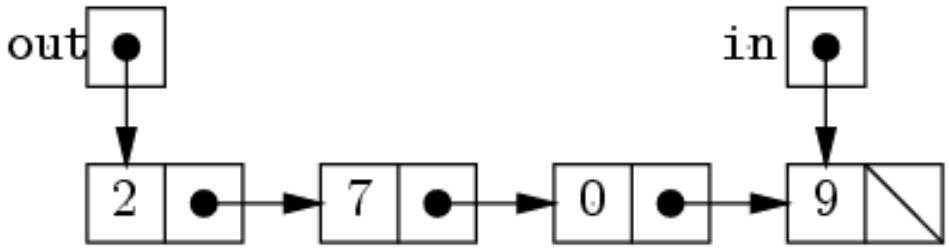
File et liste

L'idée principale

Le concept est presque le même qu'avec les tableaux. Les éléments de la file sont dans une liste : on supprime au début et on rajoute à la fin. On utilise alors (comme pour les tableaux) deux références `out` et `in` .

L'idée principale

Le concept est presque le même qu'avec les tableaux. Les éléments de la file sont dans une liste : on supprime au début et on rajoute à la fin. On utilise alors (comme pour les tableaux) deux références `out` et `in` .



Partiel du 19 Mars 2011

Rappel

- Horaire: de 14h30 à 16h30.
- Sans documents autorisés.
- Salles **`AMPHI 6C`**, **`574F`** et **`575F`**.
- Veuillez consulter l'affichage pour votre affectation.