7

Structures de tableaux

Nous savons utiliser les tableaux dans nos algorithmes pour y stocker des éléments de types primitifs et de types objets. Après avoir appris à créer nos propres classes, nous allons maintenant nous intéresser à imaginer et construire des classes permettant, tout comme les tableaux, de contenir des informations, mais avec plus de facilité et de possibilités.

La classe Vecteur

Présentation

En s'inspirant d'une classe très utile du langage Java (`java.util.Vector`), nous allons créer un outil facilitant l'utilisation d'un tableau.

> **Définition**
>
> **Vecteur**
>
> Un vecteur, tout comme un tableau, permet de stocker des éléments. Son utilisation est facilitée grâce à des méthodes d'insertion, de suppression et d'échange.

Chaque structure de stockage possède des inconvénients et des avantages. La classe `Vecteur` n'échappe pas à la règle :

- inconvénient : taille fixe ;
- avantage : rapide d'accès en lecture et en écriture.

Soit la classe `VecteurEntier` qui nous permettra de gérer un vecteur d'éléments de type entier (voir figure 7-1).

Figure 7-1
L'interface utilisateur de la classeVecteurEntier.

VecteurEntier
VecteurEntier ( ) VecteurEntier (nb: entier) setEntierAt (nb: entier, position: entier): vide getEntierAt (position: entier): entier getTaille ( ): entier echanger (pos1: entier, pos2: entier): vide

Détaillons l'utilisation de chaque méthode :

- `VecteurEntier()` permet de créer un vecteur de 5 éléments au maximum.
- `VecteurEntier(n: entier)` permet de créer un vecteur de n éléments au maximum.
- `setEntierAt(nb: entier, position: entier)` permet de fixer la valeur `nb` à l'élément numéro position.
- `getEntierAt(position: entier): entier` retourne la valeur de l'élément situé en numéro position.
- `getTaille(): entier` retourne la taille du tableau.
- `echanger(pos1: entier, pos2: entier)` échange les deux valeurs du tableau.

Pour bien comprendre l'utilisation d'un vecteur d'entiers, écrivons un petit algorithme permettant d'illustrer deux méthodes, et le schéma mémoire associé.

```
Algorithme test-Vecteur
variables: v1: VecteurEntier;
Debut
        v1 ← new VecteurEntier();
        v1.setEntierAt(35,0);
Fin
```

Représentons le schéma mémoire à la fin de l'exécution de l'algorithme précédent (figure 7-2).

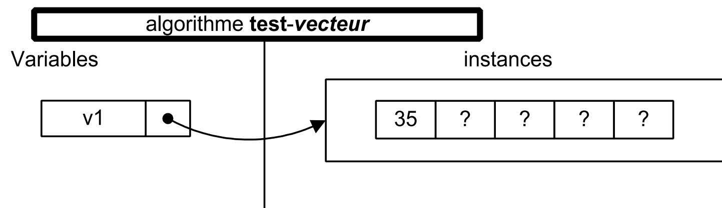


Figure 7-2
Schéma mémoire.

Écriture de la classe VecteurEntier

Les attributs

Définissons les attributs à encapsuler dans la classe `VecteurEntier` (figure 7-3). Il s'agit en fait d'un tableau d'entiers qui sera initialisé par le constructeur. Nous aurons aussi besoin de connaître la taille de ce tableau : introduisons l'attribut `taille` de type entier.

Figure 7-3

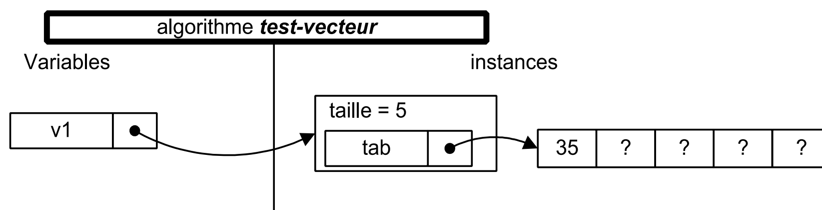
L'interface programmeur de la classe VecteurEntier.

VecteurEntier
– tab : tableau[ ] d'entiers – taille : entier
+ VecteurEntier() + VecteurEntier(nb: entier) + setEntierAt(nb: entier, position: entier): vide + getEntierAt(position: entier): entier + getTaille(): entier + echanger (pos1: entier, pos2: entier): vide

Représentons à nouveau le schéma mémoire de l'algorithme précédant, mais en introduisant les attributs encapsulés dans l'instance du vecteur d'entiers (figure 7-4).

Figure 7-4

Schéma mémoire avec les attributs.



Les constructeurs

Le constructeur de la classe `VecteurEntier` doit initialiser les attributs. Au début, il n'y a pas d'éléments dans le tableau.

Le constructeur par défaut crée un tableau de 5 éléments.

```
Classe VecteurEntier comporte methode VecteurEntier()
Debut
        this.taille ← 5;                  // l'attribut taille a une valeur
        this.tab ← new entier[5];         // l'attribut tab est initialisé
Fin
```

L'autre constructeur reçoit le nombre d'éléments en paramètre.

```
Classe VecteurEntier comporte methode VecteurEntier(taille: entier)
Debut
        this.taille ← taille;             // l'attribut taille a une valeur
        this.tab ← new entier[taille];    // l'attribut tab est initialisé
Fin
```

Les méthodes

Écrivons les autres méthodes, en commençant par celles qui permettent de modifier les éléments du tableau. Pour cela, il est utile de rappeler qu'à l'exécution de chaque méthode de la classe `VecteurEntier`, sont connus :

- l'instance courante `this` qui applique la méthode ;
- les attributs `taille` et `tab` (que l'on peut préciser par `this.taille` et `this.tab`) ;
- les paramètres et leurs valeurs qui ont été passés lors de l'appel de la méthode ;
- les variables locales.

Commençons par la méthode `setEntierAt` :

```
Classe VecteurEntier comporte methode setEntierAt(nb: entier, position: entier): vide
Debut
        tab[position] ← nb;
Fin
```

```
Classe VecteurEntier comporte methode getEntierAt(position: entier): entier
Debut
        retourne(tab[position]);
Fin
```

```
Classe VecteurEntier comporte methode getTaille(): entier
Debut
        retourne(taille);
Fin
```

La méthode qui permet d'échanger deux éléments utilise l'algorithme vu au chapitre 1. La seule différence porte sur la nature des variables manipulées : il s'agit ici des éléments de l'attribut `tab` de la classe `VecteurEntier`.

```
Classe VecteurEntier comporte methode echanger(pos1: entier, pos2: entier): vide
variables: tmp: entier;
Debut
        tmp ← tab[pos1];
        tab[pos1] ← tab[pos2];
        tab[pos2] ← tmp;
Fin
```

Le contrôle des erreurs d'utilisation

Nous n'avons pas abordé la possibilité de contrôler les erreurs éventuelles de l'utilisation de la classe `VecteurEntier` : que se passe-t-il pour l'algorithme suivant ?

```
Algorithme VecteurEntier-erreur-d-utilisation
variables: v1: VecteurEntier;
Debut
         v1 ← new VecteurEntier();
         v1.setEntier(4,100);
Fin
```

Une erreur se produit : le constructeur par défaut définit un tableau de 5 éléments, donc l'élément 100 n'existe pas.

L'écriture de votre classe n'est pas en cause : l'erreur a été commise par le programmeur de l'algorithme. Il aurait dû effectuer le contrôle et savoir comment utiliser la classe que vous lui avez donnée. Par contre, lorsque vous utilisez la classe de quelqu'un d'autre, veillez toujours à ne pas commettre ce genre d'erreur.

Un conseil

Ne mettez pas des tests dans vos méthodes pour anticiper et corriger les erreurs de ceux qui vont l'utiliser. En revanche, documentez vos méthodes pour qu'elles soient utilisées correctement.

Amélioration : le vecteur dynamique

Il est facile de faire en sorte que le vecteur ait une taille qui augmente si la limite a été atteinte. Il n'y aura alors plus de problème de taille maximale du vecteur pour l'utilisateur.

Nous avons à notre disposition deux techniques équivalentes pour ajouter cette fonctionnalité à la classe `VecteurEntier` :

- ajouter des nouvelles méthodes à la classe `VecteurEntier` ;
- créer une nouvelle classe `VecteurEntierPlus` qui hérite de la classe `VecteurEntier` où sont redéfinies seulement les nouvelles méthodes.

Choisissons de définir la nouvelle classe `VecteurEntierPlus` qui spécialise la classe `VecteurEntier` selon le schéma suivant. Profitons de cet exemple d'héritage pour rappeler les différentes questions auxquelles il faut avoir répondu.

Héritage

La classe à écrire doit répondre à la question : « Un objet de ma classe fille est-il un objet particulier de la classe mère ? » : figure 7-5.

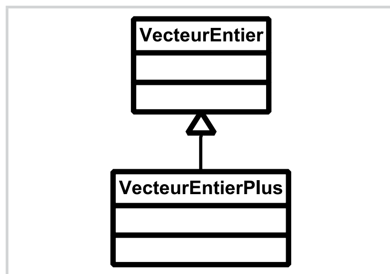


Figure 7-5
Héritage de VecteurEntier.

```
Classe VecteurEntierPlus specialise
VecteurEntier
Debut
Attributs :
        declaration des nouveaux attributs
Constructeurs :
        signature des constructeurs
Methodes :
        signature des methodes à modifier
        signature des nouvelles méthodes
Fin
```

Les nouveaux attributs

Quels sont les attributs spécifiques à la classe fille qui n'existent pas dans la classe mère ? Il faut alors les introduire dans la classe fille.

Il n'y a pas d'attribut supplémentaire.

Les constructeurs

Les constructeurs de la classe fille doivent être redéfinis : l'opérateur `super` est utilisé pour cela.

```
Classe VecteurEntierPlus comporte methode VecteurEntierPlus()
Debut
    super();
Fin
```

L'autre constructeur reçoit le nombre d'éléments en paramètre.

```
Classe VecteurEntierPlus comporte methode VecteurEntierPlus(taille: entier)
Debut
    super(taille);
Fin
```

Les méthodes

Il faut considérer chaque méthode de la classe mère, et une par une, décider s'il faut la redéfinir ou non.

Les méthodes à ne pas redéfinir

Lorsqu'une méthode sera utilisée telle quelle dans la classe fille, l'héritage nous dispense de la redéfinir (ce serait d'ailleurs une erreur).

Ainsi, la méthode `echanger(entier, entier): vide` n'a rien à voir avec la taille du vecteur : il est possible d'échanger des variables seulement si elles ont été initialisées.

La méthode `getEntierAt(position: entier): entier` fait aussi partie de cette catégorie. En effet, il n'est pas souhaitable de fournir cette valeur si elle est hors du tableau puisqu'elle n'aura pas été initialisée.

Les méthodes à modifier

Lorsqu'une méthode ne peut être identique dans la classe fille (si son action est différente, si elle doit mettre à jour un nouvel attribut…), il faut alors obligatoirement la redéfinir.

La méthode `setEntierAt(nb: entier, position: entier): vide` fait partie de cette catégorie. En effet, il faut augmenter la taille du vecteur si l'utilisateur veut placer une valeur en dehors des limites initiales.

```
Classe VecteurEntierPlus comporte methode setEntierAt(nb: entier, position: entier ): vide
Debut
   si (position > taille) alors
       nouvelleTaille(position+20);
   super.setEntierAt(nb, position);
Fin
```

Les nouvelles méthodes

Quelles sont les méthodes spécifiques à la classe fille qui n'existent pas dans la classe mère ? Il faut alors les introduire dans la classe fille.

La méthode qui augmente la capacité du tableau est nouvelle : il faut définir cette méthode en privé. Notez qu'il n'est pas possible que la dimension du tableau diminue.

Il suffit d'introduire une méthode qui crée un nouveau tableau avec plus de cases que le précédent. Cette méthode comporte 3 étapes :

1. créer un nouveau tableau avec suffisamment d'espace réservé ;
2. recopier les valeurs de `tab` dans le nouveau tableau ;
3. faire en sorte que `tab` référence le nouveau tableau, et affecter la nouvelle valeur de la taille.

```
Classe VecteurEntierPlus comporte methode nouvelleTaille(nouvelleTaille: entier): vide
variables: indice: entier;
           nouveauTab: tableau[] d'entiers;
Debut
 // (1) création d'un nouveau tableau plus grand, à l'image de l'existant
    nouveauTab ← new entier [nouvelleTaille];
 // (2) recopie
    indice ← 0;
    tant_que (indice < taille) faire
    {
             nouveauTab[indice] ← tab[indice];
             indice ← indice +1;
    }
  // (3) modification des attributs
    taille ← nouvelleTaille;
    tab ← nouveauTab;
Fin
```

Voici un schéma mémoire représentant l'évolution des valeurs lors de l'exécution de la méthode précédente (voir figure 7-6).

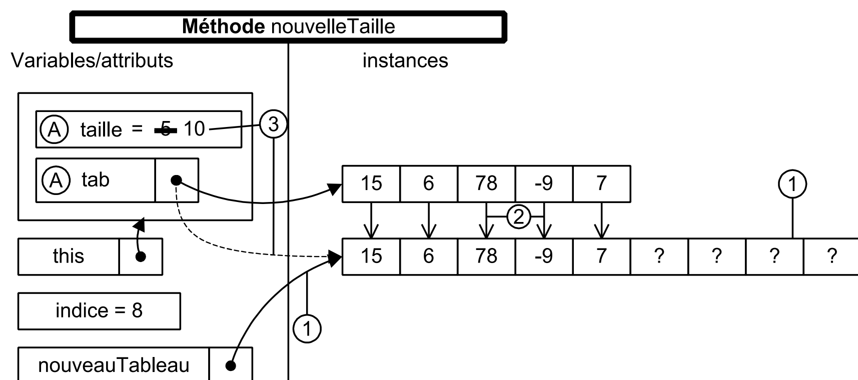


Figure 7-6
Exemple d'augmentation de la taille du tableau.

Les algorithmes de tri

Il existe de nombreuses méthodes pour trier par ordre croissant les éléments d'un tableau. Nous allons étudier les quatre plus classiques. Tous ces algorithmes sont intéressants dans la mesure où il n'y a pas de tableau intermédiaire à créer : le tableau initial est modifié par des permutations d'éléments. Ces techniques permettent de vérifier la dextérité avec laquelle vous manipulez les doubles boucles et les tableaux.

Il est utile de savoir que le verbe « trier » se traduit en anglais par « to sort ».

Nous avons à notre disposition deux techniques équivalentes pour ajouter des méthodes de tri à la classe `VecteurEntier` :

- ajouter des nouvelles méthodes à la classe `VecteurEntier` ;
- créer une nouvelle classe `VecteurTri` qui hérite de la classe `Vecteur` où sont redéfinies seulement les nouvelles méthodes.

Pour plus de simplicité, nous choisissons la première méthode. Les nouvelles méthodes de tri ainsi ajoutées à `VecteurEntier` appartiennent automatiquement à ses classes filles, donc à la classe `VecteurEntierPlus`.

Dans un souci de clarté, les tableaux à trier comportent uniquement des entiers. Mais il est possible de trier n'importe quel tableau d'éléments de type primitifs (ou types objets) à la seule condition d'avoir une opération (ou une méthode) permettant de comparer deux éléments. Il est par exemple possible de trier un tableau de `Date`, puisque la méthode `precede` permet de comparer deux dates.

Les tris simples

Le tri par sélection

Définition

Le tri par sélection

Le tri par sélection, appelé aussi tri par le min, permet de trier un tableau. L'algorithme parcourt le tableau pour identifier le plus petit élément, positionne ce dernier au début du tableau, et recommence l'opération.

Le même algorithme, le tri par le max, parcourt le tableau à la recherche de l'élément le plus grand pour le placer à la fin.

Cette méthode a l'avantage d'être facile à comprendre et à écrire mais s'avère peu rapide.

Un petit exemple permet de mieux comprendre l'évolution du tri (voir figure 7-7). Soit le tableau {7 ; 16 ; 5 ; 10 ; 2}. Chaque étape se fait en deux temps :

1. déterminer le minimum de la partie non triée du tableau ;
2. échanger le minimum avec la première case non triée.

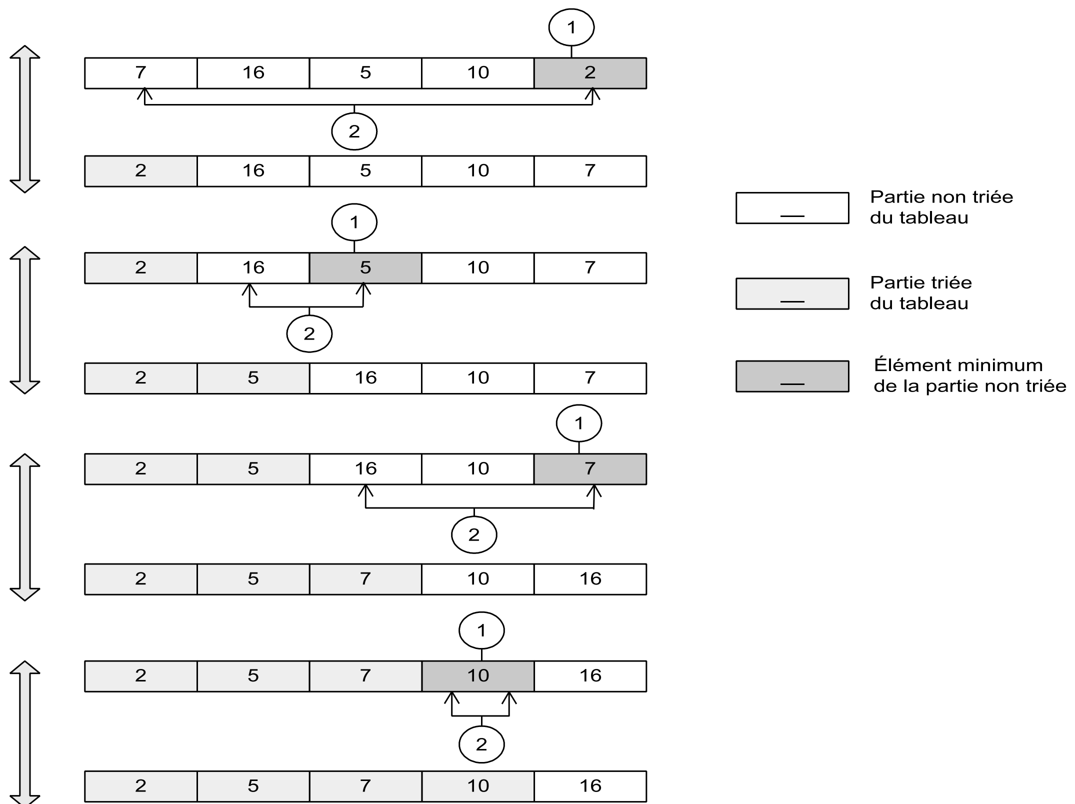


Figure 7-7

Exemple des différentes étapes du tri par sélection.

Ce tri parcourt tous les éléments de l'indice 0 au dernier. Il s'agit là de la boucle principale qui permet d'ajouter à chaque tour l'élément le plus petit restant dans la partie non triée du tableau. La variable `indice` indique cette position.

Pour chaque tour de la boucle principale, il y a deux opérations :

- Une boucle parcourt la partie non triée pour trouver le plus petit élément grâce à la variable `indiceNonTrie`.
- On procède à un échange entre le plus petit élément trouvé et le premier de la partie de tableau non triée.

```
Classe VecteurEntier comporte methode triSelection(): vide
variables: indice, indiceNonTrie, posMinimum: entier;
  // indice, indiceNonTrie et posMinimum sont des indices
           minimum: entier;
  // minimum est une valeur
Debut
  indice ← 0; // autant d'itération que d'éléments dans le tableau
```

```
    tant_que (indice < taille) faire
    {
        minimum ← tab[indice];
        posMinimum ← indice;

    // boucle de recherche du minimum
        indiceNonTrie ← indice;
        tant_que (indiceNonTrie < taille) faire
        {
            si (tab[indiceNonTrie] < minimum) alors
            {
                minimum ← tab[indiceNonTrie];
                posMinimum ← indiceNonTrie;
            }
            indiceNonTrie ← indiceNonTrie + 1;
        }
        echanger(posMinimum, indice);
        indice ← indice + 1;
    }
Fin
```

Le tri par insertion

Définition

Le tri par insertion

Le tri par insertion permet de trier un tableau. L'algorithme parcourt le tableau pour insérer chaque élément à la bonne place dans la partie triée du tableau.

Remarque

Le premier élément constitue toujours le tableau trié de départ.

Cette méthode présente l'avantage d'être très simple à comprendre et à mettre en œuvre, mais elle est lente en raison des décalages dus à l'insertion.

Ce tri parcourt tous les éléments de l'indice `1` au dernier. Il s'agit là de la boucle principale. On utilisera une variable `indice`.

Pour chaque élément de la boucle principale, l'insertion se fait en deux temps :

- Une boucle part de l'élément à insérer et décale tous les éléments plus grands d'une case vers la droite.
- Dès qu'on arrive à un élément plus petit ou au début du tableau, il suffit d'inscrire la valeur de l'élément à insérer. Cette valeur sera sauvegardée dans une variable `valeurAInserer`.

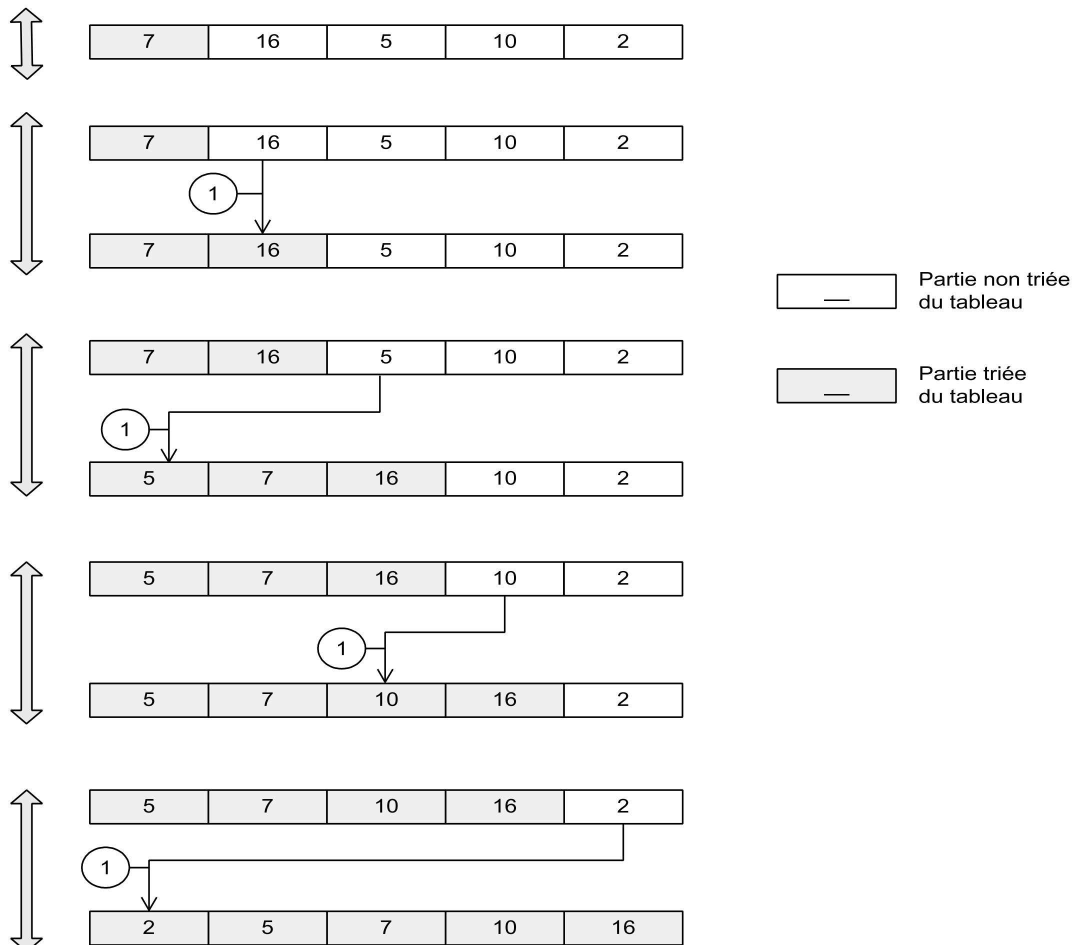


Figure 7-8

Exemple des différentes étapes du tri par insertion.

La partie la plus délicate est l'insertion : un exemple concret nous permettra de comprendre ce morceau d'algorithme avant d'écrire la méthode de tri par insertion entièrement (figure 7-8). Trois étapes sont nécessaires :

1. sauver la valeur à insérer dans une variable `valeurAInserer` ;
2. décaler d'un rang vers la droite tous les éléments plus grands que l'élément à insérer ;
3. placer la valeur à insérer (qui a été sauvée) à la place du dernier élément décalé.

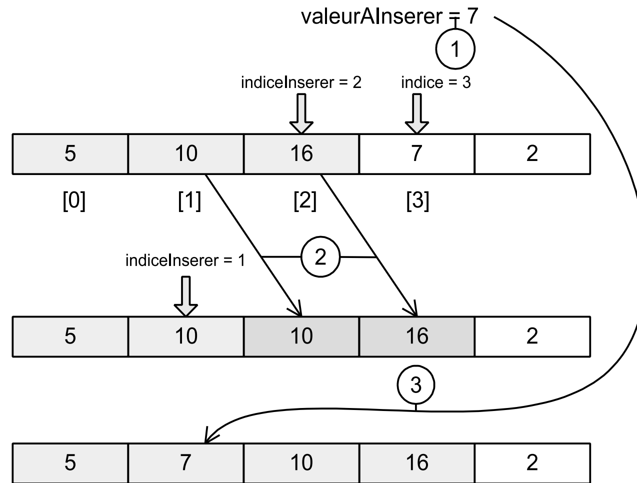


Figure 7-9

Insertion dans la partie déjà triée.

Incluons cet algorithme dans une boucle principale.

```
Classe VecteurEntier comporte methode triInsertion(): vide
variables: indice, indiceInserer: entier;
           valeurAInserer: entier;
Debut
    indice ← 1 ;
    // autant d'itération que d'éléments dans le tableau
    tant_que (indice < taille) faire
    {
    // décalage vers la gauche de tab[indice] : à sa place
        valeurAInserer ← tab[indice];
        indiceInserer ← indice – 1;
        // boucle de recherche du minimum
        tant_que ((indiceInserer ≥ 0)
             ET (valeurAInserer ≤ tab[indiceInserer])) faire
        {
            tab[indiceInserer + 1] = tab[indiceInserer];
            indiceInserer ← indiceInserer – 1;
        }
        tab[indiceInserer + 1] ← valeurAInserer;
        indice ← indice + 1;
    }
Fin
```

Le tri à bulle

Définition

Le tri à bulle

Le tri à bulle, appelé aussi tri bulle ou *bubble sort* en anglais, permet de trier un tableau. L'algorithme parcourt le tableau pour comparer les éléments deux à deux afin de faire descendre les valeurs les plus lourdes en bas du tableau.

Les éléments les plus légers (un peu comme les bulles d'air dans l'eau) vont remonter au début du tableau (à la surface).

Cette méthode présente l'avantage d'être très rapide si le tableau est presque trié (sauf quelques éléments), mais elle sera lente si les éléments du tableau ne sont pas du tout triés.

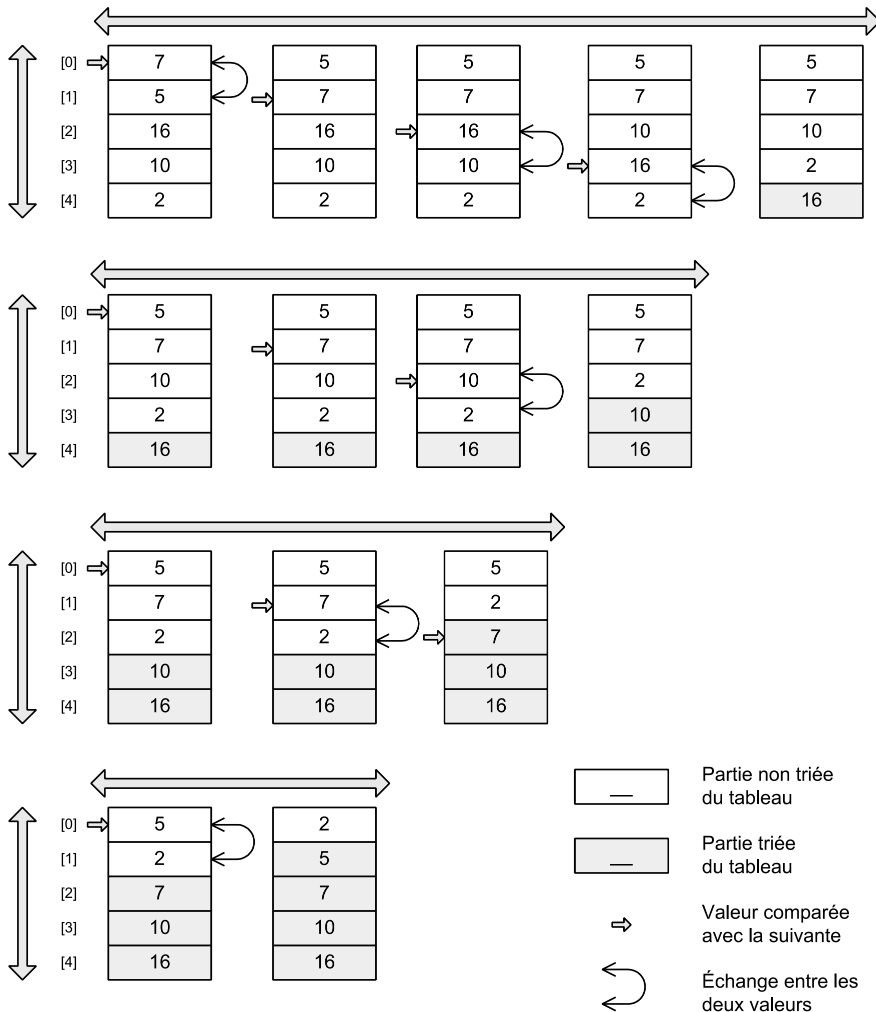


Figure 7-10

Exemple des différentes étapes du tri bulle.

La 1e itération parcourt le tableau pour descendre le plus lourd de l'indice `0` à l'indice `taille - 1`.

La 2e itération parcourt le tableau pour descendre le plus lourd de l'indice `0` à l'indice `taille - 2`, puisque le plus lourd est déjà en bas.

Appelons une variable `nbIteration` qui sera initialisée à `taille - 1`, et qui sera décrémentée jusqu'à ce que le tableau soit trié : il s'agit de la boucle principale.

Pour chaque boucle, introduisons une variable `indice` qui parcourt le tableau de `0` à `nbIteration` pour gérer les échanges successifs si nécessaire.

La boucle principale s'arrête quand le tableau est trié, ce qui se produit quand `nbIteration` atteint la valeur `1`, mais aussi si aucun échange n'a été effectué au tour précédent. C'est ce dernier cas qui sera implémenté avec l'introduction d'une variable booléenne `pasEncoreTrie`.

```
Classe TableauReelTrie comporte methode triBulle(): vide
variables: nbIteration, indice: entier;
           pasEncoreTrie: booléen;
Debut
    pasEncoreTrie ← vrai;
    nbIteration ← taille − 1;
    tant_que (pasEncoreTrie = vrai) faire
    {
        indice ← 0;
        pasEncoreTrie ← faux;
        tant_que (indice < nbIteration) faire
  // faire parcours des indices 0 à nb_iteration−1 pour comparer les éléments [indice] et [indice + 1].
        {
            si (tab[indice] ≥ tab[indice + 1]) alors
            {
                echanger(indice, indice + 1);
                pasEncoreTrie ← vrai;
  // on a fait un échange, il faut reparcourir le tableau (car 'trie' = Vrai)
            }
            indice ← indice + 1;
        }
    }   // si la variable trie n'a pas été modifiée dans la boucle, cela signifie que le tableau est complètement
        // trié : terminé !
Fin
```

La dichotomie

Avant d'étudier les méthodes de tri par dichotomie, il est indispensable de découvrir les bases de ce principe.

> **Définition**
>
> **Le traitement par dichotomie**
>
> La dichotomie consiste à subdiviser des données ou un problème en deux. Le traitement sur deux parties plus petites de moitié est en effet souvent plus simple.

La dichotomie, associée au concept de « diviser pour régner », permet de répartir le traitement d'une grande quantité de données en deux traitements de deux quantités moins importantes.

Souvent associée à la récursivité (pour diviser les données en 2, puis en 4, puis en 8…), cette technique est très performante. Attention, la performance a un coût : les algorithmes utilisant la dichotomie sont plus complexes, et de nombreuses erreurs peuvent apparaître dues aux effets de bord.

La recherche dichotomique

La méthode de recherche retourne la position de l'élément ayant la valeur recherchée. Si l'élément n'est pas dans le vecteur, la méthode retourne `-1`. Si la valeur recherchée apparaît plusieurs fois dans le vecteur, la méthode retourne l'une des positions.

Pour utiliser la recherche dichotomique, le tableau doit être déjà trié.

La recherche dichotomique consiste à partir d'un tableau déjà trié :

1. séparer le tableau en deux par un indice `milieu` (entre des indices `gauche` et `droite`) ;
2. comparer la valeur recherchée et la valeur située au milieu du sous-tableau ;
3. continuer la recherche dans un seul des deux sous-tableaux.

Il suffit de comparer la valeur recherchée et la valeur située au milieu du sous-tableau.

Recherche dichotomique itérative

```
Classe VecteurEntier comporte methode rechercherDicho(x: entier): entier
// retourne la position de la valeur cherchée
variables: gauche, milieu, droite: entier;
           trouve: booléen;
Debut
    gauche ← 0;
    droite ← taille – 1;
    milieu ← (gauche + droite) / 2;   // division entière pour trouver le milieu
    trouve ← Faux;

    tant_que ((gauche ≤ droite) ET (NON trouve)) faire
    // si on ne trouve rien, à un moment on a gauche ≥ droite
    {
        milieu ← (gauche + droite) / 2;   // recalcule le milieu (DIV)
        trouve ← (tab[milieu] = x);       // on a trouvé l'élément ?

        si (x > tab[milieu]) alors        // l'élément est à droite du milieu
            gauche ← milieu + 1;
        sinon                             // sinon, il est à gauche du milieu
            droite ← milieu – 1;
    }
    si (trouve = Vrai) alors
        retourne milieu;                  // on sort
    sinon
        retourne(–1);                     // élément introuvable : on sort
Fin
```

Recherche dichotomique récursive

La première méthode fait uniquement appel à la méthode (privée) récursive :

```
Classe VecteurEntier comporte methode rechercherDichoRecursif(x: entier): entier
Debut
    retourne(rechercherDichoRecursif(x, 0, taille-1));
Fin
```

La méthode récursive cherche la valeur `x` dans la partie de tableau située entre les indices `gauche` et `droite`.

```
Classe VecteurEntier comporte methode rechercherDichoRecursif(x: entier, gauche:
entier, droite: entier): entier
variables: milieu: entier;
Debut
    milieu ← (gauche + droite) / 2;     // division entière pour trouver le milieu
    si (tab[milieu] = x) alors          // deux conditions d'arrêt
        retourne(milieu);
    si (droite ≤ gauche) alors
        retourne(-1);

    si (x < tab[milieu]) alors          // appels récursifs
        retourne(rechercherDichoRecursif(x, gauche, milieu-1));
    sinon                               // sinon, il est à droite du milieu
        retourne(rechercherDichoRecursif(x, milieu+1, droite));
   Fin
```

Le tri par fusion : interclassement

> **Définition**
>
> **Le tri par fusion**
>
> Le tri par fusion permet de trier un tableau avec un traitement récursif et dichotomique. Par récursivité, chaque tableau est divisé en deux sous-tableaux qui sont triés puis refusionnés dans le bon ordre grâce à un tableau intermédiaire.

Nous avons à nouveau une méthode récursive à écrire, analysons une étape intermédiaire (voir chapitre 3).

Le tri demande à séparer le tableau en deux : il suffit d'introduire les indices `debut`, `milieu` et `fin` tels que `milieu = (debut+fin) / 2`. Grâce à la récursivité, ces deux sous-tableaux vont être triés. C'est la puissance de la récursivité, on doit supposer qu'ils ont été triés par la méthode qu'on est en train d'écrire !

```
Classe TableauReelTrie comporte methode triFusion(debut: entier, fin: entier): vide
variables: milieu: entier;
Debut
     si (debut ≠ fin) alors
     {
        milieu ← (fin+debut) / 2;
```

```
        triFusion(debut, milieu);
        triFusion(milieu+1, fin);
        interclasser(debut, milieu, fin);
    }
Fin
```

Cette méthode sera appelée de `0` à `taille–1` pour trier tout le tableau :

```
Classe TableauReelTrie comporte methode triFusion(): vide
Debut
   triFusion(0, taille–1);
Fin
```

Il faut alors interclasser (refusionner) les deux sous-tableaux triés comme le montre la figure 7-11.

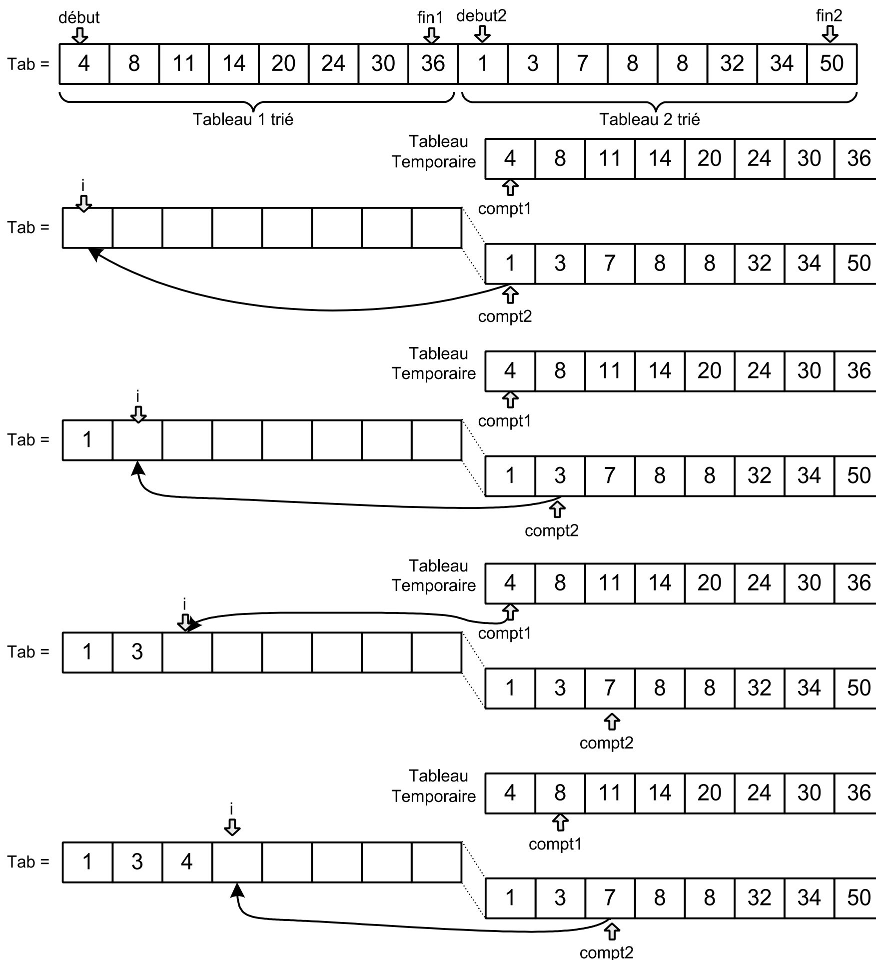


Figure 7-11

L'interclassement de deux tableaux triés.

On commence par faire une copie du 1er tableau dans le tableau temporaire. Les valeurs du 1er tableau sont alors sans importance : les cases ont été vidées sur le schéma pour l'indiquer.

On introduit 3 indices : les indices `compt1` et `compt2` pour parcourir les éléments du 1er et du 2e tableau, et l'indice `i` pour préciser l'élément du tableau `tab` qui va être modifié.

On compare l'élément de 1er tableau (grâce au tableau temporaire) avec celui du 2e tableau : le plus petit élément est copié dans `tab[i]` et on passe cet élément en incrémentant `compt1` ou `compt2`.

```
Classe TableauReelTrie comporte methode interclasser(debut: entier, fin1: entier,
fin2: entier): vide
variables: debut2: entier;
           tabTemp: entier[];
           compt1, compt2, i: entier;
Debut
    tabTemp ← new entier[fin1−debut1+1];
    debut2 ← fin1+1;
    // on recopie les éléments du début du tableau
    i ← 0;
    tant_que (i ≤ fin1) faire
    {
        tabTemp[i−debut1] = tab[i];
        i ← i + 1;
    }

    compt1 ← debut1;
    compt2 ← debut2;
    i ← debut1;

    tant_que ((i ≤ fin2) ET (compt1 ≠ debut2)) faire
    {
        si (compt2 =(fin2+1)) alors          // tous les éléments du second tableau ont été placés
        {
            tab[i] ← tabTemp[compt1−debut1]; // placer le reste du premier tableau
            compt1 ← compt1 + 1;
        }
        sinon si (tabTemp[compt1−debut1] < tab[compt2]) alors
        {
            tab[i] ← tabTemp[compt1−debut1];
            // ajouter un élément du premier tableau
            compt1 ← compt1 + 1;
        }
        sinon
        {
            tab[i] ← tab[compt2];            // ajouter un élément du second tableau
            compt2 ← compt2 + 1;
        }
        i ← i + 1;
    }
Fin
```

Le tri rapide : tri dichotomique récursif

> **Définition**
>
> **Le tri rapide**
>
> Le tri rapide permet de trier un tableau avec un traitement récursif et dichotomique. Par récursivité, un élément appelé *pivot* est choisi. Le pivot est alors placé à sa place définitive dans le tableau avec les éléments plus petits avant et les plus grands après. La récursivité traite les deux sous-tableaux avant et après le pivot.

Le tri rapide est récursif : il suffit d'analyser une seule étape du traitement pour pouvoir le comprendre et l'implémenter. En effet, comme nous l'avons vu au chapitre 3, pour écrire une fonction récursive, il suffit d'écrire la condition d'arrêt et une étape (la résolution au rang N) : les rappels récursifs (avec les appels aux rangs inférieurs) donneront la solution. Comme d'habitude, il est utile de dresser un schéma (voir figure 7-12).

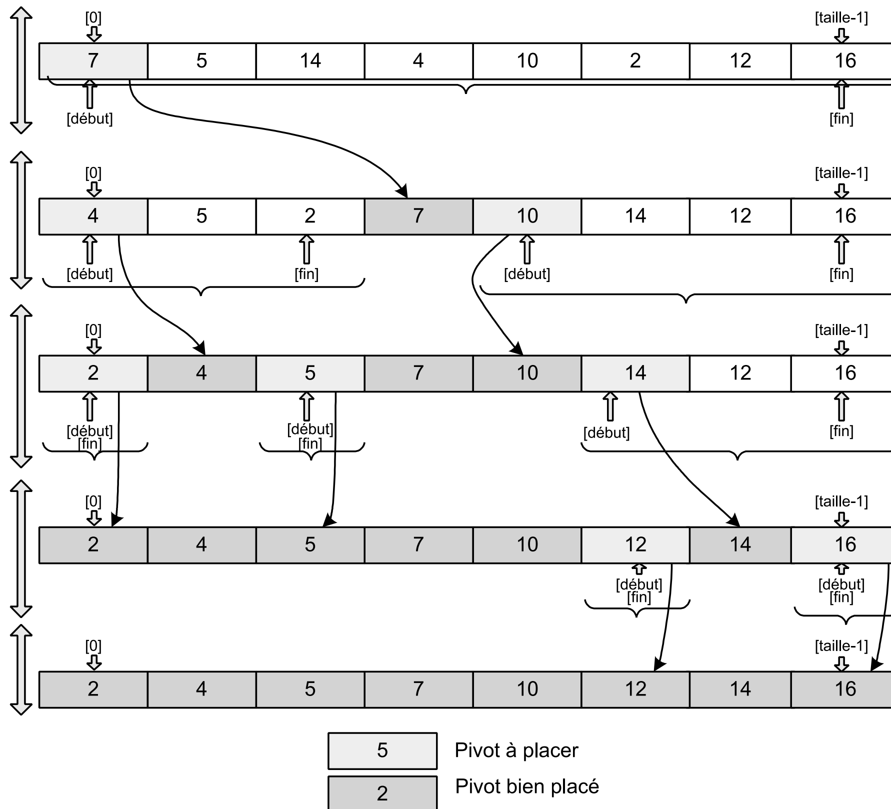


Figure 7-12

Les différentes étapes du tri rapide.

```
Classe TableauReelTrie comporte methode triRapide(debut: entier, fin: entier): vide
variables: pivot: entier;
Debut
      si (fin ≤ debut) alors
          retourne;                        // la condition d'arrêt
      pivot ← placePivot(debut, fin);      // mettre le pivot à sa place
      triRapide(debut, pivot–1);           // appel récursif de la partie gauche de tab[]
      triRapide(pivot+1, fin);             // tri récursif de la partie droite de tab[]
Fin
```

Cette méthode sera appelée de `0` à `taille–1` pour trier tout le tableau :

```
Classe TableauReelTrie comporte methode triRapide(): vide
Debut
        triRapide(0,taille–1);
Fin
```

Le problème le plus épineux reste le placement du pivot à la bonne place. Utilisons pour cela deux variables `indiceGauche` et `indiceDroite`, qui vont laisser à gauche les éléments plus petits que le pivot, et à droite les éléments plus grands (voir figure 7-13).

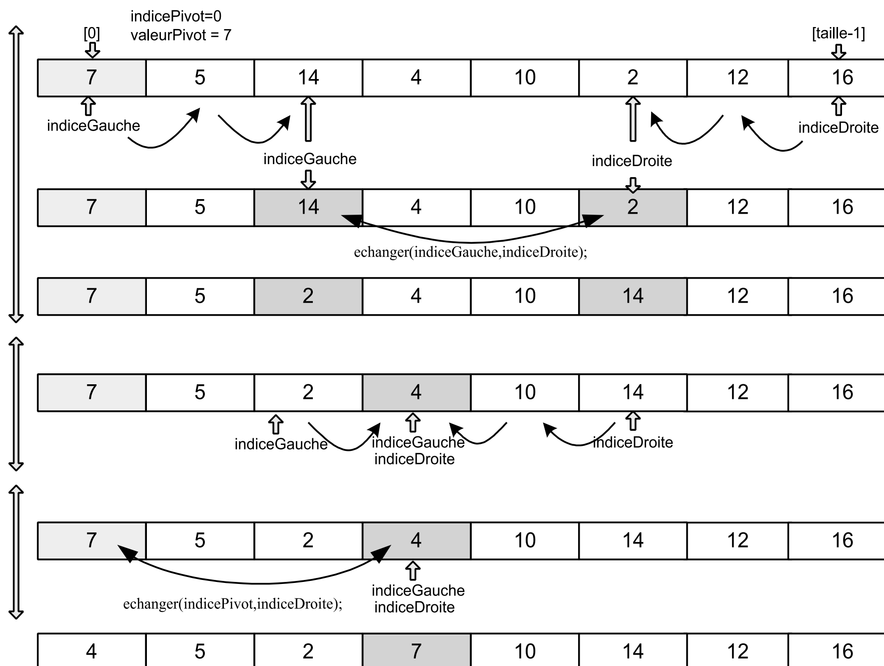


Figure 7-13

Placer le pivot à sa place.

On laisse à leur place les éléments plus petits que le pivot qui sont déjà à gauche : on passe au suivant (en incrémentant `indiceGauche`). On s'arrête quand `indiceGauche` indique un élément plus grand que le pivot (ici `14`).

De l'autre côté, on laisse à leur place les éléments plus grands que le pivot qui sont déjà à droite : on passe au suivant (en décrémentant `indiceDroite`). On s'arrête quand `indiceDroite` indique un élément plus petit que le pivot (ici `2`).

On échange alors les valeurs entre `indiceGauche` et `indiceDroite` (`14` et `2`).

Et on recommence jusqu'à ce que `indiceGauche` et `indiceDroite` désigne la même case. Il s'agit de la place finale du pivot.

```
Classe TableauReelTrie comporte methode placePivot(debut: entier, fin: entier): entier
variables: indicePivot, indiceGauche, indiceDroite: entier;
           valeurPivot: entier;
           pasPlace: booléen;
Debut
        indicePivot ← debut;
        valeurPivot ← tab[indicePivot];
        indiceGauche ← debut+1 − 1;      // on se place bien
        indiceDroite ← fin + 1;
        pasPlace ← Vrai ;                // il faut entrer dans la boucle
        tant_que (pasPlace) faire
        {
             indiceGauche ← indiceGauche + 1;
             tant_que ((indiceGauche ≤ fin) ET
                        (valeurPivot > tab[indiceGauche]) faire
             {
                indiceGauche ← indiceGauche + 1;
             }

             indiceDroite ← indiceDroite − 1;
             tant_que (valeurPivot < tab[indiceDroite]) faire
             {
                indiceDroite ← indiceDroite − 1;
             }
             // en général IndiceGauche et IndiceDroite se croisent
             si (indiceGauche ≤ indiceDroite) alors
             {
                echanger(indiceGauche, indiceDroite);
             } sinon
             {
                pasPlace ← Faux;
             }
        }
        echanger(indicePivot, indiceDroite);
        retourne indiceDroite;
Fin
```

Notion de complexité

Approche pratique

Un algorithme doit donner un résultat juste dans tous les cas, mais aussi s'effectuer de manière réaliste. La complexité représente l'évaluation du coût en mémoire utilisée et en temps de calcul d'un programme informatique. En effet, ces deux facteurs peuvent empêcher un algorithme, qui fonctionne sur le papier, de fournir un résultat, compte tenu des limites de vitesse et de capacité de stockage des ordinateurs.

De nos jours, la mémoire ne faisant pas défaut, le point sensible d'un programme reste son temps d'exécution : à quoi sert un programme qui fournira une réponse dans 200 ans ?

> **Définition**
>
> **La complexité en temps**
>
> La complexité d'un algorithme mesure le nombre d'opérations effectuées relativement au nombre N d'éléments traités.

Un algorithme peut avoir des résultats très différents en fonction des données initiales : le tri à bulle, par exemple, sera très rapide si les données sont déjà presque triées. Selon les cas favorables, défavorables ou les cas intermédiaires, il y a encore plusieurs complexités à calculer.

> **Définition**
>
> **La complexité dans le pire (respectivement le meilleur) des cas**
>
> Il s'agit de la complexité calculée lorsque les données demandent à l'algorithme le nombre maximum (respectivement le minimum) de traitements.

> **Définition**
>
> **La complexité en moyenne**
>
> La complexité en moyenne est la moyenne du nombre de traitements pour toutes les données possibles en entrée.

Nous nous intéresserons uniquement à la complexité en moyenne, en choisissant quelques jeux de données au hasard.

Comparons les algorithmes de tris vus précédemment pour des tableaux identiques. Pour cela, calculons le nombre total de comparaisons effectuées pour chaque tri : il suffit d'ajouter la variable `nbTest` et de l'incrémenter au bon endroit.

Le tableau 7-1 présente le résultat obtenu par le programme informatique après quelques minutes (N représente le nombre d'éléments).

Tableau 7-1 Comparaison de la complexité des différents exemples vus précédemment

	Sélection	Insertion	Bulle	Fusion	Rapide
N = 100	5.050	2.509	4.950	1704	539
N = 500	125.220	64.569	124.750	11324	3570
N = 1000	500.500	247.462	499.500	25165	7827
N = 2000	2.001.000	994.319	1.999.000	55229	18561
N = 5000	12.502.500	6.277.463	12.497.500	156382	48558

Nous constatons que le tri par fusion et le tri rapide sont effectivement les plus rapides (sur un tableau initialisé au hasard).

À l'aide d'une calculatrice, vous pouvez vérifier les résultats suivants (Log représente la fonction mathématique logarithme décimal, de base 10) : tableau 7-2.

Tableau 7-2 Résultats

N	N × N	N × Log(N)
N = 100	100 × 100 = 10.000	100 × Log(100) = 200
N = 500	250.000	1.349,48
N = 1.000	1.000.000	3.000
N = 2.000	4.000.000	6.602,06
N = 5.000	25.000.000	18.494,85

Nous constatons que les trois premiers tris sont à peu près proportionnels à $N \times N$ (tableau 7-3).

Tableau 7-3 Résultats

Sélection	Insertion	Bulle
~ 0,5 × N × N	~ 0,25 × N × N	~ 0,49 × N × N

Par exemple, pour le tri par insertion, pour $N = 1\,000$, $N \times N = 1\,000\,000$ et $0{,}25 \times N \times N = 250\,000$: ce nombre est peu différent du nombre de comparaisons calculé par l'ordinateur : 247 462.

Le recours à une calculatrice est nécessaire pour déterminer le facteur de proportionnalité entre les tris rapide et fusion avec $N \times \mathrm{Log}(N)$: $2{,}7 \times 100 \times \mathrm{Log}(100) = 540$, assez proche des 539 obtenus par l'expérience du tri rapide.

Tableau 7-4 Facteur de proportionnalité

Fusion	Rapide
~ 8,5 × N × Log(N)	~ 2,7 × N × Log(N)

La notation de l'efficacité d'un algorithme s'écrit sans tenir compte des valeurs de proportionnalité. Il suffit d'indiquer la croissance avec le nombre d'éléments N sous les formes suivantes :

- La complexité polynomiale de degré 2 est notée **O(N^2)** quand le traitement est proportionnel à $N \times N$: c'est le cas pour les tris par sélection, par insertion et à bulle.
- La complexité quasi-linéaire est notée **O(N × log(N))** quand le traitement est proportionnel à $N \times \mathrm{Log}(N)$: c'est le cas pour les tris par fusion et le tri rapide.
- La complexité linéaire notée **O(N)** intervient lorsque l'ensemble des éléments a été parcouru une seule fois. Il s'agit par exemple, de la complexité de l'algorithme de recherche linéaire d'une valeur dans un tableau.
- Un algorithme de complexité constante noté **O(1)** effectue toujours le même nombre d'opérations, quel que soit le nombre d'éléments N donnés.

Approche théorique

Calculons la complexité théorique du tri par sélection dans des cas simples. Supposons pour cela que le tableau à trier possède N éléments. Déterminons le nombre de comparaisons effectuées en fonction de N.

Pour trouver l'élément le plus petit, il faut parcourir tout le tableau. Cette opération est effectuée N–1 fois, avec (au début) N éléments, puis N–2 fois avec N–1 éléments restants, ainsi de suite jusqu'à ce qu'il ne reste que 2 éléments (quand il n'en reste qu'un, il n'y a plus de parcours à faire). Calculons le nombre de parcours du tableau :

Nombre de parcours = $(N - 1) + (N - 2) + \ldots + 3 + 2$

$\sim N \times (N - 1)/2$: on retrouve le terme le plus grand $\sim 0{,}5 \times N \times N$, trouvé par l'approche pratique.

La pile

Employée parfois pour sa simplicité, une structure de données peut vous être utile. Il s'agit de la pile.

Présentation

> **Définition**
>
> **Pile**
>
> Une pile est une structure de stockage de données. Les éléments sont ajoutés les uns après les autres dans la pile. L'utilisateur peut accéder seulement au dernier élément stocké.

Soit la classe `PileEntier` qui nous permettra de gérer des éléments de type entier (voir figure 7-14).

Figure 7-14
L'interface utilisateur.

PileEntier
+ PileEntier() + empiler(valeur: entier): vide + depiler(): entier + estVide(): booléen

Détaillons l'utilisation de chaque méthode :

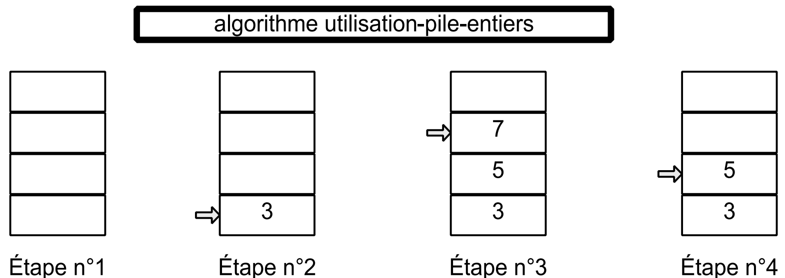
- `PileEntier()` permet de créer une pile capable de contenir 7 éléments au maximum.
- `PileEntier(n: entier)` permet de créer une pile de n éléments au maximum.
- `empiler(n: entier)` ajoute une nouvelle valeur n au sommet de la pile.
- `depiler(): entier` retire le sommet de la pile et retourne sa valeur.
- `estVide(): booléen` retourne `Vrai` si la pile n'a pas d'élément, `Faux` sinon.

Pour bien comprendre l'utilisation d'une pile d'entiers, écrivons un petit algorithme permettant d'illustrer chaque méthode et le schéma mémoire associé.

```
Algorithme utilisation-PileEntier
variables: p: PileEntier;
           valeur: entier;
Debut
     p ← new PileEntier();
  // étape n° 1
     p.empiler(3);
  // étape n° 2
     p.empiler(5);
     p.empiler(7);
  // étape n° 3
     valeur ← p.depiler();
  // étape n° 4
Fin
```

Représentons l'évolution des valeurs de la pile : la pile est vide au début, puis les valeurs s'empilent (voir figure 7-15).

Figure 7-15
Évolution de la pile.



Écriture de la classe Pile

Les attributs

La classe `PileEntier` peut être gérée par un tableau. Il faut définir un entier indiquant la position du sommet ou le nombre d'éléments déjà empilés. Choisissons la variable `nbElement` qui est égale à `0` quand la pile est vide, et qui augmente à chaque valeur empilée (voir figure 7-16).

Figure 7-16

L'interface programmeur de la classe PileEntier.

PileEntier
– tab: tableau[] d'entiers – nbElement: entier
+ PileEntier() + empiler(valeur: entier): vide + depiler(): entier + estVide(): booléen

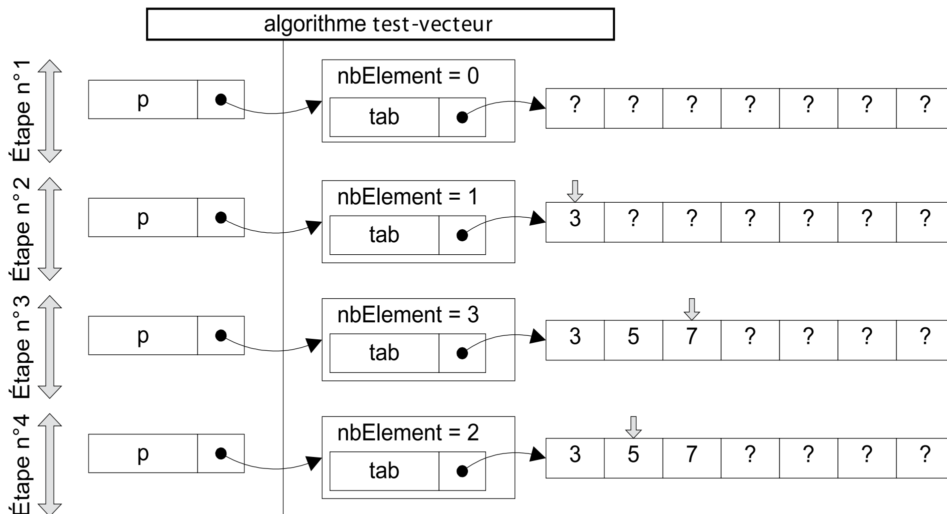


Figure 7-17

La classe PileEntier vue pour son programmeur.

Remarquons que la variable `nbElement` et la position du sommet de la pile dans le tableau ne sont pas identiques. À l'étape n° 2 par exemple, il y a un seul élément (`nbElement` vaut `1`) et la valeur `3` est à la position `0` (`tab[0]` vaut `3`).

L'écriture des méthodes est plus simple que pour le cas du vecteur. Commençons par le constructeur qui initialise les deux attributs.

```
Classe PileEntier comporte méthode PileEntier()
Debut
    this.nbElement ← 0;        // aucune valeur n'a été empilée
    tab ← new entier[7];       // l'attribut tab est initialisé
Fin
```

```
Classe PileEntier comporte méthode PileEntier(n: entier)
Debut
    this.nbElement ← 0;      // aucune valeur n'a été empilée
    tab ← new entier[n];     // l'attribut tab est initialisé
Fin
```

Avant d'écrire la méthode `empiler`, représentons le schéma montrant son action sur une pile contenant `[3 ; 5]` pour empiler la valeur `7` : figure 7-18.

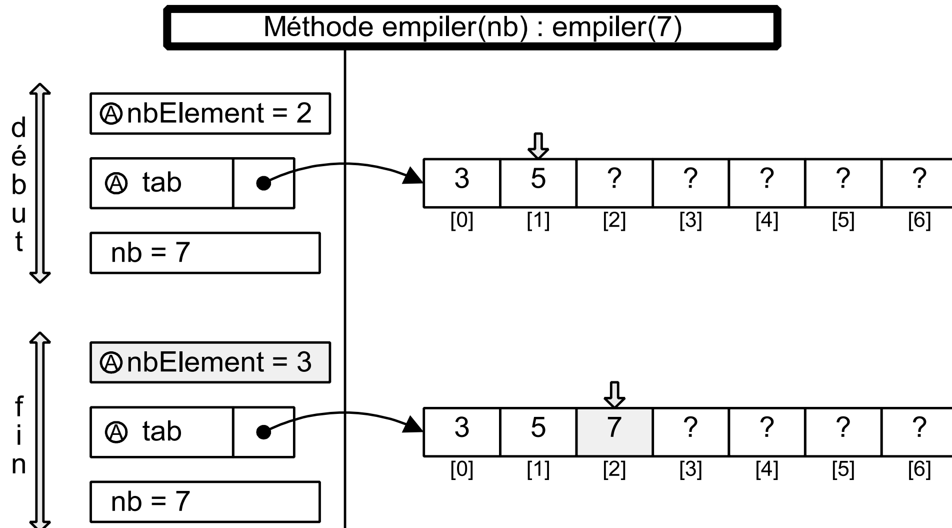


Figure 7-18

Empiler la valeur 7.

```
Classe PileEntier comporte methode empiler(nb: entier): vide
Debut
    tab[nbElement] ← nb;
    nbElement ← nbElement + 1;
Fin
```

Quant à la méthode `depiler()`, il s'agit d'empiler à l'envers.

```
Classe PileEntier comporte methode depiler(): entier
Debut
    nbElement ← nbElement – 1;
    retourne(tab[nbElement]);
Fin
```

> **Remarque**
>
> Il est inutile de modifier la valeur dépilée qui est dans le tableau : cette valeur est en effet inaccessible et sera écrasée au prochain empilement (disposer la valeur cachée 7 ou 0, c'est la même chose…).
>
> Néanmoins, si nous avions empilé des objets (des `Dates` par exemple), il faudrait obligatoirement mettre la case dépilée du tableau à `null` pour libérer l'espace mémoire de l'objet pointé.

La méthode `estVide()` ne modifiant pas les attributs, elle s'écrit simplement.

```
Classe PileEntier comporte methode estVide(): booléen
Debut
    retourne(nbElement = 0);
Fin
```

Exercices de bilan

Exercice 7.1 Implémenter la classe `VecteurEntierTrie`, un vecteur d'entiers où tous les éléments sont toujours triés.

Exercice 7.2 Écrire la classe `FileEtudiant` qui contient des étudiants : le premier étudiant entré dans la file sera le premier à sortir.

Exercice 7.3 Écrire un algorithme (simple utilisateur) pour afficher le nombre d'éléments d'une pile (donner une solution itérative et une solution récursive).

Exercice 7.4 Écrire une classe `TestTri`, comportant un tableau et une méthode de tri par sélection, et ajouter le calcul du nombre de tests effectués, utile notamment pour déterminer la performance.