Logiciels de gestion d'un ensemble d'informations

Introduction aux classes

Introduction à la gestion de tables

Introduction

Nous allons traiter des techniques de représentation en mémoire d'un ensemble d'informations

Type des éléments d'un ensemble, notion de clé

- Chaque élément d'un ensemble peut être composé de plusieurs informations
- L'une d'elles, appelée **clé** ou **indicatif**, permet d'identifier un élément de manière unique
- Par l'intermédiaire de la clé on peut accéder à la valeur de l'élément
- Par exemple, pour un répertoire téléphonique, le nom d'une personne joue le rôle de clé, le numéro de téléphone est la valeur associée
- Description du type Elément:

```
Elément : type agrégat
            cl : Clé       // peut être composée de plusieurs champs
            v : Valeur     // peut être composée de plusieurs champs
          fagregat
```

Ordre des éléments de l'ensemble

- Il est des situations où le type Elément est muni d'une relation d'ordre total portant sur la clé ou sur la valeur, ou sur une partie de la valeur.
- Les éléments de l'ensemble sont en général mémorisés en séquence.
- Cette séquence définit un ordre entre les éléments, cet ordre peut correspondre à l'ordre dont est muni le type Elément. On dit alors que la séquence est **triée**.
- L'ordre des éléments peut cependant dépendre d'autres critères comme par exemple la chronologie selon laquelle les éléments ont été ajoutés à la séquence.

Gestion d'un ensemble

- Le logiciel de gestion d'un ensemble d'éléments fournit en général les fonctions suivantes :
  - **Créer** un nouvel ensemble en l'initialisant par l'ensemble vide
  - **Ajouter** un nouvel élément à l'ensemble
  - **Modifier** un élément existant : la valeur associée à une clé donnée
  - **Supprimer** un élément de l'ensemble
  - **Enumérer** l'ensemble ou un sous-ensemble des éléments (parcours)
  - **Rechercher** un élément particulier (recherche)

Organisation du logiciel de gestion d'un ensemble

- Le logiciel comporte 2 parties :
  - Une partie dédiée à la communication avec l'utilisateur appelée interface utilisateur : saisie des commandes de l'utilisateur et présentation des résultats, message d'erreur.
  - Une partie dédiée à la gestion de l'ensemble appelée noyau fonctionnel. Partie réalisant les fonctions du logiciel.

  → Nous allons nous intéresser à la réalisation du noyau fonctionnel.

Représentation de l'ensemble à gérer

- Il faut choisir une structure de données pour représenter l'ensemble : fichier séquentiel, tableau, liste chainée, arbre, graphe, etc.
- Associés à cette représentation, nous définirons les actions et fonctions assurant la gestion de l'ensemble. On parle de méthodes dans les langages à objets.
- Toutes ces actions et fonctions accèdent donc à la même structure de données qui constitue les variables partagées par les méthodes.
- Il est donc intéressant de regrouper dans une même entité fermée, la structure de données représentant l'ensemble, et les actions et fonctions qui la manipulent : cette entité est appelée module ou classe
- Nous utiliserons donc une classe pour représenter une telle partie de logiciel.

Classe représentant l'ensemble

classe NomdelaClasse

lexique de NomdelaClasse // variables partagées par les méthodes de la classe

NMAX: entier > 0 // nombre maximum d'éléments

mem : table sur [0.. NMAX-1] d'Eléments // l'ensemble des éléments

…

méthodes de NomdelaClasse

public action creer (Consulté n :entier > 0)

// Effet : initialise l'ensemble de taille n à l'ensemble vide (= constructeur)

… lexique et algorithme de creerNomdelaClasse

public action ajouter (…)

// Effet : ajout d'un élément à l'ensemble

… lexique et algorithme de ajouter

public action supprimer (…)

// Effet : ajout d'un élément à l'ensemble

… lexique et algorithme de supprimer

public fonction recherche (…) → Elément

// Recherche d'un élément dans l'ensemble

… lexique et algorithme de recherche

… etc.

privé fonction adrElem(…) → entier sur 0..NMAX-1

…

fclasse

Utilisation de la classe

lexique principal

truc : NomdelaClasse // ensemble à gérer

… définition des données et des résultats

tt : entier > 0 // donnée : taille de truc

e : Elément // inter : un élément

Algorithme principal

// saisie des données

…

truc.creer (tt)

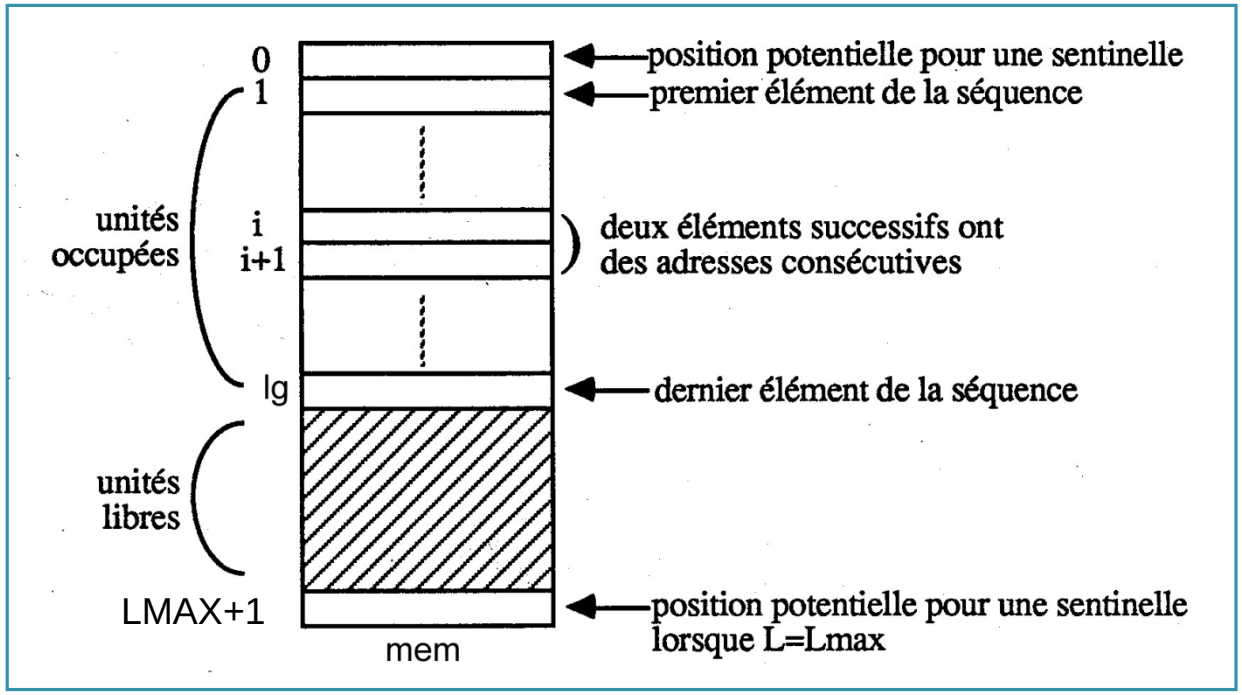
…

truc.ajouter(...)

e ← truc.recherche(…)

Représentation d'un ensemble par une séquence contiguë dans une table

- L'ensemble est représenté dans une table d'**Eléments** nommée **mem**
- On connaît la longueur **lg** de l'ensemble dont la taille maximale est **LMAX**



Principe de représentation contiguë d'une séquence

Exemple de gestion d'un ensemble: le répertoire téléphonique

- On désire gérer une répertoire simple de numéros de téléphone en nombre relativement limité
- Un élément du répertoire est composé
  - d'un nom de personne
  - d'un numéro de téléphone
- On désigne par NMAX le nombre maximum d'éléments du répertoire
- Pour simplifier de développement de l'exemple, on suppose qu'un seul numéro est associé à une personne donnée, et qu'il n'y a pas d'homonymes.

Exemple de gestion d'un ensemble: le répertoire téléphonique

- Le logiciel doit fournir les fonctions suivantes :
  - **création** d'un répertoire vide
  - **ajouter** un nouvel élément au répertoire
  - **modifier** un numéro associé à un nom
  - **supprimer** un élément du répertoire
  - **consulter** le répertoire pour obtenir le numéro de téléphone d'une personne
- Conditions exceptionnelles à détecter
  - **saturation** de l'espace mémoire : tentative d'ajout d'un élément au répertoire alors que sa taille maximale est atteinte
  - **nom erroné** : tentative d'ajout d'un élément lié à un nom déjà présent, ou tentative de suppression, de modification ou de consultation liée à un nom qui n'existe pas.

Type énuméré

- Il peut arriver que l'on veuille définir un type formé uniquement de constantes nommées
- A chaque nom on associe une valeur entière
- Par exemple les jours de la semaine pourraient être représentés par un type formées des constantes : lundi, mardi mercredi, jeudi, vendredi, samedi, dimanche.
- Pour notre répertoire, nous voudrions définir un type Condition correspondant aux deux conditions exceptionnelles à détecter et à la condition normale sans erreur.
- Représentation possible du type Condition dans le lexique partagé :

  constante NORMAL l'entier 0
  constante SATURATION l'entier 1
  constante NOMERRONE l'entier 2
  Condition : type entier entre NORMAL et NOMERRONE // entre 0 et 2

- Il s'agit en fait d'un type dit « énuméré » que l'on peut définir plus simplement de la manière suivante :

  Condition : type { NORMAL, SATURATION, NOMERRONE }

  autre exemple :

  Jour : type { lundi, mardi mercredi, jeudi, vendredi, samedi, dimanche }

  Les noms de valeurs spécifiés sont associés à aux valeurs 0, 1, 2, etc.

Spécification du Répertoire (1)

Lexique partagé

Personne : type chaine
Numero : type chaine
Elément : type agrégat nom : Personne ; num : Numero fagrégat
Condition : type { NORMAL, SATURATION, NOMERRONE }

Classe Répertoire

lexique de Répertoire // variables partagées par les méthodes du répertoire
NMAX: entier > 0 // nombre maximum d'éléments du répertoire
mem : tableau sur [0.. NMAX] d'Eléments // l'ensemble des éléments du répertoire
lg : entier entre 0 et NMAX // nombre d'éléments du répertoire
...
méthodes de Répertoire

public action **creerRepVide** (Consulté n :entier > 0)
// Effet : initialise le répertoire de taille n à un répertoire vide
// E.I. : indifférent
// E.F.: mem représente le répertoire vide de taille n

Spécification du Répertoire (2)

public action **ajouter** (Consulté e : Element, Elaboré c : Condition)
// Effet : ajout de l'élément e au répertoire
// E.I. : indifférent
// E/F.: c = NORMAL : on a ajouté e au répertoire
c = SATURATION : répertoire inchangé, il comporte NMAX éléments
c = NOMERRONE : répertoire inchangé ,il existe déjà un élément avec le même nom que e

public action **modifier** (Consulté e : Element, Elaboré c : Condition)
// Effet modifie le numéro associé à e.nom
// E.I. : indifférent
// E/F.: c = NORMAL : on modifié le répertoire en tenant compte de e
c = NOMERRONE : répertoire inchangé, e.nom est absent du répertoire

public action **supprimer**(Consulté nom : Personne, Elaboré c : Condition)
// Effet : supprime du répertoire l'élément correspondant à nom
// E.I. : indifférent
// E/F.: c = NORMAL : on a supprimé du répertoire l'élément associé à nom
c = NOMERRONE : répertoire inchangé nom est absent du répertoire

public fonction **consulter**(nom : Personne) → Numero
// consulter(nom) renvoie le numéro de téléphone associé nom
// renvoie " " si nom n'a pas été trouvé dans le répertoire

Réalisation du répertoire

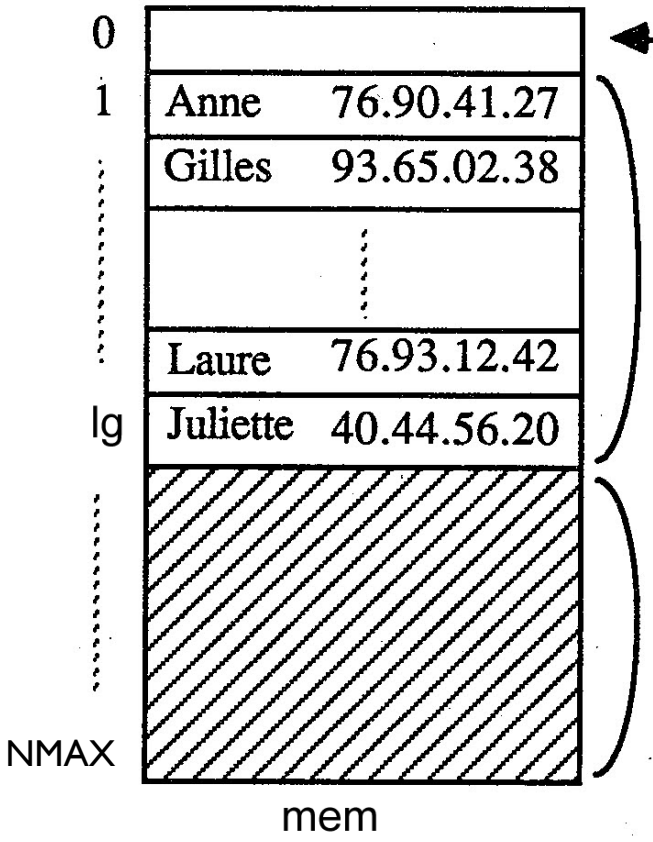
- La notion d'adresse en mémoire des éléments du répertoire est représentée par les indices du tableau
- Nous aurons besoin d'une fonction intermédiaire (privée) pour savoir si le répertoire comporte un élément avec un nom donné, et dans l'affirmative où il se trouve. Soit **adNom** le nom de cette fonction
- adNom(nm) renvoie l'adresse (l'indice dans mem) de l'Elément de nom nm. Si le nom cherché est absent, la fonction renvoie une valeur particulière nommée **ADFIC**. La valeur **0** est choisie pour ADFIC.

Réalisation du répertoire

Nous complétons donc la classe Répertoire comme suit :

```
lexique de Répertoire
    constante ADFIC l'entier 0
méthodes de Répertoire
  privé fonction adNom(nm : Personne) → entier entre 0 et NMAX
  // adNom(n) renvoie l'adresse (l'indice) de l'élément de nom nm
  // s'il existe, renvoie ADFIC sinon
  lexique de adNom
    a : entier entre 0 et NMAX  // inter : indice de recherche
  algorithme de adNom
    mem[ADFIC].nom ← nm
    a ← lg
    tantque mem[a].nom ≠ nm faire
        a ← a – 1
    ftq
    renvoyer(a)
public action creerRepVide (Consulté n :entier > 0)
    // Effet : initialise le répertoire de taille n à un répertoire vide
    // E.I. : indifférent
    // E/F.: mem représente le répertoire vide
  algorithme de creerRepVide
      NMAX ← n ; lg ← 0
```

Schéma du Répertoire



élément prévu pour la technique de sentinelle

les emplacements occupés sont contigus
lg est l'indice du dernier emplacement occupé
(c'est aussi le nombre d'emplacements occupés

les hachures représentent l'ensemble des emplacements libres (ils sont contigus)

Questions : comment procéder pour :

- ajouter un élément au répertoire ?
- supprimer un élément du répertoire ?

Ajout d'un Elément

```
public action ajouter (Consulté e : Element, Elaboré c : Condition)
// Effet : ajout de l'élément e au répertoire
// E.I. : indifférent
// E.F.:    c = NORMAL : on a ajouté e au répertoire
            c = SATURATION : répertoire inchangé, il comporte NMAX éléments
            c = NOMERRONE : répertoire inchangé ,il existe déjà un élément
                                    avec le même nom que e
lexique de ajouter
  a : entier entre 0 et NMAX  // adresse de e dans mem si présent
  fonction utilisée : adNom
algorithme de ajouter
   a ← adNom(e.nom)
   si a ≠ ADFIC alors c ← NOMERRONE
   sinon  si lg = NMAX
            alors c ← SATURATION
            sinon
               lg ← lg + 1 ; mem[lg] ← e ; c ← NORMAL
            fsi
  fsi
```

Modification d'un Elément

public action **modifier** (Consulté e : Element, Elaboré c : Condition)

// Effet modifie le numéro associé à e.nom
// E.I. : indifférent
// E/F.: c = NORMAL : on modifié le répertoire en tenant compte de e
c = NOMERRONE : répertoire inchangé, e.nom est absent du répertoire

lexique de modifier
a : entier entre 0 et NMAX // adresse de e.nom dans mem
fonction utilisée : adNom

algorithme de modifier
a ← adNom(e.nom)
si a = ADFIC
alors c ← NOMERRONE
sinon
mem[a].num ← e.num // on peut aussi écrire: mem[a] ← e
c ← NORMAL
fsi

Suppression d'un Elément

```
public action supprimer(Consulté nom : Personne, Elaboré c : Condition)
// Effet : supprime du répertoire l'élément correspondant à nom
// E.I. : indifférent
// E/F.: c = NORMAL : on a supprimé du répertoire l'élément associé à nom
         c = NOMERRONE : : répertoire inchangé nom est absent du répertoire
lexique de supprimer
  a : entier entre 0 et NMAX  // adresse de nom dans mem
  fonction utilisée : adNom
 algorithme de supprimer
   a ← adNom(nom)
   si a = ADFIC
   alors c ← NOMERRONE
   sinon
       mem[a] ← mem[lg]  // l'ordre des éléments n'étant pas significatif, on supprime l'élément
       lg ← lg – 1              // d'indice a en le remplaçant par le dernier: mem[lg]
       c ← NORMAL
fsi
```

Consultation du Répertoire

```
public fonction consulter(nom : Personne) → Numero
// consulter(nom) renvoie le numéro de téléphone associé nom
// renvoie  " "  si nom n'a pas été trouvé dans le répertoire
 lexique de consulter
  a : entier entre 0 et NMAX  // adresse de nom dans mem
  n : Numéro    // valeur renvoyée
  fonction utilisée : adNom
algorithme de consulter
   a ← adNom(nom)
   si a = ADFIC
   alors n ← " "
   sinon
      n ← mem[a].num
   fsi
   renvoyer(n)
```

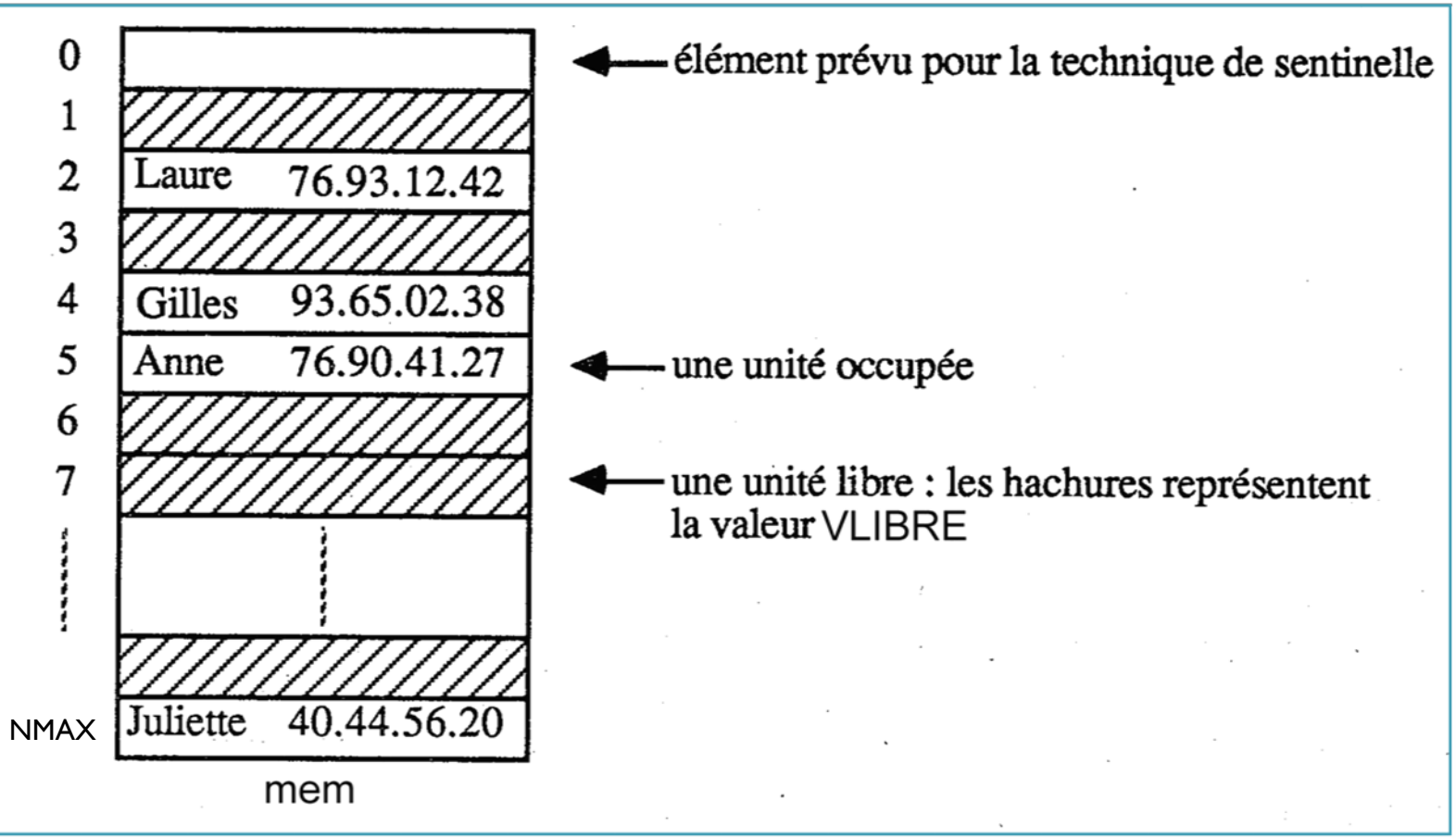
Organisation dispersée des Eléments du Répertoire

Autre organisation possible du Répertoire:

- A un instant donné les éléments présents dans le répertoire sont dispersés dans l'espace mémoire
- Les suppressions d'éléments libèrent des emplacements utilisables pour des ajouts ultérieurs
- Les emplacements libérés sont marquées par une valeur spéciale d'Elément appelée VLIBRE
- Gérer le répertoire revient à gérer un ensemble d'emplacements mémoire comportant un certain nombre de valeurs VLIBRE.

L'organisation dispersée est utilisée lorsque l'on ne peut pas déplacer des éléments dans la table, car un déplacement engendrerait des calculs couteux pour mettre à jour d'autres structures de données.

Schéma du répertoire en organisation dispersée



Questions : comment procéder pour :

- ajouter un élément au répertoire ?
- supprimer un élément du répertoire ?

Répertoire géré en dispersé

Lexique partagé

Personne : type chaine
Numero : type chaine
Elément : type agrégat nom : Personne ; num : Numero fagrégat
Condition : type { NORMAL, SATURATION, NOMERRONE }

Classe Répertoire

lexique de Répertoire
NMAX : entier > 0 // nombre maximum d'éléments du répertoire
mem : table sur [0.. NMAX] d'Eléments // éléments du répertoire
constante Elément VLIBRE = Elément("" , "")
// valeur spécifique indiquant un emplacement libre
constante ADFIC l'entier 0

...

Création du répertoire (dispersé)

méthodes de Répertoire

public action **creerRepVide** (Consulté n :entier > 0)
// Effet : initialise le répertoire de taille n à un répertoire vide
// E.I. : indifférent
// E/F.: mem représente le répertoire vide
lexique de creerRepVide
  a : entier entre 0 et NMAX // indice de parcours
algorithme de supprimer
   NMAX ← n
   a ← 1
   tantque a ≤ NMAX faire
    mem[a] ← VLIBRE
    a ← a + 1
   ftq

Fonction adNom (dispersé)

privé fonction adNom(nm : Personne) → entier entre 0 et NMAX
  // adNom(n) renvoie l'adresse (l'indice) de l'élément de nom nm
  // s'il existe, renvoie ADFIC sinon
  lexique de adNom
    a : entier entre 0 et nmax // inter : indice de recherche
  algorithme de adNom
    mem[ADFIC].nom ← nm
    a ← **NMAX**
    tantque mem[a].nom ≠ nm faire
        a ← a – 1
    ftq
    renvoyer(a)

Ajout d'un Elément (dispersé)

```
public action ajouter (Consulté e : Element, Elaboré c : Condition)
// Effet : ajout de l'élément e au répertoire
// E.I. : indifférent
// E/F.:    c = NORMAL : on a ajouté e au répertoire
            c = SATURATION : répertoire inchangé, il comporte NMAX éléments
            c = NOMERRONE : répertoire inchangé ,il existe déjà un élément
                                    avec le même nom que e
lexique de ajouter
  a : entier entre 0 et NMAX  // adresse d'insertion de e dans mem
  fonction utilisée : adNom
 algorithme de ajouter
   a ← adNom(e.nom)
   selon a
     a ≠ ADFIC : c ← NOMERRONE
     a = ADFIC : // on recherche un emplacement libre
                 a ← adNom(VLIBRE.nom)
                 si a = ADFIC
                 alors c ← SATURATION
                 sinon mem[a] ← e ; c ← NORMAL
                 fsi
  fselon
```

Modification (dispersé: inchangé)

public action **modifier** (Consulté e : Element, Elaboré c : Condition)

// Effet modifie le numéro associé à e.nom
// E.I. : indifférent
// E/F.: c = NORMAL : on modifié le répertoire en tenant compte de e
c = NOMERRONE : répertoire inchangé, e.nom est absent du répertoire

lexique de modifier
a : entier entre 0 et NMAX // adresse de e.nom dans mem
fonction utilisée : adNom

algorithme de modifier
a ← adNom(e.nom)
si a = ADFIC
alors c ← NOMERRONE
sinon
mem[a] ← e
c ← NORMAL
fsi

Suppression (dispersé)

public action **supprimer**(Consulté nom : Personne, Elaboré c : Condition)

// Effet : supprime du répertoire l'élément correspondant à nom
// E.I. : indifférent
// E/F.: c = NORMAL : on a supprimé du répertoire l'élément associé à nom
c = NOMERRONE : répertoire inchangé nom est absent du répertoire

lexique de supprimer
  a : entier entre 0 et NMAX // adresse de nom dans mem
  fonction utilisée : adNom
 algorithme de supprimer
   a ← adNom(nom)
   si a = ADFIC
   alors c ← NOMERRONE
   sinon
       mem[a] ← VLIBRE
      c ← NORMAL
   fsi

Consultation (dispersé : inchangé)

public fonction **consulter**(nom : Personne) → Numero

// consulter(nom) renvoie le numéro de téléphone associé nom
// renvoie " " si nom n'a pas été trouvé dans le répertoire

lexique de consulter

a : entier entre 0 et NMAX // adresse de nom dans mem

n : Numéro // valeur renvoyée

fonction utilisée : adNom

algorithme de consulter

```
a ← adNom(nom)
si a = ADFIC
alors n ← " "
sinon
    n ← mem[a].num
fsi
renvoyer(n)
```

Organisation triée des Eléments d'un ensemble

Il est parfois intéressant d'ordonner les éléments de l'ensemble selon un ordre particulier

- Par exemple selon les noms de personnes dans le cas du répertoire téléphonique
- Dans le cas où l'ordre des éléments est significatif, l'écriture des algorithmes de gestion de l'ensemble change :
  - On utilise la représentation contiguë de l'ensemble
  - L'insertion et la suppression d'un élément nécessite un décalage du tableau d'une position

Algorithme d'insertion d'un élément dans une table triée en ordre croissant

```
lexique
  t : tableau sur [0..NMAX] d'Eléments  // t[0] réservé pour sentinelle
  lg : entier entre 0 et NMAX // nombre d'éléments de l'ensemble
  i : entier entre 0 et NMAX   // indice de parcours de t
  e : Elément  // élément à insérer dans t
algorithme
  // on suppose lg < NMAX
  t[0] ← e
  i ← lg
  tantque e < t[i] faire
    t [i+1]  ← t[i]
     i ← i - 1
  ftq
  // e ≥ T[i] ⇒ on insère e en i+1
  t[i+1] ← e
  lg ← lg + 1
```

Application : Ajout (Répertoire trié)

```
public action ajouter (Consulté e : Element, Elaboré c : Condition)
// Effet : ajout de l'élément e au répertoire
// E.I. : indifférent
// E/F.:      c = NORMAL : on a ajouté e au répertoire
              c = SATURATION : répertoire inchangé, il comporte NMAX éléments
               c = NOMERRONE : répertoire inchangé ,il existe déjà un élément
                                avec le même nom que e
lexique de ajouter
  a : entier entre 0 et NMAX  // indice de parcours de mem
  fonction utilisée : adNom
 algorithme de ajouter
   a ← adNom(e.nom)
   si a ≠ ADFIC alors c ← NOMERRONE
   sinon  si lg = NMAX
          alors c ← SATURATION
          sinon // on décale le tableau jusqu'à la position d'insertion
             mem[0] ← e    // e.nom joue le rôle de valeur sentinelle
             a ← lg
             tantque e.nom < mem[a].nom faire
                  mem[a+1] ← mem[a] ;  a ← a – 1
             ftq
             // e.nom ≥ mem[a].nom ⇒ on insère e en a+1
             mem[a+1] ← e ;
             lg ← lg + 1 ; c ← NORMAL
          fsi
 fsi
```

Suppression (Répertoire trié)

```
public action supprimer(Consulté nom : Personne, Elaboré c : Condition)
// Effet : supprime du répertoire l'élément correspondant à nom
// E.I. : indifférent
// E/F.: c = NORMAL : on a supprimé du répertoire l'élément associé à nom
         c = NOMERRONE : répertoire inchangé nom est absent du répertoire
lexique de supprimer
  a : entier entre 1 et NMAX  // indice l'élément à supprimer
  i : entier entre 1 et NMAX   // indice de parcours de mem pour le décalage
  fonction utilisée : adNom
 algorithme de supprimer
   a ← adNom(nom)
   si a = ADFIC alors c ← NOMERRONE
   sinon  // on décale les éléments mem[a+1..lg]  vers mem[a..lg-1]
     i ← a
     tantque i < lg faire
          mem[i] ← mem[i+1] ;
          i ← i + 1
     ftq
     lg ← lg – 1 ; c ← NORMAL
fsi
```

Recherche dichotomique

- La recherche dichotomique est applicable dans le cas d'un **ensemble trié** représenté de manière **contiguë**.
- Elle peut être utilisée pour
  - Déterminer si la séquence de longueur **lg** comporte un élément **x** donné
  - Déterminer l'indice **k** d'insertion d'un élément **x** donné :
    - si lg = 0 ou x < T[1] : k est **1**
    - si lg > 0 et x ≥ T [lg] : k est **lg + 1**
    - si lg > 0 et T[1] ≤ x < T[lg] : k est l'indice vérifiant:

$$2 \leq k \leq lg \text{ et } T[k-1] \leq x < T[k]$$

Recherche dichotomique

Principe général:

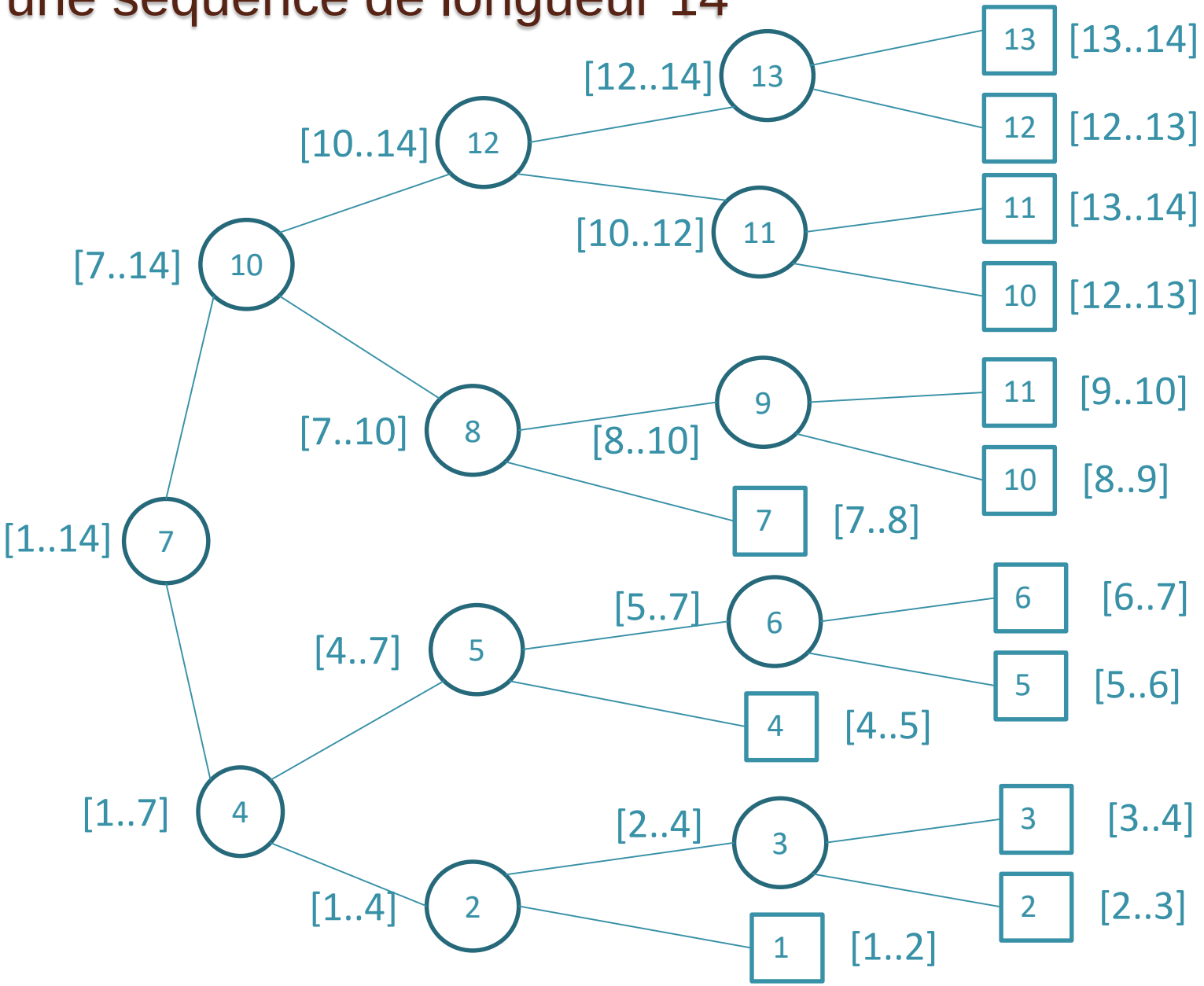
- On compare x à l'élément situé au milieu de l'ensemble et on détermine si l'on est dans l'un des 3 cas suivants :
  - On a trouvé x
  - x est inférieur à l'élément du milieu, s'il existe il est dans la partie supérieure du tableau
  - x est supérieur à l'élément du milieu, s'il existe il est dans la partie inférieure de la table
- L'intérêt de cette technique est de réduire de manière logarithmique le coût de la recherche mesuré en nombre de comparaisons d'éléments.

Recherche dichotomique

Algorithme de la recherche dichotomique

```
selon x, t
  x < t[1] : k ← 1
  x ≥ t[lg] : k ← lg + 1
  x ≥ t[1] et x < t[lg] :
        inf ← 1 ; sup ← lg              // lg > 1
        tantque inf < sup – 1 faire  // invariant :  t[inf] ≤ x < t[sup]
            m ← (inf + sup) div 2    // inf < sup – 1 ⇒ inf < m < sup
            si x < t[m]
            alors   sup ← m
            sinon  inf ← m
            fsi
        ftq
        // inf ≥ sup – 1 ⇒ t[inf] ≤ x < t[inf+1]   x est présent si x = t[inf]
        k ← inf+1
fselon
```

Recherche dichotomique : arbre de décision pour une séquence de longueur 14



Application au Répertoire trié: fonction adNom avec recherche dichotomique

```
privé fonction adNom(nm : Personne) → entier entre 0 et NMAX
   // adNom(n) renvoie l'adresse (l'indice) de l'élément de nom n
   // s'il existe, renvoie ADFIC sinon
lexique de adNom
      inf, sup : entier entre 1 et NMAX  // bornes de l'intervalle de recherche
      m : entier entre 1 et NMAX         // milieu de l'intervalle de recherche
      a : entier entre 0 et NMAX
algorithme de adNom
// on suppose les comparaisons < et > possibles entre chaines de caractères
si lg = 0 oualors (nm < mem[1].nom) ou (nm > mem[lg].nom)
alors  a ← ADFIC
sinon
      inf ← 1 ; sup ← lg
      tantque inf < sup - 1 faire     // invariant :  mem[inf].nom ≤ nm < mem[sup].nom
                  m ← (inf + sup) div 2
                  si  nm < mem[m].nom
                      alors   sup ← m
                      sinon  inf ← m
                  fsi
      ftq
      // inf ≥ sup – 1 ⇒ mem[inf].nom ≤ nm < mem[inf+1].nom    n est présent si n = mem[inf].nom
      si nm = mem[inf].nom  alors a ← inf sinon a ← ADFIC fsi
fsi
renvoyer(a)
```

Quelques techniques simples de tri d'un tableau

tri par insertion, tri par sélection, tri à bulles

Tri par insertion

Le tri par insertion est un algorithme de tri classique, que la plupart des personnes utilisent naturellement pour trier des cartes : prendre les cartes mélangées une à une sur la table, et former une main en insérant chaque carte à sa place.

Appliqué à un tableau, l'algorithme est le suivant :

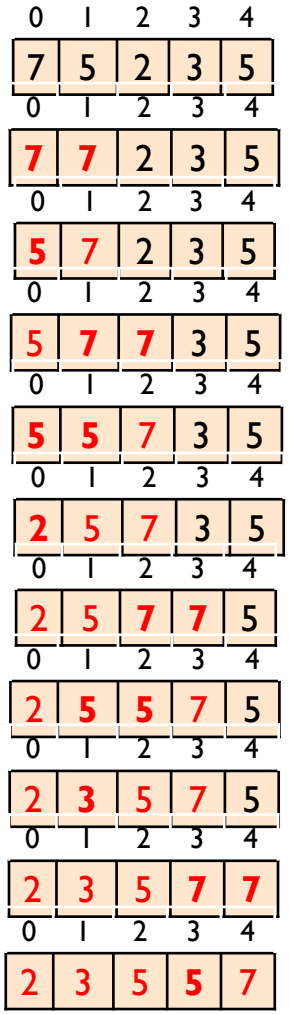
- On parcourt le tableau à trier du début à la fin. Au moment où on considère le i-ème élément, les éléments qui le précèdent sont déjà triés.
- L'objectif d'une étape est d'insérer le i-ème élément à sa place parmi ceux qui précèdent. Il faut pour cela trouver où l'élément doit être inséré en le comparant aux autres, puis décaler les éléments afin de pouvoir effectuer l'insertion. Ces deux actions sont effectuées en une passe, qui consiste à faire « remonter » l'élément au fur et à mesure jusqu'à rencontrer un élément plus petit.

Algorithme de tri par insertion d'un tableau de n éléments

```
lexique
t : tableau sur [0..n-1] d'Elements
i : entier entre 1 et n  // indice de parcours de t
j : entier entre 0 et n-1  // indice de parcours pour insertion de t[i]
e : Element                // intermédiaire : copie de t[i]

algorithme
    i ← 1 ;
    tantque i < n faire
        e ← t[i]
        j ← i - 1
        tantque j ≥ 0 etpuis e < t[j] faire
              t[j+1] ← t[j]      // on décale t[j]
              j ← j - 1
        ftq
        // j = -1 oualors e ≥ t[j] => on place e en t[j+1]
        t[j+1] ← e
        i ← i + 1
    ftq
```

Exécution sur un exemple



i	e	j	T[j]	T[j+1]
1	5	0	7	7
1	5	0	5	7
2	2	1	7	7
2	2	0	5	5
2	2	0	5	2
3	3	2	7	7
3	3	1	5	5
3	3	0	2	3
4	5	3	7	7
4	5	2	5	5

Intérêt et coût de l'algorithme

- Dans le pire cas, atteint lorsque le tableau est trié à l'envers, l'algorithme effectue de l'ordre de $n^2/2$ affectations et comparaisons.
- Si les éléments sont distincts et que toutes leurs permutations sont équiprobables, la complexité en moyenne de l'algorithme est de l'ordre de $n^2/4$ affectations et comparaisons.
- En général, le tri par insertion est beaucoup plus lent que d'autres algorithmes comme le tri rapide et le tri fusion pour traiter de grandes séquences.
- Le tri par insertion est cependant considéré comme le tri le plus efficace sur des tableaux de petite taille. Il est aussi très rapide lorsque les données sont déjà presque triées. Pour ces raisons, il est utilisé en pratique en combinaison avec d'autres méthodes.

Tri par sélection

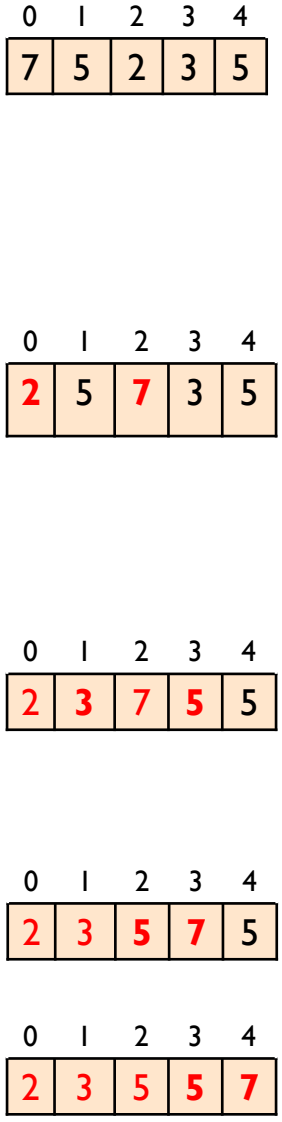
- Sur un tableau de n éléments (numérotés de 0 à n-1), le principe du tri par sélection est le suivant :
  - rechercher le plus petit élément du tableau, et l'échanger avec l'élément d'indice 0 ;
  - rechercher le second plus petit élément du tableau, et l'échanger avec l'élément d'indice 1 ;
  - continuer de cette façon jusqu'à ce que le tableau soit entièrement trié.

Algorithme de tri par sélection d'un tableau de n éléments

```
lexique
t : tableau sur [0..n-1] de Élément
i : entier entre 0 et n-1     // indice de parcours de t
j : entier entre 1 et n       // indice de recherche du plus petit élément
m : entier entre 0 et n-1     // indice du l'élément de valeur minimale rencontré
e : Element                   // intermédiaire pour échange

algorithme
    i ← 0 ;
    tantque i < n-1 faire
        // recherche du plus petit élément de t[i..n-1]
        m ← i
        j ← i + 1
        tantque j < n faire
              si t[j] < t[m] alors m ← j fsi
              j ← j + 1
        ftq
        si  i ≠ m alors  e ← t[m] ; t[m] ← t[i] ; t[i] ← e fsi     //  t[i] ↔ t[m]
        i ← i + 1
    ftq
```

Exécution sur un exemple



i	m	j	T[j]	T[m]	T[i]
0	0	1	5	7	7
0	1	2	2	5	7
0	2	3	3	2	7
0	2	4	5	2	7
0	2	5		7	2
1	1	2	7	5	5
1	1	3	3	5	5
1	3	4	5	3	5
1	3	5		5	3
2	2	3	5	7	7
2	3	4	5	5	7
2	3	5		7	5
3	4	4	5	5	7
3	4	5		7	5

Intérêt et coût de l'algorithme

- Dans tous les cas, pour trier *n* éléments, le tri par sélection effectue $n(n-1)/2$ comparaisons.
- Par contre, le tri par sélection n'effectue que peu d'échanges : $n-1$ échanges dans le pire cas, $n-(1/2+\ldots+1/n) \approx n - \ln(n)$ en moyenne.
- Ce tri est donc intéressant lorsque les éléments sont aisément comparables, mais coûteux à déplacer dans la structure.
- Il est également intéressant pour les petits tableaux ($n \leq 10$).

Tri à bulles

- Le tri à bulles ou tri par propagation est un algorithme de tri qui consiste à faire remonter progressivement les plus petits éléments d'un tableau, comme les bulles d'air remontent à la surface d'un liquide.
- L'algorithme parcourt le tableau, et compare les couples d'éléments successifs. Lorsque deux éléments successifs ne sont pas dans l'ordre croissant, ils sont échangés. Après chaque parcours complet du tableau, l'algorithme recommence l'opération.
- Lorsqu'aucun échange n'a lieu pendant un parcours, cela signifie que le tableau est trié. On arrête alors l'algorithme.

Algorithme de tri à bulles d'un tableau de n éléments

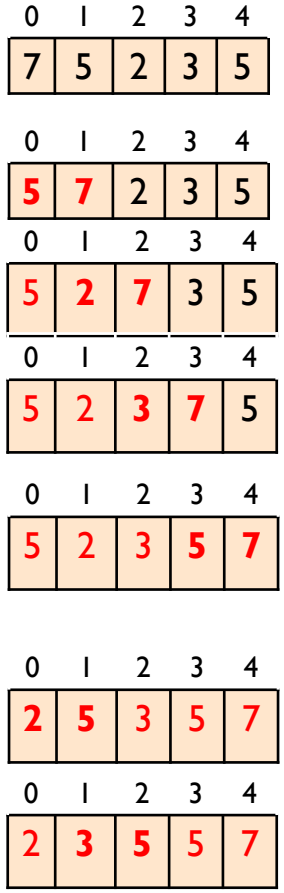
```
lexique
t : tableau sur [0..n-1] d'Elements
ifin : entier entre 0 et n-1 // borne supérieure de la partie non triée
i : entier entre 0 et n-1      // indice de parcours de t
e : Element                    // intermédiaire : pour échange
échange : booléen              // vrai s'il y a eu un échange

algorithme
    ifin ← n-1 ; échange ← vrai
    tantque ifin > 0 et échange faire
        i ← 0
        échange ← faux
        tantque i < ifin faire
            si t[i] > t[i+1] alors échange ← vrai
                e ← t[i] ; t[i] ← t[i+1] ; t[i+1] ← e
            fsi
            i ← i + 1
        ftq
        // le plus grand élément de t[0..ifin] et placé en t[ifin]
        ifin ← ifin - 1
    ftq
```

Algorithme de tri à bulles optimisé

```
lexique
t : tableau sur [0..n-1] d'Elements
ifin : entier entre 0 et n-1 // borne supérieure de la partie non triée
i : entier entre 0 et n-1     // indice de parcours de t
e : Element                   // intermédiaire : pour échange
j : entier entre 0 et n-1     // indice du dernier échange effectué, 0 si aucun échange
algorithme
    ifin ← n-1
    tantque ifin > 0 faire
        i ← 0
        j ← 0
        tantque i < ifin faire
            si t[i] > t[i+1] alors j ← i
                e ← t[i] ; t[i] ← t[i+1] ; t[i+1] ← e
            fsi
            i ← i + 1
        ftq
        // tous les éléments de t[j+1..n] sont triés et à leur place
        ifin ← j
    ftq
```

Exécution sur un exemple



ifin	i	j	T[i]	T[j]	T[i+1]
4	0	0	7	7	5
4	0	0	5	7	7
4	1	0	7	5	2
4	1	1	2	2	7
4	2	1	7	2	3
4	2	2	3	3	7
4	3	2	7	3	5
4	3	3	5	5	7
4	4	3	7	5	
3	0	0	5	5	2
3	0	0	2	2	3
3	1	0	5	2	3
3	1	1	3	3	5
3	2	1	5	3	5
3	3	1	5	3	7
1	0	0	2	2	3
1	1	0	3	2	5
0	1	0	3	2	5

Intérêt et coût de l'algorithme de tri à bulles

- Pour un tableau de taille n, le nombre d'itérations de la boucle externe est compris entre 1 et n. En effet, on peut démontrer qu'après la i-ème étape, les i derniers éléments du tableau sont à leur place. À chaque itération, il y a exactement n-i comparaisons et au plus n-i échanges.
- Le pire cas (n itérations) est atteint lorsque le plus petit élément est à la fin du tableau.
- Le nombre d'échanges d'éléments successifs est indépendant de la manière d'organiser les échanges. Lorsque l'ordre initial des éléments du tableau est aléatoire, il est en moyenne égal à n(n-1)/4.
- Le meilleur cas (une seule itération) est atteint quand le tableau est déjà trié. Dans ce cas, la complexité est linéaire.
- Le tri à bulles est souvent enseigné en tant qu'exemple algorithmique, car son principe est simple. Cependant, sa complexité en moyenne est de l'ordre de n^2,ce qui le classe parmi les plus mauvais algorithmes de tri. Il n'est donc quasiment pas utilisé en pratique.

Second exemple de gestion d'un ensemble: la course contre la montre

- On désire gérer l'affichage des résultats intermédiaires d'une course contre montre:
  - les coureurs partent l'un après l'autre à intervalles réguliers et font parcours individuel chronométré.
  - à chaque arrivée, le coureur est classé, l'affichage du classement provisoire est mis à jour.
  - un coureur peut être déclassé après son arrivée, alors qu'il apparaît déjà dans le classement.
- L'ensemble des coureurs engagés est connu à priori. On désigne par **NBCOUR** le nombre de coureurs engagés.
- A chaque coureur est associé un numéro de dossard compris entre 1 et NBCOUR.

Second exemple de gestion d'un ensemble: la course contre la montre

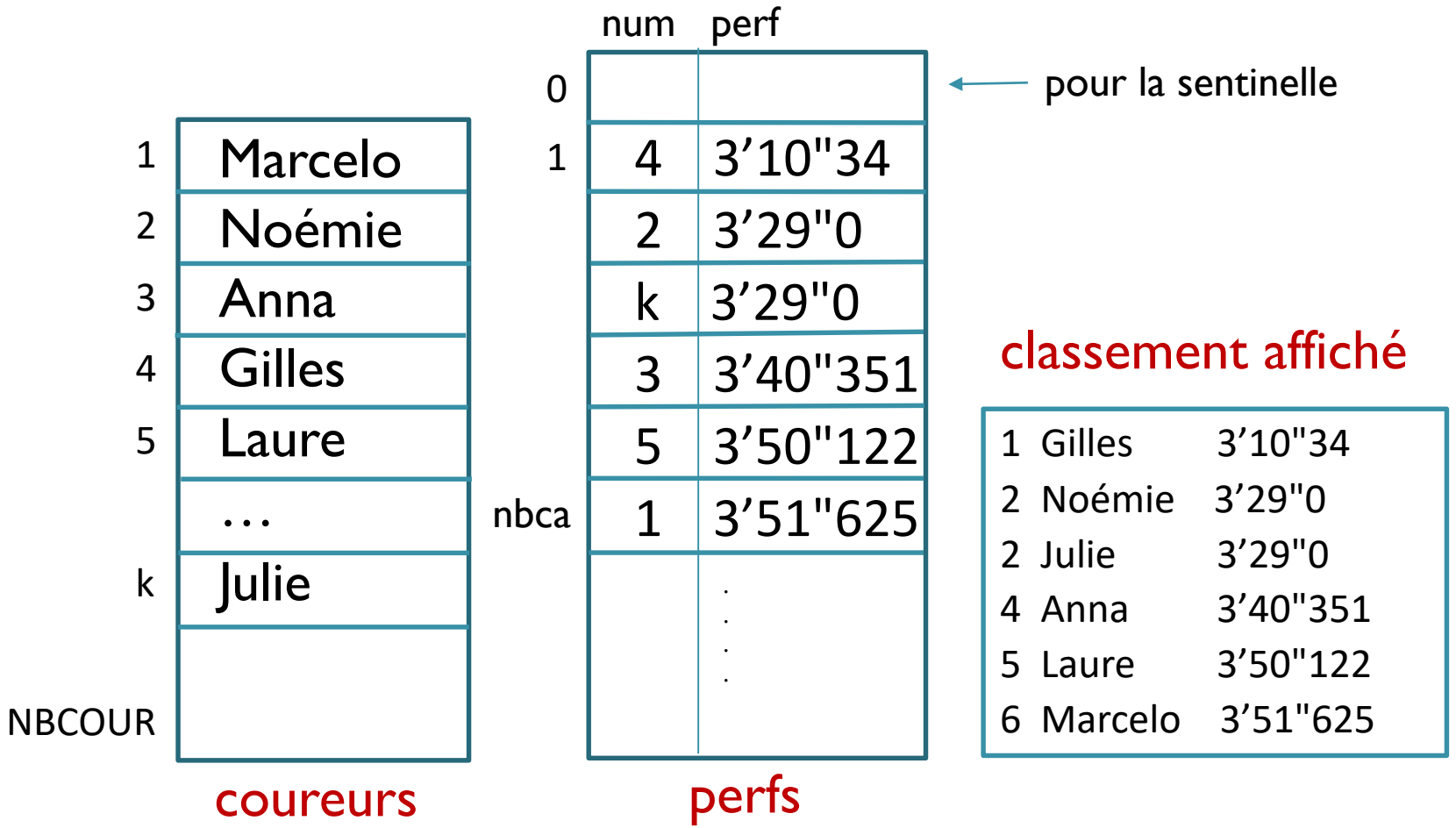
- Les performances des coureurs arrivés et classés constituent un ensemble doublets (numéro de dossard, temps de course).
- L'utilisateur du logiciel doit pouvoir :
  - **Initialiser** l'ensemble avant le début de la course
  - **Ajouter** une performance lors de l'arrivée d'un coureur
  - **Supprimer** une performance lorsqu'un coureur est déclassé
- Après chaque ajout/suppressions le logiciel réagit par l'**affichage du classement provisoire** par ordre croissant des temps de course : pour chaque coureur arrivé et classé, il fait apparaît son rang en tenant compte des ex-æquo, son nom et son temps de course.
- Comme dans l'exemple du répertoire téléphonique, nous n'étudions ici que le noyau fonctionnel.

Second exemple de gestion d'un ensemble: la course contre la montre

- La seule condition exceptionnelle correspond à un numéro de dossard erroné numéro déjà présent lors d'une arrivée, ou numéro absent lors d'un déclassement.
- L'ensemble des noms des coureurs engagés est représenté dans le tableau **coureurs** de taille NBCOUR défini sur [1.. NBCOUR], le nom placé à l'indice **i** correspond au coureur ayant **i** pour numéro de dossard.
- L'ensemble des performances des coureurs arrivés et classés est géré sous forme d'une séquence en ordre croissant des temps de course, mémorisée dans le tableau **perfs**.
- On considère des performances inférieures à une heure, données à la milliseconde.

Spécification (1)

- La figure suivante illustre l'état du système en milieu de course :



Spécification (2)

Lexique partagé

NBCOUR : entier > 0 // nombre de coureurs engagés
Dossard : type entier entre 1 et NBCOUR
Elément : type agrégat num: Dossard; perf: Durée fagrégat
Condition : type { NORMAL, NOMERRONE }
Durée : type agrégat m, s : entier entre 0 et 59 ; ms : entier entre 0 et 999 fagrégat

Classe Course

lexique de Course // variables partagées par les méthodes du répertoire
    coureurs : tableau sur [1..NBCOUR] de chaines // noms des coureurs
    perfs: tableau sur [0..NBCOUR] d'Eléments // classement de coureurs arrivés
    nbca: entier entre 0 et NBCOUR // nombre de coureurs arrivés et classés
  ...
méthodes de Course

  public action **initialiser**
   // Effet : création d'une séquence vide de coureurs arrivés
   // E.I. : indifférent
   // E/F.: mem représente le répertoire vide de taille n

Spécification (3)

public action **classer**(Consulté e : Element, Elaboré c : Condition)
// Effet : ajout de l'élément e au tableau perfs (classement)
// E.I. : indifférent
// E/F.: c = NORMAL : on a inséré e au classement
c = NOMERRONE : classement inchangé, il existe déjà un élément avec le même numéro que e.num

public action **déclasser**(Consulté d : Dossard, Elaboré c : Condition)
// Effet : supprime du classement la performance du coureur de dossard d
// E.I. : indifférent
// E/F.: c = NORMAL : on a supprimé la performance associée au dossard d
c = NOMERRONE : classement inchangé d est absent de perfs

public action **afficherClassement**
// Effet : affichage du classement des coureurs arrivés
// E.I. indifférent
// E.F. classement affiché, état du classement inchangé

Etude de l'affichage

- C'est un parcours de la séquence.
- On introduit la variable rang qui représente le dernier rang affiché. Ce rang n'est pas modifié en cas d'ex-aequo.

```
public action afficherClassement
// Effet : affichage du classement des coureurs arrivés
// E.I. indifférent
// E.F. classement affiché, état du classement inchangé
lexique de afficherClassement
  rang : entier entre 1 et NBCOUR     // dernier rang affiché
  a : entier entre 0 et NBCOUR        // indice de parcours du classement
  e : écran
  Fonction utilisée : cvDS(x: Durée) → réel > 0   // convertit une Durée en secondes
algorithme de afficherClassement
  si nbca = 0 alors e.afficher("aucun coureur classé")
  sinon   e.afficher(1, coureur[perfs[1].num], perfs[1].perf)
            a ← 2 ; rang ← 1 ;
          tantque a ≤ nbca faire
                si cvDS(perfs[a].perf)  > cvDS(perfs[a-1].perf) alors rang ← a fsi
                e.afficher(rang, coureur[perfs[a].num], perfs[a].perf)
                a ← a + 1
          ftq
  fsi
```

Classement d'un Elément

```
public action classer(Consulté e : Element, Elaboré c : Condition)
// Effet : ajout de l'élément e au tableau perfs (classement)
// E.I. : indifférent
// E/F.:  c = NORMAL : on a inséré e au classement
          c = NOMERRONE : classement inchangé, il existe déjà un élément avec le
               même numéro que e.num
 lexique de ajouter
   a : entier entre 0 et NMAX  // adresse d'insertion de e dans mem
   fonction utilisée : adNum  // adresse dans perfs d'un numéro de dossard donné
 algorithme de ajouter
     a ← adNum(e.num)
     si a ≠ ADFIC alors c ← NOMERRONE   // e.num déjà arrivé !
     sinon  c ← NORMAL
              perfs[0] ← e    // e.perf joue le rôle de valeur sentinelle
               a ← nbca
              tantque e.perf < perfs[a].perf faire
                  perfs[a+1] ← perfs[a] ;  a ← a – 1
              ftq
             // e.perf ≥ perfs[a].perf ⇒ on insère e en a+1
             perfs[a+1] ← e ;
             nbca ← nbca + 1
     fsi
```

Exercice

- Écrire l'algorithme du déclassement d'un coureur désigné par son dossard